Distributed Lock Management with RDMA: Decentralization without Starvation

Dong Young Yoon
University of Michigan, Ann Arbor
dyoon@umich.edu

Mosharaf Chowdhury
University of Michigan, Ann Arbor
mosharaf@umich.edu

Barzan Mozafari
University of Michigan, Ann Arbor
mozafari@umich.edu

ABSTRACT

Lock managers are a crucial component of modern distributed systems. However, with the increasing availability of fast RDMA-enabled networks, traditional lock managers can no longer keep up with the latency and throughput requirements of modern systems. Centralized lock managers can ensure fairness and prevent starvation using global knowledge of the system, but are themselves single points of contention and failure. Consequently, they fall short in leveraging the full potential of RDMA networks. On the other hand, decentralized (RDMA-based) lock managers either completely sacrifice global knowledge to achieve higher throughput at the risk of starvation and higher tail latencies, or they resort to costly communications in order to maintain global knowledge, which can result in significantly lower throughput.

In this paper, we show that it is possible for a lock manager to be fully decentralized and yet exchange the partial knowledge necessary for preventing starvation and thereby reducing tail latencies. Our main observation is that we can design a lock manager primarily using RDMA's fetch-and-add (FA) operations, which always succeed, rather than compare-and-swap (CAS) operations, which only succeed if a given condition is satisfied. While this requires us to rethink the locking mechanism from the ground up, it enables us to sidestep the performance drawbacks of the previous CAS-based proposals that relied solely on blind retries upon lock conflicts.

Specifically, we present DSLR (**D**ecentralized and **S**tarvation-free **L**ock management with **R**DMA), a decentralized lock manager that targets distributed systems running on RDMA-enabled networks. We demonstrate that, despite being fully decentralized, DSLR prevents starvation and blind retries by guaranteeing first-come-first-serve (FCFS) scheduling without maintaining explicit queues. We adapt Lamport's bakery algorithm [36] to an RDMA-enabled environment with multiple bakers, utilizing only one-sided READ and atomic FA operations. Our experiments show that, on average, DSLR delivers 1.8× (and up to 2.8×) higher throughput than all existing RDMA-based lock managers, while reducing their mean and 99.9% latencies by 2.0× and 18.3× (and up to 2.5× and 47×), respectively.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196890

ACM Reference Format:
Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed Lock Management with RDMA: Decentralization without Starvation. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196890

1 INTRODUCTION

With the advent of high-speed RDMA networks and affordable memory prices, distributed in-memory systems have become increasingly more common [39, 57, 65]. The reason for this rising popularity is simple: many of today's workloads can fit within the memory of a handful of machines, and they can be processed and served over RDMA-enabled networks at a significantly faster rate than with traditional architectures.

A primary challenge in distributed computing is lock management, which forms the backbone of many distributed systems accessing shared resources over the network. Examples include OLTP databases [53, 59], distributed file systems [22, 56], in-memory storage systems [39, 51], and any system that requires synchronization, consensus, or leader election [14, 28, 37]. In a transactional setting, the key responsibility of a lock manager (LM) is ensuring both serializability—or other forms of isolation—and starvation-free behavior of competing transactions [49].

Centralized Lock Managers (CLM)— In traditional distributed databases, each node is in charge of managing the locks for its own objects (i.e., tuples hosted on that node) [3, 8, 25, 34]. In other words, before remote nodes or transactions can read or update a particular object, they must communicate with the LM daemon running on the node hosting that object. Only after the local LM grants the lock can the remote node or transaction proceed to read or modify the object. Although distributed, these LMs are still *centralized*, as each LM instance represents a *central* point of decision for granting locks on the set of objects assigned to it.

Because each CLM instance has *global knowledge* and full visibility into the operations performed on its objects, it can guarantee many strong and desirable properties. For example, it can queue all lock requests and take holistic actions [61, 63], prevent starvation [25, 30, 34], and even employ sophisticated scheduling of incoming requests to bound tail latencies [26, 33, 40, 66].

