ShiftLock: Mitigate One-sided RDMA Lock Contention via Handover

Jian Gao, Qing Wang, and Jiwu Shu, *Tsinghua University*

https://www.usenix.org/conference/fast25/presentation/gao

This paper is included in the Proceedings of the
23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

Open access to the Proceedings
of the 23rd USENIX Conference on
File and Storage Technologies
is sponsored by

ShiftLock: Mitigate One-sided RDMA Lock Contention via Handover

Jian Gao Qing Wang Jiwu Shu*

Tsinghua University

Abstract

Lock is a basic building block of distributed storage systems. With the extensive deployment of the Remote Direct Memory Access (RDMA) network, RDMA lock has been brought into increasing focus since it can leverage RDMA one-sided verbs to acquire and release locks, achieving high performance without any intervention of server-side CPUs. However, existing RDMA locks are suboptimal under high contention, mainly because clients are likely to fail to acquire a locked lock and must *retry*. Excessive retries incur high latencies for clients and decrease the overall goodput as they devour the lock server's network inbound IOPS resources.

The MCS lock inspired us that instead of contending, clients can coordinate with each other by directly *handing over* locks; thus, they can wait locally without retrying. We present ShiftLock, an RDMA lock supporting lock handover among arbitrary clients. At its core is a non-blocking direct client-to-client coordination mechanism with CPU efficiency, scalability, and fault tolerance, realized with proper software design and exertion of RDMA features. Based on it, ShiftLock employs a crafted protocol with reader-writer semantics, starvation-freedom, and low latency or high goodput under low or high contention. Compared to existing locks, ShiftLock improves goodput by up to 3.62× and reduces tail latencies by up to 76.6% in microbenchmarks, respectively, while also improving transaction goodput by up to 2.85×.

1 Introduction

Distributed lock is a primary component of distributed storage systems that access shared resources over the network, including file systems [1, 2, 29, 38], key-value stores [19, 64], and OLTP databases [57, 63]. In recent years, there has been a rapidly increasing deployment of in-memory storage systems over high-speed Remote Direct Memory Access (RDMA) networks, which are capable of performing file operations, key-value queries, and database transactions in mere microseconds [7, 66, 74]. However, conventional distributed lock managers depend on server-side CPUs to grant or revoke locks, which can take tens to hundreds of microseconds to process a lock request under high concurrency [20, 70], creating a significant performance bottleneck.

This situation has motivated the design of *RDMA locks*, a type of distributed lock managers that manipulate lock entries stored on the lock server with RDMA one-sided verbs, bypassing server-side CPUs. RDMA has ultra-low latencies (e.g., $\leq 2\,\mu s$ network RTT) and high throughputs (e.g., 370 Mpkts/s for NVIDIA ConnectX-7 [50]), forming the basis for the high performance of RDMA locks. RDMA's advantages are particularly evident in low-contention scenarios where clients can acquire and release the lock almost always with only one network roundtrip without server-side CPU intervention.

However, we observe that existing RDMA locks meet performance bottlenecks under high concurrency and contention. This is because all clients' network traffic heads to the lock server, forming an N-to-1 communication paradigm. Due to high contention, a client may have to retry many times to acquire a lock. Retries cause high network traffic to the server, consuming its RNIC's inbound IOPS resources and causing performance degradation. The widely-used backoff solution to this problem is not only difficult to finetune in practice [56] but also suboptimal in performance (§2.3).

Enabling direct coordination among clients can break the N-to-1 paradigm. Specifically, a client holding the lock can explicitly and directly hand it over to another client, which we call *lock handover*. This allows clients to wait locally for the receipt of the handed-over locks instead of performing retries, reducing network traffic and improving goodput. Although this idea has been mature in single-machine locks (e.g., the MCS lock [43, 60]), we find it still underexplored and challenging to realize practically in a distributed environment. First, the conventional remote procedure call (RPC) between distributed clients is *blocking* and *CPU-consuming*, for which we need to find a proper alternative communication paradigm. Second, maintaining full reliable connections among clients is *non-scalable* but unreliable RDMA transport is *susceptible to client failures*, for which we need to ensure scalable and fault-tolerant RDMA communication among clients.

In this paper, we propose ShiftLock, an RDMA lock that resolves the challenges above and enables lock handover in a non-blocking, CPU-efficient, scalable, and fault-tolerant manner. ShiftLock realizes direct client-to-client communication with *two-sided* RDMA to reduce metadata overheads and compact lock entries. For CPU efficiency, clients only check notifications during lock acquisition or release when necessary, avoiding dedicated CPU polling efforts. To avoid maintaining full connections and achieve scalability, ShiftLock adopts RDMA dynamic connection, enabling direct reliable communication among arbitrary clients without explicit con-

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

nection management efforts. ShiftLock supports starvation-free reader-writer semantics without maintaining the current lock mode, guaranteeing single-roundtrip lock acquisition and release under low contention. ShiftLock also correctly tolerates failures with joint efforts from clients and the server.

In summary, ShiftLock can effectively reduce retries and achieve high performance. Experiments show that compared to existing RDMA locks, ShiftLock improves the throughput by up to 3.62× and reduces the tail latencies by up to 76.6% under high contention, respectively. For TATP and TPC-C transaction benchmarks, it improves the throughput by up to 2.85× with comparable or lower latencies.

2 Background and Motivation

2.1 RDMA

Remote Direct Memory Access (RDMA) is a network technology widely used in distributed storage systems [1, 9, 19, 23, 26, 29, 34, 38] that provides low latency and high throughput. As its name implies, RDMA allows access to remote memory without remote CPU intervention, offering chances of reducing CPU overheads for all kinds of distributed systems.

Nodes equip RDMA network interface cards (RNICs) to communicate with RDMA. They post RDMA verbs to queue pairs (QPs) and later poll the completion queues (CQs) to check for verbs' completions [41]. RDMA supports two-sided verbs including SEND and RECV, and one-sided verbs including READ, WRITE, atomic COMPAREANDSWAP (CAS), and atomic FETCHANDADD (FAA), which can only be fully utilized with reliably connected (RC) QPs. Each RC QP connects to only one remote RC QP; to establish a reliable connection from one QP to another, it needs the target's Global Identifier (GID, 128-bit), Local Identifier (LID, 16-bit), and QP number (QPN, 24-bit). Every one-sided verb further requires a 64-bit target virtual address and a 32-bit Remote Key (RKey).

RDMA atomics have rich semantics. Modern off-the-shelf RNICs (e.g., `mlx5` devices, ranging from Mellanox Connect-IB to NVIDIA ConnectX-7 [42, 48–50]) support extended atomics [47], which can CAS on user-chosen bits or FAA on separate fields. We will describe the semantics of extended atomics in detail in §3 when ShiftLock requires them.

2.2 RDMA Locks

The efficiency of distributed lock protocols has long been a major concern for distributed system designers. RDMA enables distributed locks to enjoy network-RTT-level lock management latencies, which motivates the design of various RDMA locks [8, 15, 46, 67, 69, 71, 72].

A typical RDMA lock protocol assumes a lock server hosting the lock entries and exposing them to clients via its RNIC. Distributed clients acquire and release locks by manipulating the lock entries with RDMA one-sided verbs. The lock server's CPUs are uninvolved in the critical paths of lock management, which avoids server-side CPU bottlenecks.

We present an example with the simplest RDMA lock: the CAS lock, a variant of the typical test-and-set (TAS) lock. A CAS lock entry is an aligned 8-byte memory region initialized to zero. To acquire a lock, a client uses CAS to atomically change the lock entry from 0 to a non-zero value (e.g., 1 or the client's ID). *If the CAS fails, the client must retry until it succeeds.* A successful CAS indicates lock acquisition, after which the client can enter the critical section. To release the lock, the client zeroes the lock entry. Throughout this process, the client only uses RDMA one-sided verbs to manipulate the lock, without any server-side CPU intervention.

2.3 Retries in RDMA Lock Acquisiton

A client is not guaranteed to successfully acquire a lock on the first attempt. Under contention, it may fail and must retry. In single-node systems, prior research has shown that retries cause performance degradation due to large contention for the lock entry and cache traffic [6, 27, 43]. In RDMA-based distributed systems, retry is also a critical challenge. Every retry consumes one RDMA roundtrip, which incurs high latencies for clients and consumes RNIC IOPS resources for the server, thereby decreasing the overall goodput.

Take the CAS lock as an example. We evaluate the impact of retries by having 240 clients contend for one lock (see §4.1 for detailed testbed setup). Each client retries immediately after a failed CAS attempt and immediately releases the lock after acquisition. Compared to the ideal case with no retries, the overall goodput decreases dramatically from 278.3 Kops/s to 15.5 Kops/s (−94.4%); 99.4% of the CAS verbs received by the server are retries.

Mitigating the retry problem is non-trivial. For example, to reduce unnecessary retries, prior studies proposed the *backoff* strategy [56, 69], where the clients wait for a short period after a failed lock acquisition attempt. However, backoff is a probabilistic approach that can only partially alleviate the problem instead of solving it. To demonstrate, we equip the CAS lock with truncated exponential backoff proposed in SMART [56], a state-of-the-art backoff scheme, and repeat the experiment. This time, it delivers 141.4 Kops/s goodput, which is 49.2% worse than the ideal case despite a 9.1× improvement over the no-backoff implementation. Most (87.1%) CAS verbs received by the server are still retries.

In summary, the negative impact of retries can be significant under high lock contention. As a conclusion, RDMA locks should try to minimize retries to achieve high performance.

2.4 Motivation, and MCS Locks

Our key insight is that enabling direct client-to-client *lock handover* is critical for reducing retries and improving the lock's performance. Lock handover means that a client transfers lock ownership to another client via direct communication without unlocking. This allows waiting clients to only wait locally (e.g., poll a CQ) for a handover notification instead of performing costly retries to the lock entry via RDMA.

