Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock

Sumit Kumar Monga
Virginia Tech

Sanidhya Kashyap
EPFL

Changwoo Min
Virginia Tech

Abstract

RDMA-capable networks are gaining traction with datacenter deployments due to their high throughput, low latency, CPU efficiency, and advanced features, such as remote memory operations. However, efficiently utilizing RDMA capability in a common setting of high fan-in, fan-out asymmetric network topology is challenging. For instance, using RDMA programming features comes at the cost of connection scalability, which does not scale with increasing cluster size. To address that, several works forgo some RDMA features by only focusing on conventional RPC APIs.

In this work, we strive to exploit the full capability of RDMA, while scaling the number of connections regardless of the cluster size. We present Flock, a communication framework for RDMA networks that uses hardware provided reliable connection. Using a partially shared model, Flock departs from the conventional RDMA design by enabling connection sharing among threads, which provides significant performance improvements contrary to the widely held belief that connection sharing deteriorates performance. At its core, Flock uses a connection handle abstraction for connection multiplexing; a new coalescing-based synchronization approach for efficient network utilization; and a load-control mechanism for connections with symbiotic send-recv scheduling, which reduces the synchronization overheads associated with connection sharing along with ensuring fair utilization of network connections. We demonstrate the benefits for a distributed transaction processing system and an in-memory index, where it outperforms other RPC systems by up to 88% and 50%, respectively, with significant reductions in median and tail latency.

CCS Concepts: • **Networks → Data center networks**; • **Hardware → Networking hardware**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP '21, October 26–29, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483576

Keywords: Remote Memory Access, Network hardware

ACM Reference Format:
Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. 2021. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483576

1 Introduction

Network communication is one of the key ingredients in determining the performance of datacenter applications. These applications require good performance in the form of high throughput and low latency from the network, which the Remote Direct Memory Access (RDMA) capable networks promise. These networks have not only become a commodity but also are now part of datacenters. An RDMA-capable network avoids the overhead of conventional network protocol stacks by bypassing them. Moreover, it provides reliable one-sided reads and writes of remote memory that provides off-the-shelf hardware-based reliable packet delivery and no remote CPU involvement. Because of such performance implications, large-scale distributed applications, such as RPC-based systems for lower latency [14, 20, 41, 42] and even resource disaggregation [13, 17, 32, 35] are using it.

Despite the popularity of RDMA, there is a long standing debate on finding the balance between efficiently utilizing the advanced capabilities of RDMA and scaling the number of connections. In particular, several works use one-sided operations over the reliable connection that ensures no packet drop [13]. On the other hand, some works forgo this in the favor of two-sided operations over unreliable datagram that provides conventional RPC like APIs [20, 21] and scales to a large number of nodes without any hardware issues, such as cache thrashing on RDMA-capable NIC (RNIC). Meanwhile, other works use the best of both worlds as they specialize various algorithm phases (*e.g.*, distributed transactions) with particular RDMA operations [41].

Although one-sided operations do not involve remote CPU, it comes at the cost of scalability bottleneck in hardware. In particular, the RNIC, which maintains connection information for each pair of connection, suffers from cache thrashing because of its limited memory. Unfortunately, this information can spill over the host memory resulting in a huge penalty on a cache miss because RNIC has to fetch data over the PCIe, which can be several microseconds. Meanwhile, developers minimize this thrashing by using unreliable datagram (UD) that avoids pair-wise connection information.

Moreover, it becomes a suitable choice for high fan-in, fan-out cluster scenarios [12, 22, 30]. However, this approach leads to developers forgoing the hardware-provided reliable connection and they must ensure application reliability and congestion control mechanisms that can impose additional overhead [21]. Moreover, unreliable datagram does not support one-sided operations, which limits the design of RDMA-enabled distributed applications.

In this work, we revisit the idea of sharing connections among threads, especially in a high fan-in fan-out cluster [12, 30]. In particular, we advocate for a partially shared model—sharing connections among threads—that earlier approaches forgo, as the synchronization cost overshadows the benefits of sharing a connection [13]. For instance, we find that a lock-based connection sharing is up to 2.3× slower than our approach. Thus, we tackle the problem of connection sharing by designing a scalable RDMA communication framework that focuses on three dimensions: First is maintaining the scalability of connections regardless of the number of client threads and nodes in a cluster. Second is versatility, *i.e.*, allowing users to leverage all the features of RDMA—such as RPC, memory, and atomic operations—unlike RPC-only frameworks [18, 20–22]. Finally, ensuring the packet reliability from the ground up by using the hardware-provided reliable connection that removes the software overhead of congestion control and fault tolerance.

Thus, we design a new RDMA-based framework, called Flock. With versatility and reliability as important design goals, Flock relies on the hardware-supported reliable connection (RC) that not only removes the aforementioned software overheads but also supports all RDMA operations, such as RPC, memory, and atomic. We particularly optimize Flock for multi-threaded applications that are widely used within datacenters [7, 40].

At its core, Flock uses a combination of three critical components to maintain connection scalability, *i.e.*, high throughput and low latency, without RNIC cache thrashing for a high fan-in, high fan-out network traffic pattern. First is using an indirection layer, *i.e.*, one more abstraction atop the existing RDMA-based connection: queue pair (QP). We call this *connection handle* abstraction that multiplexes application threads with connections. We propose a programming interface that exposes the setup and all RDMA operations. Second, instead of using lock-based synchronization, we introduce a new synchronization mechanism, called Flock synchronization. It is based on a *leader-follower* coalescing-based coordination, in which a thread becomes the leader and coalesces requests from other concurrent threads (followers) into a single message request. Our leader selection procedure is dynamic and efficient. Thus, with our synchronization approach, we save more CPU cycles by reducing serialization and even improve the network bandwidth utilization. Finally, we avoid the cache thrashing with our *symbiotic send-recv scheduling* by efficiently mapping connections to threads.

	MTU size	One-sided verbs			Two-sided verbs
		`read`	`atomic`	`write`	`send/recv`
RC	2 GB	✓	✓	✓	✓
UC	2 GB	✗	✗	✓	✓
UD	4 KB	✗	✗	✗	✓

Table 1. RDMA operations and MTU sizes supported by each transport type. In RC transport, the RNIC is responsible for retransmission following a packet loss. UC and UD do not use ACKs for reliable packet delivery requiring packet loss to be handled by the application. The MTU for UD is limited to 4KB and transferring data larger than this requires splitting payload into 4KB chunks. As packet reordering is possible in UD, application receiving the data must handle packet reassembly.

Our approach consists of receiver-side connection scheduling that controls which connection should be used by the client threads by activating or deactivating them dynamically to avoid cache thrashing on RNIC; and sender-side thread scheduling that assigns threads to share an active connection. This approach leads to minimizing head-of-line blocking issue that arises due to varying message size as well as better network bandwidth utilization.

This paper makes the following contributions:

- We address the problem of cache thrashing for the reliable connection with: connection handle abstraction, leader-follower based synchronization mechanism, and symbiotic send-recv scheduling.
- We design and implement a new RDMA communication library, called Flock, that ensures the goal of achieving connection scalability, programming versatility, and off-the-shelf hardware reliability.
- Our evaluation shows that Flock outperforms FaSST [20] by up to 140% and 88% for distributed transaction processing with a read-intensive and a write-intensive workload respectively along with a reduction in 99th percentile latency by up to 71% and 50% respectively. Moreover, Flock delivers up to 50% throughput improvement and a reduction in 99th percentile latency by up to 32% over eRPC [21] with HydraList [25].

2 Background and Motivation

We first briefly describe RDMA and RNIC (§2.1), then introduce the challenges and research efforts in scaling RDMA communication (§2.2). Finally, we summarize the RDMA scalability challenges that we aim to solve in this paper (§2.3).

2.1 Remote Direct Memory Access (RDMA)

RDMA NIC (RNIC). RDMA enables a host to access the memory of a remote host bypassing the kernel. This allows zero-copy transfers with low latency and reduced CPU consumption. To achieve these gains, RDMA NICs (RNICs) implement several layers of network stack in hardware so that applications can directly access the RNIC from user space. Applications use RDMA by registering memory regions (MRs) with the RNIC for remote access.

Queue pair (QP). An RDMA host establishes communication by creating queue pairs (QPs) wherein each QP consists of a send queue and a receive queue. An application submits requests (verbs) to these queues via a user-space library (*i.e.*, `libibverbs` [2]). Each QP is associated with a completion queue (CQ), which contains events indicating the completion status of previously submitted verbs. The RNIC performs the DMA of completion events onto the CQ, on which the application polls.

Primitive types. RDMA supports two types of verbs namely, message and memory verbs. *Message verbs* include `send` and `receive` verbs. They require a receiver to post receive buffers before a sender posts a send request. The sender's payload gets written to one of the receive buffers. Since these verbs require CPU involvement at both the sender and the receiver, such verbs are called *two-sided verbs*. On the other hand, *memory verbs*, which include `read`, `write` and `atomic` operations, are called *one-sided verbs*, since they do not involve CPU on the remote host. The sender specifies the remote memory address to operate upon in its request, which gets executed by the RNIC on the remote host without consuming any CPU. As a result, these verbs have lower latency and higher throughput than message verbs.

