CITRON: Distributed Range Lock Management with One-sided RDMA

Jian Gao, Youyou Lu, Minhui Xie, Qing Wang, and
Jiwu Shu, *Tsinghua University*

https://www.usenix.org/conference/fast23/presentation/gao

This paper is included in the Proceedings of the
21st USENIX Conference on File and
Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings
of the 21st USENIX Conference on
File and Storage Technologies
is sponsored by

NetApp®

CITRON: Distributed Range Lock Management with One-sided RDMA

Jian Gao Youyou Lu Minhui Xie Qing Wang Jiwu Shu*
Tsinghua University

Abstract

Range lock enables concurrent accesses to disjoint parts of a shared storage. However, existing range lock managers rely on centralized CPU resources to process lock requests, which results in server-side CPU bottleneck and suboptimal performance when placed in a distributed scenario.

We propose CITRON, an RDMA-enabled distributed range lock manager that bypasses server-side CPUs by using only one-sided RDMA in range lock acquisition and release paths. CITRON manages range locks with a static data structure called segment tree, which effectively accommodates dynamically located and sized ranges but only requires limited and nearly constant synchronization costs from the clients. CITRON can also scale up itself in microseconds to adapt to a shared storage of a growing size at runtime. Evaluation shows that under various workloads, CITRON delivers up to 3.05× throughput and 76.4% lower tail latency than CPU-based approaches.

1 Introduction

Large-scale distributed applications have high demands to access shared storage resources concurrently [60, 65]. File systems designed for high-performance computing (HPC), for example, are usually required to handle massive parallel I/O requests to different parts of a single data file [11, 32, 49]. Disaggregated memory pools have the need to allow multiple clients to access the same memory space simultaneously, possibly with different access patterns [19, 40, 50]. These systems require the capability to correctly and efficiently coordinate concurrent accesses to a large-scale shared storage.

Lock is a common and essential approach to enabling correct concurrent accesses to a shared storage. A wealth of research contributes to designing mutual exclusive locks (i.e., *mutexes*) and their variants [20, 22, 29, 33, 76], which grant exclusive access (or write) permission of the shared storage to at most one client at any time. Still, mutexes can be too coarse-grained and, thus, inefficient. For this reason, range locks become a preferable alternative since they allow finer-grained concurrency, i.e., clients simultaneously operating at disjoint parts of the same shared storage resource [35, 39].

Existing distributed range lock managers (DRLMs) grant and revoke locks with *centralized* server-side[1] CPUs through a remote procedure call (RPC) interface. However, with prevalent high-speed networks, CPU-oriented DRLMs cause performance bottlenecks. First, they limit throughput because of the mismatch between high network packet rate (e.g., 215 Mops/s for NVIDIA ConnectX-6 [58]) and limited CPU resources and that they need to perform CPU-consuming traversal and modification to complex dynamic data structures upon lock operations (e.g., the interval trees in Lustre [47]). Second, they also incur high queueing latencies (4-5 network roundtrip times, §2.2) due to the RPC paradigm. In latency-sensitive scenarios like memory pools, a CPU-based DRLM can become a major latency contributor in the critical path.

Remote Direct Memory Access (RDMA) offers a chance to avoid the CPU bottleneck with its one-sided verbs that can bypass server-side CPUs. However, taking this chance requires a comprehensive re-design of the range lock protocol. First, existing DRLMs are built atop dynamic data structures, but RDMA incapacitates them due to the lack of support for dynamic remote memory allocation. Second, these DRLMs are also RDMA-unconscious and perform many memory accesses to their data structures in lock operations, which turn into excessive network roundtrips when using one-sided RDMA, overshadowing RDMA's high performances.

This paper proposes CITRON, an efficient distributed range lock manager. CITRON acquires and releases range locks using only one-sided RDMA to lift the burden off server-side CPUs with an RDMA-conscious lock protocol based on static data structures to exploit the full performance potentials of the RDMA hardware. Specifically, CITRON retrofits *segment tree*, an RDMA-friendly static data structure, to manage lock entries. Thus, CITRON simplifies lock conflict resolution into the communication between ancestor and descendant nodes on the tree. To effectively handle dynamically positioned and sized lock requests, CITRON develops a protocol tightly interwoven with the range lock specs, the segment tree's memory layout, and the one-sided RDMA semantics. Clients lock at different levels of the segment tree and pay nearly constant costs to synchronize with conflicting peers. In the best case, lock acquisition takes only two RDMA roundtrips.

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

[1] For disambiguation, in this paper, we use different terms for different purposes: *servers* are counterparts of clients; *machines* are computers in the distributed system; *nodes* are components of tree data structures.

For a shared storage whose size grows, CITRON provides a mechanism to scale itself up (i.e., expand its capacity), leveraging the structural self-similarity of segment trees. CITRON enables scaling up the lock tree to a proper size with one-sided RDMA and minimum server-side CPU intervention.

CITRON offers several benefits. First, it is CPU-efficient. To our knowledge, CITRON is the first DRLM that uses only one-sided RDMA for lock acquisitions and releases, which obviates the server-side CPU bottleneck. Second, CITRON delivers high performance. Evaluation shows that CITRON outperforms CPU-based range lock managers by up to 3.05× in throughput and 76.4% in latency under different workloads.

2 Background

2.1 RDMA

RDMA is a network protocol with low latency, high throughput, and low CPU overhead. Due to these benefits, numerous distributed file systems [2, 3, 26, 42, 44, 46, 47, 77], transaction systems [5, 16, 31, 41, 72], and lock managers [13, 54, 79] are built atop or compatible with RDMA.

Machines must equip RDMA-capable NICs (RNICs) to communicate with RDMA. Clients first post RDMA verbs to queue pairs (QPs) and later poll the completion queues (CQs) associated with the QPs for completion events. RDMA supports one-sided verbs, including `read`, `write`, atomic `compare-swap` (CAS), and atomic `fetch-add` (FAA). Furthermore, a wide range of off-the-shelf RNICs (e.g., from Mellanox Connect-IB to NVIDIA ConnectX-7 [51, 57–59]) also support *masked atomic verbs* [56], which perform similarly to standard atomic verbs but have more flexibility.

For masked-CAS, users need to provide a compare bitmask and a swap bitmask. The compare and swap steps are each performed with regard to the corresponding bitmask. The masked-out bits will not get compared or swapped.

For masked-FAA, users need to provide a bitmask that splits the 8 bytes into different fields. Each set bit in the bitmask indicates the left boundary of a field, and FAA is performed separately within every field. The field boundaries can occur at any position; non-byte-aligned fields are allowed.

2.2 Distributed Range Lock Management

CPU-based centralized solutions. Most existing DRLMs rely heavily on server-side CPUs [3, 9, 47]. However, this kind of solutions are notorious for their high CPU overheads and the ensuing CPU bottleneck, including limited throughput and high queueing latencies; see Figure 1(a).

First, executing complex range lock operations with limited CPU resources not only bottlenecks the throughput but also inevitably harms co-locating CPU-demanding services that have little chance of being offloaded to the RNIC (e.g., path traversal in a distributed file system). Second, in the RPC paradigm, server-side CPUs fetch and process RPC requests

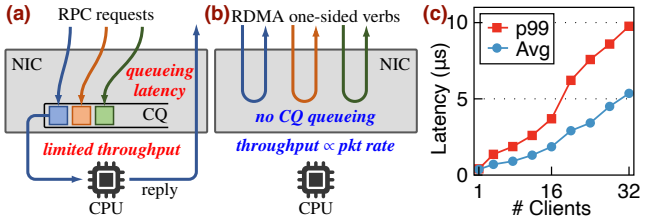

Figure 1: Flaws of RPC-based DRLMs and our motivation.

from the RNIC-side queues. Under high concurrency, the requests will queue in the RNIC, which results in high queueing latencies. Also, when processing a range lock request, a CPU core cannot process other ones in the same queue, even if they do not conflict with each other logically.

We demonstrate the high latencies by running eRPC [30] on a testbed consistent with §4.1. Clients synchronously send RPCs to one server, and the RPC handler runs for 100 ns. We measure the server-side queueing latency, i.e., the time between the RPC arrives at the NIC and the CPU processes the RPC, similarly to 2LClock [27]. Figure 1(c) shows the results. With 32 clients, the average queueing latency is 5.4 µs, more than 2× RDMA roundtrip times (RTTs). The p99 latency even reaches 9.8 µs (4-5 RTTs).

Mutex-based decentralized solutions. Dividing ranges into segments and associating each with a mutex is a strawman solution to decentralized range lock management [35], but it is only efficient when the access granularity is static and priorly known. In the case of unaligned or dynamically-sized ranges, this solution can suffer from a significant 92% throughput decline and 5.65× higher tail latencies; see §4.2.

3 Design

Our design goal is a high-performance DRLM that leverages one-sided RDMA to eliminate server-side CPU bottlenecks. As shown in Figure 1(b), a one-sided RDMA-based DRLM can remove the queueing latencies and offer higher throughput by offloading all lock operations to the RNIC's tailored ASIC, thus exploiting the full performance potentials of the RNIC.

3.1 Challenges and Design Principles

Challenge 1. We need a one-sided RDMA-conscious data structure that can efficiently manage dynamically positioned and sized range locks and resolve their conflicts.

➥ **Static tree structure for dynamic ranges.** CITRON maps each requested range as precisely as possible to a constant number of nodes on a *segment tree*, a static data structure, to effectively manage dynamic range lock entries.

Challenge 2. To achieve low latency and high throughput, we must tailor the lock protocol to reduce the critical path lengths despite the complex range lock semantics.