Unfortunately, the operational simplicity of having global knowledge is not free [23], even with low-latency RDMA networks. First, the CLM itself—specifically, its CPU—becomes a performance bottleneck for applications that require high-throughput locking/unlocking operations and as transactional workloads scale up or out. Second, the CLM becomes a single point of failure [23]. Consequently, many modern distributed systems do not use CLMs in practice; instead, they rely on decentralized lock managers, which

offer better scalability and reliability by avoiding a single point of contention and failure [23].

Decentralized Lock Managers (DLM)— To avoid the drawbacks of centralization and to exploit fast RDMA networks, decentralized lock managers are becoming more popular [15, 18, 47, 62]. This is because RDMA operations enable transactions to acquire and release locks on remote machines at extremely low latencies, *without* involving any remote CPUs. In contrast to CLMs, RDMA-based decentralized approaches offer better CPU usage, scalability, and fault tolerance.

Unfortunately, existing RDMA-based DLMs take an extremist approach, where they either completely forgo the benefits of maintaining global knowledge and rely on blind fail-and-retry strategies to achieve higher throughput [15, 62], or they emulate global knowledge using distributed queues and additional network roundtrips [47]. The former can cause starvation and thereby higher tail latencies, while the latter significantly lowers throughput, undermining the performance benefits of decentralization.

Challenges— There are two key challenges in designing an efficient DLM. First, to avoid data races, RDMA-based DLMs [15, 62] must only rely on RDMA atomic operations: fetch-and-add (FA) and compare-and-swap (CAS). FA atomically adds a constant to a remote variable and returns the previous value of the variable. CAS atomically compares a constant to the remote variable and updates the variable only if the constant matches the previous value. Although CAS does not always guarantee successful completion, it is easy to reason about, which is why all previous RDMA-based DLMs have relied on CAS for implementing exclusive locks [15, 18, 47, 62]. Consequently, when there is high contention in the system, protocols relying on CAS require multiple retries to acquire a lock. These blind and unbounded retries cause starvation, which increases tail latency. Second, the lack of global knowledge complicates other run-time issues, such as deadlock detection and mitigation.

Our Approach— In this paper, we propose a fully Decentralized and Starvation-free Lock management (DSLR) algorithm to mitigate the aforementioned challenges. Our key insight is the following: a distributed lock manager can be fully decentralized and yet exchange the partial knowledge necessary for avoiding blind retries, preventing starvation and thereby reducing tail latencies. Specifically, DSLR adapts Lamport's bakery algorithm [36] to a decentralized setting with RDMA capabilities, which itself poses a new set of interesting challenges:

(1) The original bakery algorithm assumes two unbounded counters per object. However, the current RDMA atomic operations are limited to 64-bit words, which must accommodate two counters—for shared and exclusive locks—in order to implement the bakery algorithm, leaving only 16 bits per counter. Because one is forced to use either RDMA CAS or FA, it is difficult to directly and efficiently apply bounded variants of bakery algorithms [29, 58] in an RDMA context.[1]

(2) To compete with existing RDMA-based DLMs, our algorithm must be able to acquire locks on uncontented objects using a *single* RDMA atomic operation. However, the original bakery algorithm requires setting a semaphore, reading the existing tickets, and assigning the requester the maximum ticket value.

DSLR overcomes all of these challenges (see §4) and, to the best of our knowledge, is the first DLM to extend Lamport's bakery algorithm to an RDMA context.

Contributions— We make the following contributions:

(1) We propose a fully decentralized and distributed locking algorithm, DSLR, that extends Lamport's bakery algorithm and combines it with novel RDMA protocols. Not only does DSLR prevent lock starvation, but it also delivers higher throughput and significantly lower tail latencies than previous proposals.

(2) DSLR provides fault tolerance for transactions that fail to release their acquired locks or fall into a deadlock. DSLR achieves this goal by utilizing *leases* and determining lease expirations using a locally calculated elapsed time.

(3) Through extensive experiments on TPC-C and micro-benchmarks, we show that DSLR outperforms existing RDMA-based LMs; on average, it delivers 1.8× (and up to 2.8×) higher throughput, and 2.0× and 18.3× (and up to 2.5× and 47×) lower average and 99.9% percentile latencies, respectively.