Algorithm 1 Vanilla MCS lock.

```
1:  procedure MCS-Acquire(C)
2:      prev ← tail.FetchAndStore(C)          ▷ 'tail' is the lock entry.
3:      if prev ≠ null then
4:          prev.next ← C
5:          Wait until c.flag = True

6:  procedure MCS-Release(C)
7:      if C.next = null then
8:          if tail.CompareAndSwap(C, null) then
9:              return
10:         Wait until C.next ≠ null
11:     C.next.flag ← True
```

Following this insight, we look towards the MCS lock [43, 60], a lock handover scheme widely used in single-node scenarios but underexplored in distributed systems. MCS lock uses the queueing principle, in which waiters join the queue and get the ownership in a first-in-first-out (FIFO) order.

Algorithm 1 details the MCS lock protocol. The lock entry consists of a pointer to the queue tail. Each client maintains a local pointer to its successor in the queue and a boolean flag signaling lock acquisition. Initially, the local pointer is null and the flag is `False`. To acquire the lock, a client C joins the queue by updating the lock entry to C and setting its predecessor's local pointer also to C (Lines 2 and 4). It then waits until its flag becomes `True` (Line 5), after which it can enter the critical section. To release the lock, if no successors exist, C sets the lock entry to null (Line 8); otherwise, it hands over the lock by setting the successor's flag to `True` (Line 11).

MCS lock enables clients to hand over the lock without contending on the lock entry. However, it has significant disadvantages in a distributed environment. First, direct communication between clients is CPU-consuming (due to the RPC paradigm) and space-consuming (due to the need to maintain clients' routing information). Second, vanilla MCS lock do not support reader-writer semantics. While there are MCS lock variants that add such support [27, 28, 31, 33, 44, 61], in a distributed environment, they incur high latencies by accessing the lock entry for multiple times (each consuming a network roundtrip) in acquisition/release procedures. Third, client failures are catastrophic as they can break the queue, preventing subsequent clients from acquiring the lock. These disadvantages must be addressed to enjoy its benefits, which motivates us to design ShiftLock.

3 Design

ShiftLock is an RDMA-based, fault-tolerant reader-writer lock that effectively enables lock handover among clients.

3.1 Design Goals and Overview

ShiftLock has two major design goals:

(G1) Low latency under low contention: To take advantage of RDMA's low latency, ShiftLock must always use *only one* network roundtrip to acquire a lock if there is no contention.

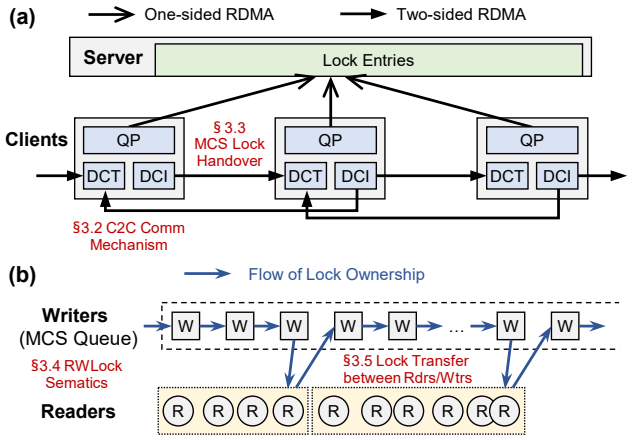

Figure 1: Overview of ShiftLock's design.

(G2) High goodput under high contention: ShiftLock must reduce the total number of RDMA verbs compared to existing locks to save RNIC IOPS and improve goodput.

Figure 1 gives an overview of ShiftLock. The (a) part shows ShiftLock's architecture. Clients access lock entries on the server with one-sided RDMA and efficiently communicate with each other with two-sided RDMA (§3.2) to hand over locks (§3.3). The (b) part shows the logic of ShiftLock. Based on MCS, ShiftLock implements reader-writer semantics (§3.4) and proactively transfers lock ownership between readers and writers to avoid starvation (§3.5). ShiftLock also incorporates mechanisms to tolerate client failures (§3.6).

3.2 Enabling Client-to-Client Communication

Direct client-to-client communication is fundamental to lock handover, yet it is challenging to implement practically. We detail each challenge and our solution below.

Choosing the proper RDMA verbs. In Algorithm 1, client-to-client communications (Lines 4 and 11) involve modifying another client's memory. While it is intuitive to translate them into Write verbs in RDMA, we argue for using Send and Recv instead. The reasons are twofold. First, Send and Recv are one-sided on the data path and perform similarly to Write [65]. Second, Write requires 96 bits of extra metadata (a 64-bit address and a 32-bit RKey), which must be maintained in lock entries and can be space-consuming.

Achieving non-blocking and CPU efficiency. Typically, nodes perform two-sided communication with RPCs. For example, to implement Line 4 of Algorithm 1, the current client would send an RPC containing its routing information to its predecessor. However, this approach blocks the sender during the RPC (incurring high latencies) and requires the receiver to run an RPC server (consuming CPU resources).

We have two insights for this challenge. First, unlike single-node systems, client should not actively poll others because RDMA latency is high. Instead, they should wait locally for

messages from others. Second, messages between clients are not *calls* that require instant replies, but are non-blocking *notifications* that can be buffered and processed later when needed. In the example above, the predecessor does not need to handle the notification until it releases the lock. Thus, it does not need to spare CPU resources to monitor incoming notifications, but simply prepares enough RECVs and inspects the received messages upon lock release. The current client only needs to ensure the reliable delivery of the message, for which we can rely on the hardware acknowledgment mechanisms of RDMA reliable connections.

Achieving scalability. It is impractical and unscalable to maintain full reliable connections between massive clients in a real system, as it requires a connection between each pair of clients, producing an exploding number of QPs. Worse, establishing an RDMA connection can take milliseconds [66], which is detrimental if it occurs in ShiftLock's critical paths.

To solve this challenge, we make use of dynamic connection (DC), a feature widely supported by modern RNICs. DC QPs can dynamically connect to different remote nodes in the data path, eliminating the control-path overhead of explicitly establishing connections. In practice, each client creates a DC Initiator QP (DCI) and a DC Target QP (DCT); the DCI can reliably send data to any DCT by specifying its routing information in RDMA requests. DCI takes less than 1 µs to connect to a new DCT [66], which is efficient enough.

Achieving reasonable space consumption. RDMA's routing information contains a 128-bit GID, a 16-bit LID, and a 24-bit QPN, adding up to 168 bits (21 B). Maintaining all these in a lock entry is impractical due to their large size.

To solve this challenge, we observe that GIDs and LIDs of nodes are stable after network setup. We assign each node a unique non-zero ID and cache the ID-to-GID/LID mapping in each client. We use 16-bit integers for node IDs, allowing a cluster of up to 65535 nodes, which should be sufficient for data center usage [39]. Thus, we can use a 16-bit node ID instead of the 144-bit GID+LID to locate a node, reducing the total size of routing information to 40 bits (16-bit ID + 24-bit DCT QP number). This design allows new clients to join the system without any effort from existing clients.

3.3 Implementing Distributed MCS Lock

We now start implementing ShiftLock as a distributed version of the MCS lock in Algorithm 1. The MCS lock can be acquired with only one FETCHANDSTORE to the lock entry without contention, thus meeting G1. It also enables clients to hand over the lock without retrying, thus meeting G2.

Based on our discussion in §3.2, we use a 16-bit `NodeId` and a 24-bit `DctQpNum` as the queue tail pointer; for convenience, we denote their combination as `TailPtr`. We also replace Lines 4 and 11 with RDMA SENDS. Specifically, in Line 4, the client sends a `Successor`-type message to its predecessor carrying its routing information; in Line 11, the client sends a `Handover`-type message to its successor.

The main challenge here is that we need a FETCHANDSTORE verb in Line 2 that atomically sets the destination operand and returns its previous value. However, standard RDMA does not support FETCHANDSTORE. A strawman solution is to replace it with CAS, but this would incur significant retry traffic under high contention, which decreases goodput and violates G2.

To solve this problem, we use RDMA extended atomic CAS verbs (denoted as EXTCAS hereinafter). EXTCAS allows its user to specify a compare bitmask and a swap bitmask; only the masked bits will be compared and swapped, respectively. Conveniently, EXTCAS will return the original value entirely regardless of the masks. This allows us to implement FETCHANDSTORE with an EXTCAS whose compare mask is set to zero, which deterministically takes one network roundtrip.

3.4 Adding Reader-Writer Semantics

We now add reader-writer semantics to ShiftLock. A reader can acquire the lock simultaneously with other readers, so it cannot join the queue, or it will block other readers. Therefore, we need another mechanism to maintain readers.

Similar to prior work [59, 69], we use a reader counter to implement reader locks. We embed a 23-bit `RdrCnt` into the lock entry that counts the number of active readers, which allows 8 million readers to simultaneously acquire the same lock. We choose the bit width 23 to fully utilize the lock entry's memory, which we will explain in §3.7. A reader increments this counter when it tries to acquire the lock and decrements it when it releases the lock.

To enable modifying the counter without tainting `TailPtr`, we again leverage RDMA extended atomics. The extended FETCHANDADD (FAA) verb, denoted as EXTFAA hereinafter, allows the user to specify a mask that splits the operand into multiple fields; FAA will be performed separately within each field (i.e., carrying is suppressed at field boundaries). This allows us to modify `RdrCnt` with an EXTFAA verb whose mask separates `RdrCnt` from other fields, keeping them intact.

Intuitively, readers may enter critical section if `TailPtr` is null (i.e., `NodeId` and `DctQpNum` are both zero); similarly, a writer may enter its critical section if `TailPtr` is null and `RdrCnt` is zero. However, this design can lead to deadlocks: when readers and writers coexist, readers will be blocked by a non-null `TailPtr`, while writers also cannot proceed because `RdrCnt` might never reach zero as new readers come in.

Introducing the `RelCnt` field. The root cause of the problem above is that the queue head writer cannot know when all preceding readers have left. To solve this problem, we need an extra counter for lock releases. Specifically, we embed a counter `RelCnt` into the lock entry, which will be incremented once a client releases its lock. Now, if the queue head writer finds a non-zero `RdrCnt`, it checks the current value of `RelCnt` (denoted as `CurRelCnt`). When all existing readers have left, `RelCnt` should become `CurRelCnt` + `RdrCnt`. Therefore, the writer can spin-wait until `RelCnt` reaches this value, after which it can acquire the lock.