➥ **Minimized synchronization overhead.** CITRON's protocol couples tightly with RDMA semantics and the segment tree's

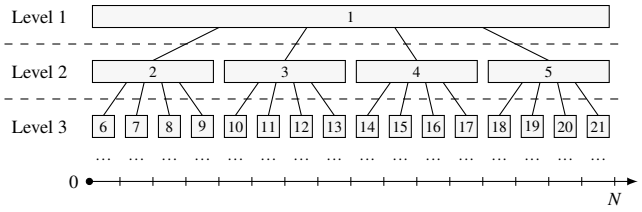

Figure 2: The structure and node indices of the lock tree.

layout, minimizing synchronization costs to nearly constant. Acquiring a lock requires a minimum of only two roundtrips.

Challenge 3. Real-world storage resources are not always fixed-size. Therefore, we must also efficiently handle a possibly dynamically growing storage size.

➥ **Runtime capacity expansion.** While segment trees are static, smaller trees can be seen as subtrees of larger ones. Citron leverages this characteristic to enable scaling up the tree's capacity at runtime using a few server-side CPU cycles.

3.2 Basic Assumptions

Address space. Citron maintains range locks within an abstract address space $[0,\infty)$. Multiple real-world scenarios fit in this model, e.g., LBA ranges in distributed NVMe-oF namespaces [62] and byte ranges in file systems [3, 47].

Cluster infrastructure. Aside from a lock server that hosts Citron's components (§3.3) in its DRAM, Citron requires that there is a cluster manager (CM) and a metadata server (MDS). The CM coordinates configuration changes (§3.5.5) and detects client failures (§3.10). The MDS maintains the addresses of Citron's components to enable the use of one-sided RDMA. The CM and the MDS need not run on independent machines: they can run on the lock server behind an RPC interface. There are already mature solutions for CM and MDS [17, 23, 48], so we need not discuss them here.

Clock well-behavedness. Citron assumes that the clocks of all clients are *well-behaved*, i.e., they advance at nearly the same speeds. Note that Citron does not require the clocks to be *synchronized*. Prior studies report that the clock frequency variation in a productional network is at most ±100 ppm [43] or even ±20 ppm when static errors are filtered out [53], which means that the clock drift is only up to ±0.1 ns or ±0.02 ns per microsecond, more than sufficient for Citron.

3.3 Components of Citron

Citron maintains range locks with a *lock tree* and a *spillover mutex*. The lock tree is responsible for locks within $[0,N)$, and the spillover mutex is for $[N,\infty)$, where N is specified at initialization time. Citron further includes a *maximizer* to enable clients to scale up the lock tree, i.e., to increase N.

Lock tree. The lock tree is a segment tree [4] – a perfectly balanced tree in which each node represents a continuous range. The root represents the entire range $[0,N)$; for every non-root node, it and its siblings each receive an equal and continuous share of the range represented by their parent. Such a structure determines that *the range represented by any node intersects only with its ancestors and its descendants*.

Orthodox segment trees are binary trees [4]. However, we define the lock tree in Citron as a quaternary segment tree, which means that the degrees (i.e., numbers of children) of all internal nodes are all four. Also, leaf nodes represent ranges of size 64, not the 1 in the orthodox definition. These designs aim to limit the tree height and, thus, the number of necessary RDMA verbs to post per lock request.

Since all internal nodes have the same degrees, there is no need for pointers in the lock tree. Instead, all nodes are placed in a continuous flat array by level order and indexed by positive integers (cf. heaps [74]). Tree navigation is simply node index arithmetics. For example, Figure 2 shows the lock tree's first three levels and the node indices, in which the widths of nodes correspond to their represented ranges. From this figure, we can easily verify that for a node with index x,

$$\textsc{Child}(x,i) = 4x-2+i \quad (i=0,1,2,3)$$
$$\textsc{Parent}(x) = \lfloor (x+2)/4 \rfloor$$

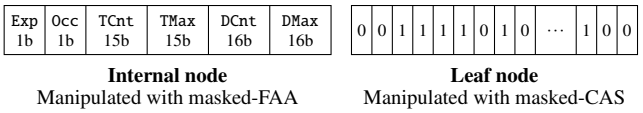

Figure 3: 64-bit representations of internal and leaf nodes.

Spillover mutex. The spillover mutex represents $[N,\infty)$, i.e., it handles out-of-bound parts (w.r.t. the lock tree) of range lock requests. It can adopt any design that is friendly to one-sided RDMA. We use DSLR [79] to implement this mutex.

Maximizer. The maximizer is an initially-zero 8-byte variable accessible by one-sided RDMA. A client modifies this variable when it locks a range that is not contained within $[0,N)$. We will detail the usage of the maximizer in §3.9.

3.4 Formats of Lock Tree Nodes

All nodes in the lock tree are 8-byte variables accessible by all kinds of RDMA one-sided verbs. Internal nodes and leaf nodes have different formats and are manipulated by different RDMA atomic verbs, as shown in Figure 3.

Leaf nodes. Leaf nodes are 8-byte bitmaps in which each bit is associated with a unit of the shared storage. A set bit in the bitmap indicates the corresponding unit of the resource occupied by some client, and vice versa. Clients use RDMA masked-CAS to set and clear each of the 64 bits.

Internal nodes. Each internal node divides into six fields. `Exp` and `Occ` are flags, and the remaining four are counters. Clients use RDMA masked-FAA to modify these fields.

`{TCnt,TMax}` and `{DCnt,DMax}` are two counter pairs that follow the idea of Lamport's bakery algorithm [38, 79]. Specifically, in each counter pair, `Max` is the next available

Algorithm 1 Acquire range locks from CITRON

```
1: procedure AcquireRangeLock(l, r)
2:     A ← [l, r) ∩ [0, N)
3:     if [l, r) ∩ [N, ∞) ≠ ∅ then                ▷ Out-of-bound
4:         Acquire the spillover mutex
5:     if A ≠ ∅ then                              ▷ In-bound
6:         AcquireLockOnTree(A.left, A.right)
```

"ticket number," and `Cnt` is the ticket number that is currently holding the lock. A client gets a ticket by performing an FAA on the `Max` field, polls the `Cnt` field until it matches the ticket, enters the critical section, and finally performs an FAA on the `Cnt` field when the client finishes. The two counter pairs are for different purposes: `{TCnt,TMax}` counts lock requests at "this node," while `{DCnt,DMax}` counts those at descendants.

As for the flags, `Exp` (stands for *expanded*) notifies clients of a lock tree scale-up event. `Occ` (stands for *occupied*) blocks conflicting lock requests at descendants if it is set. A node with the `Occ` flag set will be called an occupied node.

Like prior studies [79], the bit widths of counters impose a hard limit on the maximum concurrency of the system. There may not be more than $2^{15} - 1 = 32767$ clients accessing the same CITRON instance concurrently; otherwise, the overflowing counters can put CITRON into an erroneous state. Nevertheless, this restriction is tolerable in most scenarios.

3.5 Lock Acquisition

Algorithm 1 shows how a client acquires a lock on a range $[l, r)$. Since mutexes are already well-studied, here, we omit the details about the spillover mutex and focus on the lock tree. Without loss of generality, we now assume $[l, r)$ is fully contained within $[0, N)$. Algorithm 2 shows the whole lock acquisition procedure, which consists of two steps:

1. split the range properly into sub-ranges, such that each of which corresponds to a single tree node;
2. acquire locks on each sub-range in ascending order.

For each sub-range and the corresponding node on the lock tree (denoted as *node* hereinafter), the second step further decomposites into four phases:

2(a). lock *node* if it is internal;
2(b). wait until all locks at *node*'s ancestors are released;
2(c). lock *node* if it is a leaf, otherwise occupy it;
2(d). notify *node*'s ancestors and wait for its descendants.

Below, we elaborate on each of the two steps and the four phases of the second step. For convenience and readability,

- we call our protagonist "Alice": she is a client trying to acquire a range lock, and we describe what she will do;
- we use the adjectives *low* and *high* to describe tree nodes that are far from and close to the root;
- we describe masked-FAA with variadic arguments (a pair per field to FAA) instead of bitmasks (Line 3).

Algorithm 2 Acquire a range lock from the lock tree

```
1: ▷ Function signatures of RDMA masked atomic verbs ◁
2: def MaskedCAS(addr, cmp, cmpMask, swap, swapMask) → boolean
3: def MaskedFAA(addr, field_1, add_1, [field_2, add_2, [...]]) → uint64

4: procedure AcquireLockOnTree(l, r, k = 2, m = 4)        ▷ Step 1
5:     nodes ← SolveKnapsack(l, r, k)
6:     for all node ∈ nodes in ascending order do
7:         repeat ret ← LockNode(node, l, r, m) until ret = Acquired

8: procedure LockNode(node, l, r, m)                      ▷ Step 2
9:     if node is internal then                           ▷ Phase (a)
10:        ticket ← MaskedFAA(node, TMax, 1)
11:        repeat val ← RdmaRead(node) until val.TCnt = ticket.TMax
12:    cleared ← node                                     ▷ Phase (b)
13:    while cleared ≠ root do
14:        {anc} ← RdmaRead(all ancestors of cleared)
15:        if root.Exp = 1 then return Aborted
16:        next ← the lowest node in {anc} with Occ ≠ 0
17:        if next = nil then break
18:        repeat val ← RdmaRead(next) until val.Occ = 0
19:        cleared ← next
20:    if node is a leaf then                             ▷ Phase (c)
21:        mask ← bitmask of [l, r) ∩ node.range
22:        if not MaskedCAS(node, 0, mask, mask, mask) then
23:            if MaskedCAS kept failing for too long then
24:                return LockNode(Parent(node), l, r, m)
25:            else
26:                goto Line 12
27:    else
28:        MaskedFAA(node, Occ, 1)
29:    t_0 ← current time                                 ▷ Phase (d)
30:    {anc_notify} ← every m-th ancestor of node
31:    MaskedFAA({anc_notify}, DMax, 1), RdmaRead(root)
32:    if possible time limit excess then return Aborted
33:    if {anc_notify}.highest.Exp = root.Exp = 1 then return Aborted
34:    if node is internal then wait until t_0 + T_wait
35:    for desc ∈ {node and its internal descendants within m levels} do
36:        repeat val ← RdmaRead(desc) until val.DCnt = val.DMax
37:    return Acquired
```

3.5.1 Step 1: Split the range

In this step, Alice decides which node(s) to lock. This step incurs zero network traffic because Alice knows the structure of the lock tree in advance and can do all computations locally.

