ALock: Asymmetric Lock Primitive for RDMA Systems

Amanda Baran
adb321@lehigh.edu
Lehigh University
Bethlehem, PA, USA

Jacob Nelson-Slivon
jjn217@lehigh.edu
Lehigh University
Bethlehem, PA, USA

Lewis Tseng
LTseng@clarku.edu
Clark University
Worcester, MA, USA

Roberto Palmieri
palmieri@lehigh.edu
Lehigh University
Bethlehem, PA, USA

ABSTRACT

Remote direct memory access (RDMA) networks are being rapidly adopted into industry for their high speed, low latency, and reduced CPU overheads compared to traditional kernel-based TCP/IP networks. RDMA enables threads to access remote memory without interacting with another process. However, atomicity between local accesses and remote accesses is not guaranteed by the technology, hence complicating synchronization significantly. The current solution is to require threads wanting to access local memory in an RDMA-accessible region to pass through the RDMA card using a mechanism known as loopback, but this can quickly degrade performance. In this paper, we introduce ALock, a novel locking primitive designed for RDMA-based systems. ALock allows programmers to synchronize local and remote accesses without using loopback or remote procedure calls (RPCs). We draw inspiration from the classic Peterson's algorithm to create a hierarchical design that includes embedded MCS locks for two cohorts, remote and local. To evaluate the ALock we implement a distributed lock table, measuring throughput and latency in various cluster configurations and workloads. In workloads with a majority of local operations, the ALock outperforms competitors up to 29x and achieves a latency up to 20x faster.

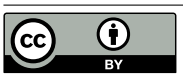
CCS CONCEPTS

• **Hardware → Emerging technologies**; • **Computer systems organization → Peer-to-peer architectures**; • **Software and its engineering → Distributed memory**.

KEYWORDS

RDMA, Lock, Synchronization, Locality

ACM Reference Format:
Amanda Baran, Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri. 2024. ALock: Asymmetric Lock Primitive for RDMA Systems. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3626183.3659977



SPAA '24, June 17–21, 2024, Nantes, France

ACM ISBN 979-8-4007-0416-1/24/06
https://doi.org/10.1145/3626183.3659977

1 INTRODUCTION

Coordinating access to shared resources is a fundamental challenge in concurrent computation and has motivated the widespread adoption of hardware enabled synchronization mechanisms like compare-and-swap (CAS), whose importance in concurrent computing is indubitable. An atomic CAS operation enables an arbitrary number of threads to agree on a value in a wait-free manner [13], making it a powerful tool for building mutual exclusion primitives. The value of atomic CAS operations is likewise evident in the increasingly popular network communication technology, remote direct memory access (RDMA).

Remote direct memory access is a popular network communication technology that aims at implementing the shared-memory abstraction in the distributed setting by allowing a thread to access memory on a remote machine *without* interacting with another process [15, 30, 31]. In addition to its ability to read and write memory on another machine, RDMA also enables threads to perform atomic read-modify-write (RMW) operations on remote memory, like compare-and-swap.

An important factor to consider when dealing with RDMA operations is that the atomicity between local (i.e., shared-memory) and remote (i.e., RDMA) accesses *is not guaranteed*. That means:

i) RDMA reads and writes are only atomic with their local counterparts for accesses within a single cache line [10] (typically only 64 bytes); and
ii) RDMA RMW operations are *not* atomic with local RMW operations.

Essentially, from the perspective of local memory, a remote RMW is nothing more than a read followed by a write to the same memory location. As a result, the lack of atomicity for RMW operations makes it significantly more difficult to synchronize local and remote accesses.

In practice, RDMA RMW operations are frequently used in RDMA-based distributed systems when synchronizing concurrent accesses (e.g., [6, 7, 35, 36]). To ensure atomicity between operations in these systems, threads performing local accesses must use the *loopback* mechanism, which allows a thread to access RDMA memory on its own machine by passing through the local RDMA network interface controller (RNIC) [37]. Although fast compared to other network communication, RDMA is still at least an order of magnitude slower than shared memory operations [17, 37], and suffers from non-uniform memory access (NUMA) behaviors [25], degrading performance for local accesses. Additionally, the RDMA

loopback mechanism can cause poor performance due to internal congestion [19]. In Section 2, we discuss an empirical study we conducted to confirm our claims about loopback congestion and the scalability issues of RDMA.

Alternatively, systems could leverage remote procedure calls (RPCs) to allow all synchronization to be handled exclusively by threads on receiving nodes. However, RPCs follow the traditional send/receive programming model, which has been shown to nullify the performance benefit of directly accessing remote memory [29]. However, in practice, RPCs are still used in RDMA-based systems (e.g., [10, 18, 24, 33, 38]), and their prevalence is attributed in part to the challenges associated with synchronizing local and remote accesses.

In this paper, we describe a new locking primitive, *ALock*, designed to capture the nuanced requirements of synchronizing local and remote accesses in RDMA-aware systems *without* using loopback or RPCs. In our solution, we enforce that threads performing local accesses only use local (i.e., shared-memory) operations. However, since atomicity is not guaranteed among local and remote (i.e., RDMA) RMW operations, we require a new mutual exclusion algorithm between these two groups of operations.

Our Design. The intuition behind the ALock design begins with reducing the problem to a two-process mutual exclusion. A natural choice is to find inspiration from Peterson's algorithm [27], a well-known, starvation-free, and fair mutual exclusion protocol for two processes using only shared memory to communicate (more details about Peterson's algorithm can be found in Section 3.1).

To allow for *multi*-thread synchronization, we extend Peterson's algorithm to include two *cohorts* (as opposed to just two threads), *remote* and *local*. Threads within each cohort compete amongst each other to determine a *leader*. The leaders of the two cohorts then compete using our modified version of Peterson's lock algorithm to successfully acquire the ALock, finally entering the critical section. Thanks to this hierarchical design, threads within each cohort can use APIs that are guaranteed to be atomic with each other. Specifically, threads performing local accesses can use shared-memory operations, and threads performing remote accesses can use RDMA operations. Notably, there is no need to use the loopback mechanism anymore.

In addition to eliminating loopback for local accesses, it is critical to limit the number of RDMA operations for remote accesses in order to prevent congestion in the RNIC, which causes performance deterioration (see Section 2). For this reason, we take advantage of the design of the widely used MCS queue lock [23], which allows threads to spin locally while waiting in a queue to acquire the lock. This combination of locks has similar characteristics to lock cohorting [9], which decouples the synchronization within a cohort with the synchronization among cohorts.

To evaluate the ALock, we build a distributed lock table and measure the throughput and latency on multiple cluster configurations under various contentions, ratios between local and remote accesses, and workloads. We compare against the commonly used RDMA CAS-based spinlock and an RDMA-enabled MCS Queue lock. In the presence of a majority of local operations, ALock outperforms competitors up to 29x. As a result, data repositories that use one-sided RDMA operations and currently rely on specialized hardware or loopback to achieve atomicity between remote and local operations can relax these constraints and improve performance with ALock.

To the best of our knowledge, we are the first to fully develop a solution that solves mutual exclusion specifically for RDMA in a manner that does *not* require involving the loopback RNIC or RPC handling for local accesses, is *starvation-free* and *fair*.

The ALock is released as an open-source project and can be found at [4]. A preliminary design of this work appears in [26].

The formal TLA+ specification of the ALock's correctness, liveness, and fairness can be found in our Technical Report [5].