The only remaining problem is that existing fields in the lock entry (aside from `RelCnt`) already occupy 24 + 16 + 23 = 63 bits of space. Given that conventional RDMA atomics can only operate on 64-bit operands, we only have 1 bit left for `RelCnt`, which is obviously insufficient. Thankfully, we find that extended atomics support 128-bit operands [47]. We can use the extra 64 bits to accommodate `RelCnt`.

3.5 Starvation Avoidance

We now discuss how to prevent any client from starving in ShiftLock. When only readers exist, every client can immediately acquire the lock with no starvation; when only writers exist, they queue up and acquire the lock in an FCFS manner, which is also starvation-free. Therefore, the only situation to consider is when readers and writers simultaneously exist: the lock must not be held by one type of clients indefinitely. Below, we discuss how ShiftLock avoids starvation by transferring the lock's ownership between readers and writers.

3.5.1 Transfer from readers to a writer

ShiftLock does not need extra mechanisms to transfer the lock ownership from readers to a writer. As described in §3.4, after a writer enters the queue, `TailPtr` will become non-null and block new readers. The writer will eventually acquire the lock when all preceding readers have left, thus transferring lock ownership from previous readers to itself.

This design is *write-preferring*: readers subsequent to the queue head writer cannot acquire the lock before it. This makes ShiftLock efficient in read-intensive scenarios where writers will be served as quickly as possible, meeting G2.

3.5.2 Transfer from a writer to readers

If writers do not proactively transfer the lock to readers, the lock can be held indefinitely in the queue, causing reader starvation. Therefore, we need a mechanism for the queue head writer to give up the lock and transfer it to readers. This raises the following three problems. First, we must use a new field to enable such transfers since existing fields are insufficient: `NodeId`, `DctQpNum`, and `RdrCnt` each has its fixed purposes, while `RelCnt` is useless because readers cannot know the queue length. Second, there must be a notification mechanism for readers to eventually hand back the lock to the writers. Third, there must be a criterion for writers to decide when to transfer the lock to readers.

The new field `Epoch`. Finding an appropriate semantic for this new field is challenging. An intuitive design is to make it a flag indicating the current lock mode [59,70], e.g., 0 for reader (shared) lock and 1 for writer (exclusive) lock. Unfortunately, this design violates G1 as it necessitates explicit lock mode management, which may consume an extra network roundtrip during lock acquisition even without contention.

To address this challenge, we first reveal that the entire lifetime of the lock can be divided into *epochs*. Each epoch consists of three sequential parts: (1) zero or more readers acquire and release the lock; (2) a writer appears, taking the lock when all preceding readers have left; (3) writers hand over the lock in the queue until a writer releases the lock and gives it to readers, either because there are no more succeeding writers or to avoid reader starvation.

Now, it is obvious that transferring the lock from a writer to readers is equivalent to starting a new epoch. We can record the epoch number in the lock entry in an `Epoch` field; the writer increments `Epoch` and readers know they receive the lock once `Epoch` changes. With this approach, as long as there are no conflicts, a client can directly acquire a lock regardless of its current epoch and without extra network roundtrips.

Note that readers only need to be aware of epoch changes instead of actual epoch numbers, so we can reduce `Epoch` to 1 bit. Writers can flip it with either ExtCAS or ExtFAA.

Also, note that when there is a waiting writer, changing `Epoch` will only admit currently waiting readers into critical sections. New readers will still get blocked until `Epoch` changes again, thus ensuring starvation freedom for writers.

The notification mechanism. Instead of a `Handover`-type message, a writer must send a different message to its successor (denoted as *succ* below) in the queue when it decides to transfer the lock to readers. We name it `ModeChanged` to indicate a change from writer lock to reader lock.

The problem is: what information should we encapsulate into the `ModeChanged` message? Since the queue is non-empty when the lock is given to the readers, new readers will see a non-null `TailPtr` in the lock entry, so they cannot acquire the lock before *succ*. Therefore, we find it possible to use `RelCnt` as a signal for *succ* to get the lock back. Specifically, when the queue head writer flips `Epoch` with an extended atomic verb, it will get the current `RdrCnt` and `RelCnt` in the return value (denoted as `CurRdrCnt` and `CurRelCnt`, respectively). Before *succ*, the queue head writer and `CurRdrCnt` readers will all release their locks. As a result, the writer can encapsulate `CurRelCnt` + `CurRdrCnt` + 1 in the `ModeChanged` message, while *succ* should spin-wait on `RelCnt` until it becomes that value.

The transfer criterion. We use the number of consecutive writers as the criterion to determine whether a writer should transfer the lock to readers. We embed this number in the `Handover` messages and increment it upon every lock handover. If it reaches a certain threshold N (we use $N = 16$), the writer will flip `Epoch` and send a `ModeChanged` message to its successor in the queue (if there is one).

We may further optimize ShiftLock when `RdrCnt` is zero as the lock may continue to be handed over among writers. We leave this as future work.

3.6 Fault Tolerance

Clients may fail. A failed client will never release the lock, thus preventing the update of `RelCnt`. To tolerate client failures, we adopt the idea of leasing [21], setting a *lease time* T_{lease} as the maximum time a client can hold the lock from

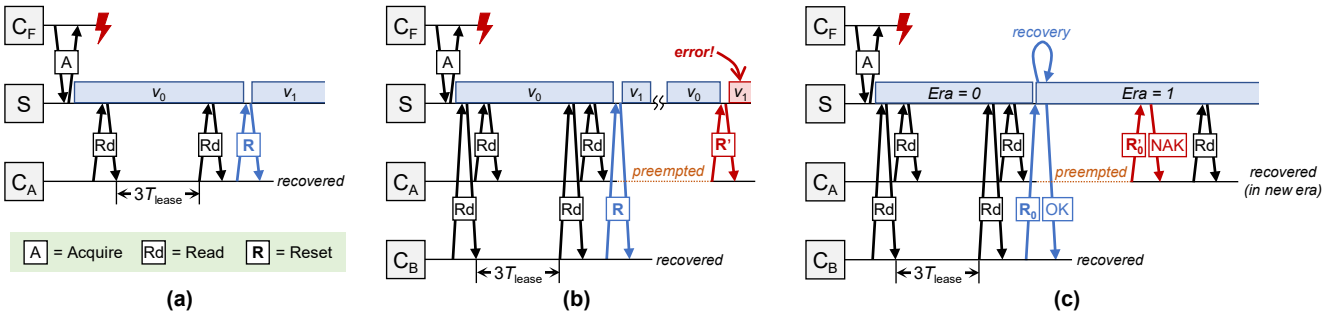

Figure 2: Timeline of the recovery process. S is the lock server, and C_F, C_A, C_B are clients. *(a) One-sided recovery of existing RDMA locks. (b) An errorneous situation of the one-sided recovery process. (c) Two-sided recovery of ShiftLock.*

acquisition to release. T_{lease} should be much larger than the clients' clock resolution and the RDMA transmission time; we set it to 10 ms in our implementation.

Clients check `RelCnt`'s value periodically (e.g., once per $T_{\text{lease}}/2$) when acquiring or releasing a lock. Meanwhile, the lock server maintains a 64-bit *era* and exposes it to clients via RDMA. If `RelCnt` remains unchanged for a timeout of $3\times T_{\text{lease}}$ (the reason for this timeout value is explained below) in the same era, the client assumes a failure and starts recovery by sending an RPC to the server. The RPC contains the lock's address and the era number read by the client.

Upon receiving the RPC, the lock server increments the era and uses an ExtCAS to reset the lock, zeroing all fields except `RelCnt` and increasing `RelCnt` by 2^{63}. Other clients waiting for the lock will see a leap of 2^{63} in `RelCnt`, which would normally take 292 years even if the lock could be released 10^9 times per second (an impossibly high speed for RNICs today). Therefore, they recognize the recovery process and restart lock acquisition once observing the `RelCnt` leap.

The lock server performs recovery *only once per era* and rejects recovery requests with previous era numbers. A client will wait for T_{lease} if its recovery request is rejected and then check `RelCnt`'s value again. If it sees the `RelCnt` leap, it restarts lock acquisition; otherwise, it will re-read the era counter and send another RPC to request recovery.

Assumptions for timing. ShiftLock assumes that the clocks of all clients advance at similar speeds so that their timing results are reliable. Prior studies show that cluster-wide clock speed variation is small and bounded [35,45], with the relative clock drift usually up to $\delta = 10^{-4}$. This makes our timing approach practical: when a client needs to wait for T, it actually waits for $(1+\delta)\times T$ to tolerate clock drift.

Reason for the timeout value $3\times T_{\text{lease}}$. The timeout value is a pessimistic estimation of the interval between two consecutive updates to `RelCnt` by writers; readers always release the lock and update `RelCnt` within T_{lease} since lock acquisition, so they do not affect the timeout value here.

Specifically, a writer updates `RelCnt` when it releases the lock. Starting from this point, it first waits for up to T_{lease} for a `Successor` message (if it times out here, it assumes a failure and drops the lock; the lock will be recovered subsequently). Then, it sends `Handover` to the successor. The successor should finish its critical section within T_{lease} since it receives the `Handover` (we use RNIC timestamps [24,51] to get the receive time). Finally, the successor releases the lock and updates `RelCnt` again. Using a T_{lease} to cap the time spent by lock handover and `RelCnt` updates, we estimate $3\times T_{\text{lease}}$ to be the maximum interval between two updates to `RelCnt`.

Reason for our recovery design. Existing RDMA locks allow clients to recover the lock in a one-sided fashion, i.e., clients use an RDMA one-sided verb (e.g., a CAS) to reset the lock entry [69], as shown in Figure 2(a). Here, a client C_F fails while holding the lock, and another client C_A times out during lock acquisition and resets the lock from the current state v_0 to another state v_1 to recover. ShiftLock does not take this approach because it suffers from the *ABA problem* [14]. Figure 2(b) shows a problematic scenario with this approach. Here, clients C_A and C_B both detect the failure and decide to reset the lock, but C_A is preempted by the operating system before doing so. During the preemption, C_B recovers the lock. When C_A resumes execution, it continues the "recovery". Coincidentally, the lock entry is in state v_0 again (because other clients are acquiring/releasing it), so the "recovery" succeeds. However, this "recovery" actually corrupts the lock entry's state, which can cause fatal errors.