With a segment tree, any continuous range can be expressed as an aggregate of $O(\log N)$ tree nodes [4]. As a result, Alice has to lock $\Theta(\log N)$ nodes to precisely lock the range $[l, r)$ in the worst case. However, this can result in high latencies since the nodes must be locked sequentially to prevent deadlocks.

A strawman solution is to simply lock the lowest node whose represented range completely covers $[l, r)$. However, this can result in severe false conflicts. For example, imagine that Alice wishes to lock $[N/2 - 1, N/2 + 1)$: the lowest node that covers this small range would be the root, which unfortunately conflicts with all other lock requests.

CITRON strikes a balance by allowing Alice to lock up to k nodes that cover the requested range together. To reduce false lock conflicts, CITRON tries to minimize the covered but unrequested range. This optimization goal can be formulated into a tree knapsack problem [37] and solved by existing algorithms. Our knapsack algorithm has a time complexity of $O(k^2 \log N)$,

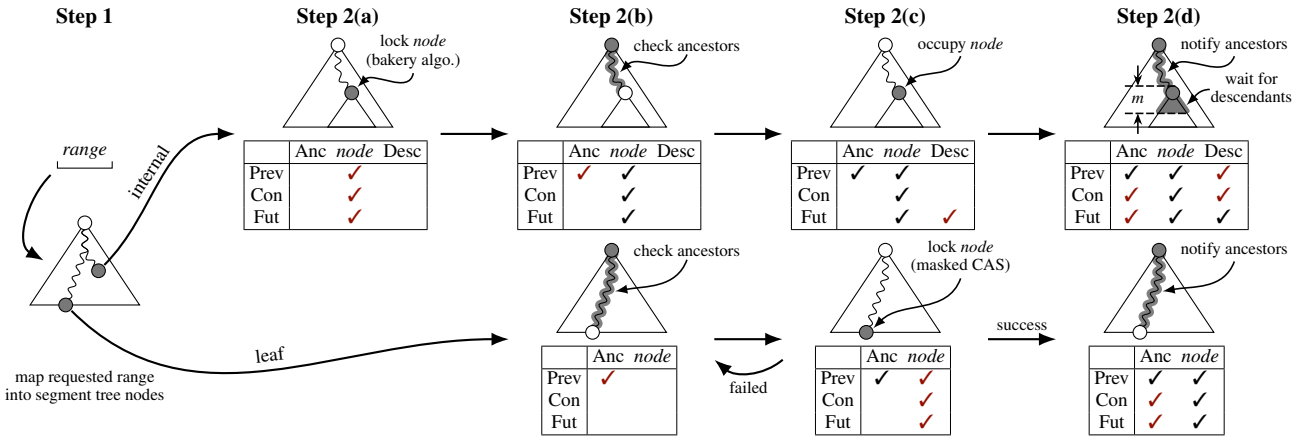

Figure 4: Demonstration of the lock acquisition workflow and the lock conflicts resolved by each phase. *Table rows are the time dimension (Prev = Previous, Con = Concurrent, Fut = Future), and table columns are the space dimension (Anc = Ancestors, Desc = Descendants). Lock conflicts occurring at any time and any position will all be resolved.*

which usually finishes within 0.5 µs when properly optimized and hardly harms lock acquisition performances. Increasing k trades latencies for fewer false conflicts and vice versa. In our implementation, we fix k to 2 for low latency purposes.

Procedure AcquireLockOnTree in Algorithm 2 shows how this step works. Alice first runs a knapsack algorithm to find the optimal combination of the nodes to lock (Line 5). Then, she locks the nodes sequentially (Lines 6-7).

3.5.2 Step 2(a): Lock an internal *node*

In this phase, Alice acquires the lock at *node* if it is an internal node. The workflow follows Lamport's bakery algorithm [79]. Specifically, Alice first increments the `TMax` field of *node* to get a "ticket." Then, she polls the `TCnt` field until it matches the `TMax` field of the ticket (Lines 10-11).

Alice does not lock *node* if it is a leaf. Instead, she defers locking *node* to Step 2(c) to facilitate failure recovery (§3.10). Were she to lock *node* here, Citron would be unable to recover to a normal state if Alice crashed before releasing her lock.

3.5.3 Step 2(b): Wait for *node*'s ancestors

From this phase, Citron starts to resolve conflicts between different nodes. The major principle is that among multiple concurrent lock requests, Citron prioritizes the smallest range because it is usually also the most latency-sensitive one.

As we have discussed before, all ancestors of *node* conflict with it. If there is a held lock at one of *node*'s ancestors, Alice must wait until it is released. Furthermore, there can be higher occupied ancestors of *node*, which belong to lock requests that arrive earlier than Alice's but are still waiting because Citron prioritizes smaller ranges. To ensure fairness, Alice should also wait for these lock requests to complete.

This phase consists of multiple iterations. In each iteration, Alice first reads the ancestors of *node* from the lowest one possibly occupied (Line 14) and checks their `Occ` flags to see if occupied nodes exist. If there are any, the lowest one is selected (Line 16). Alice waits until the lock at the selected node gets released (Line 18), which ends her current iteration. In the next iteration, Alice only needs to check the ancestors of the previously selected node (Line 19). She repeats this process until *node*'s all occupied ancestors are released.

Note that Alice cannot read the ancestors of *node* only once because other clients might issue new range lock requests to nodes higher than any existing lock. Due to unlucky timing, these clients can occupy the nodes they are locking without being aware of the lock requests below. The lock protocol ensures that these clients will get notified of all lock conflicts in Step 2(d) (§3.5.5), so there are no correctness concerns. However, Alice must repeatedly check *node*'s ancestors to detect these possible new lock requests.

3.5.4 Step 2(c): Occupy or lock *node*

Alice can ensure no held locks at *node*'s ancestors now. The remaining lock conflicts can only locate at *node*'s descendants for an internal *node*, or *node* itself for a leaf *node*.

If *node* is internal, Alice needs to set its `Occ` flag with an RDMA masked-FAA. The reason is that Alice should wait for lock requests at *node*'s descendants (because they are more prioritized than Alice's), but she must not wait indefinitely. By setting *node*'s `Occ` flag, newly arriving lock requests at *node*'s descendants will detect and wait for Alice in their Step 2(b), which ensures finite wait time for Alice.

If *node* is a leaf, Alice needs to post an RDMA masked-CAS verb to lock the corresponding bits of *node*. On success, Alice finishes this phase. On failure, she must return to the beginning of Step 2(b) because other clients could have set the `Occ` flags of *node*'s ancestors, as discussed above.

Starvation avoidance. When *node* is a leaf, a series of failed masked-CASs might cause lock starvation. Citron offers a workaround: if Alice keeps getting masked-CAS failures for a certain period, instead of returning to Step 2(b), she can

optionally set *node* to its parent, restart the lock acquisition procedure, and switch to the starvation-free Lamport's bakery algorithm since *node* is now internal (Line 24).

3.5.5 Step 2(d): Notify ancestors, wait for descendants

Let *desc* be an arbitrary descendant of *node*. Assume Bob is another client that is trying to acquire a range lock at *desc* concurrently with Alice. Although Bob conflicts with Alice, they are both unaware of each other's existence. A mechanism is therefore needed to allow Bob to notify Alice of a lock conflict and also to allow Alice to detect this conflict.

There are two strawman solutions. In the first solution, Bob is responsible for notifying all ancestors of *desc*. However, this can result in high latencies for small range lock requests: the lower *desc* is, the more ancestors it needs to notify. In the second solution, Alice is responsible for checking all descendants of *node* for possible conflicts. However, this can result in excessive network traffic since the number of *node*'s descendants increases exponentially as *node* becomes higher.

Inspired by *meet-in-the-middle* (MITM), a common idea in computer science, we employ a combination of the solutions above to synchronize Alice and Bob. Specifically, Citron maintains a globally consistent parameter: m, the MITM distance. Starting from *desc*'s parent node, Bob notifies *desc*'s every m-th ancestor (Lines 30-31). Alice, on the other hand, checks all *node*'s descendants within m layers on the lock tree, as well as *node* itself (Lines 35-36). This solution ensures that no matter where *node* and *desc* locate, Alice will check a node that Bob notifies and thus detect the lock conflict.

For Bob, this solution reduces his notification overheads by a factor of m, which is efficient enough even with a small m. For Alice, she needs to read and check $(4^m - 1)/3$ nodes. Recall that the nodes of the lock tree are placed in the memory by level order and will only form m continuous blocks in the memory layout. Therefore, Alice only needs to post m RDMA reads, which is an acceptable cost. In our implementation, we set $m = 4$. Also, to avoid contention of RDMA atomic verbs, Citron does not notify nodes in the top $m-1$ levels of the lock tree except for *node*'s parent. Instead, Citron replaces them with nodes in the m-th level.