2 RDMA SCALABILITY AND PITFALLS

RDMA is a technology increasingly affecting the design of many distributed systems due to its unquestionably fast operations. These operations include APIs to send/receive messages but also to operate directly on remote memory (named one-sided operations). The presence of these one-sided operations is considered RDMA's true breakthrough because it challenges the message-passing model, which is typically used to design distributed interactions. Although promising as a technology, RDMA's programmability still requires innovations, such as our ALock, as its performance scalability is hampered by two factors, namely the loopback network congestion and QP thrashing [19, 32].

To assess the performance of RDMA loopback and show the network saturation pitfalls it comes with, we perform a simple experiment to verify our intuition. We run a simple spinlock algorithm using 1000 locks, which shows low to no logical contention, on a single machine equipped with an Intel Zeon E5-2450 processor with 8/16 cores/threads and one Mellanox dual port CX3 RNIC. As shown in Figure 1, the peak throughput is reached at a few threads. After that, the loopback traffic drains the PCIe bandwidth, causing accumulation in the RNIC's RX buffer and slowing down the CAS operations despite the lack of logical contention.

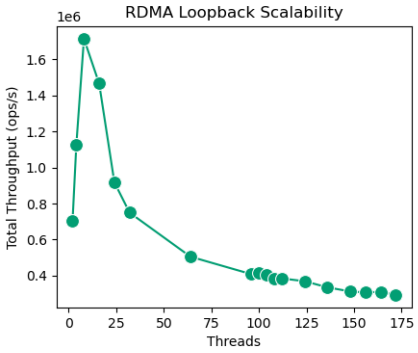


Figure 1: RDMA Spinlock with 1k locks on 1 Node.

Another well-documented problem with RDMA in large-scale data centers is connection scalability. In order for one-sided RDMA operations to avoid the involvement of the host CPU, the RNIC also manages the connection context. This connection context, sometimes referred to as QP Context (QPC), maintains all the attributes required for both sending data and keeping track of the connection

state. Typical commercial RNICs such as the ConnectX-4 require 256 bytes to maintain the QPC for each RDMA connection [22]. The Infiniband specification states that the RNIC can handle at most 2^{24} QPCs [15]. However, at 256 bytes each, this would require the RNIC to have 4GB of space to store QPCs for all 2^{24} connections. In reality, the RNIC has a very small cache on its chip, which is not nearly enough to maintain all of its connections. Recent work has shown that the message rate of commodity RNICs declines after 450 connections [32]. As a result, RDMA-enabled systems can experience a side effect known as *QP thrashing*. This occurs when the number of QPs being used does not fit in the RNIC cache, so the RNIC is continuously loading and evicting QPCs from the cache, quickly degrading performance. Systems using RDMA loopback for local accesses can still be affected by QP thrashing regardless of whether the memory is local to the requesting thread.

The design of the ALock directly solves the first issue of using loopback in the presence of workloads accessing local memory. Also, it limits QP thrashing by removing $\frac{1}{n}$ QPs from the system (where n represents the number of nodes) if we assume an identical workload on each node.

3 BACKGROUND

3.1 Peterson's Algorithm

Peterson's algorithm (also referred to as Peterson's lock) by Gary L. Peterson [27] is a well-known algorithm for mutual exclusion that is both starvation-free and fair. It only relies on read and write operations to coherent shared memory to synchronize. Since atomic read and write operations can be used over remotely accessible memory, Peterson's algorithm can also be implemented directly over RDMA, with the appropriate memory fences, to coordinate access between a local and remote thread.

Peterson's algorithm uses two variables: a boolean array `flag` of size 2 and an integer `victim`. A process announces its interest in entering the critical section by setting its flag to `true` and setting itself as the victim. By doing so, the algorithm guarantees fairness as the waiting process will always access the critical section next, regardless of any performance differences between the two threads. After completing the critical section, the process sets its flag to `false` to indicate it does not want to execute anymore. The waiting thread waits until the executing process sets its flag to `false`, or the thread is no longer the victim. By alternating access to the critical section through the use of the "victim" variable, the algorithm ensures both processes eventually gain access.

3.2 MCS Lock

Mellor-Crummy and Scott presented a scalable mutual exclusion algorithm for multiprocessors in 1991, the MCS lock [23]. The algorithm is fair because it is equivalent to a FIFO-pattern queue. It is also contention-free, as each processor spins on a local variable. This property is desirable when extending the MCS lock to be used in a distributed system, as remote spinning severely limits performance due to network congestion.

A process interested in acquiring the lock first allocates a locally-accessible descriptor, which contains a boolean flag to spin on and a pointer to form the queue. To acquire the lock, the process adds its descriptor to the end of the queue. If the process has a predecessor, the process spins on its local boolean variable until its predecessor passes the lock by writing to the variable. Otherwise, the process is the head of the queue and is the lock holder. To release the lock, the process writes to the boolean variable of its successor. If the process has no successor, the tail is set to `NULL`.

4 SYSTEM MODEL

We model an RDMA-based distributed system as a set of nodes N and threads[1] T, for which t_i^j is a thread running on node n_i with identifier j. All threads can access an RDMA-accessible shared-memory partitioned among the nodes. In our model, all data and metadata are stored in RDMA-accessible memory. Similar to previous RDMA system models [1], we assume that threads are asynchronous and that accesses to memory are failure-free.

The notion of locality is crucial in our model. Although not new [12], we define it in relation to the APIs a thread uses to operate on RDMA-accessible memory. Definitions 4.1 and 4.2 define local and remote access.

Definition 4.1 (Local Access). A thread t_i^j, executing on node n_i, performs a *local access* using shared-memory operations if the RDMA-accessible memory being accessed is stored on n_i .

Definition 4.2 (Remote Access). A thread t_i^j, executing on node n_i, performs a *remote access* using RDMA operations if the RDMA-accessible memory being accessed is *not* stored on n_i .

For each class of access (local or remote), memory is accessed through three operations: read, write, and compare-and-swap. We denote the shared-memory (or local) operations using `Read`, `Write` and `CAS`, and the RDMA (or remote) operations with `rRead`, `rWrite` and `rCAS`. As shown in Table 1, the atomicity of operations between classes is *not* guaranteed due to the design of RDMA.

Access (8B)		Remote (RDMA)		
		`Read`	`Write`	`CAS`
Local	`Read`	Yes	Yes	Yes
	`Write`	Yes	Yes	No
	`RMW`	Yes	Yes	No

Table 1: Atomicity between 8-byte local and remote accesses.

Since remote operations complete asynchronously, local and remote access to a given memory location may be reordered [8]. As a result, an `rCAS` operation appears to a thread performing local accesses as if it were a `Read` followed by a `Write`. Hence, interleaves with other local operations are possible. Thus, the programming model requires that programmers wait until operations are complete by using appropriate memory fences to guarantee correct ordering [15].

We say a memory object is *operation-asymmetric* if, given two threads, the intersection of the operations they can perform on that object is *not equal* to their union. ALocks are designed to deal with the operation asymmetry of RDMA-accessible memory, hiding the complexity from the programmer while simultaneously optimizing for it.

[1]The theory literature often refers to threads as processes. These names are interchangeable in our context.

5 ALOCK ALGORITHM AND DESIGN

ALock innovates over the idea of lock cohorting [9] by defining cohorts with respect to the APIs used by threads operating on the lock. More specifically, in our case, two cohorts are defined as follows: one cluster of threads performing local accesses, and one cluster of threads performing remote accesses.