To solve this problem, as discussed above and shown in Figure 2(c), ShiftLock delegates the recovery task to the lock server, which rejects outdated recovery attempts with old era numbers. Although this approach relies on the lock server's CPU, we argue that client failures are rare, and recovery should not cause a CPU bottleneck at the lock server.

Improving recovery performance. When a client holding multiple locks fails, the entire recovery process may span multiple eras and can be time-consuming. This problem can be mitigated by maintaining multiple era counters and assigning each lock to one of them (e.g., by hashing their virtual addresses). Recoveries subject to different era counters are independent and can progress simultaneously.

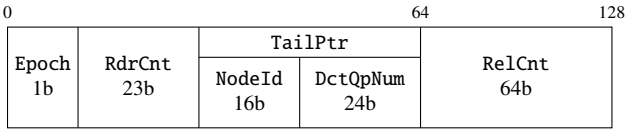

Figure 3: Format of a ShiftLock entry. In this figure, `TailPtr` is the combination of `NodeId` and `DctQpNum`.

Algorithm 2 ShiftLock acquire algorithm.

```
1:  function Reader-Acquire(lock)
2:      entry ← lock.ExtFAA({RdrCnt, 1})
3:      if entry.TailPtr = null then
4:          return
5:      Wait until entry.Epoch ≠ lock.Epoch

6:  function Writer-Acquire(lock) → (u64, u32, u1)
7:      entry ← lock.ExtCAS(
            compare: ∅,
            swap: {TailPtr ← (self.NodeId, self.DctQpNum)}
        )
8:      curRelCnt ← entry.RelCnt
9:      consWrt ← 0
10:     if entry.TailPtr ≠ null then          ▷ A previous writer exists
11:         pred ← DCI QP to entry.TailPtr
12:         pred.Send(Successor{payload ← self})
13:         Wait until received Handover or ModeChanged message
14:         if received ModeChanged{payload → ExpectedRelCnt} then
15:             Wait until lock.RelCnt = ExpectedRelCnt
16:             curRelCnt ← ExpectedRelCnt
17:             entry.Epoch ← 1 − entry.Epoch
18:         else                              ▷ Received Handover
19:             curRelCnt, consWrt ← payload of the Handover message
20:     else if entry.RdrCnt ≠ 0 then         ▷ Previous readers exist
21:         Wait until lock.RelCnt = entry.RdrCnt + entry.RelCnt
22:         curRelCnt ← entry.RdrCnt + entry.RelCnt
23:     return curRelCnt, consWrt, entry.Epoch
```

3.7 Putting It All Together

Figure 3 shows the format of a ShiftLock entry. Each entry occupies aligned 16 bytes of memory. It contains five fields: the 1-bit `Epoch`, the 23-bit `RdrCnt`, the 16-bit `NodeId`, the 24-bit `DctQpNum`, and the 64-bit `RelCnt`. The first four fields together occupy exactly 8 bytes (which explains the bit width 23 of `RdrCnt`), and `RelCnt` occupies the remaining 8 bytes.

Algorithm 2 shows the lock acquisition routines, in which we omit failure detection and recovery logic for simplicity. A reader first increments `RdrCnt` (Line 2) and waits for preceding writers (Line 5). A writer first enqueues itself in the queue (Line 7) and waits for preceding writers (Line 12) and readers (Lines 15 and 21). Writer-Acquire returns three values (Line 23): the current value of `RelCnt` (*curRelCnt*), the consecutive writer count (*consWrt*), and the current *epoch*.

Algorithm 3 shows the lock release routines. A reader simply increments `RelCnt` and decrements `RdrCnt` to release the lock (Line 2). A writer tries to zero the queue tail pointer, increment `RelCnt`, and flip `Epoch` if it has no successors in the queue (Line 6), during which it leverages the return values of Writer-Acquire to combine all modifications to the lock entry into one ExtCAS. Otherwise, if there is a successor (Line 9), it will send a `ModeChanged` (Lines 11-14) or `Handover` (Lines 15-19) depending on *consWrt*'s value.

Algorithm 3 ShiftLock release algorithm.

```
1:  function Reader-Release(lock)
2:      lock.ExtFAA({RdrCnt, −1}, {RelCnt, 1})

3:  function Writer-Release(lock, curRelCnt, consWrt, epoch)
4:      succ ← the successor in the queue
5:      if succ = ∅ then
6:          entry ← lock.ExtCAS(
                compare: {TailPtr = (self.NodeId, self.DctQpNum)},
                swap: {TailPtr ← null, RelCnt ← curRelCnt + 1,
                       Epoch ← 1 − epoch}
            )
7:          if (entry.TailPtr) = (self.NodeId, self.DctQpNum) then
8:              return
9:          Wait until received Successor message
10:         succ ← payload of the Successor message
11:     if consWrt reaches the threshold then
12:         entry ← lock.ExtFAA({Epoch, 1}, {RelCnt, 1})
13:         expected ← entry.RelCnt + entry.RdrCnt + 1
14:         succ.Send(ModeChanged{payload ← expected})
15:     else
16:         lock.ExtFAA({RelCnt, 1})
17:         curRelCnt ← curRelCnt + 1
18:         consWrt ← consWrt + 1
19:         succ.Send(Handover{payload ← (curRelCnt, consWrt)})
```

3.8 Proof of Correctness

Inspired by [59], we show ShiftLock's correctness by proving two critical properties: mutual exclusion and deadlock freedom. Starvation freedom is also an important property, but we have discussed it thoroughly in §3.5 and can omit it here.

3.8.1 Mutual exclusion

A writer cannot have the lock simultaneously with another writer or reader. We discuss both cases as follows.

Writer & writer. A writer must be the queue head to have the lock. Since ExtCAS is atomic and a writer has up to one successor, by induction, no two writers can simultaneously recognize themselves as the queue head.

Writer & reader. When a reader has the lock, `RdrCnt` must be non-zero. However, a subsequent writer must wait until `RdrCnt` becomes zero to acquire the lock. When a writer has the lock, a subsequent reader must wait until the current queue head writer flips `Epoch`. Together, a notification will be sent to the next writer, making it wait until the reader leaves.

3.8.2 Deadlock freedom

A writer and another writer/reader cannot wait mutually for each other. We discuss both cases as follows.

Writer & writer. Writers must form a cycle in the queue to form a deadlock. This requires that an ExtCAS observes the effects of another ExtCAS happening-after it, which violates its atomicity and is impossible.

Writer & reader. We have discussed this case in §3.4.

Lock	One-sided	R/W	Handover	Fault-tolerance
CAS [56]	✓	✗	✗	✗
DrTM [67]	✓	✓	✗	✗
DSLR [69]	✓	✓	✗	✓
MCS [43, 60]	✓	✗	✓	✗
RMA-RW [59]	✓	✓	✓	✗
RPC	✗	✓	✗	✗
ShiftLock	✓	✓	✓	✓

Table 1: Summary of evaluated lock managers.

4 Evaluation

We implement ShiftLock in 7.2K lines of Rust and evaluate it to answer the following questions:

- How does ShiftLock perform? (§4.2-4.5)
- How to understand ShiftLock's performance? (§4.6-4.7)
- How can ShiftLock recover from client failures? (§4.8)

4.1 Experiment Setup

We use a testbed of 6 machines, one as the lock server and the other five as clients. Each machine is equipped with two 24-core Intel® Xeon® E5-2650v4 CPUs running at 2.20 GHz, 128 GB DDR4-2400 DRAM, and a Mellanox ConnectX-5 100 Gbps RNIC connected to a Mellanox QM8790 Infiniband switch. All machines run Ubuntu 18.04 with Linux kernel version 4.15.0. Each client machine runs a client on each core, so the total client count is $48 \times 5 = 240$.

Baselines. We evaluate ShiftLock against baselines shown in Table 1. `CAS` uses RDMA CAS to acquire locks and adopts truncated exponential backoff from SMART [56]. `DrTM` also uses RDMA CAS but supports readers with timestamp-based leases. `DSLR` implements Lamport's bakery algorithm [32] using RDMA FAA with a heuristic dynamic backoff scheme guided by the number of waiting clients. `MCS` is Algorithm 1. `RMA-RW` hand over locks among a pre-determined set of clients with MPI. `RPC` relies on server-side CPUs to serve lock acquisition and release RPCs in a retry-on-fail manner.

Implementation details. For fairness in evaluation, we implement and evaluate ShiftLock and baselines meticulously.

- ShiftLock, `MCS`, and `RMA-RW` allow direct communication between two clients, which experiences significantly lower intra-node latency than inter-node. We detect intra-node communication via the `NodeId` field and compensate for latency by introducing a delay when receiving a message from a client on the same node. Based on our latency measurements, we set the compensation delay to 1 µs.
- `DrTM` requires clock synchronization among clients. To this end, we use the Precision Time Protocol (PTP) [36] to synchronize clients' clocks to the server's. We use hardware PTP and keep the PTP daemons running in all experiments. `DrTM` assumes a clock drift of up to 50 µs.

Lock settings. Unless otherwise stated, we use 10 million locks in all experiments. The lease time is set to 10 ms.

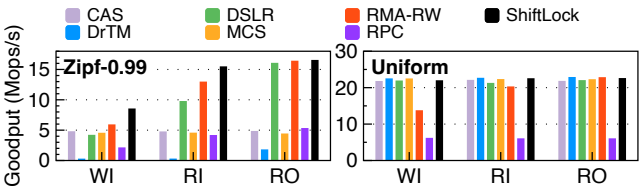

Figure 4: Goodputs of locks for the microbenchmark.

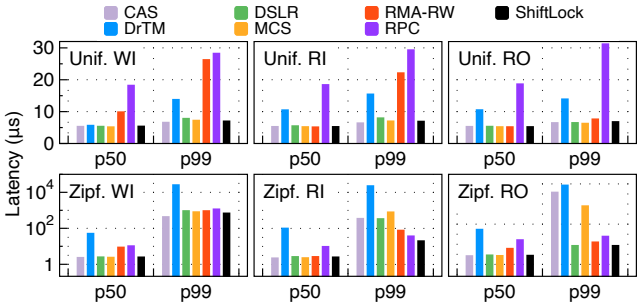

Figure 5: Latencies of locks for the microbenchmark.