Transport types. Current RDMA implementations offer three transport types : *reliable connection (RC)*, *unreliable connection (UC)*, and *unreliable datagram (UD)*. Connected transports like RC and UC require a one-to-one connection between QPs, whereas UD enables one QP to communicate with multiple remote QPs. Additionally, RC ensures reliable packet delivery, unlike UC and UD. The transports also differ in the verbs supported by each of them as well as the maximum transmission unit (MTU), which we summarize in Table 1.

Thus, there are some advantages of using RC. First, it enables access to the full suite of message and memory verbs, thereby providing the opportunity for minimizing CPU usage at the remote host. Second, reliability provided by the hardware reduces complexity and overheads in the software stack. On the other hand, UC and UD transports lack these benefits and hence require integrating reliability concerns (*e.g.*, packet loss/reordering) within the application in addition to their other limitations (*e.g.*, limited MTU size).

2.2 Scalability in RDMA

Challenges. Scaling RC connection is challenging in large clusters [8, 13, 20, 21, 32, 37, 39]. This stems from RNIC's inability to cache the state for all the connections handled by it due to its limited size. Figure 1 illustrates that the RNIC uses its SRAM to cache information, such as virtual-to-physical address translation for the registered memory regions as well as their permissions and the state corresponding to each of its QPs. Unfortunately during cache misses, RNIC issues PCIe reads [13, 19] to fetch the necessary state from memory that degrades performance. These observations have lead to a long debate over which RDMA transport and verbs should be used for scalable communication with good performance.

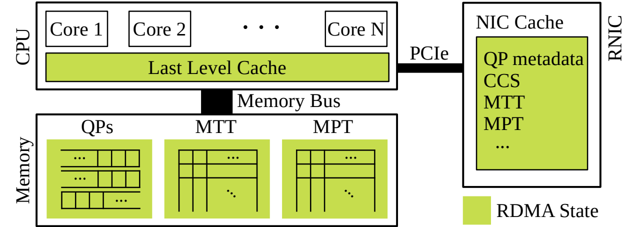

Figure 1. RDMA NIC (RNIC) Architecture. RNIC caches active connection state – queue pair (QP) metadata, congestion control state (CCS) [27, 43] – as well as the memory translation table (MTT) and memory protection table (MPT). Upon NIC cache misses, RNIC fetches the data from main memory via DMA over PCIe.

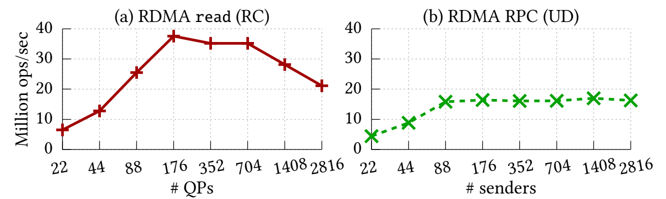

Figure 2. Performance of RDMA `read` (RC) and UD-based RPC with increasing number of QPs.

RC-based approach. Systems, such as FaRM [13] and Storm [32], use RC QPs to take advantage of the low latency and CPU-efficient memory verbs. To tackle the scalability issue, FaRM uses QP sharing between threads and registers memory regions using 2 GB huge pages. Storm, which uses recent RNICs (Mellanox ConnectX-4) that are equipped with improved cache management, registers physical memory to eliminate the metadata needed by the RNIC for translating virtual addresses to physical ones. However, the connection-oriented nature of RC means the state RNIC caches will increase as the cluster size increases, making it particularly worse for high fan-in, high fan-out communication patterns.

To validate this, we run an experiment using Mellanox ConnectX-5 NICs with 22 client nodes and one server node. The clients issues 16-byte RDMA `reads` to the server. Figure 2(a) shows that the read performance peaks between 176-704 QPs followed by a sharp drop, as the number of QPs increase further. This is due to the RNIC's inability to cache the state corresponding to all the connections.

UD-based approach. On the other hand, systems, such as HERD [18], FaSST [20], and eRPC [21] use UD to communicate with multiple remote hosts using a single QP.[1] Reduction in the number of QPs and QP-attached memory regions translate to a significant reduction in the RNIC managed state. However, UD has several drawbacks such as (1) access to only message verbs which have high CPU cost;

[1]HERD uses a mix of UC and UD.

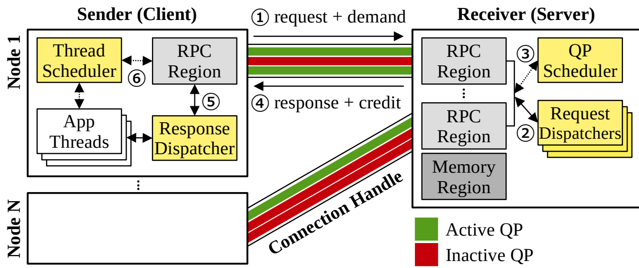

Figure 3. Flock architecture.

(2) software overheads as re-transmissions and congestion control need to be handled in software (*e.g.*, eRPC [21]).

Besides the congestion control overhead, Figure 2 shows that performance gap can be up to 2× with high remote CPU utilization. At 352 senders in Figure 2(b), the server consumes more than 90% of its CPU cycles within the Mellanox user-space libraries. Most of these cycles are spent on recycling receive buffers (`ibv_post_recv`) and polling the completion queue (`ibv_poll_cq`) for detecting incoming requests. Such high CPU overheads lead to fewer cycles for application processing.

Hybrid approach. Another class of systems use a hybrid of message and memory verbs. One such example is DrTM+H [41], a distributed transaction processing system that uses different types of verbs for different phases of a transaction. For instance, DrTM+H uses a mix of one-sided `reads` and UD-based RPC for the transaction execution phase. However, their findings are limited to OLTP systems and may not apply to other applications.

2.3 Challenges in Scaling RDMA Communication

Although latest RNICs have improved with better caching and prefetching, they do not come with more memory probably because of cost factors and increased power consumption [21]. Hence, we do not expect that the newer generation of RNICs will not suffer from this scalability problem.

In summary, using RC is beneficial, as it allows flexibility at the application to choose from different types of verbs and provides reliability out-of-the-box without remote CPU intervention. However scaling its performance for large connection counts is imperative, which remains an open challenge. Our aim is to address this challenge with Flock.

3 Flock Overview

Flock aims to provide a scalable RDMA communication framework targeting a challenging traffic pattern: a high fan-in, high fan-out network that is commonly used in today's datacentres [12, 30]. Moreover, Flock exposes all RDMA primitives—RPC and memory—to applications and minimizes software-induced overheads [21] by utilizing off-the-shelf hardware-supported reliable connection (RC). In addition, Flock departs from the conventional design of RDMA stacks. For instance, we revisit the idea of QP sharing that is known to suffer from synchronization overheads. Flock addresses this overhead and maintains scalable application performance, *i.e.*, high throughput and low latency, with three contributions: a *connection handle* abstraction, coalescing-based *Flock synchronization*, and a load-control mechanism with *symbiotic send-recv scheduling*, comprising of sender-side thread scheduling and receiver-side QP scheduling.

Figure 3 shows Flock in action. Each sender node and receiver node are connected by QPs. After establishing the connection with the *connection handle* abstraction, the sender uses Flock synchronization to send requests, *i.e.*, one of the sender's threads collects and coalesces a set of requests in one message, which it sends to the server using an active QP. The sender also asks for keeping the QP active with a credit renewal scheme and provides other QP utilization metrics to the server. After receiving the message, the server performs two tasks: First, it uses the request dispatcher to process all requests in the message (②). Second, it initiates the QP scheduler for QP scheduling, which activates/deactivates them based on the received utilization metrics (③). The server then responds back with processed requests as well as the renewed credit if required (④). Once the sender receives the response, its response dispatcher notifies appropriate application threads (⑤). Now, the sender uses the thread scheduler for sender-side thread scheduling, *i.e.*, the thread scheduler schedules/migrates threads from a deactivated QP to an active one (⑥) based on the server response for credits.

Connection handle. To efficiently utilize QPs, we introduce a connection handle abstraction on top of QPs that allows multiple threads to share them. Table 2 lists Flock's major APIs, which we categorize into (1) connection set up and memory region registration, (2) RPC client/server (sender/receiver), and (3) memory APIs. The sender can issue RPC and memory operations, while the receiver only needs to handle incoming RPC requests because memory operations do not consume any CPU on the receiver side.

Our programming API is based on the *connection handle*, which establishes one-to-one connectivity between two RDMA nodes, as illustrated in Figure 3. When a user establishes a connection to a remote node (`fl_connect`), Flock creates a representative connection handle and all other APIs operate on the connection handle to communicate to the remote node. Internally, Flock manages a set of RC QPs for a connection handle. Each connection handle attaches one or more RDMA MRs (`fl_attach_mreg`), wherein separate MRs are used for RPC and memory operations.