The ALock is a composition of two of our modified MCS queue locks, one for each cohort, and our modified Peterson's algorithm. Elements in the MCS queues of the above MCS locks are made of descriptors, which are memory regions describing the state of the lock request. Since lock requests can be generated by threads from any node in the system, these MCS queues are effectively linked to possibly distributed memory. For simplicity, we refer to each MCS queue by its *tail* (i.e., the pointer to the last element of the queue). To maximize performance and reduce the amount of metadata required, the remote and local tails ($tail\_r$ and $tail\_l$, respectively) are also interpreted as the status flags used in Peterson's algorithm [14]. Recall from Section 3.1 that, when the flag is set to `true` for process i, that means i is either interested in capturing the lock or is currently in the critical section. Similarly, in our algorithm, tails contain either a `NULL` pointer or a pointer to a descriptor. The latter indicates that the remote ($tail\_r$) or local ($tail\_l$) cohort is interested in or has acquired the lock. Embedding Peterson's flag semantics into the MCS queues avoids an additional, possibly remote, memory operation. Also, as required for the Peterson' algorithm, we include the boolean `victim` field, which is used to yield the lock to a waiting thread of the other cohort.

Metadata. The metadata used by a thread to interact with an ALock is represented in Algorithm 1. Two descriptors are allocated in RDMA-accessible memory, one `LocalDescriptor`, which is used by threads that are local to the ALock, and one `RemoteDescriptor`, which is used by threads that are remote to the ALock. Each of these descriptors has a budget field associated with it. In short, the budget is used to introduce fairness between ALock's cohorts. A comprehensive discussion about that can be found later in the section.

Algorithm 1: ALock Metadata

```
1  Struct ALock contains
2      rdma_ptr<RemoteDescriptor> tail_r;
3      rdma_ptr<LocalDescriptor> tail_l;
4      int victim;
5  Struct LocalDescriptor contains
6      int local_budget;
7      LocalDescriptor* next;
8  Struct RemoteDescriptor contains
9      int remote_budget;
10     rdma_ptr<RemoteDescriptor> next;
```

The ALock provides two operations, `Lock(rdma_ptr<ALock>)` and `Unlock(rdma_ptr<ALock>)`, shown in Algorithm 2.

Lock Procedure. Upon initiating a `Lock(rdma_ptr<ALock>)` operation, it is first determined whether the thread is performing a local (see Definition 4.1) or remote (see Definitions 4.2) access on the requested lock. This is done by checking the ID of the node where the lock is located, which is embedded in the first 4 bits of the RDMA pointer. Based on the classification of the lock access request, the thread first competes to be the leader of either the remote or local cohort for that lock via our modified remote or local MCS Queue algorithm (see Algorithm 3). In our algorithm, if the call to `qLock(rdma_ptr<ALock>)` returns `false` on Line 6, that means there is no current lock holder. Therefore, the thread needs to then compete in our modified Peterson's algorithm (see Algorithm 4) in order to finally acquire the ALock. Otherwise, if `qLock(rdma_ptr<ALock>)` returns `true` on Line 13 of Algorithm 3, we say the MCS lock was *passed*. This means that the thread did not swap onto an empty MCS queue. Instead, it waited for a predecessor to hand over the lock by spinning on a local variable in its descriptor. This local variable is written by a predecessor in order to notify the thread that it now owns the lock.

Algorithm 2: ALock Operations

```
1  Lock(rdma_ptr<ALock>) begin
2      passed ← qLock(rdma_ptr<ALock>)
3      if passed = false then
4          pReacquire(rdma_ptr<ALock>)
5  Unlock(rdma_ptr<ALock>) begin
6      return qUnLock(rdma_ptr<ALock>)
```

Algorithm 3 shows the modified MCS algorithm for remote accesses, whereby the algorithm for local accesses simply requires replacing each remote access with a local (i.e., shared-memory) one. For simplicity, let us assume this is the first time the lock is ever acquired. Because of that, `qLock(rdma_ptr<ALock>)` returns `false`, meaning the lock was not passed, and therefore, the thread needs to participate in our modified Peterson's algorithm (Algorithm 4) to finally obtain the ALock.

The steps of ALock's `Lock(rdma_ptr<ALock>)` operation (Alg. 2 Line 1) for a remote access are as follows. First, the thread announces its interest in obtaining the cohort lock by swapping its descriptor onto the remote tail ($tail\_r$) of the lock (Line 3). Assuming that this is the first time the ALock is ever acquired, the rCAS on Line 3 will succeed on its first attempt. Otherwise, if the queue was not empty, the rCAS uses the learned value, $prev$, to retry the rCAS and swap the descriptor onto the tail of the queue. The MCS queue designed for the remote cohort now only contains this thread's descriptor. As mentioned earlier, a thread requesting a local lock performs the same steps to acquire the ALock but uses shared-memory APIs to interact with the local tail ($tail\_l$), compete to become the leader of the local cohort, and participate in Peterson's algorithm to synchronize with the leader of the remote cohort.

Unlock Procedure. Unlocking the ALock is done by invoking the `Unlock(rdma_ptr<ALock>)` operation. The thread first attempts to remove its descriptor from the appropriate (i.e., local or remote) tail of the MCS queue. Under the assumption that no other concurrent lock request occurres, the modified tail now points to `NULL`, which means the thread is leaving the critical section defined by Peterson's algorithm and thus indicates the ALock is successfully unlocked. On the other hand, if another thread has issued a

Algorithm 3: Modified Remote MCS Queue Lock

```
Data: (constants) kInitBudget; (global) tail_r;
      (process-local) desc
1  qLock(rdma_ptr<ALock>) begin
2      desc ← RemoteDescriptor(null)
3      prev ← rCAS(tail_r, nullptr, &desc)
4      if prev = nullptr then
5          desc.remote_budget ← kInitBudget
6          return false ;          // Lock was not passed
7      desc.remote_budget ← −1
8      rWrite(&(prev->next), &desc)
       // Spins locally
9      while desc.remote_budget = −1 do wait ;
10     if desc.remote_budget = 0 then
11         pReacquire() ;          // Provides fairness
12         desc.remote_budget ← kInitBudget
13     return true ;                  // Lock was passed
14 qUnlock(rdma_ptr<ALock>) begin
15     curr = rCAS(tail_r, &desc, nullptr)
16     if curr ≠ &desc then
           // Wait for successor to enqueue
17         while desc.next = nullptr do wait;
           // Pass the lock
18         rWrite(&(desc.next->remote_budget),
            desc.remote_budget - 1)
19 qIsLocked() begin
20     return rRead(tail_r) ≠ nullptr
```

Algorithm 4: Modified Peterson's Lock

```
Data: (global) cohort[2], victim
1  pReacquire() begin
2      id ← getCid() ;    // Get ID of process cohort
3      other ← 1− id
4      victim ← id ;  // Yield lock to waiting cohort
5      while cohort[other].qIsLocked() or victim = id
        do
6          wait
```

lock request to the ALock in the meantime, the tail will no longer be equal to the thread's descriptor. Following the original MCS algorithm, releasing the lock when a successor is present requires notifying the successor of the lock release by writing to a special state field within the successor's descriptor. As mentioned earlier, we refer to notifying the successor of the lock release as *passing* the lock. This process is further described in Section 5.1.