We still need to ensure that Bob notifies *desc*'s ancestors before Alice checks one of them. To this end, Alice waits for a period of time T_{wait} before she checks *node*'s descendants (Lines 29 & 34). Bob ensures that he finishes notifying *desc*'s ancestors before Alice stops waiting; otherwise, he aborts his lock request (Line 32). The foundations of this solution are (1) the well-behavedness of the clients' clocks and (2) the fact that the server-side RNIC executes inbound writes and atomics as if in a global order (i.e., linearizability).

For convenience, we assume an imagined global wall clock in the following discussion. Suppose Alice waits in the time interval $[t_0, t_0 + T_{\text{wait}})$, where t_0 is a global time point. Therefore, the deadline for Bob's notification is $t_0 + T_{\text{wait}}$.

Recall that Bob performs RDMA reads to ancestors of *desc* in Step 2(b) to check if there are any occupied nodes. From this phase, Bob can find a time point t_1 at which Alice has not started waiting. Specifically, if any RDMA reads find an occupied ancestor of *node*, t_1 is the time when Bob posts the last of those. Otherwise, t_1 is the post time of the last RDMA read in Step 2(b). Since Alice only starts waiting after she sets *node*'s `0cc` flag in Step 2(c), $t_1 < t_0$ must hold because of RDMA's linearizability. Say Bob finishes notifying *desc*'s ancestors at time t_2. Bob verifies that

$$t_2 - t_1 \le (1 - \delta) \cdot T_{\text{wait}} \tag{1}$$

where δ is the bound of clock drift. Since Bob does not know where *node* locates at, he needs to record a t_1 and verify the equation above for every ancestor of *desc*.

Despite the non-existence of an imagined global wall clock, Bob can use his local clock to compute $t_2 - t_1$ because it is well-behaved. We use the number from Sundial [43] and set $\delta = 10^{-4}$. Bob will abort his lock acquisition process if Inequation (1) does not hold. The process of aborting a lock request is the same as releasing the lock, and we will detail the procedure in §3.6.

To decide T_{wait}, we count the maximum number of RDMA roundtrips from t_1 to t_2 and reserve a unit of time for each roundtrip. On our testbed, the RDMA RTT is around 2 µs; conservatively, we reserve 5 µs for each roundtrip. Therefore, $T_{\text{wait}} = 15$ µs: Alice waits for up to two RDMA reads in Step 2(b) and a batch of RDMA masked-FAAs in Step 2(c)+(d).

A complete Step 2(d). In the discussions above, we make Bob notify Alice and make Alice wait for Bob. However, we can imagine swapping the roles of Alice and Bob to see that they are actually symmetric clients and need to do what each other does. In other words, Alice needs to both notify *node*'s ancestors and wait for notification from *node*'s descendants.

Tuning the parameters. Step 2(d) relies on properly selected m and T_{wait} to perform well. Increasing either parameter will trade performance of large range lock requests for small ones, and vice versa. Therefore, clients can profile the performance of lock requests (e.g., throughput and lock abort rate, §4.7) and send the profiled data to the cluster manager (CM), enabling the CM to make tradeoffs and tune the parameters.

The CM can employ a two-phase commit (2PC) protocol to adjust the parameters. Suppose we wish to change (m, T_{wait}) from $(m_{\text{old}}, T_{\text{old}})$ to $(m_{\text{new}}, T_{\text{new}})$. The CM first broadcasts $(m_{\text{new}}, T_{\text{new}})$ to all clients. Upon receiving the parameters, a client acknowledges the CM and, in lock acquisition, uses

- $\min\{T_{\text{old}}, T_{\text{new}}\}$ to determine whether it should abort,
- $\max\{T_{\text{old}}, T_{\text{new}}\}$ when waiting for *node*'s descendants,
- $\min\{m_{\text{old}}, m_{\text{new}}\}$ when notifying *node*'s ancestors, and
- $\max\{m_{\text{old}}, m_{\text{new}}\}$ when checking *node*'s descendants.

After confirming that all clients have already received the new parameters, the CM sends "commit" messages to make clients switch entirely to $m = m_{\text{new}}$ and $T_{\text{wait}} = T_{\text{new}}$.

Algorithm 3 Release a range lock back to the lock tree

```
1:  procedure ReleaseLockOnTree(l, r)
2:      for all node ∈ locked nodes do
3:          if node is a leaf then
4:              mask ← corresponding bitmask of [l, r)
5:              MaskedCAS(node, mask, mask, 0, mask)
6:          else
7:              MaskedFAA(node, Occ, −1, TCnt, 1)
8:          for all anc ∈ notified ancestors of node do
9:              val ← MaskedFAA(anc, DCnt, 1)
10:             if anc is the highest notified node and val.Exp ≠ 0 then
11:                 Fetch and update the new tree configuration if needed
12:                 Continue the for all loop for new ancestors of anc
```

3.6 Lock Release

Algorithm 3 shows the lock release procedure. If *node* is a leaf, Alice unlocks it with masked-CAS (Lines 3-5); otherwise, she vacates *node* by adding the `TCnt` counter and clearing the `Occ` flag with masked-FAA (Line 7). Also, for all *node*'s ancestors that have been notified during lock acquisition, Alice adds their `DCnt` counters (Lines 8-9). All these RDMA verbs can be batched together to reduce latency.

3.7 Proof Sketch of Correctness

Range locks in Citron consist of nodes on the lock tree and possibly a spillover mutex, all of which are acquired separately and sequentially. The correctness of the spillover mutex is already proven [79]. Therefore, we only need to prove the correctness of a lock on a single tree node. Alice is still our protagonist in the proof sketch.

Safety. Safety means that Citron does not simultaneously grant locks to Bob – a conflicting client – and Alice. As shown in Figure 4, no matter when Bob arrives and where he locates on the lock tree, Citron will resolve the lock conflict between Alice and him. Specifically, Steps 2(a)+(c) ensures that no conflicting clients exist at *node*: 2(a) for an internal *node*, and 2(c) for a leaf *node*. Steps 2(b) and 2(c)+(d) ensure respectively that no held locks exist at *node*'s ancestors and descendants. Step 2(d) further ensures that:

1. if Bob is at a descendant of *node*, Alice will wait until he releases his lock or aborts;
2. if Bob is at an ancestor of *node*, he will wait until Alice releases her lock or aborts.

Therefore, when Alice enters the critical section, no conflicting held locks may exist. □

Liveness. Liveness means that without infinitely long critical sections, Alice's lock acquisition procedure will always take some finite amount of time to return (the result could be Aborted, though). Specifically, Step 1 is finite. Step 2(a) employs Lamport's bakery algorithm, which is starvation-free. Step 2(c) is obviously finite for an internal *node*, and is also finite for a leaf *node*, thanks to the starvation avoidance mechanism. For Steps 2(b) and 2(d), Alice can only wait for a finite number of clients in each phase. Because these clients will eventually abort or release their locks, these phases are also finite. To sum up, lock acquisition takes a finite time. □

3.8 Fast Path Optimization

Several optimizations apply to the lock acquisition path when Citron is not under severe contention.

First, RDMA ensures that it will not reorder any one-sided verb before previous writes and atomics in the same QP [66]. Thanks to this ordering guarantee, Alice can batch the RDMA verbs in the lock acquisition path together. In Step 2(b), all reads in an iteration can be batched (Line 14). In Step 2(d), the notification to *node*'s ancestors and the read to the root can be batched (Line 31). Further, Steps 2(a) and 2(b) can be optimistically batched in the hope that Step 2(a) immediately succeeds. More aggressively, Steps 2(c) and 2(d) can also be batched, but Alice needs to roll back the notification to *node*'s ancestors in Step 2(d) (by adding `DCnt`) if *node* is a leaf and the masked-CAS verb in Step 2(c) fails.

Second, if *node*'s children are all leaf nodes, Alice can explicitly lock all *node*'s descendants to skip the wait time in Step 2(d). Specifically, she post masked-CAS verbs to all its children to set all 256 bits from 0 to 1. If all these masked-CAS verbs succeed, she can skip the wait and directly enter her critical section. Otherwise, she needs to fall back to the regular lock acquisition path and also clear the bits of modified children nodes. The RDMA masked-CASs can be batched with the atomic verbs in Steps 2(c) and 2(d).

With these optimizations, optimistically, acquiring a lock takes only two RDMA roundtrips, ensuring low latencies.

3.9 Scaling the Lock Tree

In real-world scenarios, a shared storage resource can be of a dynamic size (e.g., append-only log). If the storage size grows, Citron's lock tree might be unable to cover the lock requests, which can cause performance degradation. On the other hand, if the storage size shrinks, maintaining unused nodes in the lock tree will result in extra memory consumption. Therefore, it is necessary for Citron to react to storage size changes.

3.9.1 Scale up

The size of a storage resource can grow upon writes. When this happens, out-of-bound lock requests not contained within $[0, N)$ will contend for the spillover mutex. Overprovisioning the lock tree will inflate Citron's DRAM footprint, of which a considerable percentage is wasted, while a stop-the-world synchronized scaling up mechanism will cause significant synchronization overheads. To solve this problem, leveraging the structural self-similarity of segment trees, i.e., small trees can be viewed as subtrees of large ones, Citron provides an option to scale up the lock tree at runtime.