As for our implementation, Flock uses native OS threads. It does not implement or depend on a specific thread library compared to prior works in high-performance networking [33, 34]. However, it can be easily extended to integrate with a user-level thread library (*e.g.*, §8.5.2).

Supported functionality. Our API currently supports the most common operations including RPCs, remote memory

	API	Description
Init	`fl_connect`	Connect to a remote node
	`fl_attach_mreg`	Attach a memory region for memory operations
RPC client	`fl_send_rpc`	Send an RPC request with an RPC ID and data
	`fl_recv_res`	Receive RPC responses
RPC server	`fl_reg_handler`	Register an RPC handler function with an RPC ID
	`fl_recv_rpc`	Fetch RPC requests
	`fl_send_res`	Send an RPC response with data
Memory Atomics	`fl_read`	Read from a remote memory for a given size
	`fl_write`	Write to a remote memory for a given size
	`fl_fetch_and_add`	Atomically fetch data and add a remote memory
	`fl_cmp_and_swap`	Atomically compare and swap a remote memory

Table 2. Major Flock APIs.

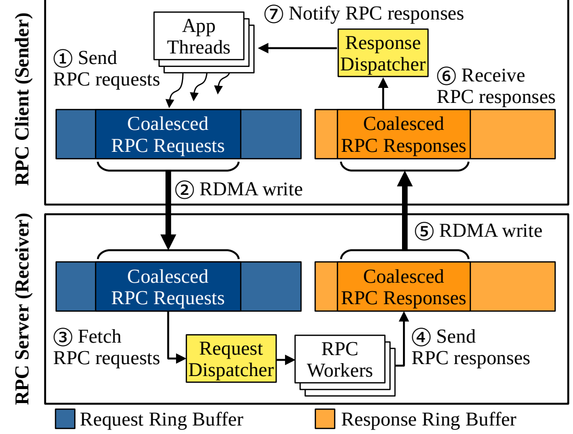

Figure 4. Overview of Flock's RPC.

verbs and atomics. Extending Flock to support more advanced functionality available in current RDMA implementations (*e.g.*, cross-channel communication, extended atomics) is part of our future work.

Summary. Our connection handle abstraction shares multiple threads with QPs, which implies sharing hardware resources. Flock addresses this with its coalescing-based Flock synchronization, in which threads that share a QP, progress concurrently with minimal synchronization. In addition, our approach efficiently utilizes the network bandwidth by coalescing small messages and further reduces the CPU overhead. Meanwhile, our symbiotic send-recv scheduling maintains a balance of QPs that avoids RNIC cache thrashing on the server, while maintaining client's performance.

4 Flock Remote Procedure Call

We describe the detailed design of Flock's zero-copy RPC. Flock registers a set of memory regions between nodes after establishing the connection (`fl_connect`). We logically create two ring buffers, a *request buffer* and a *response buffer*, as shown in Figure 4. Clients prepare RPC requests in their local request buffer (①), and then submit to the server's request buffer using RDMA writes (②). Subsequently, the server dispatches the RPC requests from its local request buffer (③), then it uses RDMA writes to send RPC response from its local response buffer (④) to the client-side response buffer (⑤). Finally, the client dispatches the RPC responses from the local response buffer (⑥) and notifies application threads (⑦).

Although this is a simple messaging primitive, the challenge is dynamically sharing a QP and its associated memory region among application threads on the client side. Hence, our RPC is designed to efficiently support the above described messaging primitive. We first describe our message layout and then present the key components powering our RPC layer.

4.1 Message Layout

Figure 5 shows the message layout for both RPC request and response. This layout enables coalescing of multiple small requests (or responses) into a single larger message, transferring the coalesced message using a single RDMA `write` (① in Figure 5). Each message has four parts: the header (H), metadata (M), data (D), and canary (C). The header contains information, such as total coalesced message length, the number of requests (or responses) in the message, and the expected canary value. The header is followed by one or more pairs of metadata and data of individual requests (or responses). Metadata contains the size of the associated data, the thread ID, its sequence ID, and the RPC handler ID to execute a request on the server with the request payload (data). Sequence ID is a thread-local monotonically increasing identifier that the server returns along with a response. This allows one-to-one mapping between a thread's outstanding request and an incoming response. The canary is a 64-bit random value. The thread generates the canary and places it in the header as well as the end of the message. The receiver has the complete message if the canary matches at both places, assuming RDMA writes are performed in order of increasing memory addresses [13].

The receiver detects a new message by polling its request buffer using the control variable `Head` (② in Figure 5). Upon receiving the message, the receiver first checks whether message is complete through the canary value. It then updates `Head` to point to the next message location. Finally, it decodes the message into individual requests with the number of requests present in the message header.

Before sending a message, a sender ensures that there is free space on the receiver's ring buffer. For this, the receiver's `Head` is cached locally at the sender (*i.e.*, sender's copy of `Head`), which it can fetch using an RDMA `read`. However, the sender rarely reads because the receiver sends its updated `Head` value as a part of the response message. This allows the sender to update its `Head` without performing an RDMA `read`.

4.2 Flock Synchronization

Since multiple threads share a QP, the major challenge lies in efficiently utilizing the RNIC. To address this issue, we employ a form of leader-follower coordination named *Flock*

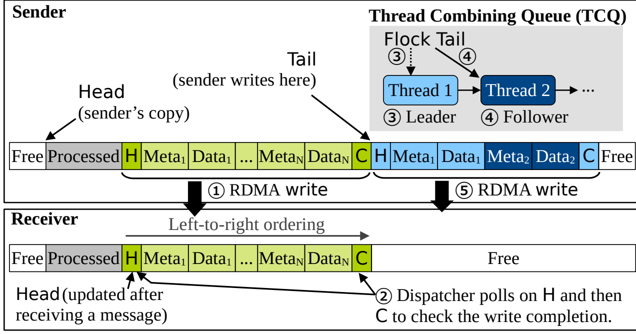

Figure 5. RPC message layout and message coalescing using Flock synchronization.

synchronization for QP sharing. An application thread acting as the *leader* coalesces RPC requests from other concurrent threads—*followers*—and later issues an RDMA `write` for the coalesced message.

To efficiently choose a leader in a dynamic fashion, Flock maintains a *thread combining queue (TCQ)* for each QP (③, ④ in Figure 5). Each TCQ maintains a queue of concurrent threads, where `Flock Tail` points to the queue tail. An application thread first enqueues itself to the TCQ to access a QP by updating `Flock Tail` using an atomic `swap` operation. The TCQ follows an update protocol similar to the MCS queue lock [26]; If `Flock Tail` is `null` after the `swap` operation, the thread is at the head of the TCQ and it becomes a leader (③). Otherwise, it becomes a follower (④).

A leader provides memory buffers to other concurrent threads as per their requested payload. Every thread first copies its payload into the provided buffer and updates its copy-completion flag. The leader polls on each thread's copy-completion flag; once set, it finally sets up the message header and canary, and issues an RDMA `write` for the coalesced message (⑤ in Figure 5). Note that the leader provides a bounded number of buffers to the followers to ensure application-level progress of the leader. On reaching this limit, the leader performs the aforementioned steps of copying, coalescing, and writing the message. It then hands over the leadership to the first follower in the TCQ whose request is not part of the coalesced message. We would like to emphasize that the notion of an application thread being a leader or follower in Flock is a transient one with a scalable leader selection based on TCQ.

In Flock, the number of requests within a coalesced message depends on the concurrent threads using a shared QP. In the absence of concurrency, a coalesced message contains only the request belonging to the leader. On the other hand, coalescing in the company of concurrency comes with a certain synchronization cost but has several advantages to it as well. Coalescing is beneficial because it reduces the number of MMIO operations required to submit RPC requests, saving CPU cycles in the process (§8.3.1). Also, sending a single large message can be more efficient than sending many small messages since every message on the wire requires additional metadata, such as network transport information. Additionally, our message format (§4.1) requires per message headers and canary. Thus, coalescing reduces the number of bytes sent, thereby making more efficient use of network bandwidth. To reduce contention on the shared resources, the leader is responsible for polling completion events, in addition to managing the request buffer.

4.3 Request Processing and Response Notification

The server registers an RPC handler for each RPC ID by invoking `fl_reg_handler` during bootstrap. It detects new coalesced messages containing RPC requests by polling its request buffer (② in Figure 5). It then executes the RPC handler corresponding to each request and prepares the response in its response buffer, as shown in Figure 4. An application can execute the request either with a RPC dispatcher thread or with an application-managed pool of RPC workers. The server tags the request sequence ID to its response metadata so the client can match a response to a corresponding request with that ID. Similarly, thread ID is tagged to the response metadata. As the message format used by the server is similar to the one used at the client, RPC response are also coalesced into larger messages. Moreover, the server piggybacks its control variable `Head` along with the coalesced response, which the server posts using an RDMA `write`.