Adding Fairness. As described so far, the above algorithm is unfair because the lock may be passed indefinitely among threads of the same cohort. To address fairness, we introduce a budget policy that uses a counter to decide when a cohort must release the ALock. To enforce the budget policy, we extend the classic Peterson's algorithm to support a reacquire operation. With this operation, a thread releases the lock by first setting its cohort as the victim and then immediately attempts to reacquire the lock. In the following sections, we refer to the counter variable budget as an indicator of whether a lock should be released or passed to the waiting successor. Specifically, if the budget is 0, the lock must be released regardless of a present waiting request. Since the lock is released after a bounded number of cohort lock acquisitions, and the lock is itself fair (i.e., a waiting thread cannot be overtaken), our approach is fair [9].

Example. For the sake of clarity, we define an example to show the procedure that two competing threads of opposite cohorts follow in order to synchronize on the ALock. We model a cluster with two nodes, n_1 and n_2. One lock resides on each node, l_1 on n_1 and l_2 on n_2. We also have one thread on each node, t_1 and t_2. For t_1, all access requests made on l_1 will be considered local, and all access requests made on l_2 will be considered remote. Likewise, for t_2, access requests made on l_2 will be considered local, and access requests made on l_1 will be considered remote. In Figure 2, we show the steps taken by t_1 to lock l_2 followed by t_2 attempting to lock l_2 concurrently.

In the first frame ①, we show the 2-node cluster as described with one lock and one thread on each node. As shown in frame ②, each thread has a remote descriptor (RD) and local descriptor (LD) that are both initialized with -1 and NULL values for the budget and next pointer, respectively. The first step taken by t_1 to lock l_2 is to perform a rCAS on the tail of its cohort's MCS queue, which in this case is the remote cohort. After successfully performing a rCAS on the remote tail (Frame ③), the thread engages in Peterson's algorithm as the queue was empty when the thread enqueued its descriptor. As shown in Frame ④, t_1 sets the victim variable to be its cohort (REMOTE), and then enters the while loop, which checks that either the victim is no longer set to REMOTE, or that the local tail's cohort is unlocked. In this example, the local tail is currently NULL (unlocked), so T_1 immediately acquires Peterson's lock. After seeing that its budget is now non-negative, t_1 learns that it has acquired the lock and enters its critical section (Frame ⑤).

While t_1 is in its critical section, t_2 also attempts to lock l_2. In Frame ⑤, we see t_2 locks its cohort by swapping its local descriptors address into the local tail. Again, since the queue is empty when the thread adds itself, it must engage in our Peterson's algorithm. Frame ⑥ shows t_2 first sets the victim to be LOCAL, then spins while the victim is still LOCAL or the remote tail is locked (i.e., not null). In Frame ⑦, we see that t_1 unlocks l_2 by performing an rCAS on the remote tail to set it back to NULL, removing its descriptor from the queue. At this point, t_2 will exit the while loop since the remote tail is now unlocked, and upon seeing its now non-negative budget (Frame ⑧), t_2 has acquired the lock and is able to enter its critical section.

In summary, our technique enables local and remote threads to synchronize via two independent locks integrated into the hierarchical ALock algorithm. By using a budgeted MCS queue lock as the embedded cohort lock, we can provide fairness. Lastly, our approach is RDMA-aware since threads performing local accesses avoid RDMA loopback, and threads performing remote accesses limit remote spinning, completely avoiding it in the case competing for Peterson's lock is not required.

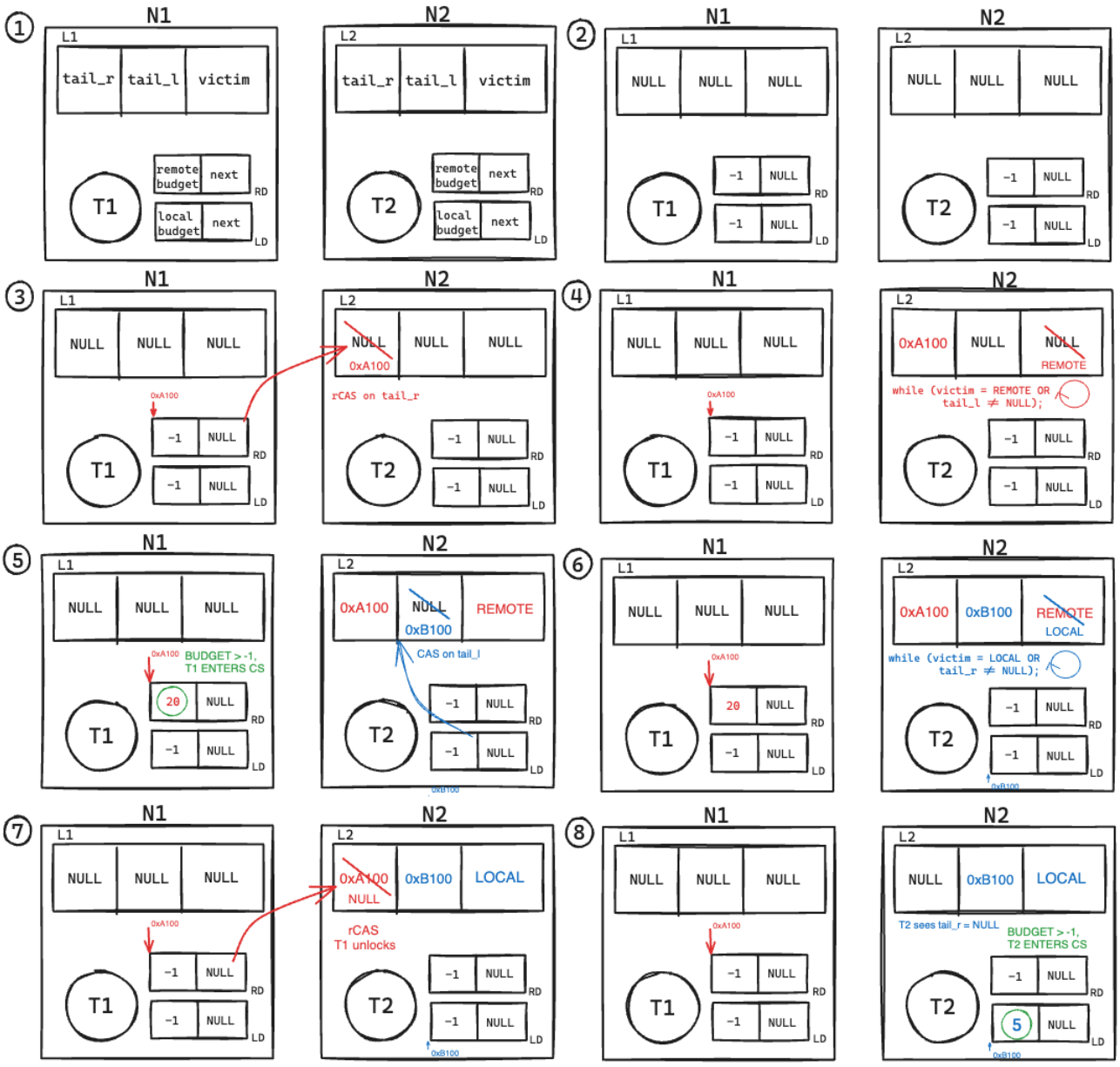


Figure 2: Execution of the ALock algorithm in a system with two nodes, one lock per node, and one thread per node.

5.1 ALock's MCS lock

In this section, we describe the modifications we made to the original MCS queue algorithm for remote accesses. It should be noted that the local version of the algorithm can be obtained by directly replacing each remote operation with a local one. This algorithm maintains a global variable, $tail\_r$, which is a remote reference to the corresponding slot in the `cohort` array of Algorithm 4 that acts as the tail of the lock queue.