Citron uses masked-CAS to decide how large the lock tree should scale up to. Note that *masked-CAS offers bitwise-OR semantics when the compare mask is zero*. For this reason, Citron contains a maximizer: the clients can OR the right

Algorithm 4 Scale up the lock tree

```
1:  procedure ScaleUpLockTree
2:      Acquire the spillover mutex
3:      Send an RPC to server to allocate free and zeroed memory
4:      for all node ∈ top m levels of the old lock tree, by index order do
5:          v ← MaskedFAA(node, Exp, 1)
6:          for i = m, 2m, 3m, ... do
7:              anc ← i-th ancestor of node in the new lock tree
8:              MaskedFAA(anc, DCnt, v.DCnt, DMax, v.DMax + v.Occ)
9:      Update the metadata service to renew the lock tree configuration
10:     RdmaWrite(maximizer, 0)
11:     Release the spillover mutex
```

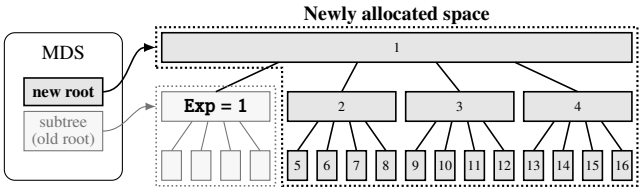

Figure 5: Demonstration of a lock tree scale-up process.

boundaries of out-of-bound lock requests to the maximizer with one-sided masked-CAS, enabling the detection of such lock requests. The maximizer's value is at most 2× of the actual maximum, which is accurate enough because the minimum scale-up factor of the lock tree is 4×.

If Alice is willing to scale up the lock tree, she can read the maximizer via RDMA and perform the scale-up if she finds a non-zero value. Algorithm 4 shows the procedure of scaling up. Alice first acquires the spillover mutex (Line 2) to both ensure lock safety and prevent simultaneous scale-up attempts. Then, Alice sends an RPC to the server to allocate the expanded part of the segment tree (Line 3). The original lock tree is not moved and will form a new enlarged tree with the newly allocated nodes, as shown in Figure 5. Alice then sets the `Exp` bits of all nodes in the top m levels of the old lock tree to notify other clients of the scale-up event (Lines 4-5). She also needs to propagate these nodes' `DCnt` and `DMax` counters to their new ancestors (Lines 7-8). Note that an occupied node accounts for an extra unit of `DMax`. Finally, Alice updates the metadata service with a renewed configuration containing addresses of both original and expanded parts of the lock tree, clears the maximizer, and releases the spillover mutex to finish scaling up the lock tree (Lines 9-11).

With off-the-shelf RNICs that do not support one-sided memory allocation, we must rely on the server-side CPUs to allocate memory and register it to the RNIC. However, this is a lightweight task compared with CPU-based lock management and can hardly cause any server-side CPU bottleneck.

Handling scale-ups in lock acquisition. When acquiring a lock (Algorithm 2), Bob, another client, must take into consideration that Alice can concurrently scale up the lock tree. Specifically, in Step 2(b), Bob checks the `Exp` flag whenever he reads the root (Line 15): a set `Exp` indicates a concurrent scale-up. In Step 2(d), Bob needs to read the root after notifying *node*'s ancestors. If the `Exp` flags of the highest notified ancestor of *node* and the root are both set (Line 33), there must be a concurrent scale-up. Bob handles concurrent scale-ups trivially: he aborts and retries.

Handling scale-ups in lock release. The lock tree can also be scaled up after Bob acquires a lock and before he releases it. Alice will notify the new ancestors of *node* on behalf of Bob. Therefore, Bob should also add the `DCnt` counters of the new ancestors of *node* (Algorithm 3, Lines 10-12).

Node index arithmetics. For a node x in the enlarged part of the lock tree, by simply offsetting the number of nodes in the old lock tree that lie ahead of x in the level order, the node index arithmetics rules described in §3.3 still hold.

Impact to Step 2(d) of lock acquisition. Scaling up the lock tree can break the continuous memory layout of the tree nodes in the memory, which can affect Step 2(d) of the lock acquisition path. To detect Bob, Alice originally only needs to post m RDMA reads, but with a scaled-up lock tree she will possibly need to post more. However, even in the worst case, the number of RDMA reads is only $m(m+1)/2$, which is still acceptable when m is small (e.g., for our $m = 4$ setting, the number of RDMA reads is 10) because all these reads can be parallelized. The extra reads can also be reduced by setting a larger minimum scale-up factor (e.g., 16×).

3.9.2 Scale down

Different to scaling up, scaling down cannot be triggered by writes and is in most cases intrinsically a blocking operation. For example, in file systems, calling `ftruncate` to shrink a file will take its inode mutex and block all other I/O attempts. During a blocking scale-down operation, Citron can safely shrink its lock tree by removing all nodes except a subtree.

3.10 Handling Client Failures

To enable recovery, all clients must agree with a lease time T_{lease} and that a range lock must be released within T_{lease}.

Detection. Citron relies on the cluster manager (CM) to detect client failures. The CM notifies the lock server to destroy the RDMA QPs that were connected with the failed clients.

Recovery. Citron recovers lazily. Alice detects a failure if she spins at a place for longer than T_{lease} during lock acquisition, including Lines 11, 18, and 36 in Algorithm 2. Also, Alice suspects a failure if she fails too many times at Line 22.

Line 11. Alice detects a failure when `TCnt` and `Occ` are both unchanged for T_{lease}. Shen then waits for up to $(\Delta - 1) \cdot T_{\text{lease}}$, where Δ is the gap between *node*'s `TCnt` and her ticket's `TMax`. If `TCnt` and `Occ` remain unchanged, Alice sets *node*'s `TCnt` field to her ticket's `TMax` and clears *node*'s `Occ` flag to recover.

Line 18. Alice detects a failure when `TCnt` of an ancestor of *node* is unchanged for T_{lease}. She aborts the current lock request and tries to lock that ancestor instead, reducing the problem to the situation of Line 11, which we have already discussed above.

Codename	Type	Lock Management Scheme	Description
MT	I	Maple tree [21]	A modern data structure dedicated to efficiently managing disjoint ranges, ported from Oracle Linux UEK.
IT	I	Interval tree [47]	A representative implementation of interval tree ported from Lustre, in which it is used to manage range locks upon file I/O requests.
LLC	I	Lock-free linked list [36]	A range lock manager that chains lock entries in a lock-free linked list.
LLD	II		Same as above, but all the CPU atomic instructions are replaced with RDMA one-sided atomic verbs to make the lock manager decentralized.
SS	II	Static segmentation [35]	The whole range is divided into fixed-size segments, each associated with a DSLR [79] instance, a state-of-the-art RDMA-based decentralized mutex.

Table 1: Baseline systems used in evaluation.

Line 36. Alice detects a failure when DCnt is unchanged for $H \cdot T_{\text{lease}}$, where H is *node*'s height in the lock tree. Since $H \geq 1$, Alice is sure that no clients are holding locks at *node*'s descendants and can set *node*'s DCnt to its DMax to recover.

Line 22. Alice cannot distinguish between lock starvation and client failures when she repeatedly fails to lock *node* with masked-CAS. However, she can acquire a lock at *node*'s parent and then check if *node* is zero. If not, Alice detects a failure and zeroes *node* with an RDMA write to recover.

The recovery time is dominated by the user-defined lease time T_{lease}. Aside from waiting for lease expiration, Alice only needs one RDMA operation to perform the recovery. In practice, T_{lease} is usually set to several milliseconds (e.g., 10 ms in [79]); with larger ranges, T_{lease} can also be longer.

4 Evaluation

In this section, we use a number of benchmarks to evaluate Citron, seeking to answer the following questions:

- How does Citron compare against existing lock managers? (§4.2, §4.3)
- What are the performance effects of the fast path? (§4.4)
- How well does Citron scale up itself? (§4.5)
- How does splitting ranges reduce false conflicts? (§4.6)
- What is the lock abort rate of Citron? (§4.7)

4.1 Experiment Setup

Our testbed consists of 4 machines, one acting as the lock server and the other as clients. Each machine is equipped with two Intel® Xeon® Gold 5220 CPUs running at 2.20 GHz, 256 GB DDR4-2666 DRAM, and a Mellanox ConnectX-6 RNIC via PCIe 3.0 ×16 interface. All machines run Ubuntu 18.04 with Linux kernel version 4.15.0 and are connected by a Mellanox QM8790 InfiniBand switch.

Lock tree configuration. Except in §4.5 (in which we need to scale up the lock tree), the lock tree is always initialized with $N = 2^{28}$. This is to simulate a large-scale scenario where 1 TB space is divided into 4 KB pages and managed by Citron. As a result, the lock tree contains 5.6 million nodes and the Citron instance takes up 42.7 MB of memory, which is only about 0.004% of the total storage amount.

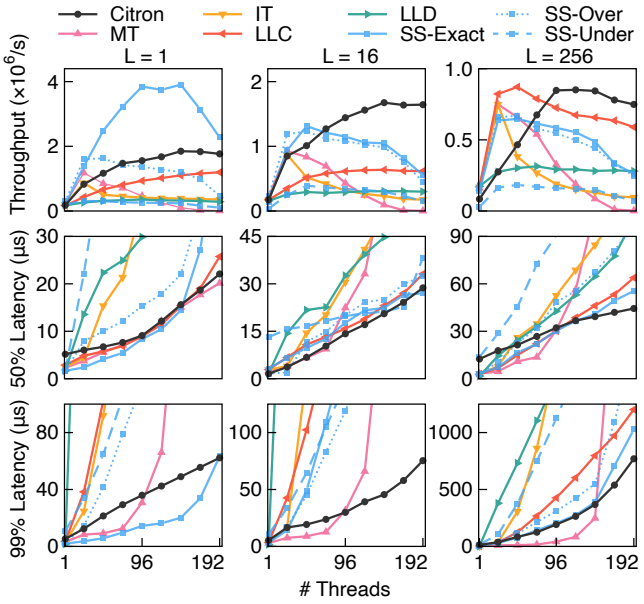

Figure 6: Throughputs and latencies of Citron and baseline systems with different range lock sizes.