4.2 Microbenchmarks

We build a microbenchmark to study how fast ShiftLock and baselines can grant locks, in which clients repeatedly acquire a lock and then immediately release it. It consists of 3 typical workloads: write-intensive (WI, 50% reads), read-intensive (RI, 95% reads), and read-only (RO, 100% reads). Every client, in each iteration, independently chooses a lock according to a uniform (Unif.) or Zipfian-0.99 (Zipf.) distribution. Figures 4 and 5 show the goodputs and the latencies, respectively. We make the following observations.

First, ShiftLock meets the design goal G1, preserving low latencies under low contention. Under uniform workloads, ShiftLock delivers the lowest latencies and the highest goodputs together with `CAS`, `DSLR`, and `MCS`. Other baselines, `DrTM` and `RMA-RW`, have higher latencies (1.99× and 2.26× in p99 latencies, respectively). `DrTM` introduces extra wait time since it must tolerate clock drift between clients, while `RMA-RW` incurs overheads on maintaining the lock mode. `RPC` has both high latencies and low goodputs because it suffers from the server-side CPU bottleneck. On the other hand, for the skewed read-only workload, ShiftLock performs close to `DSLR` and `RMA-RW`. `CAS` and `MCS` now perform significantly worse than ShiftLock as they do not support reader locks and have to serialize readers, which results in poor performance.

Second, ShiftLock meets the goal G2, achieving high performance under high contention. Under WI and RI workloads, ShiftLock outperforms `MCS` and `RMA-RW` by 1.65×, 1.27× and 3.62×, 1.26× in goodput, respectively; also, it has 6.25% and 76.6% lower p99 latencies. Compared to other baselines, ShiftLock delivers 1.56× to 26.91× goodput and has comparable or significantly lower latencies.

To understand more clearly why ShiftLock can deliver improved performance than the baselines, we obtain the average

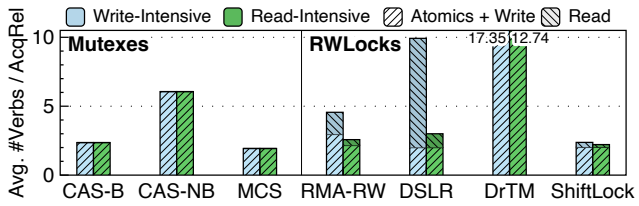

Figure 6: Average numbers of RDMA one-sided verbs required to acquire and release a lock.

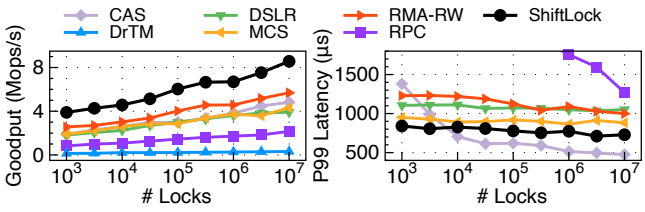

Figure 7: Performance of locks with different lock counts.

numbers of RDMA atomics, Reads, and Writes used by each lock acquisition-release cycle by inspecting hardware counters. We use Zipfian-0.99 WI and RI workloads. Figure 6 shows the results, in which CAS has two variants with backoff enabled (CAS-B) and disabled (CAS-NB).

On average, ShiftLock requires 2.01 atomics and 0.36 (WI) or 0.20 (RI) Reads per acquisition-release cycle. Under WI, compared to RMA-RW, ShiftLock reduces 0.94 atomics (31.9%) and 1.25 Reads (77.6%) per cycle, effectively cutting network IOPS consumption. DSLR requires 7.92 Reads despite its heuristic backoff scheme, which is 22.00× more than ShiftLock and 4.92× more than RMA-RW. DrTM must tolerate clock drift and incurs excessive CAS retries. CAS's two variants demonstrate backoff's effectiveness: CAS-B reduces RDMA CASs by 73.1% than CAS-NB. However, compared to ShiftLock, CAS-B still requires 0.35 more atomics per cycle.

Under RI, mutexes are unaware of reader-writer semantics and deliver the same results as WI. RWLocks deliver various degrees of performance improvement, after which ShiftLock still outperforms others. Compared to DSLR, ShiftLock guarantees bounded wait time for readers even when many writers contend for the same lock, thus reducing 0.80 Reads (79.7%) per lock acquisition on average. For DrTM and RMA-RW, the reasons ShiftLock outperforms them are similar to WI.

The analyses above support our argument that *backoff only alleviates the retry problem instead of solving it*, and that lock handover is an effective solution.

4.3 Scalability

We study the scalability of ShiftLock and baselines with different lock counts, client counts, and workload skewnesses.

For the Zipfian-0.99 WI workload, Figure 7 shows how the locks scale with different lock counts. Here, DrTM's p99 latencies are too high (> 10^4 µs) and hence omitted from the latency figure. ShiftLock constantly delivers the highest goodput and performs relatively better with fewer locks (1.52× of baselines with 1K locks, but 1.38× with 10 million locks) due to higher contention. Although CAS can have lower latencies than ShiftLock, it does not allow concurrent readers and suffers from lower goodputs.

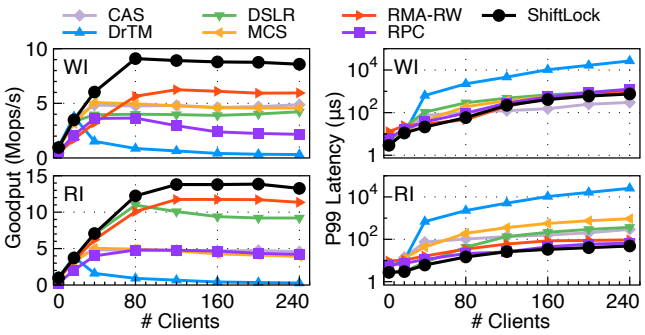

Figure 8: Performance of locks with different client counts.

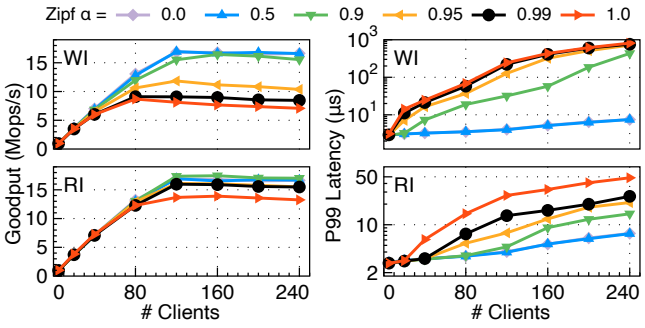

Figure 9: ShiftLock under different workload skewnesses.

For the Zipfian-0.99 WI/RI workloads, Figure 8 shows how the locks scale with different client counts. For WI/RI workloads, Figure 9 shows how ShiftLock scales with different workload skewnesses. We make the following observations. First, ShiftLock scales to 80 (WI) or 120 (RI) clients and keeps a stable goodput when the client count further increases. Skewness and writer ratio are both dominating factors of ShiftLock's scalability. Second, the goodput plateaus prove that the server-side RNIC is the throughput bottleneck. ShiftLock outperforms baselines because it intentionally and effectually reduces network traffic to the server.

4.4 Distributed Transactions

We evaluate ShiftLock and baselines with distributed transaction workloads including TATP [3] and TPC-C [4], which are generated from tools in FissLock [70]. For TPC-C, we use 100 warehouses per node. The TATP workload has 680 236 total locks and is read-intensive (80.6% of lock requests are readers); the TPC-C workload has 60 490 total locks and is write-intensive (86.5% of lock requests are writers). We adopt the two-phase locking (2PL) concurrency control protocol: clients acquire all locks required by the transaction before executing it and release them after the transaction commits.

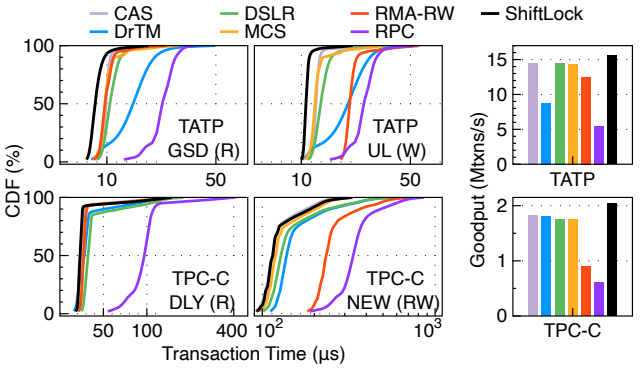

Figure 10: Performance of locks under transactions.

Clients simulate the execution of a transaction by spin-waiting for its typical execution time; for TATP and TPC-C, the wait times are 2.8 µs and 7 µs per transaction, respectively [70].

There are 7 transaction types in TATP and 5 in TPC-C, and it would be too verbose to analyze all of them. For brevity, we only discuss the most frequent read and write transactions in TATP and TPC-C, which are GetSubscriberData (GSD, read-only, 35%) and UpdateLocation (UL, write-only, 14%) in TATP, and Delivery (DLY, 4%, read-only) and NewOrder (NEW, read-write, 45%) in TPC-C, respectively. Figure 10 shows the latency CDF curves and the overall goodputs. We make the following two observations.

First, for the TATP workload, ShiftLock performs mildly better than `CAS`, `DSLR`, and `MCS` in both latency and goodput. ShiftLock outperforms `DSLR` due to fewer RDMA accesses to the lock entries, delivering 19.6% and 16.8% lower median latencies for GSD and UL transactions, respectively. ShiftLock also outperforms `CAS` and `MCS` by 6.9% in goodput because it supports reader locks. `CAS`'s and `MCS`'s goodputs are close to ShiftLock and `DSLR` because in TATP, most locks are cold and only a few are hot, and the difference between mutexes and RWLocks cannot manifest. Compared to other RDMA locks, ShiftLock improves goodput by 1.25× to 2.85×.