In `qLock()`, a thread atomically swaps a new descriptor into $tail\_r$ and then waits until the thread is at the head of the queue, finally returning whether the queue was empty at its onset. The value swapped into the tail of the queue contains an address of the remotely accessible descriptor (`desc`), which is the requesting thread's local metadata. If $tail\_r$ was not previously set (i.e., was `NULL`), the call to `qlock()` on Line 2 of Algorithm 2 returns `false` since the cohort lock was not passed to the calling thread. Otherwise, if the tail was not `NULL`, the thread performs an `rWrite` operation to the current tail's `next` pointer containing the address of the thread's `RemoteDescriptor`. The locking thread then spins locally on its budget field shown in Line 9 of Algorithm 3 while waiting for its predecessor to pass the lock. Once the budget field is set to a

non-negative value via a `rWrite` operation by the predecessor, the lock is considered to be passed to the successor.

After acquiring the lock and performing its critical section, a thread attempts to release the lock following the conventional MCS queue algorithm, which tries to `rCAS` the tail of the queue back to a `NULL` value. Recall that if the `rCAS` operation in `qUnlock()` is successful, then it also releases the Peterson's lock, since the corresponding `cohort` is now unset. Otherwise, the thread passes the lock to the next waiting thread by performing a `rWrite` to the location returned by the attempted `rCAS`. At worst, each `Unlock()` operation requires an `rCAS` operation followed by an `rWrite`.

To support our fairness policy, we alter the original MCS queue algorithm to support a budget, similar to the technique used by Dice et al. [9]. A lock is passed during the `Unlock()` operation by setting the budget of a waiting thread's descriptor to a non-negative integer that represents the number of remaining lock acquisitions. When the budget reaches zero, a requesting thread must call `pReacquire()` on the ALock. If there is a waiting thread of the opposite cohort, it will be allowed to proceed when the thread sets itself to be the victim (Algorithm 4 Line 4). Otherwise, the calling thread reacquires the ALock and resets the budget.

As mentioned earlier, the choice of the MCS algorithm is in part due to its inherent reduction of network contention. When the queue is empty, a lone thread requires a single `rCAS` to acquire the MCS lock, followed by a single `rRead` to participate in Peterson's algorithm. Otherwise, if the queue is not empty, the calling thread spins on its local descriptor, avoiding remote spinning and thus reducing network traffic.

5.2 ALock's Peterson's Algorithm

Our modified Peterson's lock algorithm (Algorithm 4) has two global variables: `cohort`, which is a two-element array of cohort locks, and `victim`, which determines the cohort that yields execution. We replace the flag variables in the classic Peterson's algorithm with this array of cohort locks. Threads engage in Peterson's algorithm after becoming the leader of their cohort's MCS Queue if they are *1)* the only thread currently requesting the lock; or *2)* passed a budget of 0, requiring the cohort to release the ALock.

In `pReacquire()`, a thread first yields the lock to the waiting cohort by setting the `victim` to be its own cohort. Recall that a thread's cohort is defined for each operation as remote or local based on whether the access for the lock is remote or local. The wait condition (Line 5) in our modified algorithm is similar to the original Peterson's algorithm. Checking if the competing thread has finished executing is done by checking whether the other MCS queue cohort has been unlocked. As mentioned in Section 4, the memory semantics of RDMA is *not* sequentially consistent. This requires adding atomic thread fencing instructions after locking and before unlocking and using atomic load and stores for shared memory operations. Crucially, any interleaving of instructions in concurrent calls to `Lock(rdma_ptr<ALock>)` will only allow a single thread access to the critical section, assuming that sequential consistency is enforced.

6 EVALUATION

In this section, we present the performance results of our ALock. As competitors, we ported two well-known locking algorithms, spinlock, and MCS lock, to RDMA. The former simply repeats RDMA `rCAS` until it succeeds. For the latter, we implement an RDMA-aware queue and integrate it into the original MCS lock algorithm. Both these implementations use RDMA for all their operations, regardless of locality. In other words, while ALock only performs RDMA operations on remote memory, the competitors use the local RDMA loopback card to perform RDMA operations on local memory.

All our code is written in C++ and compiled with `-std=c++20` at optimization level -O3. All competitors are optimized for RDMA operations. Each metadata is padded to a size of 64 bytes in order to prevent false cache-line sharing (pictured in Figure 3 for ALock), and RDMA pointers remain small at 8 bytes to be friendly to RDMA atomic operations. We use `rdma_ptr<T>` to represent an RDMA pointer to an object of type T, which is located in RDMA-accessible memory. The first 4 bits of the pointer embed the node ID where the memory resides, followed by 60 bits to represent the memory address on that node. For our experiments, we used 20 machines from the CloudLab [28] testbed running on Ubuntu 22.04. Each machine is equipped with an Intel Zeon E5-2450 processor with 8/16 cores/threads and one dual-port Mellanox ConnectX-3 RNIC.

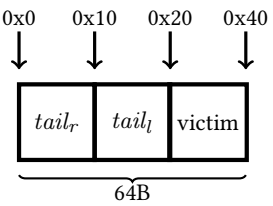


Figure 3: 64B-aligned ALock containing 8B pointers to the remote and local cohort tails, and an integer victim field to indicate the current victim cohort. Values are padded to the address alignments shown.

To test our competitors, we implement a distributed lock table in which locks are partitioned equally across nodes. We measure the throughput and latency of operations that encompass both one lock and one unlock operation. We vary the logical contention by changing the size of the lock table (i.e., the number of locks). In our experiments, we use 20 locks for a high-contention scenario, 100 for medium contention, and 1000 for low contention. We also experiment with different workload levels by varying the number of application threads per node, as well as the number of nodes. Lastly, we also vary the percentage of operations performed by application threads on local locks. For example, when we refer to 95% locality, it means 95% of the operation a thread performs during the experiment target locks that are stored in the same node where the threat executes (i.e., local accesses as defined in Section 4).

We conduct our study by first analyzing the impact of the budget on the performance of ALock. The results of these experiments inform our decision to choose a remote and local budget for our remaining experiments. We then continue by measuring throughput and latency under different conditions.

6.1 Choosing a Budget

As described in Section 5, the ALock algorithm provides fairness through the use of a budget. However, the budget has a double effect: preventing starvation while also minimizing the cost of the lock reacquire operation. There is an *asymmetric* cost associated with the reacquire operation for remote and local threads because a local thread needs just shared-memory operations to participate in the reacquire, while a remote thread needs RDMA operations. For this reason, we have two separately defined budget configurations: one local budget and one remote budget(see Algorithm 1).

Our intuition is that keeping the local budget low minimizes the time that a thread requesting a remote lock may spin in the high-contention workload scenario. On the other hand, threads that are requesting a remote lock and waiting for it to be released benefit from a higher remote budget as it reduces the number of times the reacquire operation is invoked.

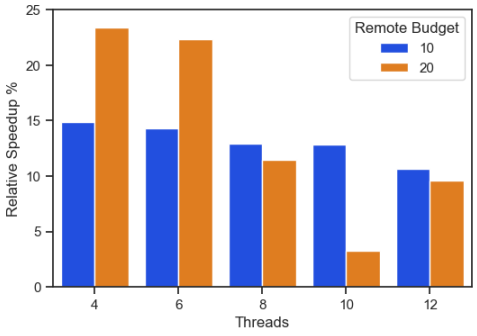


Figure 4: Relative speedup compared to the baseline remote budget of 5.