Baseline systems. All baseline systems are shown in Table 1, which can be classified into the following two types.

I. Server-side CPUs are fully responsible for acquiring and releasing locks, and they accept clients' requests using eRPC [30], a state-of-the-art RDMA RPC engine.

II. Clients leverage one-sided RDMA to acquire and release range locks and server-side CPUs are idle.

The number of threads. The server machine runs 18 RPC server threads when evaluating baselines of type I. For clients, we enable hyperthreading and run up to 64 worker threads in each client machine, each thread on a separate logical core. As a result, the maximum number of clients is $3 \times 64 = 192$.

4.2 Microbenchmarks

In this experiment, we set the range sizes to $L = 1$, $L = 16$, and $L = 256$ respectively. For static segmentation (SS), we consider three different situations in terms of the segment size: (1)

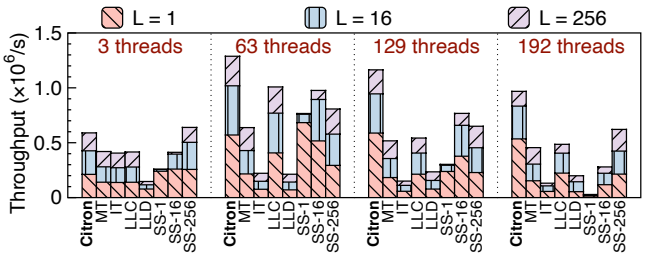

Figure 7: Throughputs by range size of Citron and baseline systems under a mixed-size workload.

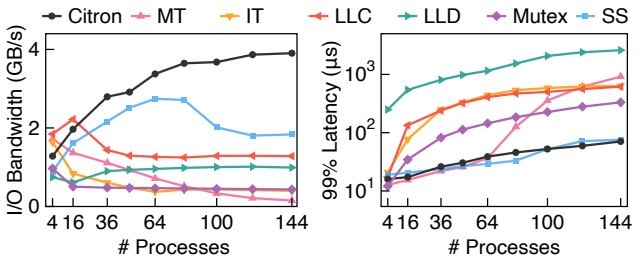

Figure 8: Throughputs and latencies of Citron and baseline systems under the BT-IO workload. *Latency is log scale.*

exactly the range size L (`SS-Exact`), (2) overestimated to $8L$ (`SS-Over`), and (3) underestimated to $L/8$ (`SS-Under`). The left borders of the requested ranges are subject to a Zipfian-0.9 distribution on $[0, N-L]$. Figure 6 shows the results.

In terms of median latency, in almost all cases, Citron performs comparably to the best of the baselines, namely `MT`, `LLC`, and `SS-Exact`. This matches our expectation because Citron needs a similar number of RDMA roundtrips to these baselines to acquire a range lock. We focus on the lock manager's tail latencies below.

When $L = 1$, range locks are equivalent to mutexes. As expected, Citron underperforms `SS-Exact`. It delivers 44.6% lower throughput (i.e., locks granted per second) and 1.83× higher p99 latency on average. However, this gap is because the access granularity is static and correctly known in advance. If this requirement is not met, the performance of `SS` will drop dramatically: `SS-Over` and `SS-Under` deliver peak throughputs of only 83.2% and 16.6% compared to that of Citron, and they suffer from 3.77× and 4.68× higher p99 latencies, respectively on average. The results demonstrate that the static segmentation mechanism is unfit for dynamic workloads whose I/O granularities vary.

When L is 16 or 256, because of unaligned ranges, static segmentation causes severe false lock conflicts and degrades performance. Citron delivers 28.7% and 38.6% higher peak throughputs and significantly lower tail latencies than `SS`.

Type I baseline systems that rely heavily on server-side CPUs are all bottlenecked by CPUs under high contention. Citron avoids such bottleneck and has 1.56× and 1.76× peak throughputs than these baselines for $L = 1$ and $L = 16$. When $L = 256$, Citron shows similar peak throughput to `LLC` but in average 24.6% lower tail latencies. Under low contention, due to the efficient eRPC engine and low CPU burdens, the queueing latencies are lowered to a sub-microsecond level and the baselines can show tail latency advantages to Citron. Unfortunately, such advantages vanish quickly as the number of clients increases.

`LLC` performs significantly better than `LLD` because their lock management scheme, i.e., the lock-free linked list, is CPU-friendly but RDMA-unfriendly. `LLC` performs pointer chasing which has very limited overheads on the CPU. However, with RDMA, each step of pointer chasing takes one RDMA roundtrip, leading to high latencies and low performance. This demonstrates the unfeasibility of simply porting existing range lock managers to one-sided RDMA.

We also test a mixed workload where each of the range sizes described above accounts for one-third of the client threads. For `SS`, we test three granularities: 1, 16, and 256. Figure 7 shows the results. Citron delivers higher throughputs than baselines under high contention: it outperforms the best baseline by 27.7%, 51.7%, and 55.9% with 63, 129, and 192 client threads, respectively. Also, Citron grants higher throughput to small range locks without starving large ones.

In summary, Citron delivers the overall best performance for different range sizes under high contention. However, Citron can be suboptimal for mutex-only workloads.

4.3 Application Benchmarks

We build a distributed in-DRAM file system CitronFS to evaluate Citron and baseline lock managers under realistic workloads. CitronFS follows Octopus's design [46] but stores all data in DRAM. It implements cacheless file I/O that can be protected by either per-file byte-range locks or inode mutexes. The server machine serves file metadata, while the three client machines stores file data.

4.3.1 BT-IO: a non-conflicting I/O workload

This experiment runs the Class D BT-IO [75] workload in the NAS Parallel Benchmarks [55] with different process counts. This application performs non-conflicting interleaved writes and reads to a total of 135.8 GB of data in a single file; different process counts lead to different I/O granularities ranging from 2040 B to 16 320 B. Figure 8 shows the results.

With Citron, the I/O bandwidth of CitronFS reaches a maximum of 3.90 GB/s, which is 3.05× and 2.13× to those with `LLC` and `SS-Exact`, the best CPU-based and one-sided RDMA-based baselines, respectively. Citron outperforms `LLC` and `SS-Exact` by 1.89× and 1.52× on average. Compared with `LLC` and other CPU-based baselines, Citron delivers a 73.4% p99 latency reduction on average.

The underlying reason is that the range locks uniformly span the whole range because BT-IO is a non-conflicting workload. Therefore, CPU-based range lock managers need to maintain larger data structures for more concurrent lock

Lock Scheme	Throughput (kops/s)	Reader 99% Latency (µs)
CITRON	781.4	59.6
MT	220.6	2043.4
IT	503.1	118.7
LLC	847.5	430.9
LLD	144.8	442.0
SS-4KB	575.3	56.1
Mutex	173.8	295.2

Table 2: Throughputs and latencies of CITRON and baseline systems under the Filebench OLTP workload.

entries and perform memory (de)allocations more frequently, aggravating the CPU bottleneck. CITRON, SS-1, and LLD avoid such bottlenecks by using only one-sided RDMA. Compared with LLD, CITRON requires much fewer network roundtrips and has significantly lower latencies. Compared with SS-1, CITRON leverages RDMA masked-CAS, which can obviate false lock conflicts and also minimize the number of locks to acquire, resulting in higher performances.

4.3.2 Filebench OLTP: a conflicting I/O workload

This experiment runs the Filebench [68] OLTP workload modified to run distributedly. This application runs reader and writer threads that operate on a dataset of 10 data files and a log file, all 10 MB sized. Each client runs 1 log file writer, 10 data file writers, and 50 readers. In each client, readers perform random 8 KB reads to data files, while writers perform 100 random 8 KB random writes evenly to all data files per 1000 reads and one 256 KB write to the log file per 3200 reads. We use 4 KB (i.e., page size) as the granularity for static segmentation (SS). Table 2 shows the results.

Overall, CITRONFS delivers the second highest I/O throughput with CITRON, 7.8% lower than that with LLC. However, LLC shows 7.2× p99 latencies compared with CITRON, which demonstrates that CITRON can avoid the CPU bottleneck by eliminating the RDMA CQ queueing latencies.

Another baseline system, SS-4KB, shows similar latencies to CITRON for readers because they share similar numbers of necessary RDMA roundtrips to acquire and release a lock. However, CITRON delivers 35.8% higher throughput for two reasons. First, when using SS-4KB, CITRONFS is bottlenecked by the inefficient log writer that needs to acquire 64 mutexes to perform a write. Second, CITRON is more friendly to the CPU cache because it reduces memory footprint by compressing lock entries into bitmaps, which brings higher performance thanks to Intel's Data Direct I/O technology [24].

4.4 Effects of the Fast Path

We measure the performance of CITRON with and without the fast path optimization (§3.8) to understand its benefits. We use the same fixed-size microbenchmark as in §4.2 and set the range sizes to $L = 16$ and $L = 256$, respectively, to evaluate the fast path for both small and large ranges. The left borders of the requested ranges are subject to a Zipfian-α distribution on

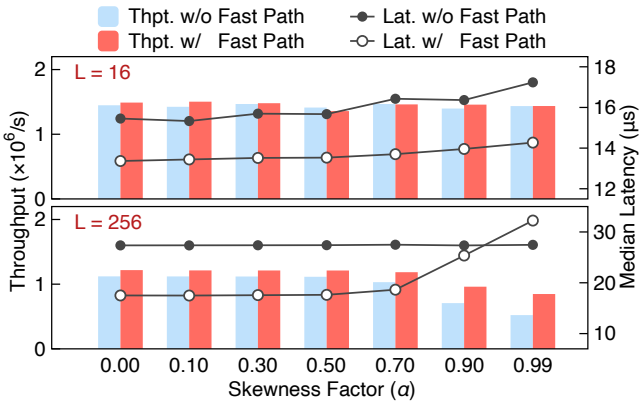

Figure 9: Performance effects of the fast path optimization.