Besides reducing the overhead of QP sharing among threads with our Flock synchronization (§4.2), we further reduce contention on the response buffer to maintain RPC scalability. As shown in Figure 4, we use a response dispatcher thread that polls the response buffer and relays the available response to appropriate application threads according to the tagged thread ID in each response. Since application threads access their respective response disjointly, the dispatcher enables parallel data access to the response buffer as well as its memory reclamation. A dispatcher thread does not interact with the RDMA stack or is involved with application-specific processing, making its job relatively lightweight. As a result, it can work across multiple QPs thereby managing multiple response buffers.

5 Flock Symbiotic Send-Recv Scheduling

Achieving good performance and scalability are goals that require somewhat contrasting design choices particularly with RC. Although threads using dedicated QPs enable more parallelism within the RNIC [19], it hampers scalability in a client-server setting with high fan-in, high fan-out communication patterns. The reason is due to the RNIC cache thrashing and CPU overload on the server (Figure 2). On the other hand, sharing a QP among application threads at the client improves connection scalability but comes at the cost of performance [20, 21], which we also observe (§8.3.1).

We propose *symbiotic send-recv scheduling* to achieve these two contrasting design goals. QPs at the server are categorized into active and inactive so as to limit the number of QPs actively served by the server. FLOCK distributes active QPs among client threads based on their recent communication activity with the server. The approach of limiting active QPs enables us to remain scalable with increasing client count and FLOCK synchronization limits any performance degradation caused due to QP sharing. On the client side, FLOCK assigns active QPs to application threads that mitigates head-of-line blocking.

5.1 Receiver-side QP Scheduling

FLOCK's QP scheduler (Figure 3) on the server executes the receiver-side QP scheduling algorithm, which minimizes RNIC cache thrashing and server CPU overload. The scheduler activates or deactivates QPs based on the utilization metrics exposed by clients. These metrics include a cooperative credit renewal scheme and a contention metric on a QP. Based on these metrics, the QP scheduler periodically distributes QPs; *i.e.*, it allocates more active QPs to contended or active clients.

QP scheduler. The server runs a dedicated scheduler thread, *QP scheduler*. QP scheduler is in charge of (1) handling (granting) credit renew requests, (2) updating QP utilization statistics based on the reported coalescing degree, and (3) periodically redistributing QPs among the senders based on the collected QP utilization statistics for each sender. QP scheduler assigns more QPs to senders that suffer from higher QP contention and send requests more frequently. The sender, particularly the leader (§4.2), requests for more credits as well as shares the QP contention metric with the receiver.

Metric: Credit renewal. A credit represents the number of requests a client can send to a receiver, which decreases after posting the request. The credit system allows the scheduler to control the maximum QP load the server can handle. Moreover, the scheduler issue credits on a per QP basis to avoid cross-QP synchronization on both the sender and the receiver. The renewal of credits works as follows: During bootstrap, each sender gets `C` (default = 32) credits to initiate its application processing. After consuming half of the credits, a sender requests for `C` more credits to avoid any delay after consuming the other half of its credits. The scheduler can decline credit requests, which deactivates that QP for both the sender and the receiver.

Metric: Coalescing degree as a QP contention metric. The leader also reports the *coalescing degree* to the receiver. Coalescing degree (> 1) represents the number of RPC requests coalesced within a message, *i.e.*, multiple threads concurrently submitting requests, which indicates QP contention. The QP scheduler tracks this metric on each QP and activates more QPs to decrease contention. Moreover, the leader also piggybacks this metric when requesting credits from the receiver. Specifically, it reports the median of coalescing degree since last renew request, representing the median of QP contention level.

QP assignment. To reflect these two factors—level of QP contention and request frequency—in scheduling, we define *QP utilization* of an active QP j for sender i denoted $U_{i,j}$, as the sum of the reported coalescing degrees in credit renew requests since last QP redistribution. A higher $U_{i,j}$ means either higher QP contention or frequent credit renewal requests. We define U_i as the aggregated active QP utilization for a sender i, *i.e.*, the sum of active QP utilization ($U_{i,j}$).

Thus, in every scheduling interval, the QP scheduler redistributes active QPs to each sender i based on its U_i. It categorizes the senders into *functioning* and *dormant* with each sender starting in the functioning state and becoming dormant if the client does not issue any request (*i.e.*, $U_i = 0$) within a scheduling interval. Specifically, we determine the number of active QPs (denoted AQP_i) for a sender i as:

$$AQP_i = \begin{cases} MAX_AQP * \dfrac{U_i}{\sum_{k \in senders} U_k}, & \text{if } U_i > 0 \\ 1, & \text{otherwise} \end{cases} \tag{1}$$

where MAX_AQP is the maximum QP count that the server keeps active. The QP scheduler allows dormant senders to have one QP for future communication. Moreover, if a new client joins, it gets an average number of QPs per total functioning senders. On the other hand, the QP scheduler ensures that only the set of active QPs continue to receive credits, while dormant ones receive no credits from the next scheduling interval. We choose MAX_AQP (256) to avoid performance degradation caused by RNIC cache trashing (see Figure 2).

5.2 Sender-side Thread Scheduling

As discussed in §5.1, once a QP becomes inactive, a sender cannot use it for request processing. Thus, application threads using the currently inactive QP should be migrated to an active QP. The *thread scheduler* enables this migration via sender-side thread scheduling approach.

Thread scheduler. A sender node runs a dedicated scheduler thread that assigns application threads to QPs. Similar to receiver-side scheduling, the thread scheduler does three tasks: (1) it collects the statistics of application thread behavior; (2) it assigns application threads to new active QPs; and (3) it also notifies those threads who have a different assignment of a QP. Before sending any requests using the new QP, these threads ensure that all requests sent using their old QP are completed and all responses have been received. We avoid the request-response inconsistency using the sequence ID (§4.1), which enables mapping a thread's outstanding requests to their corresponding responses.

Scheduling goals. When deciding the assignment of application threads to QPs, the thread scheduler maintains two optimization objectives: First, the scheduler tries to avoid

Algorithm 1: Mapping sender threads to active QPs

threads: an array of threads sorted first by the median request size and second by the number of request sent since last scheduling
total_bytes: total data sent since last scheduling
tq_map: a map from a thread to an assigned QP

```
1 quota = total_bytes / number of active QPs, qp_id = 0, qp_load = 0
2 for thread T in threads do
3     qp_load += total data sent by thread T since last scheduling
4     tq_map[T] = qp_id
5     if qp_load >= quota then
6         qp_id ++, qp_load = 0
7     endif
8 end
```

head-of-line blocking at the RPC layer by minimizing the placement of a thread with a large payload with a smaller one on the same QP. This is important because co-locating threads with small payloads increase the coalescing opportunities and it reduces (1) MMIO operations used for posting work requests and (2) the number of messages sent over the network, thereby saving bytes for transmitting network and our message-specific headers. Meanwhile, such advantages are insignificant for large payloads. Finally, the scheduler tries to use all active QPs fairly. This results in minimizing the overall QP contention by making all active QPs process a similar amount of data.

Mapping threads to QPs. Finding an optimal assignment that achieves all of our scheduling goals is a challenging combinatorial optimization problem. In algorithm 1, we introduce an approximate algorithm that tries to finish the thread-QP assignment in linear time—$O(n)$ where n is the number of application threads. The scheduler maintains the median request size, total requests sent, and total data sent for each thread since the last scheduling. We assign application threads to active QPs first based on the thread's median request size and then the number of requests sent since the last scheduling (`threads` in algorithm 1) to mitigate the head-of-line blocking problem. For each QP, we limit the number of threads based on the load (`qp_load`), which is the sum of data transferred by the assigned threads. This approach enables each QP to process an almost similar amount of data (`quota` at Line 5). For a new application thread, which does not have any request statistics, the scheduler randomly decides the QP assignment initially. We then update this assignment during the next scheduling interval if a QP gets deactivated.

6 Flock Memory and Atomic Operations

The RDMA's RC transport natively supports memory operations, such as one-sided reads, writes, and atomic. Since Flock relies on the RC transport, it also provides programming APIs for them (Table 2). Similar to RPC, these operations also use the connection handle abstraction, Flock synchronization, and integrate with our symbiotic scheduling. In conjunction with RPC, a thread performing memory operations should explicitly register a set of memory regions (MRs) using our `fl_attach_mreg` API. This API attaches an MR to a QP for memory operations later.