Second, for the TPC-C workload, ShiftLock delivers the highest overall goodput, outperforming other RDMA locks by 1.09× to 2.14× with comparable latencies. Due to TPC-C's high contention level, the benefits of the lock handover approach manifest. Note that `MCS` and `RMA-RW` both support handover but deliver very different results: `MCS`'s goodput is only slightly worse than ShiftLock since it does not support reader locks (which also explains `CAS`'s performance), while `RMA-RW` performs the worst among all RDMA locks. We find that this phenomenon is because `RMA-RW` has extra lock mode maintenance overheads (which we have discussed before) and that its design is read-preferring. Specifically, when acquiring a writer lock, `RMA-RW` waits until no readers are holding this lock, which can be blocked for a long time if there are continuously new readers. `RMA-RW`'s bad performance, as a negative example, supports ShiftLock's write-preferring design choice in which a writer immediately blocks new readers after it.

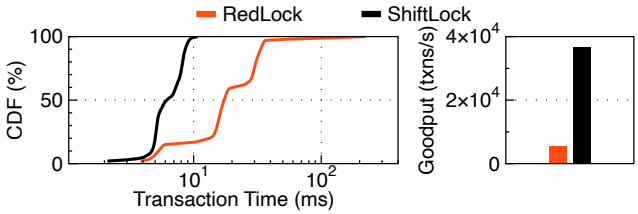

Figure 11: Performance of the banking application.

4.5 Application

We build a banking application backed by Redis [54] to evaluate ShiftLock against RedLock [55], the official implementation of Redis's distributed lock[1]. This application follows the design of SmallBank [5] and uses the 2PL protocol for concurrency control. The application maintains 1 million accounts. It performs 15% read-only transactions that check account balances and 85% read-write transactions that transfer funds between accounts [18]. We run Redis on the lock server machine and configure it to use 16 I/O threads. Figure 11 shows the transaction latency CDFs and the overall goodputs.

RedLock achieves 5.56 Kops/s average throughput, while ShiftLock improves it by almost an order of magnitude to 36.55 Kops/s. We have confirmed that the application reaches its bottlenecks with both locks (by observing no goodput improvement with various numbers of clients), proving that RedLock is the major factor that hinders performance. ShiftLock lowers the concurrency control overheads, cutting the median latency by 62.9% and the p99 latency by 95.2%.

4.6 Latency Breakdown

We analyze the breakdown of latencies delivered by ShiftLock and baselines. First, we run ShiftLock with different client counts to understand its own performance; second, we run ShiftLock, `DSLR`, and `CAS` with 240 clients to compare them with each other. We use the Zipfian-0.99 WI/RI workloads and measure the average time a client spends on different tasks. Table 2 shows the items we timed, and Tables 3 and 4 show the results. We make the following observations.

First, in ShiftLock, writers consistently spend one RDMA roundtrip to enter the queue. For the WI workload, the time spent by the initial RDMA atomic (IA) only rises marginally as contention increases, showing that ShiftLock preserves a low overhead to coordinate writers. For the RI workload, since readers poll the lock entries and the goodput is large, the server-side RNIC experiences a high pressure, causing a 3.13× increase in IA's latency from 5 to 240 clients.

Second, for the WI workload, our write-preferring and starvation-free design achieves its desired effects. We derive this conclusion from the comparison of the time a reader waits for writers (WW) with the time a writer waits for preceding

[1] Redis uses IPoIB in this experiment. We have attempted to make RedLock use userspace RDMA with Mellanox VMA [40], but surprisingly, the transaction goodput drastically decreases (by 96%). Hence, we did not use it.

Time Item	Description
IA	The initial RDMA atomic that tries acquring the lock, entering the queue, or releasing the lock.
RT	Retries, including backoff.
NP	The RDMA SEND to notify predecessor.
WP	CQ polls to wait for predecessor notification.
WR	RDMA READS to wait for previous readers.
WW	RDMA READS to wait for a lock transfer to readers.
NS	CQ polls to wait for successor notification + the RDMA SEND to notify it.

Table 2: Items that are timed in latency breakdown.

writers (WP) or preceding readers (WR). With 80 or fewer clients, WP and WR are both shorter than WW, demonstrating that ShiftLock can grant locks faster to waiting writers than to waiting readers. When the client count increases beyond 80, a writer will wait proportionally longer in WP because its preceding writers queue up. However, WW only increases marginally from 2.00 µs to 2.86 µs when the client number increases from 80 to 240, demonstrating that ShiftLock avoids reader starvation and guarantees bounded wait time.

Third, for the RI workload, readers wait less time for writers compared with WI, and writers wait more for readers. This follows our intuition and expectation. Different from WI, the client-to-client communication time (NP) significantly increases, which we believe is caused by the increased IOPS pressure at client machines due to increased goodput.

Fourth, ShiftLock achieves the lowest latency. Specifically, `CAS-B` has a lower average latency (−12.7%) for writer locks, but ShiftLock outperforms it by 7.44× for reader locks. For `CAS-NB` and `DSLR` for the WI workload, due to excessive retries, RDMA atomics suffer from significantly higher latencies: `CAS-NB`'s CAS is 15.39× slower than ShiftLock's CAS, while `DSLR`'s FAA is 3.10× slower than ShiftLock's FAA. The root cause is that the RNIC queues up non-parallelizable accesses to the same memory address [25, 30], and the excessive RDMA READS and CASs increase the queueing latency. In contrast, ShiftLock avoids most retries with lock handover and shortens the hardware queue. For the RI workload, however, ShiftLock's IA latency is 2.29× to `DSLR`. The reasons are twofold: (a) 16-byte extended atomics intrinsically suffer from lower performance than `DSLR`'s 8-byte conventional atomics, and (b) readers in ShiftLock cannot benefit from `DSLR`'s heuristic-based dynamic backoff scheme. Nevertheless, when transferring the lock to readers, ShiftLock allows all currently waiting readers to acquire the lock, while `DSLR` only permits those who arrive before the next writer. Therefore, with ShiftLock, readers experience a significantly lower wait time (−91.6%) than `DSLR` on average.

4.7 Parameter Selection

We analyze the choice of the parameter N, which represents the maximum consecutive writer count in ShiftLock. We use the Zipfian-0.99 write-intensive workload and vary N from 1 to ∞. Figure 12 shows ShiftLock's lock acquisition latencies and goodputs with different N values.

ShiftLock experiences a stable goodput only after $N \geq 16$ because when N is small, writers frequently transfer the lock to readers, underutilizing the chances of lock handover. The median latency remains stable when $N \leq 32$, while the tail latency decreases as N increases. This is because tail latencies are attributed to the long-waiting writers, who wait shorter for readers when the lock can be handed over more frequently. When $N = \infty$, we observe a maximum of 261 consecutive writers in repeated experiments. The goodput slightly increases by 3.3% compared to $N = 32$, but the median latency also significantly increases by 36.5%. Based on the results above, we believe our choice, $N = 16$, achieves a good balance between goodput and latency and prevents starvation.

WI	Acq (W)				Acq (R)		Rel	
# Clients	IA	NP	WP	WR	IA	WW	IA	NS
5	2.05	0.01	0.01	0.01	2.03	0.01	2.05	0.02
20	2.24	0.07	0.10	0.05	2.22	0.20	2.25	0.10
40	2.27	0.15	0.25	0.07	2.25	0.78	2.27	0.18
80	2.41	0.23	1.88	0.09	2.38	2.00	2.39	0.28
120	2.41	0.22	10.14	0.09	2.38	2.14	2.40	0.29
160	2.43	0.22	19.51	0.09	2.40	2.32	2.42	0.31
200	2.49	0.23	31.75	0.09	2.46	2.63	2.49	0.33
240	2.50	0.23	42.63	0.09	2.47	2.87	2.51	0.34
RI	**Acq (W)**				**Acq (R)**		**Rel**	
# Clients	IA	NP	WP	WR	IA	WW	IA	NS
5	2.41	0.09	0.37	0.32	2.32	0.00	2.29	0.00
20	2.49	0.11	0.22	1.24	2.41	0.01	2.38	0.00
40	2.64	0.25	0.50	2.09	2.49	0.04	2.46	0.02
80	3.60	0.70	1.83	3.57	2.68	0.24	2.69	0.08
120	5.14	1.02	3.48	4.45	3.44	0.59	3.48	0.14
160	5.91	1.07	4.18	4.97	4.52	0.87	4.58	0.18
200	6.96	1.13	5.23	5.83	5.79	1.21	5.87	0.21
240	7.84	1.18	7.03	6.57	6.95	1.66	7.03	0.24

Table 3: Latency breakdown of ShiftLock (in microseconds, on average) with different client numbers.

WI	Acq (W)			Acq (R)			Rel	
Lock	IA	NP+WP+WR	RT	IA	WW	RT	IA	NS
ShiftLock	2.50	42.95	-	2.47	2.87	-	2.51	0.34
`CAS-B`	2.48	-	37.23	2.48	-	37.23	2.34	-
`CAS-NB`	38.50	-	412.98	38.50	-	412.98	37.92	-
`DSLR`	7.76	-	75.36	7.76	-	74.86	7.73	-
RI	**Acq (W)**			**Acq (R)**			**Rel**	
Lock	IA	NP+WP+WR	RT	IA	WW	RT	IA	NS
ShiftLock	7.96	15.21	-	7.01	1.70	-	7.09	0.23
`DSLR`	3.47	-	18.01	3.46	-	20.26	3.51	-

Table 4: Latency breakdown of different lock schemes (in microseconds, on average) with 240 clients.

4.8 Fault Tolerance

We now evaluate how ShiftLock can handle client failures. We run the Zipfian-0.99 write-intensive microbenchmark under different failure rates from 10^{-5} to 10^{-2}. Given a failure rate p, a client will skip releasing the lock it just acquired with

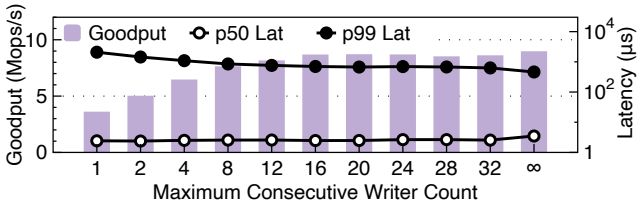

Figure 12: Performance of ShiftLock with different maximum consecutive writer counts.