$[0, N-L]$, for which we adjust the skewness factor α from 0 (i.e., uniform) to 0.99 (i.e., highly skewed). We fix the number of threads to 96. Figure 9 shows the results.

For $L = 16$, the fast path does not significantly affect CITRON's throughputs because it simply batches the RDMA verbs in Steps 2(c) and 2(d) in the lock acquisition workflow. However, by batching RDMA verbs together, the fast path saves a network roundtrip from the critical path, reducing the median latency by 2.4 µs (21.5%) on average.

For $L = 256$, the fast path contributes to higher throughput and lower latency. The fast path effectively eliminates the wait time in Step 2(d), which reduces lock acquisition latency and increases the throughput. On average, enabling the fast path improves the throughput by 34.3%. When $\alpha \leq 0.70$, the fast path reduces the median latency by 11.5 µs (39.4%). However, when most lock requests conflict with each other ($\alpha > 0.70$), the fast path lengthens the critical path and wastes RDMA IOPS, resulting in increased median latency (4.8 µs, 17.4% higher than that without the fast path when $\alpha = 0.99$). Note that non-conflicting lock requests still benefit from the fast path, which brings higher throughput.

Also, we observe that the fast path shows no significant impact on the p99 latencies. The reason is straightforward: the tail latencies stem from lock requests that cannot benefit from the fast path. We omit the results due to limited space.

4.5 Performance with Scale-ups

We use a trace collected from the hard-write workload of the IO500 benchmark [25] to evaluate the scale-up process of CITRON. In this workload, 64 I/O threads repeatedly write 47 008 B data to a large shared data file in parallel. Offsets of the writes continue to increase, resulting in a constantly growing file size. The whole trace consists of 12.8 million writes. We only acquire and release range locks without performing writes to avoid shadowing the impacts of scale-up events.

We initialize CITRON with $N = 2^{10}$ (i.e., 4 MB size, has scale-ups) and compare the results with $N = 2^{28}$ (i.e., no need for scale-ups). When $N = 2^{10}$, each client machine runs a

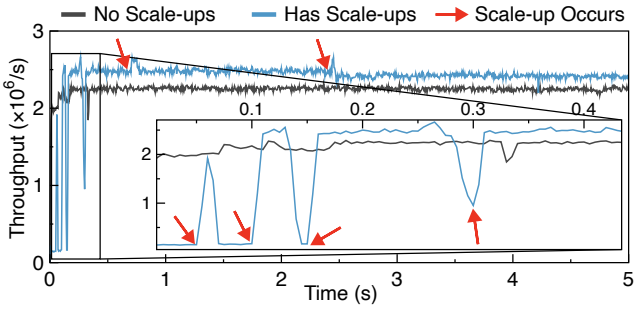

Figure 10: Lock/unlock-only throughput of Citron with and without scale-ups under the IO500 hard-write workload.

background thread that polls the maximizer once per 10 ms and scales up the lock tree whenever necessary. The server runs one eRPC thread to serve lock tree metadata queries and memory allocation requests. Figure 10 shows the results.

The scale-up process takes only tens of microseconds to complete and hardly blocks other lock requests. Hence, upon a scale-up, there will be an immediate increase in the throughput, as shown in Figure 10. In the first 300 ms of the experiment, the whole lock tree is still small despite being scaled up. The write offsets quickly grow beyond its size, causing the clients to contend for the sole spillover mutex and thus a throughput decline of up to 92.6%. After that, however, the throughput decline before the 4th scale-up is only 57.5%. The reason is that the lock tree is already large enough, and client threads are not perfectly synchronized; therefore, only a part of the threads contend for the spillover mutex. The throughput keeps almost stable afterward for similar reasons.

It is worth noting that the stable throughput with scale-ups is slightly higher than that without. Specifically, before and after the 6th scale-up, the throughput advantages are 9.7% and 7.4%, respectively. The reason is that the lock tree only grows larger when necessary, resulting in a smaller tree height, reducing the number of RDMA verbs per lock request, thus bringing higher performance.

4.6 False Conflict Rate

We measure the false conflict rate of Citron to understand the effects of our range splitting mechanism in Step 1 of the lock acquisition path (§3.5.1). Two lock requests constitute a false conflict if they do not overlap but lock conflicting nodes. We measure the false conflict rate with different L (i.e., requested range lock size) and different k (i.e., the maximum number of nodes to lock per request). The left borders of the requested ranges are independently subject to a uniform distribution on $[0, N-L]$. We repeatedly issue two concurrent lock requests and detect whether they conflict with each other logically and actually. The false conflict rate is calculated as the number of false conflicts divided by the number of all lock requests. Figure 11 shows the results.

For all $L \leq 64$, $k = 2$ is sufficient to split the requested range into leaf nodes on the lock tree, eliminating false conflicts because Citron employs RDMA masked-CAS. As a result, Citron can achieve its highest throughput for prevalent small range lock requests in real-world workloads.

Increasing k beyond $k = 2$ brings minor benefits. Compared with $k = 1$, setting $k = 2$ reduces the false conflict rate by two orders of magnitude (to relatively 3.6% on average), whose absolute value is around 10^{-4}, virtually negligible. To further reduce this rate for an order of magnitude, we need $k = 5$, which results in 3 more nodes to lock and more than doubled lock acquisition latencies. Therefore, we trade that marginal throughput improvement for lower latency and adopt $k = 2$ in our implementation of Citron.

4.7 Lock Abort Rate

We measure Citron's lock abort rate to understand the efficacy of the synchronization mechanism in Step 2(d) of the lock acquisition path (§3.5.5). Low abort rates indicate the strong practicability of Citron. Aside from hardware issues such as the RNIC capabilities, the abort rate can be affected by the following three configurable factors:

1. *#Threads*: the thread count (i.e., contention severity),
2. m: the meet-in-the-middle distance in Step 2(d), and
3. T_{wait}: the time to wait in Step 2(d).

Therefore, we conduct three experiments, in each of which we fix two of these parameters, adjust the remaining one, and measure the lock abort rate. We retry for each aborted lock request until it succeeds, so the abort rate also reflects the amount of the retry traffic. We set the fixed parameters to *#Threads* = 96, $m = 4$, and $T_{\text{wait}} = 15\,\mu s$, respectively, as is described in §3. We use the same mixed-size microbenchmark as in §4.2. The abort rate is calculated as the number of lock aborts divided by the number of all lock requests. Figure 12 shows the results.

#Threads. As the number of threads increases from 3 to 192, the RNIC suffers from an increased IOPS pressure and therefore delivers higher latencies, causing the lock abort rate to increase from 10^{-5} level to 10^{-2} level. However, the overall throughput (i.e., successful locks) does not drop with the increase in the abort rate after reaching the maximum. This shows that Citron's lock protocol causes acceptable numbers of lock aborts and retries under both low and high contention.

m. When m is small, clients need to notify many ancestors of *node* in Step 2(d) with RDMA masked-FAA, resulting in low throughput and a high possibility of lock aborting. When m is large, the burden to notify *node*'s ancestors is low, but the networking cost to detect conflicts at *node*'s descendants suffers from exponential growth. We adopt $m = 4$ to balance throughput, abort rate, and network traffic.

T_{wait}. Increasing the wait time in Step 2(d) reduces the chance that lower clients exceed the time limit but makes higher clients wait longer and degrades throughput. We adopt $T_{\text{wait}} =$

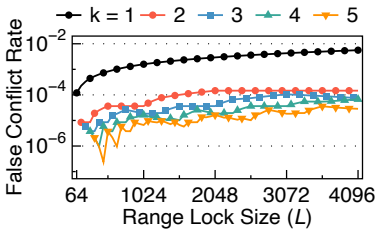

Figure 11: False conflict rate under different k settings. *Log scale.*

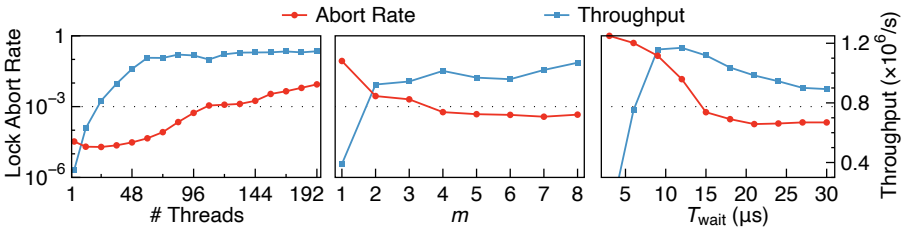

Figure 12: Lock abort rate and throughput of Citron under different workloads and configurations. *Lock abort rate is log scale.*

15 µs to balance between throughput and abort rate. Further increasing T_{wait} contributes little to reducing the abort rate since there are always occasional long-lasting RDMA verbs due to unpredictable hardware-level issues of the RNIC.

To sum up, with our configuration, the lock abort rate of Citron is acceptably low and has minimal negative effects on the overall performance.

5 Related Work

Lock management. Locks have been a major research topic ever since the outset of concurrent programming. A wealth of previous studies aim for efficient locking within a single machine [7, 8, 14, 15, 33, 34, 45, 52].