These memory operations differ from RPC in two places. The first is during the send phase, specifically the Flock synchronization protocol. During this phase, each application thread prepares its work individually using work requests [1]. Later, all followers delegate the job of posting requests to the leader, which first links these work requests together and then issues the relevant memory operations. The second is the receive phase. Unlike RPC, there is no associated response for memory operations. Thus, application threads rely on separate completion events to check for their request completion. In summary, if multiple threads, which share the same QP, perform both RPC and memory operations, Flock ensures handling them separately by annotating operations with different work requests IDs, then polling on respective completion events and finally delivering the response back to appropriate threads. Specifically, we use the work request id (`wr_id` [1]) in the work request and its completion event for annotating operations and checking its completion. These complexities are hidden under our programming interface.

Memory operations integrate with our symbiotic scheduling similarly to RPC. For receiver-side QP scheduling, the client uses the coalescing degree representing the number of concurrent memory operations submitted as an indicator of QP contention. At the client, we track the number as well as the size of read/write requests to decide the QP to thread assignment as part of our sender-side thread scheduling.

7 Implementation

We implement Flock in C/C++ with the core library consisting of about 8,000 lines of code. We now mention two implementation-level optimizations in Flock.

Besides centralizing both the buffer management and completion handling (§4.2), we use selective signaling [19] for efficient RNIC utilization. A signaled or an unsignaled RDMA work request indicates whether the RNIC will enqueue a completion event after completing the request. We can select at most `N-1` consecutive unsignaled work requests out of `N`. Since the leader posts requests, it exclusively handles the signaling of work requests. This results in a significant reduction in the number of completion entries DMA-ed by the RNIC, saving PCIe bandwidth as well as extra processing by the RNIC.

As part of our symbiotic send-recv scheduling, a sender uses RDMA `write-with-imm` verb to acquire credits from the receiver. In addition to updating remote memory, this verb generates a completion event in the receive completion

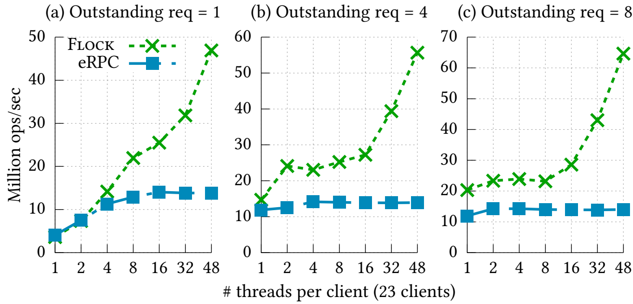

Figure 6. Throughput for Flock and eRPC

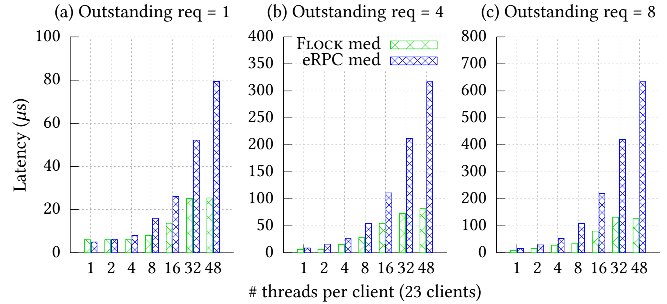

Figure 7. Median latency for Flock and eRPC

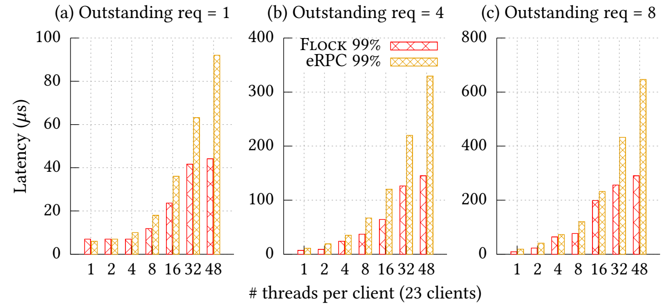

Figure 8. 99th percentile latency for Flock and eRPC

queue (RCQ) on the remote node. The QP scheduler at the receiver polls the RCQ to detect new credit requests from senders. The choice of using RDMA writes for submitting requests and `write-with-imm` for renewing credits by the sender is beneficial. At the receiver, application requests are detected by polling memory whereas polling RCQ is used to check for credit requests. This avoids synchronization between the RPC request dispatcher threads and the QP scheduler thread, while working on the same QP.

8 Evaluation

We evaluate Flock by answering the following questions:

- Can Flock achieve better performance (*i.e.*, higher throughput and lower latency) than the state-of-the-art RDMA RPC-based systems, especially eRPC [21]? (§8.2)
- How effective is Flock's symbiotic send-recv scheduling compared to a simple QP sharing approach [13]? (§8.3)
- Can Flock scale to a large number of machines? (§8.4)
- What is the impact of Flock's RPC layer on real world applications: a distributed transaction processing system (§8.5) and a network ordered key-value store (§8.6)?

8.1 Evaluation Environment

We use 24 nodes from the Cloudlab d6515 cluster [3, 15]. Each node has a 32-core AMD 7452 2.35 GHz CPU, 128 GB RAM, and a Mellanox ConnectX-5 100Gbps NIC. A Dell Z9264F-ON 100 Gbps switch connects these nodes. Each node is running Ubuntu 18.04 on the Linux kernel version 4.15.0. We used Mellanox OFED 5.0-1.0.0.0 for RDMA drivers and user-space libraries. The Maximum Transmission Unit (MTU) used across all nodes in our evaluations is 4096 bytes. For Flock evaluation, we set an upper bound of 256 QPs (*MAX_AQP*), which avoids NIC cache thrashing (Figure 2(a)) unless stated otherwise.

8.2 Comparison with the state-of-the-art

We compare the throughput and latency of Flock with eRPC, a state-of-the-art RDMA RPC system, which utilizes UD to remain scalable with increasing connection counts. We use one server and 23 clients, and vary the number of application threads on clients with the server using all its physical cores. The workload consists of 64-byte RPC request and response. To increase the load at the server, we allow application threads at the client to keep multiple outstanding requests with the server. Figures 6, 7 and 8 show the throughput, median latency and 99th percentile latency, respectively.

With one outstanding request, both systems have comparable performance up to four threads because of minimal load on the server. As we increase the number of threads per client, eRPC performance saturates at 16 threads with a sharp increase in latency at 32 threads. This happens because the server CPU becomes the bottleneck; *i.e.*, the server recycles receive buffers and polls the completion queue for incoming requests. This behavior is in line with our prior observation in §2.2. Meanwhile, Flock maintains scalability with increasing thread count by effectively sharing QPs among threads. For example, with 16 threads per client, the QP scheduler multiplexes a total of 368 (23×16) threads among 256 QPs. Moreover, because of its coalescing-based synchronization, Flock further enables parallelism at high thread count, as more threads efficiently submit requests concurrently. For example, Figure 6 shows that throughput improves by 25% and 47%, when we increase thread count from 16 to 32 and then from 32 to 48, respectively. Overall, Flock improves throughput by 1.25×–3.4× in comparison to eRPC. In terms of latency, eRPC has equal or slightly better latency up to four threads per client. Increasing the thread count leads to latency spikes in eRPC with 2× worse median latency and 1.5× worse 99th percentile latency at 32 threads.

Increasing the server load, with multiple outstanding requests per thread at the client, improves throughput in Flock, but at the cost of increased latency. However, eRPC suffers worse, besides showing performance improvement up to

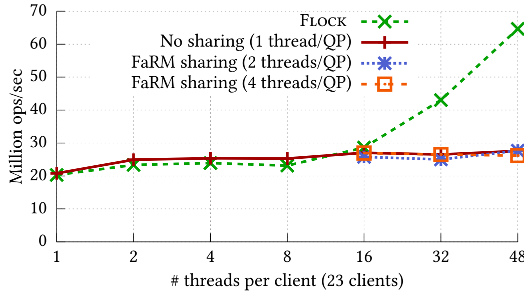

Figure 9. Comparison of RPC performance in different QP sharing approaches: FLOCK synchronization based QP scheduling, no sharing (per-thread QP), and FaRM-like QP sharing using spinlock wherein 2 or 4 threads share a QP, respectively.

eight threads. For example, at high thread count, the throughput with four or eight outstanding requests remains similar to one request with increased latency. That is, with multiple outstanding requests, FLOCK decreases the median latency by 2.9× and 3.9× at 32 and 48 threads, respectively.

8.3 Effectiveness of Symbiotic Send-Recv Scheduling

8.3.1 Receiver-side QP Scheduling To understand the effectiveness of receiver-side QP scheduling, we compare how QP sharing approaches affect the performance and scalability of RPC. In particular, we compare two configurations: (1) no sharing: each thread in a client has a dedicated QP; and (2) FaRM [13]-like QP sharing: 2 or 4 threads share a QP using a spinlock. For a fair comparison, these two configurations also implement RPC using two RDMA writes. We use one node as the server and 23 nodes as clients. The number of application threads is varied from 1 up to 48 at the clients with each thread submitting 64-byte RPC requests to the server. Each thread keeps 8 outstanding requests with the server. At the server, all physical cores are used to handle incoming requests and the server replies with a 64-byte response. Figure 9 shows the results.