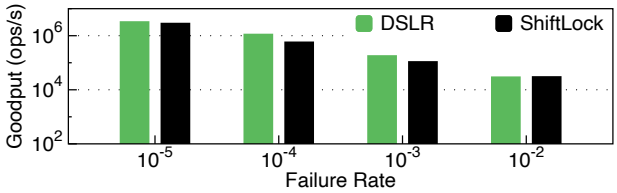

Figure 13: Performance of ShiftLock and DSLR under different client failure rates.

probability p in each iteration. This simulates a client failure scenario in which the locks will be held forever and must be recovered to allow clients to make further progress. Figure 13 shows the goodputs of ShiftLock and DSLR, the only baseline that supports failure recovery.

Overall, ShiftLock has the ability to recover in the presence of client failures, but it has lower goodputs than DSLR. DSLR and ShiftLock detect client failures in similar ways: if they time out when waiting for a lock to be released, they will assume a client failure and start recovery. The reason for ShiftLock's lower performance is that the timeout is $2\times T_{\text{lease}}$ for DSLR and $3\times T_{\text{lease}}$ for ShiftLock, which means that ShiftLock must wait longer than DSLR before each recovery. Also, in ShiftLock, when a client sends a message to a failed client, it will get a Receive-Not-Ready (RNR) error and the DCI QP will shift to the error state; in this situation, ShiftLock has to reset the QP and bring it up again. Despite these drawbacks, ShiftLock can still achieve the same order of magnitude of performance as DSLR.

5 Related Work

Distributed lock management. Lock is a fundamental research topic in distributed systems as it is the keystone of correct concurrency. There has been a mass of previous work focusing on locks within a single machine [11, 12, 16, 17, 27, 28, 37, 43], and it was common practice to manage distributed locks with a single-machine lock manager. With the wide deployment of high-throughput and low-latency RDMA, such CPU-based lock managers become bottlenecks, motivating the design of RDMA locks that use RDMA one-sided verbs to manage locks [20, 46, 67, 69].

However, in the above RDMA lock designs, clients only use RDMA one-sided verbs to access the lock table stored on the server to manipulate locks. Under high contention, the inbound RDMA one-sided verb traffic can overwhelm the lock server's RNIC and result in suboptimal performance. These locks are unaware that clients can directly communicate with each other in a two-sided manner since their CPUs are available to run the lock management logic. ShiftLock takes this chance and properly employs lock handover to reduce network traffic at the lock server and improve performance.

Handover. The idea of handover appears in multiple different forms in previous work. Some studies hand over locks. There are a host of studies focusing on this topic in a single machine [27, 43, 60]. In distributed systems, RMA-RW translates MCS into its distributed version directly using MPI's one-sided communication verbs [59]. Thakur et al. propose maintaining a lock table in the server so that every client can see which other clients are waiting and wake them up upon lock release [62]. Sherman [64] employs machine-local lock tables, allowing clients on the same machine to hand over locks before releasing them to the global lock table. A few studies (e.g., SCL [52]) rely on a scheduler to decide the next lock holder instead of handing over locks directly.

Some other studies hand over (i.e., delegate) tasks to avoid lock contention [10, 13, 22, 53, 58, 68, 73]. Task delegation can effectively resolve conflicts among clients, but only when the tasks' arguments are small (thus transmittable over the network) and when concurrency control is simple (e.g., each transaction acquires only one lock). ShiftLock, as a lock approach, can be applied to wider scenarios than task delegation, including when clients have large local states that are cost-prohibitive to transmit, when tasks can not be delegated due to privacy concerns, and when transactions require multiple locks and do not have clear target clients to be delegated to.

6 Conclusion

We present ShiftLock, an RDMA lock optimizing for high contention by reducing retries with lock handover. ShiftLock realizes efficient and practical handover with proper software designs and exertion of RDMA features. It supports starvation-free reader-writer locks, delivers optimal latency under low contention, and can tolerate client failures. Evaluation shows that ShiftLock delivers similar or better performance in various scenarios compared to existing distributed locks.

Acknowledgment

We sincerely thank our shepherd Sanidhya Kashyap for helping us improve the paper. We are also grateful to the reviewers of this paper for their helpful comments and feedback. This work is supported by the National Natural Science Foundation of China (Grant No. U22B2023, Grant No. 62472242), the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2023QNRC001), the Postdoctoral Fellowship Program of CPSF (Grant No. GZC20231296), and Alibaba Group through the Alibaba Innovative Research Program.

A Artifact Appendix

Abstract

The artifact consists of the Rust code of ShiftLock and baseline systems used in evaluation, runner scripts for reproducing the experiments, plotter scripts for visualizing the results, and related workload trace and description files. It is intended for validating the claims made in the paper and facilitating further research on one-sided RDMA locks.

Scope

At a high level, the artifact allows its users to validate the following claims in the paper:

- ShiftLock outperforms baselines under high contention, and performs similarly under low contention.
- ShiftLock's design effectively solves the retry problem, and this is where it outperforms baselines. It makes proper performance trade-offs and incurs acceptable overheads.
- ShiftLock can efficiently recover from client failures.

These claims correspond to the questions raised at the start of the Evaluation section (§4).

The main purpose of the artifact is to reproduce the experiments in the paper and validate the claims above. Aside from that, since the artifact contains full implementations of ShiftLock and baselines, it can be used as a starting point for further research on one-sided RDMA locks, especially those based on the Rust programming language.

Contents

The artifact contains source code, scripts, workload trace files, and README files. Below, we explain the contents aside from the README. More detailed descriptions can be found in `HOWTOUSE.md` in the artifact repository.

Source code. The `src` directory contains the source code of ShiftLock and baseline systems. `main.rs`, the entry point of a typical Rust project, contains a hello-world example of ShiftLock and can run independently on a single machine. ShiftLock's implementation is in the `shiftlock` subdirectory, while the baseline systems are in the `baselines` subdirectory. Experiment infrastructure is in the `bin` subdirectory.

Scripts. The `scripts` directory contains runner scripts for running experiments and plotter scripts for visualizing the results. Scripts directly reproducing the experiments (i.e., AE scripts) in the paper are in the `ae` subdirectory. Scripts not in the `ae` subdirectory serve as building blocks of the AE scripts, such as running a single experiment.

Workload traces. The `traces` directory is a git submodule pointing to an external repository containing the workload traces used in the evaluation. The main reasons for using a submodule are: (1) the traces are large, and (2) they are produced by FissLock [70] instead of ShiftLock and do not necessarily need to be versioned with the artifact. Specifically, FissLock's source code repository can be found on GitHub. With the tools provided in the `experiments/trace_gen` directory in FissLock's repository, one can independently generate the traces used in the evaluation.

Hosting

ShiftLock's artifact repository is hosted on GitHub at `https://github.com/thustorage/shiftlock` on the `master` branch. The first usable commit version is the initial commit (`82555c0`). However, due to possible future bugfixes and updates, please always use the latest commit.

Requirements

- A cluster of at least 6 Linux servers with Internet access
- Infiniband RDMA network
- Mellanox ConnectX-5 or newer RDMA NICs
- MLNX OFED v4.x, which can be downloaded here in the "Archived Versions" tab
- Rust 1.83 or newer
- The lock server has password-less SSH access to all clients

Detailed Claims to Verify

In detail, the artifact aims to verify the following claims made throughout the Evaluation section (§4) in the paper:

- Section 4.2:
  - Figure 4: ShiftLock *outperforms baselines* in terms of goodput for highly contended, non-read-only workloads, and *performs similarly to the best baselines* for other workloads.
  - Figure 5: ShiftLock delivers *the lowest latencies*.
  - Figure 6: ShiftLock produces the *lowest network traffic* towards the lock server.
- Section 4.3:
  - Figure 7: ShiftLock *consistently delivers the highest goodput* with different lock counts.
  - Figure 8: ShiftLock *scales with an increasing number of clients* until bottlenecked by the server-side RNIC.
- Section 4.4 (Figure 10): ShiftLock *outperforms baselines* under TPC-C and TATP workloads.
- Section 4.5 (Figure 11): ShiftLock *outperforms RedLock* in a banking application.
- Section 4.6:
  - Table 3: ShiftLock *has a low overhead* for writers and *ensures bounded wait time* for readers.
  - Table 4: ShiftLock avoids most retries, hence it *reduces RDMA verb latencies and wait time*.
- Section 4.7 (Figure 12): ShiftLock's choice of $N = 16$ *achieves a good balance between goodput and latency*.
- Section 4.8 (Figure 13): ShiftLock *recovers from client failures* with acceptable performance.

References

[1] BeeGFS - The Leading Parallel Cluster File System. https://www.beegfs.io/c/.

[2] Lustre® Filesystem. https://www.lustre.org/.

[3] TATP Benchmark. https://tatpbenchmark.sourceforge.net/.

[4] TPC-C. https://www.tpc.org/tpcc/.

[5] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585, April 2008. https://ieeexplore.ieee.org/abstract/document/4497466.

[6] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990. https://ieeexplore.ieee.org/document/80120.

[7] Thomas E Anderson, Simon Peter, Marco Canini, Jongyul Kim, Dejan Kostic, Youngjin Kwon, Waleed Reda, Henry N Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 1011–1027. USENIX, November 2020. https://www.usenix.org/conference/osdi20/presentation/anderson.

[8] Amanda Baran, Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri. ALock: Asymmetric Lock Primitive for RDMA Systems. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '24, pages 15–26, New York, NY, USA, June 2024. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3626183.3659977.

[9] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: it's time for a redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, March 2016. https://dl.acm.org/doi/10.14778/2904483.2904485.

[10] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2477–2489, Virtual Event China, June 2021. ACM. https://dl.acm.org/doi/10.1145/3448016.3457560.

[11] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High performance locks for multi-level NUMA systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '15)*, pages 215–226, San Francisco CA USA, January 2015. ACM. https://dl.acm.org/doi/10.1145/2688500.2688503.