With the advent of RDMA, studies above have become less valuable in distributed systems as they fall short in alleviating the CPU bottleneck. Such a situation led to the birth of decentralized [10, 54, 72, 79] and hardware-offloaded [28, 80] lock managers. Among them, DrTM [72] uses RDMA CAS to grant writer locks and reader leases. DSLR [79] uses RDMA FAA to implement the starvation-free Lamport's bakery algorithm [38]. NetLock [80] offloads lock management to a programmable switch, achieving both high performance and the benefits of centralized lock management.

While most existing studies focus on mutexes, Citron aims at range locks and supports locking disjoint parts of the same shared storage for finer-grained concurrency.

Range locks. Range locks are widely adopted in key-value stores [18, 61], file systems [3, 9, 35, 47], and memory management systems [21, 36]. These systems usually use carefully designed *dynamic* tree data structures to manage range locks, including the range tree in RocksDB [18], the interval tree in Lustre [47], the red-black tree in BeeGFS [3], and the maple tree in Oracle Linux UEK [21]. Kogan et al. also propose using a lock-free linked list to maintain range locks [36] since the number of cores is limited and the list cannot be too long.

Citron targets distributed range lock management where far more clients exist than within a single machine. Citron avoids the CPU bottleneck by using only one-sided RDMA on the critical paths of range lock operations.

Lock conflict resolution. Allowing more types of communication aside from direct one-sided RDMA between the clients and the server brings different lock conflict resolution means. For example, Sherman [70] proposes a hierarchical lock scheme that maintains a local lock table within each client machine to avoid unnecessary remote retries and enable lock handing-over, which is also applicable to Citron. Thakur et al. proposes maintaining a lock table entry in the lock server for each client, thus enabling a lock holder to read the whole lock table and wake up conflicting clients when it releases the lock [69]. Other prior research [12, 63, 67] also discusses work delegation among clients to eliminate conflicts.

One-sided RDMA systems. In addition to decentralized lock management, existing studies employ one-sided RDMA for various purposes, including file I/O [2, 46, 77, 78], transaction processing [16, 64, 71–73], and memory disaggregation [1, 6, 19, 40, 50]. A recent study, RedN [66], even proves the Turing-completeness of one-sided RDMA and shows its efficacy in RNIC-offloading multiple functionalities.

Citron shares the same goals with most one-sided RDMA systems: eliminating server-side CPU bottlenecks and improving performance. However, Citron is the first to develop an efficient distributed range lock manager with one-sided RDMA and to outperform the state-of-the-art.

6 Conclusion

We present Citron, a distributed range lock manager that relies only on one-sided RDMA to acquire and release locks. Citron employs a lock protocol that operates a segment tree and efficiently coordinates conflicting range lock requests. Citron together offers a fast path optimization and supports dynamic scaling as the size of its managed range changes. Our evaluation shows that Citron significantly outperforms existing distributed range lock managers.

Acknowledgment

We sincerely thank our shepherd Youjip Won 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 Key R&D Program of China (Grant No. 2021YFB0300500), the National Natural Science Foundation of China (Grant No. 61832011 & 62022051), and Huawei (Grant No. YBN2019125112).

References

[1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems 2020*, pages 1–16, Heraklion Greece, April 2020. ACM.

[2] 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)*, page 18. USENIX, November 2020.

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

[4] John Louis Bentley and Derick Wood. An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, July 1980.

[5] 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.

[6] 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, July 2018.

[7] 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.

[8] 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.

[9] Qi Chen, Shaonan Ma, Kang Chen, Teng Ma, Xin Liu, Dexun Chen, Yongwei Wu, and Zuoning Chen. SeqDLM: A Sequencer-based Distributed Lock Manager for Efficient Shared File Access In a Parallel File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '22)*, page 11, Dallas TX USA, November 2022. IEEE/ACM.

[10] Yeounoh Chung and Erfan Zamanian. Using RDMA for Lock Management. *arXiv:1507.03274 [cs]*, July 2015.

[11] Giuseppe Congiu, Sai Narasimhamurthy, Tim Süß, and André Brinkmann. Improving Collective I/O Performance Using Non-volatile Memory Devices. In *2016 IEEE International Conference on Cluster Computing (CLUSTER '16)*, pages 120–129, September 2016.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] etcd Authors. Etcd. https://etcd.io/, 2022.

[18] Facebook Open Source. RocksDB | A persistent key-value store. http://rocksdb.org/.

[19] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient Memory Disaggregation with InfiniSwap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, page 21, March 2017.

[20] A.B. Hastings. Distributed lock management in a transaction processing environment. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems (SRDS '90)*, pages 22–31, October 1990.

[21] Liam Howlett. Introducing the Maple Tree. https://lwn.net/Articles/884840/, February 2022.

[22] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas F. Wenisch. A Top-Down Approach to Achieving Performance Predictability in Database Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 745–758, Chicago Illinois USA, May 2017. ACM.

[23] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC '10)*, page 14, Boston MA USA, June 2010. USENIX.

[24] Intel Corportation. Intel® Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, 2012.

[25] IO500 Foundation. IO500. https://io500.org/pages/running.

[26] 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.

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

[28] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soule, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, page 16, Renton WA USA, April 2018. USENIX.

[29] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock Immunity: Enabling Systems To Defend Against Deadlocks. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, page 14, San Diego, California, USA, December 2008. USENIX.

[30] Anuj Kalia and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, page 17, Boston MA USA, February 2019. USENIX.

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

[32] Qiao Kang, Scot Breitenfeld, Kaiyuan Hou, Wei-keng Liao, Robert Ross, and Suren Byna. Optimizing Performance of Parallel I/O Accesses to Non-contiguous Blocks in Multiple Array Variables. In *2021 IEEE International Conference on Big Data*, pages 98–108, December 2021.

[33] 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.

[34] 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.

[35] June-Hyung Kim, Jangwoong Kim, Hyeongu Kang, Chang-Gyu Lee, Sungyong Park, and Youngjae Kim. pNOVA: Optimizing Shared File I/O Operations of NVM File System on Manycore Servers. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19)*, pages 1–7, Hangzhou, China, 2019. ACM Press.

[36] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, pages 1–15, Heraklion Greece, April 2020. ACM.

[37] Stavros G. Kolliopoulos and George Steiner. Partially-ordered knapsack and applications to scheduling. In Rolf Möhring and Rajeev Raman, editors, *Algorithms — ESA 2002*, pages 612–624, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[38] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[39] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write optimization of log-structured flash file system for parallel I/O on manycore servers. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 21–32, Haifa Israel, May 2019. ACM.

[40] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. Hydra: Resilient and Highly Available Remote Memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*, page 19, Santa Clara, CA, February 2022.

[41] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data*, pages 355–370, San Francisco California USA, June 2016. ACM.

[42] 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.

[43] 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.

[44] Zhen Liang, Johann Lombardi, Mohamad Chaarawi, and Michael Hennecke. DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory. In Dhabaleswar K. Panda, editor, *Supercomputing Frontiers*, Lecture Notes in Computer Science, pages 40–54, Cham, 2020. Springer International Publishing.

[45] 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)*, page 13, June 2014.

[46] 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)*, page 15, Santa Clara, CA, July 2017. USENIX.

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

[48] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. INFINIFS: Efficient metadata service for Large-Scale distributed filesystems. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*, Santa Clara CA USA, February 2022. USENIX.

[49] Xiaosong Ma, M. Winslett, Jonghyun Lee, and Shengke Yu. Improving MPI-IO output performance with active buffering plus threads. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

[50] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, page 16. USENIX, July 2020.

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

[52] 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.

[53] 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)*, page 15, Renton WA USA, April 2022. USENIX.

[54] 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.

[55] NASA Advanced Supercomputing (NAS) Division. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html.

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

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

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

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

[60] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, and Abhinav Sharma. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, page 16. USENIX, February 2021.

[61] Percona LLC. PerconaFT. https://github.com/percona/PerconaFT, March 2022.

[62] Scott Peterson. Adaptive Distributed NVMe-oF Namespaces. https://www.snia.org/educational-library/adaptive-distributed-nvme-namespaces-2020, September 2020.

[63] 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.

[64] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118, Portland Oregon USA, June 2015. ACM.

[65] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 51–63, Chicago Illinois USA, May 2017. ACM.

[66] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. RDMA is Turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*, page 15, Renton WA USA, April 2022. USENIX.

[67] 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.

[68] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework For File System Benchmarking. *USENIX ;login:*, 41(1):6–12, 2016.

[69] Rajeev Thakur, Robert Ross, and Robert Latham. Implementing Byte-Range Locks Using MPI One-Sided Communication. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666, pages 119–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[70] 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.

[71] 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 '18)*, page 20, Carlsbad CA USA, October 2018. USENIX.

[72] 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.

[73] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM. In *Proceedings of the 2021 USENIX Annual Technical Conference*, page 16. USENIX, July 2021.

[74] John William Joseph Williams. Algorithm 232 - Heapsort. *Communications of the ACM*, 7(6):347–348, June 1964.

[75] Parkson Wong and Rob F. Van der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4, 2003.

[76] Cong Yan and Alvin Cheung. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment*, 9(5):444–455, January 2016.

[77] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, page 15, Boston MA USA, February 2019. USENIX.

[78] Jian Yang, Joseph Izraelevitz, and Steven Swanson. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *Proceedings of the 17th USENIX Symposium on Networked System Design and Implementation (NSDI '20)*, page 17, Santa Clara CA USA, February 2020. USENIX.

[79] 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.

[80] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. 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 (SIGCOMM '20)*, pages 126–138, Virtual Event USA, July 2020. ACM.