We performed a series of experiments with various workloads and cluster configurations to confirm our intuition and inform our choice for both the local and remote budgets. We found that the most impactful factor for the budget is the contention as provided by the workload. Figure 4 shows the average performance improvement relative to a baseline configuration having both remote and local budgets set to 5. In the plot, we show ALock's performance while increasing the remote budget from 5 to 10 and 20. Results are averaged under 95%, 90%, and 85% locality workloads on a 20-node cluster with 100 locks (medium contention).

As shown by the plot, limiting the local budget while increasing the remote budget improves total throughput by up to 23%. We attribute this to the fact that competing in the `Reacquire` operation is much more costly for threads requesting remote locks rather than local locks due to the cost of the remote spinning encountered in Peterson's algorithm when contention from the local cohort is present. Even without local contention, the cost of two additional remote operations at the minimum needed for the remote cohort to reacquire is much more costly than the cost of two additional local operations needed for the local cohort to reacquire [17]. These findings informed us to choose a remote budget of 20 and a local budget of 5 for the rest of the experiments.

6.2 Throughput Evaluation

In Figure 5, we report the throughput in terms of operations (i.e., one successful lock and unlock) per second. We vary the size of the system, the workload, and the contention level.

We first discuss the case of 100% locality. Because of the significant gap in performance, we isolated this workload on separate plots (Figures (d), (h), (l)). As evident by the plots, the ALock significantly outperforms the spinlock and MCS lock at low, medium, and high contention levels. Specifically, the ALock performs up to 24x as many operations as the MCS Lock and 22x as many operations as the spinlock, even under high contention with just 20 locks. Our ALock is able to take full advantage of the beneficial operation asymmetry here and *only* use shared-memory operations. Additionally, there is no contention in Peterson's algorithm due to the absence of the remote cohort.

Moving away from 100% locality, we focus on the high-contention case (Figures (a), (e), (i)). The spinlock and MCS lock are both overwhelmed in these cases. The most extreme contention is in Figure (i). With up to 240 threads competing for just 20 locks in the 20-node configuration, the workload creates congestion in the RNIC, which quickly degrades performance. Conversely, the ALock outperforms the MCS lock by up to 29x and the spinlock by up to 24x. Avoiding the performance bugs introduced by loopback traffic allows the ALock to take full advantage of the passing lock mechanisms.

As the contention decreases, ALock continues to perform well, but competitors' performance increases, resulting in a smaller gap (Figures (b), (f), (j)). Changing from 20 to 100 locks, the MCS lock is no longer overwhelmed by the logical contention and, like the ALock, takes advantage of spinning locally and passing the lock as the workload scales. Interestingly, and somehow counter-intuitive, although the ALock maintains its scalability, it experiences a decrease in throughput as the contention decreases. This is due to the fact that passing the lock cannot always be extensively used without enough contention.

The spinlock, however, saturates quickly and suffers as the workload scales. Since the spinlock uses remote spinning, the card's congestion increases with the number of threads, causing undesirable performance. We attribute this behavior to the congestion introduced by the loopback mechanism, and QP thrashing. In fact, with 8 threads and 10 nodes, the RNIC already needs to maintain 1280 QPs. However, recent work has shown that performance degrades as the number of concurrent connections grows past 450 [32]. In comparison, the ALock throughput scales with the number of threads as it avoids loopback and limits QP thrashing, outperforming the simple spinlock approach by more than 4x with 240 total threads in the system (Figure (j)).

Finally, as contention decreases (Figures (c), (g), (k)), the ALock increases its performance gains as the workload's locality increases. Without logical contention, passing the lock becomes less relevant, but the asymmetry cost of remote and local operations becomes more evident. The ALock improves by 40% when increasing from 85% to 90% locality and improves by an additional 75% when increasing to 95% locality using five nodes. The spinlock and MCS lock are also able to scale their performance with the number of threads but at a slower pace than ALock. However, they do not experience the performance benefits that ALock can from the use

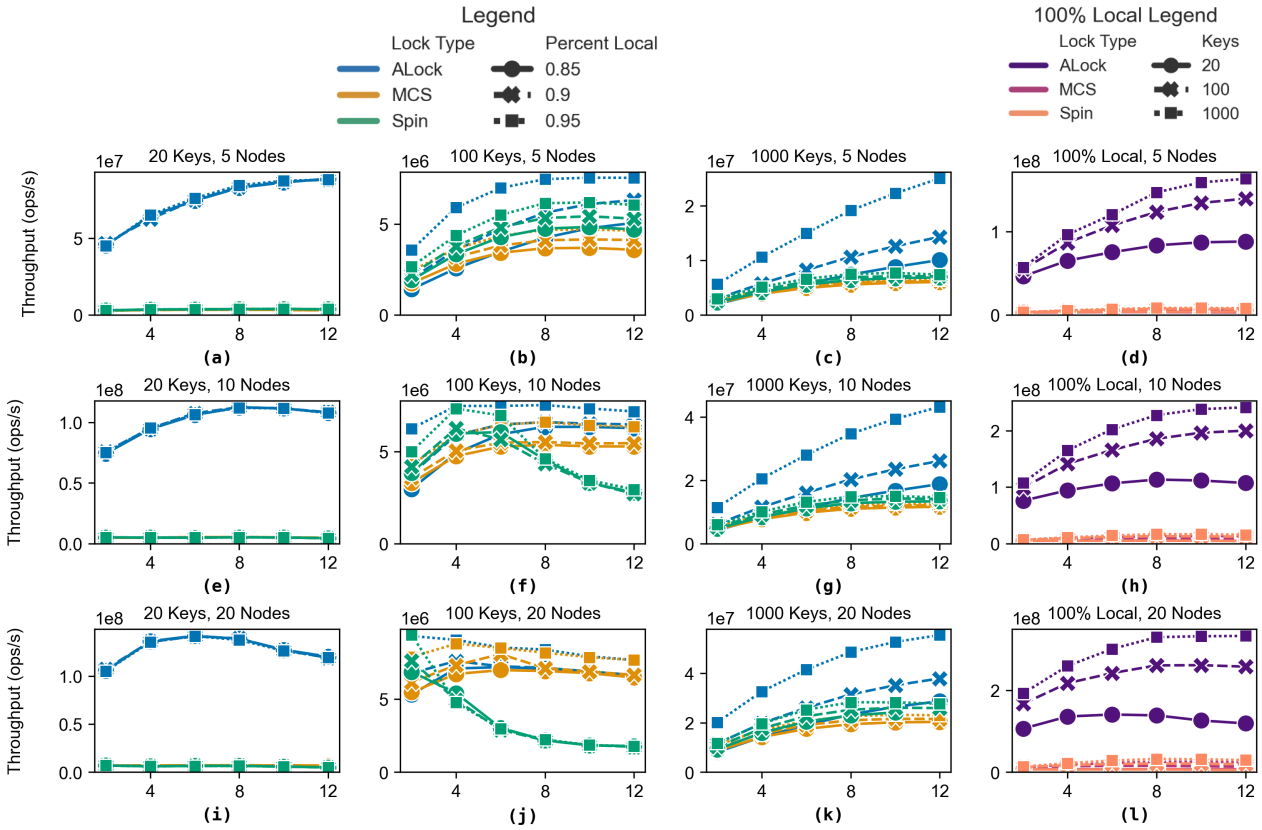


Figure 5: Throughput for a variety of workloads on 5, 10, and 20 nodes. The X-axis is the number of threads per node.

of shared-memory operations as the workload locality increases. In this case, the ALock outperforms the MCS lock by up to 3.8x and the spinlock by up to 3.3x.