Up to eight threads, FLOCK shows similar performance with the no sharing setting, although it has additional operations, such as QP and thread scheduling and FLOCK synchronization. Note that FLOCK does not experience any QP sharing up to eight threads in our cluster because the total QP count (184 QPs = 23×8) is less than the threshold (*MAX_AQP* = 256 QPs). As a result, no coalescing takes place up to eight threads and the observed median and 99th percentile latency for FLOCK and the no sharing config are similar. At 16 threads, FLOCK has slightly better throughput than other approaches because of its QP scheduling. For instance, the no sharing setting requires 368 QPs (23×16), whereas FLOCK limits QPs to 256. The no sharing approach consumes more CPU for polling the QPs at the server. Still it does not suffer from cache thrashing, as shown in Figure 2. On the other hand, FaRM-like sharing reduces the polling overhead but each thread submit its request individually, which does not provide any gains possible from coalescing.

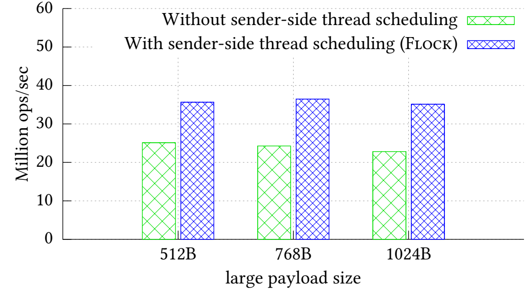

Figure 11. Performance impact of sender-side thread scheduling.

At 32 and 48 threads, because of its coalescing, FLOCK outperforms other configurations by at least 62% and 133%, respectively. At the same time, FLOCK's 99th percentile latency is 27% and 49% lower than the no sharing config at 32 and 48 threads, respectively. Meanwhile QP sharing using spinlock performs similarly to the no sharing config. With coalescing, the number of messages sent by the clients to the server is reduced with a similar effect in the opposite direction as the server also coalesces the RPC responses into larger messages. This further leads to low CPU utilization at the server due to reduced polling overhead and MMIO operations.

We quantify the effect of coalescing with each client running 32 threads and running FLOCK with and without coalescing (see Figure 10). With one outstanding request per thread, coalescing achieves 1.4× better throughput. On average, the server receives 1.56 requests within one coalesced message, which allows coalescing similar number of responses. With one MMIO operation for every 1.56 responses, CPU cycles on MMIO operations reduce by 36%. Also, detecting multiple requests in one go reduces the time spent by the server on polling ring buffers, thereby freeing up cycles for executing the application logic. Finally, coalescing enables better network utilization as it reduces the number of packets, which saves packet headers. These advantages of coalescing are more pronounced when threads keep multiple outstanding requests. With four and eight requests, coalescing improves throughput by 1.7× with the server receiving about 1.7 and 2 requests per message, respectively.

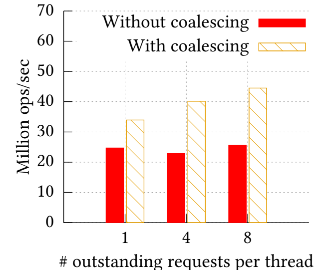

Figure 10. Coalescing impact.

8.3.2 Sender-side Thread Scheduling We evaluate the impact of sender-side thread scheduling with the following workload, in which 10% of threads submit large RPC requests, while 90% of threads issue small RPC (64 bytes) to the server, whose response is 64 bytes. The evaluation setup is the same

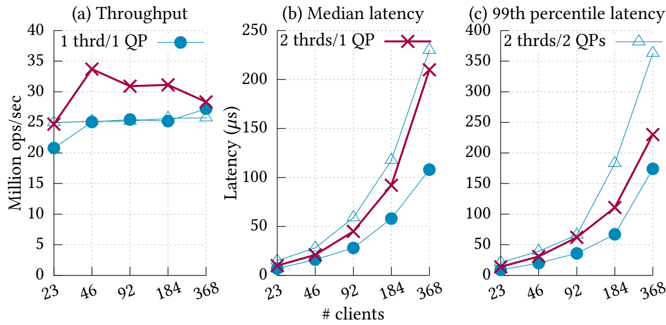

Figure 12. Node scalability in Flock varying the numbers of threads and QPs in a client.

as §8.3.1, with each client running 32 threads. For the case without the server-side thread scheduling, two threads share the same QP. Figure 11 shows that the thread scheduler improves the throughput up to 1.5× with similar latency for varying payload sizes for the large RPC. The thread scheduler effectively groups thread with smaller RPCs using one QP, while it dedicates other QPs to handle large payloads. On the other hand, sharing QPs among threads without our approach leads to head-of-line blocking, thereby reducing performance.

8.4 Node Scalability in Flock

In §8.2 and §8.3, we evaluated Flock using multi-threaded clients with the client count fixed at 23. Since multi-threaded applications are common [7, 40], Flock is helped by its coalescing-based design in multi-threaded settings. We now turn our attention to evaluating Flock's scalability as the number of clients increase.

To simulate a large number of clients in our 24-node cluster, we used one node for the server and spawned an increasing number of client processes on the rest of the 23 nodes. We consider two configurations in this evaluation, one wherein each process uses a single application thread and another in which every process uses two application threads. In the first configuration (1 thrd/1 QP in Figure 12), coalescing is not possible, so this effectively presents the worst-case scenario for Flock. For the second configuration, we further consider two settings wherein each client process uses 2 application threads, either using separate QPs (2 thrds/2 QPs) or sharing a QP among threads (2 thrds/1 QP). This configuration in effect presents a comparison of Flock against native RC QPs. As each machine has 32 physical cores, we limit the maximum number of client processes per machine to 16, resulting in a maximum client count of 368 (23×16). All physical cores at the server are used for handling client requests. Similar to §8.3.1, clients submit 64-byte requests and each application thread keeps 8 outstanding requests with the server. The server replies with a 64-byte response. Figure 12 shows the performance results.

With 1 application thread per process (1 thrd/1 QP), coalescing is not possible, so throughput saturates at 46 clients and is bottlenecked on the packet rate. However, the observed performance is expected due to a lack of coalescing which is the primary reason for performance improvements in multi-threaded settings. To compare node scalability against eRPC, we use their performance results from §8.2. As eRPC does not use QP sharing or coalescing, their performance is dependent only on the number of senders irrespective of a multi-threaded or multi-process setting. Flock's throughput is at least 74% higher with lower median and 99th percentile latency. At 368 clients, the median and 99th percentile latency in Flock is 51% and 25% lower than eRPC, respectively.

With 2 application threads per process, we evaluate whether coalescing in Flock can deliver better performance than native RC while using fewer QPs. As shown in Figure 12, a shared QP among the 2 threads (2 thrds/1 QP) delivers superior throughput and lower latency than each thread using a dedicated QP (2 thrds/2 QPs) across all client counts. Between 46 and 368 clients, the throughput improvement ranges between 10%-30% with similar reductions in 99th percentile latency. These results demonstrate that Flock can deliver better performance while using fewer QPs per machine, allowing it to scale better than native RC.

8.5 FlockTX: Distributed Transactions with Flock

We first present the design of a Flock based distributed transaction system, called FlockTX, that uses optimistic concurrency control (OCC), two-phase commit (2PC), and primary-backup replication (§8.5.1). Later, we compare it with FaSST [20] using two workloads (§8.5.2).

8.5.1 Design of FlockTX We implement a distributed transaction processing system using Flock for communication between clients and servers. We use a partitioned and replicated key-value store at the servers. A server executes transactions on the stored key-value pairs. It provides transactions with serializability when multiple key-value pairs are involved within a transaction. A transaction submitted by a coordinator at the client executes in multiple phases. Our transaction protocol is similar to FaSST [20], as it also uses optimistic concurrency control (OCC) and two-phase commit (2PC) with primary-backup replication for high availability. Figure 13 shows our transaction protocol. `R` and `W` denote two keys read and written within a transaction, respectively. Below we describe the various phases of a transaction.

1) **Execution**: The coordinator reads the key-value items in `R` and `W` by sending RPCs to servers. The server tries to lock the keys in `W`. If failed, it leads to transaction abort. The server returns the key-value pair information for both the read and write set along with the address for keys in `R`.

2) **Validation**: The coordinator uses RDMA reads (*i.e.*, `fl_read`) to verify the versions for keys in `R` using the address returned in the execution phase. If the version of a key is modified or the key is locked, validation fails and the transaction is aborted.

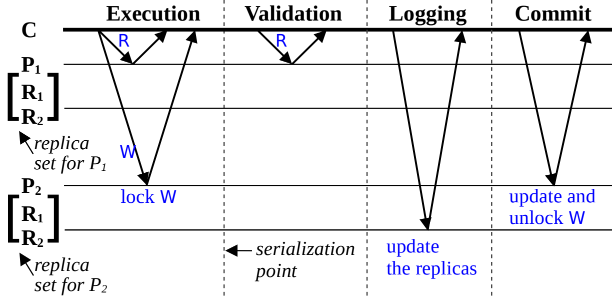

Figure 13. Distributed transaction involving two keys `R` and `W`. **C**, **P**, and **R** stand for the coordinator, the primary and the replica.