[12] Milind Chabbi and John Mellor-Crummey. Contention-conscious, locality-preserving locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*, pages 1–14, Barcelona Spain, February 2016. ACM. https://dl.acm.org/doi/10.1145/2851141.2851166.

[13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48, Farminton Pennsylvania, November 2013. ACM. https://dl.acm.org/doi/10.1145/2517349.2522714.

[14] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192, May 2010. https://ieeexplore.ieee.org/document/5479555.

[15] A. Devulapalli and P. Wyckoff. Distributed Queue-based Locking using Advanced Network Features. In *2005 International Conference on Parallel Processing (ICPP '05)*, pages 408–415, June 2005.

[16] Dave Dice and Alex Kogan. Compact NUMA-aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, Dresden Germany, March 2019. ACM. https://dl.acm.org/doi/10.1145/3302424.3303984.

[17] David Dice, Virendra J Marathe, and Nir Shavit. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*, page 10, February 2012.

[18] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, September 2014.

[19] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle WA USA, April 2014. USENIX. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87.

[20] Jian Gao, Youyou Lu, Minhui Xie, Qing Wang, and Jiwu Shu. Citron: Distributed Range Lock Management with One-sided RDMA. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST '23)*, pages 297–314, Santa Clara CA USA, February 2023. USENIX. https://www.usenix.org/conference/fast23/presentation/gao.

[21] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, November 1989. https://dl.acm.org/doi/10.1145/74851.74870.

[22] Vishal Gupta, Kumar Kartikeya Dwivedi, Yugesh Kothari, Yueyang Pan, Diyu Zhou, and Sanidhya Kashyap. Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLOCKS. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 1–16, 2023. https://www.usenix.org/conference/osdi23/presentation/gupta.

[23] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, pages 1–12, November 2012.

[24] 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, 2022. https://www.usenix.org/conference/fast22/presentation/jiang.

[25] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16)*, page 15, Denver CO USA, June 2016. USENIX.

[26] 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 '16)*, pages 185–201, Savannah GA USA, November 2016. USENIX. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia.

[27] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, pages 586–599, Huntsville Ontario Canada, October 2019. ACM. https://dl.acm.org/doi/10.1145/3341301.3359629.

[28] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, page 15, July 2017.

[29] 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 CD-ROM*, pages 756–771, Virtual Event Germany, October 2021. ACM. https://dl.acm.org/doi/10.1145/3477132.3483565.

[30] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, 2022. https://www.usenix.org/conference/nsdi22/presentation/kong.

[31] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A Fair Fast Scalable Rea,der-Writer Lock. In *1993 International Conference on Parallel Processing - ICPP'93*, volume 2, pages 201–204, August 1993. https://ieeexplore.ieee.org/document/4134208.

[32] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974. https://dl.acm.org/doi/10.1145/361082.361093.

[33] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 101–110, New York, NY, USA, August 2009. Association for

Computing Machinery. https://dl.acm.org/doi/10.1145/1583991.1584020.

[34] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. LocoFS: a loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Denver Colorado, November 2017. ACM. https://dl.acm.org/doi/10.1145/3126908.3126928.

[35] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, page 17. USENIX, November 2020. https://www.usenix.org/system/files/osdi20-li_yuliang.pdf.

[36] Linux PTP Project. Linux PTP Project - Documentation. https://linuxptp.nwtime.org/documentation/, 2024.

[37] Ran Liu, Heng Zhang, and Haibo Chen. Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pages 219–230, Philadelphia PA, USA, June 2014. USENIX. https://www.usenix.org/conference/atc14/technical-sessions/presentation/liu.

[38] Youyou Lu, Jiwu Shu, Tao Li, and Youmin Chen. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu.

[39] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. X-RDMA: Effective RDMA Middleware in Large-scale Production Environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, September 2019. https://ieeexplore.ieee.org/document/8891004/?arnumber=8891004.

[40] Mellanox. libvma: Linux user space library for network socket acceleration based on RDMA compatible network adaptors. https://github.com/Mellanox/libvma.

[41] Mellanox Technologies. RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015.

[42] Mellanox Technologies. Mellanox Connect-IB® Firmware Release Notes. https://network.nvidia.com/pdf/firmware/ConnectIB-FW-10_16_1200-release_notes.pdf, 2017.

[43] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991. https://dl.acm.org/doi/10.1145/103727.103729.

[44] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '91, pages 106–113, New York, NY, USA, April 1991. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/109625.109637.

[45] Ali Najafi and Michael Wei. Graham: Synchronizing Clocks by Leveraging Local Clock Properties. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*, pages 453–466, Renton WA USA, April 2022. USENIX. https://www.usenix.org/conference/nsdi22/presentation/najafi.

[46] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 583–590, May 2007.

[47] NVIDIA Corporation. Advanced Transport. https://docs.mellanox.com/display/MLNXOFEDv531001/Advanced+Transport.

[48] NVIDIA Corporation. NVIDIA ConnectX-5 InfiniBand Adapter Cards Datasheet. https://nvdam.widen.net/s/pkxbnmbgkh/networking-infiniband-datasheet-connectx-5-2069273.

[49] NVIDIA Corporation. NVIDIA ConnectX-6 Datasheet. https://nvdam.widen.net/s/5j7xtzqfxd/connectx-6-infiniband-datasheet-1987500-r2.

[50] NVIDIA Corporation. NVIDIA ConnectX-7 Datasheet. https://nvdam.widen.net/s/m6pt7j5rlb/networking-datasheet-infiniband-connectx-7-ds---1779005.

[51] NVIDIA Corporation. RoCE Time-Stamping. https://docs.nvidia.com/networking/display/rdmacore50/RoCE+Time-Stamping, September 2023.

[52] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Avoiding scheduler subversion using scheduler-cooperative locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, Heraklion Greece, April 2020. ACM. https://dl.acm.org/doi/10.1145/3342195.3387521.

[53] Darko Petrović, Thomas Ropars, and André Schiper. On the Performance of Delegation over Cache-Coherent Shared Memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, pages 1–10, Goa India, January 2015. ACM. https://dl.acm.org/doi/10.1145/2684464.2684476.

[54] Redis. Community Edition. https://redis.io/docs/latest/get-started/, 2024.

[55] Redis. Distributed Locks with Redis. https://redis.io/docs/latest/develop/use/patterns/distributed-locks/, 2024.

[56] Feng Ren, Mingxing Zhang, Kang Chen, Huaxia Xia, Zuoning Chen, and Yongwei Wu. Scaling Up Memory Disaggregated Applications with SMART. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 351–367, La Jolla CA USA, April 2024. ACM. https://dl.acm.org/doi/10.1145/3617232.3624857.

[57] Kun Ren, Alexander Thomson, and Daniel J. Abadi. VLL: a lock manager redesign for main memory database systems. *The VLDB Journal*, 24(5):681–705, October 2015. https://doi.org/10.1007/s00778-014-0377-7.

[58] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 342–358, Shanghai China, October 2017. ACM. https://dl.acm.org/doi/10.1145/3132747.3132771.

[59] Patrick Schmid, Maciej Besta, and Torsten Hoefler. High-Performance Distributed RMA Locks. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 19–30, Kyoto Japan, May 2016. ACM. https://dl.acm.org/doi/10.1145/2907294.2907323.

[60] Michael L. Scott and William N. Scherer. Scalable queue-based spin locks with timeout. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pages 44–52, New York, NY, USA, June 2001. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/379539.379566.

[61] Ge Shi, Ziyi Yan, and Tianzheng Wang. OptiQL: Robust Optimistic Locking for Memory-Optimized Indexes. *Proc. ACM Manag. Data*, 1(3):216:1–216:26, November 2023. https://dl.acm.org/doi/10.1145/3617336.

[62] Rajeev Thakur, Robert Ross, and Robert Latham. Implementing Byte-Range Locks Using MPI One-Sided Communication. In Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 119–128, Berlin, Heidelberg, 2005. Springer.

[63] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. Contention-aware lock scheduling for transactional databases. *Proceedings of the VLDB Endowment*, 11(5):648–662, January 2018. http://dl.acm.org/citation.cfm?doid=3187009.3177740.

[64] 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*, pages 1033–1048, Philadelphia PA USA, June 2022. ACM. https://dl.acm.org/doi/10.1145/3514221.3517824.

[65] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. Replicating Persistent Memory Key-Value Stores with Efficient RDMA Abstraction. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*, pages 441–459, Boston MA USA, July 2023. USENIX. https://www.usenix.org/conference/osdi23/presentation/wang-qing.

[66] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 121–136, 2022. https://www.usenix.org/conference/atc22/presentation/wei.

[67] 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, Monterey California, October 2015. ACM. https://dl.acm.org/doi/10.1145/2815400.2815419.

[68] Ziwei Xiong, Dejun Jiang, and Jin Xiong. DiStore: A Fully Memory Disaggregation Friendly Key-Value Store with Improved Tail Latency and Space Efficiency. In *Proceedings of the 53rd International Conference*

on Parallel Processing, ICPP '24, pages 607–617, New York, NY, USA, August 2024. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3673038.3673088.

[69] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*, pages 1571–1586, Houston TX USA, May 2018. ACM. https://dl.acm.org/doi/10.1145/3183713.3196890.

[70] Hanze Zhang, Ke Cheng, Rong Chen, and Haibo Chen. Fast and Scalable In-network Lock Management Using Lock Fission. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*, pages 1–18, Santa Clara CA USA, July 2024. USENIX. https://www.usenix.org/conference/osdi24/presentation/zhang-hanze.

[71] Ming Zhang, Yu Hua, and Zhijun Yang. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 801–819, 2024. https://www.usenix.org/conference/osdi24/presentation/zhang-ming.

[72] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*, pages 51–67, Santa Clara CA USA, February 2022. USENIX. https://www.usenix.org/conference/fast22/presentation/zhang-ming.

[73] 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, 2022. https://www.usenix.org/conference/osdi22/presentation/zhou-diyu.

[74] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC '21)*, pages 15–29, Santa Clara CA USA, July 2021. USENIX. https://www.usenix.org/conference/atc21/presentation/zuo.