6.3 Latency Evaluation

In Figure 6, we report the cumulative distribution of the latency of operations (i.e., one lock and unlock). In our experiments, we varied the workload, contention level, and size of the system. We chose only to include results for the 20-node cluster in order to show the latency at scale. That is because there are minimal differences in the plots with 5 and 10 nodes, except for the performance of spinlock, whose scalability can be better assessed with 20 nodes.

In the 100% local workloads (Figures (a), (b), (c)), the ALock significantly outperforms its competitors. Since the ALock is composed of 100% local operations, the latency is close to that of local memory accesses. Specifically, in the high-contention case with 20 locks (Figure (a)), the ALock is as much as 17x faster than the MCS lock and 33x faster than the spinlock. As the number of locks increases, ALock's tail latency also increases since the passing of the lock can no longer be always used as the contention decreases. On the contrary, the spinlock experiences a much more drastic tail in the high-contention cases with 20 locks (Figures (a), (d), (g), (j)) as it is overwhelmed by the logical contention, causing congestion in the card. The MCS lock, similar to the ALock, takes advantage of spinning locally in these high-contention cases to avoid such a large tail. However, since it still requires many RDMA operations, the MCS lock competitor still experiences a latency of up to 17x longer than the ALock under high contention. In low contention and 100% locality (Figure (c)), the ALock latency is still an average of 10x faster than spinlock and 13x faster than MCS lock.

Moving away from 100% locality and looking at the case of medium contention with 100 locks (Figures (e), (h), (k)), we see that ALock and MCS lock, which share a similar structure, perform very similarly. In these workloads, the logical contention is enough for both the ALock and MCS lock to take advantage of passing the lock and spinning locally, allowing the MCS lock to avoid saturation. On the other hand, the spinlock's long tail latency can be attributed to the congestion caused by remote spinning.

In the case of low contention (Figures (f), (i), (l)), ALock outperforms MCS by an average of 2.1x in the 95% local workload and 1.35x in the 85% local. With the absence of logical contention, both ALock and MCS lock are unable to benefit from passing. However, being unable to pass the lock results in the MCS lock requiring many more remote operations, whereas the ALock can benefit from the locality of the workload. This is evident when looking at Figures (i) and (l). As the locality decreases from 90% to 85%, the gap between the ALock and MCS lock's performance shrinks.

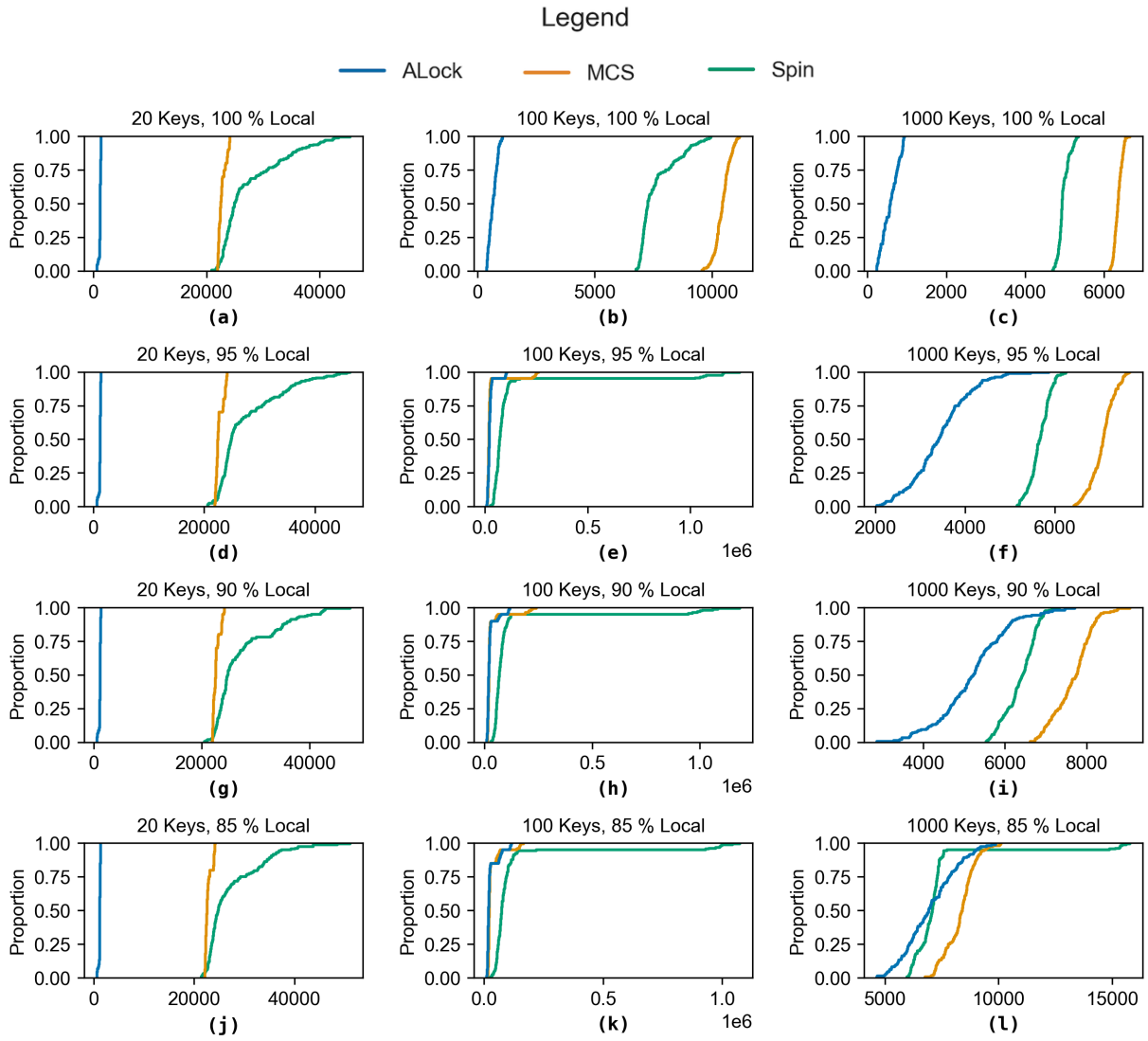


Figure 6: Latency CDF for a 10 node cluster with 8 threads under different workloads. The X-axis is latency in nanoseconds.

7 RELATED WORK

Mutual exclusion is a known problem in which access to a shared resource is coordinated among two or more concurrent threads [21]. Specifically, at most one thread may execute its critical section at a time. A naive solution to multi-thread mutual exclusion is a *filter lock* [27], which extends Peterson's lock for multiple threads. Briefly, threads compete for access to successive levels that each hold back one thread. The number of levels is equal to one less than the number of threads that might acquire the lock. Unfortunately, this would require both remote spinning and a number of remote operations proportional to the number of threads that might contend for the lock, even if a thread executes in isolation. Lamport's Bakery algorithm [14] also demonstrates the same undesirable behavior for remote threads.

Similar to lock cohorting [9], a strategy for NUMA-aware synchronization, our approach allows a group of threads to compete amongst themselves before acquiring a global lock, in our case the ALock. Our technique explicitly couples the Peterson's and MCS locks to achieve behavior tailored to the respective access types in our system, remote and local. By embedding the cohort locks in Peterson's lock, locking and unlocking the MCS queue simultaneously sets and un-sets the Peterson's flag variable. As a result, we avoid an additional remote operation for remote accesses while maintaining the integrity of the ALock. The application of lock cohorting in a distributed setting is a natural extension of the technique but it requires rethinking the design to optimize for operation asymmetry between local and remote accesses, yielding a lock primitive that is of independent interest.