3) **Logging**: The updates to keys in `W` are first sent to the replicas by posting RPCs to them. The replicas send ACKs to the coordinator after applying updates. This way, a replica ensures the update ordering as the primary.

4) **Commit**: After receiving ACKs from the replicas, the coordinator sends RPCs to the primary servers containing the updated values for the keys in the write set. The primary servers apply updates and then unlocks these keys.

8.5.2 Evaluation of FlockTX We compare FlockTX with FaSST, which provides distributed transactions with scalability via optimized RPCs over UD. For a fair comparison, we use MICA [23], the same key-value store used in FaSST, and do not cache key-value pairs. As transactions need multiple rounds of communication with a server, we also use coroutines to hide the network latency as FaSST. We use 3 server nodes and 20 client nodes[2] for our evaluation. Servers use 3-way replication. Thus, each server acts as the primary for one partition and is a replica for the other two. We use two benchmarks for evaluation: (1) *TATP* [29], a read-intensive OLTP benchmark simulating telecom database consisting of 70% single key reads, 10% multi-key reads with the rest of transactions updating keys; (2) *Smallbank* [6], a write-intensive benchmark simulating bank account transactions with 85% of transactions updating keys.

(1) TATP. We use the same setup as that of FaSST: each server uses one million subscribers, and each client and server use an equal number of threads. A client only communicates with its peer thread at the server. Each thread at the client uses 20 coroutines of which 19 are used to submit requests and one is used for processing incoming responses. This is a challenging traffic pattern, as there are 380 senders[3] submitting transactions to each server thread. Figure 14 shows throughput and latency results. Although, FaSST is slightly better than Flock up to four threads, its performance saturates at four threads, and the latency dramatically increases at higher thread counts. On the other hand, Flock's throughput increases with more threads. Although FaSST requires extra CPU cycles due to UD, it still

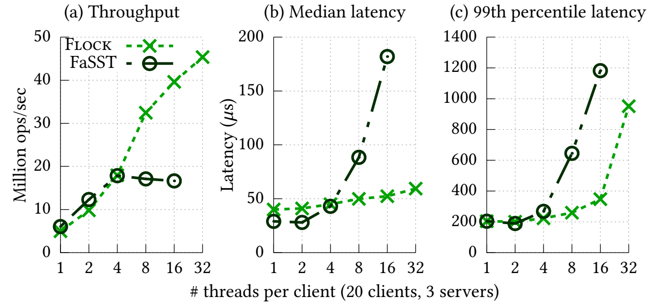

Figure 14. Throughput and median & 99th percentile latency for TATP between Flock and FaSST.

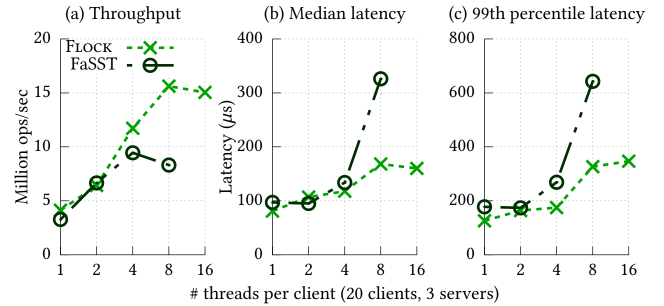

Figure 15. Throughput and median & 99th percentile latency for Smallbank in Flock and FaSST.

maintains a constant load per-server core. However, its performance drop is primarily due to the increased network traffic. Flock minimizes the network traffic thanks to its coalescing-based synchronization. Up to eight threads, there is no coalescing on the client, as the QP count is 160 (20×8) and coroutines of a single thread do not coalesce. However, server coalesces response for coroutines in a larger message, which reduces the number of packets sent to the clients.

At 16 threads, the server maintains a constant number of QPs (256), which allows coalescing between coroutines of threads sharing a QP, thereby amplifying the benefit of coalescing. As a result, Flock's throughput is about 1.9× and 2.4× as that of FaSST at 8 and 16 threads respectively. Similarly, Flock has superior latency than FaSST especially at high thread counts. We omit the 32-thread results for FaSST as some client coroutines do not make progress, which is considered as a packet loss in their RPC implementation.

(2) Smallbank. We use the same FaSST configuration in our evaluation. At each server node, we use 100,000 bank accounts per thread. The workload generated by clients is such that 4% of the total accounts are accessed by 90% of transactions. Each thread at the client runs 20 coroutines. Figure 15 presents the performance results. The throughput of Flock and FaSST are similar up to two threads. However even with a single thread, the 99th percentile latency of FaSST is higher than Flock: 178 μs and 126 μs. This trend is different from TATP because Smallbank is write-intensive. With write-intensive workloads, server threads have more

[2] We are able to use 23 nodes in total due to a faulty node in the cluster.

[3] # senders = # clients (20) × # sender coroutines per thread (19)

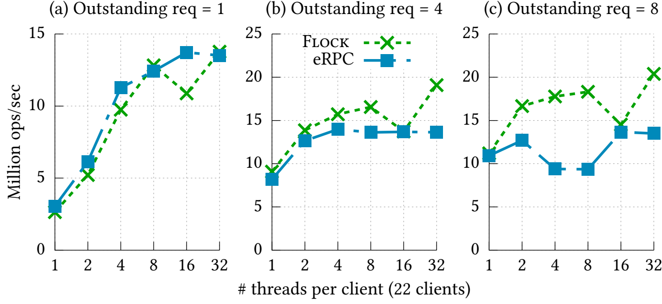

Figure 16. HydraList performance for Flock and eRPC with 90% `get` and 10% `scan` operations.

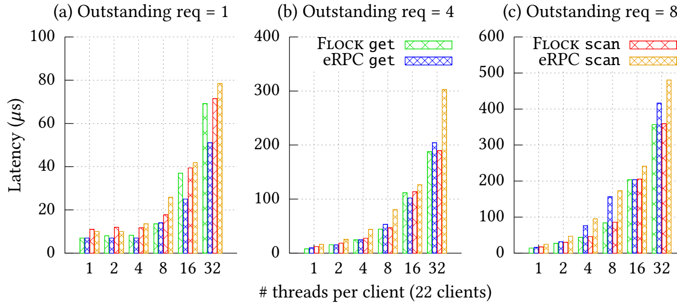

Figure 17. Median latency for `get` and `scan` operations in HydraList with Flock and eRPC.

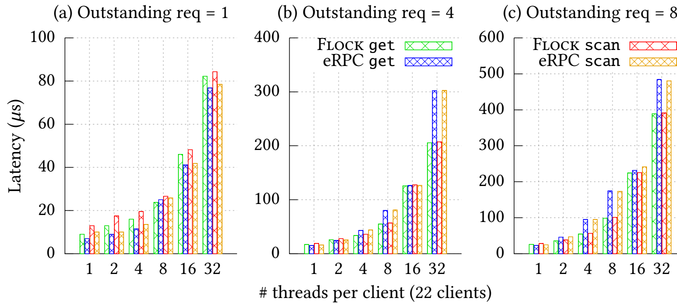

Figure 18. 99th percentile latency for `get` and `scan` operations in HydraList with Flock and eRPC.

work to do as a successful write transaction requires replicating the information on all three servers. With four and eight threads per node, Flock outperforms FaSST by up to 24% and 88% in throughput, respectively. It has similar improvements in median and tail latency. Apart from the lower CPU overheads at the server, these improvements are possible due to our Flock synchronization. At 16 threads, FaSST faces packet loss.

8.6 HydraList over Flock

HydraList [25] is a recent in-memory index optimized for multi-core machines. Using HydraList, we ran a single node index populated with 32 million key-value pairs using 8B keys and 8B values. 22 client nodes [4] submit 90% `get` (lookup) and 10% `scan` queries using either Flock or eRPC. The scan queries use a range of 64 where the server replies with the number of keys found as an 8B response. We vary the number of threads on clients while all physical cores are used at the server to handle and respond to incoming requests. To increase the load on the server, application threads keep multiple outstanding requests with the server. Figures 16, 17, and 18 show the performance results.

With one outstanding request per thread, eRPC's throughput is similar or slightly better than Flock up to eight threads. With 16 threads per client, the total QP count increases to 352 (22×16) on the server, which initiates QP sharing on the clients. However, as each thread uses a single outstanding request, there is little chance of coalescing on the clients due to variable service times for the `get` and `scan` queries, resulting in a performance drop. At 32 threads, Flock performs slightly better than eRPC due to increased coalescing, which overshadows the synchronization cost.

Increasing the load at the server by enabling multiple outstanding requests leads to better performance in Flock for all thread counts except at 16 threads where Flock and eRPC have similar performance. This behavior stems from the limited coalescing happening at 16 threads, similar to our result with one outstanding request. At 32 threads, Flock outperforms eRPC by 1.4×, with lower median and 99th percentile latency for both `get` and `scan` queries.

9 Discussion

Applicability to future hardware. The next generation of RNIC hardware may come with larger caches. Also, recent works, such as Scale-out NUMA [31] and RPCValet [11], target on-die integration of RNIC, thereby eliminating the PCIe overhead experienced by current RNICs. On the other hand, the core count per processor (socket) is already reaching up to 128 hardware threads [4, 5] and will continue to grow further given the current trend. These trends point towards a future where the scalability limitations of today's RNICs might be less eligible. However, we believe that the key ideas of our system, namely Flock synchronization and symbiotic send-recv scheduling will remain effective given both the CPU and RNIC trend. As Figure 9 depicts, increasing the number of connections does not always result in an equivalent increase in performance. However, Flock is able to deliver performance improvement when constrained to fewer QPs by reducing the number of packets exchanged due to its more efficient network utilization. This in turn allows better CPU utilization due to reduced polling overheads and MMIO operations. With the network improving at a much faster rate than the CPU, reducing the cycles spent processing per packet is imperative to make efficient use of the network. As a result, co-optimizing the network and CPU becomes more important than ever. Addressing connection scalability in vanilla RDMA hardware along with co-optimizing the network and CPU have been our two primary objectives in this work.

[4] This setup uses 23 nodes in total due to 1 faulty node in the cluster.

Generalizability. Although Flock is based on RC, we believe our contributions are applicable to UD as well. Our evaluation shows that UD suffers from high CPU overheads with majority of time spent within the network stack. While a reduction in the number of small packets via coalescing allows for better network utilization, it will also reduce the CPU cycles spent within the network stack benefiting application processing.

Support for multiple applications. We have designed and optimized Flock for multi-threaded applications in this work. While it can support multiple applications, this has not been our focus. However, Flock can be extended to support multiple applications and multi-tenancy using a similar approach as Snap [24]. Specifically, a central user-space process that manages network resources and allocates them to application processes as per their utilization is a potential starting design which bears similarities to our current scheduling mechanism. 1RMA [36] is another recent system targeted at supporting multi-tenancy for RDMA networks within datacenters. Drawing insights from prior work, we will further investigate and support multi-tenancy as part of our future work.

10 Related Work

Scalability limitations of RDMA. Improving RDMA scalability is a problem that has lead to a large body of work [8, 13, 18, 20, 21, 32, 36, 39, 41]. FaRM [13] is an RDMA-based distributed computing platform which uses 2GB hugepages and lock-based QP sharing to address the scalability challenge. Our design of a lock-free and coalescing-based QP sharing is inspired by them. LITE [39] targets data center applications and provides a kernel indirection layer to incorporate security with RDMA. It uses physical addresses to register memory regions so as to avoid RNIC cache pressure. We believe our load-control mechanism for connection sharing can allow better hardware and network utilization in these systems. ScaleRPC [8] uses a time-sharing approach to limit the number of QPs served by the RNIC at any time. This requires additional coordination when communicating with multiple remote nodes, thereby increasing tail latency.

HERD [18], FaSST [20] and eRPC [21] use UD for its inherent scalability. However, the high CPU overheads involved leaves fewer cycles for application processing resulting in performance saturating with fewer senders. We believe that forgoing CPU-efficient one-sided verbs is the missing feature in these systems which enables effective utilization of RNIC as well as CPU.

1RMA [36] is a recent system designed to address the RDMA scalability limitations in multi-tenant data centers. It re-architects the NIC hardware for a connectionless design, enabling it to overcome the limitations of standard RDMA including connection scalability, FIFO operation execution order, and close coupling of security policies with connections. The one-shot requests and solicited data transfers in 1RMA facilitates scaling to a large number of nodes along with first-class support for multi-tenancy. In comparison, our work focuses on overcoming the connection scalability in vanilla RDMA hardware while being performant and efficiently utilizing the network bandwidth.

Dynamically Connected Transport (DCT) [10] from Mellanox is another solution designed to tackle the challenges posed by connection scalability. DCT is still connection-oriented but dynamically creates and destroys queue pairs to limit the number of active connections. Although transparent to the application, frequently switching a connection to communicate with multiple remote machines lead to performance degradation as noted by prior studies [20, 38].

Credit-based scheduling in network. Receiver-driven transports like pHost [16], Homa [28] use a credit scheme at the receiver to grant credits to the sender before it submits a request. These systems are targeted at handling network congestion. Breakwater [9] provides an RPC library on top of TCP which uses a similar scheme to avoid overload at the server. Server provides credits to the clients from a global pool representing the maximum load it can handle which allows the server to maintain latency SLOs. Inspired by these systems, Flock uses a credit-based receiver-driven scheme to decide the allocation of QPs to the sender nodes.

11 Conclusion

We present Flock, a communication library that balances the performance-scalability trade-off in RDMA fan-in fan-out networks. Flock revisits the idea of QP sharing using three contributions. Flock adds an indirection in the form of connection handle abstraction to multiplex QPs among threads. It uses a leader-follower synchronization mechanism that enables QP sharing among threads with low overhead, delivering performance improvement through efficient network utilization and reduced CPU overheads. This is complemented by symbiotic send-recv scheduling, the key enabler for controlling the maximum client load at the server as well as ensuring fair utilization of QPs. Our extensive evaluation shows significant performance improvements for distributed transaction processing and an in-memory index structure, which opens the door for rethinking the existing RDMA stacks.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Adam Belay for their comments and suggestions to improve the paper. We appreciate Cloudlab [15] for providing the evaluation platform. This work was supported in part by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035).

References

[1] Infiniband Architecture Specification, Volume 1. https://cw.infinibandta.org/document/dl/8567.

[2] RDMA Core Userspace Libraries and Daemons. https://github.com/linux-rdma/rdma-core.

[3] The Cloud Lab Manual: 12. Hardware. https://docs.cloudlab.us/hardware.html.

[4] AMD Unveils EPYC Milan 7003 CPUs, Zen 3 comes to 64-Core server chips. https://www.tomshardware.com/news/amd-unveils-epyc-milan-7003-cpus-zen-3-comes-to-64-core-server-chips, 2021.

[5] ARM puts some muscle into future Neoverse server CPU designs. https://www.nextplatform.com/2021/04/27/arm-puts-some-muscle-into-future-neoverse-server-cpu-designs/, 2021.

[6] Mohammad Alomari, Michael J. Cahill, Alan D. Fekete, and Uwe Röhm. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 576–585. IEEE Computer Society, 2008.

[7] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[8] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–14, 2019.

[9] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for μs-scale rpcs with breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314, 2020.

[10] Diego Crupnicoff, Michael Kagan, Ariel Shahar, Noam Bloch, and Hillel Chapman. Dynamically-connected transport service, 2012. US Patent 8,213,315.

[11] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[12] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[13] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, Seattle, WA, March 2014.

[14] Aleksandar Dragojevic, 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 ACM Symposium on Operating Systems Principles (SOSP)*, pages 54–70, Monterey, CA, October 2015.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In *2019 USENIX Annual Technical Conference (ATC 19)*, pages 1–14, 2019.

[16] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–12, 2015.

[17] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 649–667. USENIX Association, 2017.

[18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 295–306, 2014.

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference (ATC 16)*, pages 437–450, 2016.

[20] 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 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 185–201, Savannah, GA, November 2016.

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

[22] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.

[23] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, Seattle, WA, March 2014.

[24] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, Ontario, Canada, October 2019.

[25] Ajit Mathew and Changwoo Min. Hydralist: a scalable in-memory index using asynchronous updates and partial replication. *Proceedings of the VLDB Endowment*, 13(9):1332–1345, 2020.

[26] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[27] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[28] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[29] Simo Neuvonen, Antoni Wolski, Markku manner, and Vilho Raatikka. Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.net/.

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[31] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. *ACM SIGPLAN Notices*, 49(4):3–18, 2014.

[32] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, et al. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 97–108, 2019.

[33] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for

Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, Boston, MA, February 2019.

[34] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 145–160, Carlsbad, CA, October 2018.

[35] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.

[36] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 708–721, 2020.

[37] 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*, pages 1–15, 2017.

[38] Hari Subramoni, Khaled Hamidouche, Akshey Venkatesh, Sourav Chakraborty, and Dhabaleswar K Panda. Designing mpi library with dynamic connected transport (dct) of infiniband: early experiences. In *International Supercomputing Conference*, pages 278–295, 2014.

[39] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 306–324, Shanghai, China, October 2017.

[40] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107, 2017.

[41] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 233–251, Carlsbad, CA, October 2018.

[42] Wei, Xingda and Shi, Jiaxin and Chen, Yanzhe and Chen, Rong and Chen, Haibo. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, Monterey, CA, October 2015.

[43] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.