To the best of our knowledge, our approach is the first mutual exclusion primitive designed for RMDA that provides local-only access for threads requesting local locks while maintaining fairness and avoiding RPCs. A notable alternative is the technique pioneered by Wei et al. [34], which allows local accesses to be protected by hardware transactional memory (HTM) while remote accesses acquire a lock using RDMA `rCAS`. This technique only applies to architectures supporting HTM, which is increasingly disabled due to security concerns [16, 20]. Due to cache coherent I/O, a local hardware transaction is aborted whenever a remote thread acquires the lock. Local operations use local accesses in the common case, but a fallback path using RDMA is also required. Another potential option is to leverage RDMA-accessible memory permissioning, which atomically revokes remote access [2, 15], to devise a mutual exclusion algorithm. However, this approach is known to be slow [3] and is not easily made starvation-free since remote access may be continuously revoked by threads performing local accesses.

Recently, new cache coherent interconnects like CXL have attracted attention and are expected to play a role in implementing disaggregated memory patterns. Cache coherence can make it possible to use both RDMA and local atomic operations without the need for an additional synchronization mechanism. However, to fully take advantage of CXL will require RNIC redesign and may still come with a performance tradeoff for coherency. At the current state of the technology (not yet released), these considerations are mostly speculation [11].

8 CONCLUSION

In this paper, we face the challenge of synchronizing accesses in RDMA-based systems and define operation asymmetry to capture the disparate behavior of how remote and local threads operate on RDMA memory. We propose a fair, starvation-free mutual exclusion primitive that enables local and remote accesses to synchronize globally in a manner that is abstracted away from the programmer while optimizing for their individual characteristics. Inspired by lock cohorting, we embed RDMA-aware MCS locks into a modified version of the well-known Peterson's algorithm to create the ALock. To the best of our knowledge, our technique is the first mutual exclusion solution that allows synchronizing local and remote accesses while avoiding known RDMA scalability issues (abuse of RDMA loopback and RPCs).

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-2045976. This research was also funded by a CORE grant from Lehigh University and by a gift grant from the Stellar Dev. Foundation.

REFERENCES

[1] Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. 2018. Passing Messages while Sharing Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, 51–60.

[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. 2019. The Impact of RDMA on Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 409–418. https://doi.org/10.1145/3293611.3331601

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

[4] Amanda Baran and Jacob Nelson-Slivon. 2024. ALock Source Code. https://github.com/sss-lehigh/alock.git.

[5] Amanda Baran, Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri. 2024. *ALock Technical Report.* Technical Report. Lehigh University. https://arxiv.org/abs/2404.17980

[6] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *PVLDB* 9, 7 (2016), 528–539. https://doi.org/10.14778/2904483.2904485

[7] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. 2017. Fast In-Memory Transaction Processing Using RDMA and HTM. *ACM Transactions on Computer Systems* 35, 1 (jul 2017), 1–37. https://doi.org/10.1145/3092701

[8] Andrei Marian Dan, Patrick Lam, Torsten Hoefler, and Martin T. Vechev. 2016. Modeling and Analysis of Remote Memory Access Programming. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 129–144. https://doi.org/10.1145/2983990.2984033

[9] David Dice, Virendra J. Marathe, and Nir Shavit. 2012. Lock Cohorting: A General Technique for Designing NUMA Locks. In *PPoPP '12* (New Orleans, Louisiana, USA). 247–256. https://doi.org/10.1145/2145816.2145848

[10] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. 401–414.

[11] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 287–294. https://www.usenix.org/conference/atc22/presentation/gouk

[12] Danny Hendler, Alex Naiman, Sebastiano Peluso, Francesco Quaglia, Paolo Romano, and Adi Suissa. 2013. Exploiting Locality in Lease-Based Replicated Transactional Memory via Task Migration. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8205)*, Yehuda Afek (Ed.). Springer, 121–133. https://doi.org/10.1007/978-3-642-41527-2_9

[13] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (jan 1991), 124–149. https://doi.org/10.1145/114005.102808

[14] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2021. In *The Art of Multiprocessor Programming (Second Edition)* (second edition ed.). Morgan Kaufmann, Boston. https://doi.org/10.1016/B978-0-12-415950-1.00005-7

[15] InfiniBand Trade Association 2007. *InfiniBand Architecture Specification Volume 1.* InfiniBand Trade Association. https://cw.infinibandta.org/document/dl/8567 Release 1.2.1.

[16] Intel. 2021. Performance Monitoring Impact of Intel Transactional Synchronization Extension Memory Ordering Issue. https://www.intel.com/content/dam/support/us/en/documents/processors/Performance-Monitoring-Impact-of-TSX-Memory-Ordering-Issue-604224.pdf.

[17] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[18] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSD 16)*. 185–201.

[19] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. 2022. Collie: Finding Performance Anomalies in RDMA Subsystems. In *NSDI '22*. Renton, WA, 287–305.

[20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[21] Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann.

[22] Mellanox. 2020. Mellanox Adapters Programmer's Reference Manual (PRM). https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf

[23] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (feb 1991), 21–65. https://doi.org/10.1145/103727.103729

[24] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[25] Jacob Nelson and Roberto Palmieri. 2020. Performance Evaluation of the Impact of NUMA on One-sided RDMA Interactions. In *2020 International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. https://doi.org/10.1109/srds51746.2020.00036

[26] Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri. 2022. Brief Announcement: Asymmetric Mutual Exclusion for RDMA. In *36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 50:1–50:3. https://doi.org/10.4230/LIPIcs.DISC.2022.50

[27] Gary L. Peterson. 1981. Myths About the Mutual Exclusion Problem. *Inform. Process. Lett.* 12 (1981), 115–116.

[28] Robert Ricci, Eric Eide, and The CloudLab Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *USENIX* 39, 6 (Dec. 2014).

[29] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. 2018. Tailwind: Fast and Atomic RDMA-based Replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 851–863. https://www.usenix.org/conference/atc18/presentation/taleb

[30] Mellanox Technologies. 2015. *RDMA Aware Networks Programming User Manual* (1.7 ed.).

[31] The Gen-Z Consortium. 2016. Gen-Z Specifications. https://genzconsortium.org/specifications/.

[32] Xizheng Wang, Guo Chen, Xijin Yin, Huichen Dai, Bojie Li, Binzhang Fu, and Kun Tan. 2021. StaR: Breaking the Scalability Limit for RDMA. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. 1–11. https://doi.org/10.1109/ICNP52444.2021.9651935

[33] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. 2015. HydraDB: A Resilient RDMA-driven Key-value Middleware for In-memory Cluster Computing. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*. IEEE, 1–11.

[34] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 117–135. https://www.usenix.org/conference/osdi20/presentation/wei

[35] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed Lock Management with RDMA: Decentralization without Starvation. In *SIGMOD Conference*. ACM, 1571–1586.

[36] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment* 10, 6 (feb 2017), 685–696. https://doi.org/10.14778/3055330.3055335

[37] S"amy Zehnder. 2015. *A Scalable Distributed Lock Manager using One-Sided RDMA Atomic Operations.* Master's thesis. ETH-Z"urich.

[38] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the 2019 International Conference on Management of Data*. ACM. https://doi.org/10.1145/3299869.3300081