The rest of this paper is organized as follows. Section 2 provides background material and the motivation behind DSLR. Section 3 discusses the design challenges involved in distributed and decentralized locking. Section 4 explains DSLR's algorithm and design decisions for overcoming these challenges. Section 5 describes how DSLR offers additional features often used by modern database systems. Section 6 presents our experimental results, and Section 7 discusses related work.

2 BACKGROUND AND MOTIVATION

This section provides the necessary background on modern high-speed networks, particularly RDMA operations, followed by an overview of existing RDMA-based approaches to distributed lock management. Familiar readers can skip Section 2.1 and continue reading from Section 2.2.

2.1 RDMA-Enabled Networks

Remote Direct Memory Access (RDMA) is a networking protocol that provides direct memory access from a host node to the memory of remote nodes, and vice versa. RDMA achieves its high bandwidth and low latency with no CPU overhead by using *zero-copy transfer* and *kernel bypass* [52]. There are several RDMA implementations, including InfiniBand [5], RDMA over Converged Ethernet (RoCE) [1], and iWARP [7].

2.1.1 RDMA Verbs and Transport Types. Most RDMA implementations support several operations (verbs) that can broadly be divided into two categories:

[1] Existing bounded bakery algorithms only support exclusive locks. Also, they either need additional memory and extra operations [58] or rely on complex arithmetics (e.g., modulo [29]) beyond the simple addition offered by FA.

(1) **Two-Sided Verbs (Verbs with Channel Semantics).** SEND and RECV verbs have channel semantics, meaning a receiver must publish a RECV verb (using RDMA API) prior to a sender sending data via a SEND verb. These two verbs are called *two-sided* as they must be matched by both sender and receiver. These verbs use the remote node's CPU, and thus have higher latency and lower throughput than one-sided verbs [42].

(2) **One-Sided Verbs (Verbs with Memory Semantics).** Unlike the two-sided verbs, READ, WRITE, and atomic verbs (CAS and FA) have memory semantics, meaning they specify a remote address on which to perform data operations. These verbs are *one-sided*, as the remote node's CPU is not aware of the operation. Due to their lack of CPU overhead, one-sided verbs are usually preferred over two-sided verbs [31, 42]. However, the best choice, in terms of which verb to use, always depends on the specific application.

RDMA operations take place over RDMA connections, which are of three transport types: *Reliable Connection (RC)*, *Unreliable Connection (UC)*, and *Unreliable Datagram (UD)*. With RC and UC, two queue pairs need to be *connected* and explicitly communicating with each other.

In this paper, we focus our scope on Reliable Connection (RC), as it is the only transport type that supports atomic verbs, which we use extensively to achieve fully decentralized lock management in Section 4. Next, we focus on atomic verbs in more detail.

2.1.2 Atomic Verbs. RDMA provides two types of atomic verbs, compare-and-swap (CAS) and fetch-and-add (FA). These verbs are performed on 64-bit values. For CAS, a requesting process specifies a 64-bit *new value* and a 64-bit *compare value* along with a remote address. The value (i.e., *original value*) at the remote address is compared with the *compare value*; if they are equal, the value at the remote address is swapped with the *new value*. The *original value* is returned to the requesting process. For FA, a requesting process specifies a value to be added (i.e., *increment*) to a remote address. The *increment* is added to the 64-bit *original value* at the remote address. Similar to CAS, the *original value* is returned to the requesting process.

Atomic verbs have two important characteristics that dictate their usage and system-level design decisions:

- Atomic verbs are guaranteed to never experience data races with other atomic verbs.
- The guarantee does not hold between atomic and non-atomic operations. For example, a data race can still occur between CAS and WRITE operations [4].

These characteristics effectively restrict how one can mix and match these verbs in their design, which is evident in existing RDMA-based lock management solutions—they all rely heavily on CAS [15, 18, 47, 62]. A CAS operation will only succeed if its condition is satisfied (i.e., the compare value equals the previous value). This characteristic can lead to unbounded and blind retries, which can severely impact performance and cause starvation. For this reason, our approach avoids the use of CAS as much as possible; rather, we primarily rely on FA which, unlike CAS, is guaranteed to succeed. This also solves the issue of lock starvation, as we explain in Section 3.1.

2.2 Distributed Lock Managers

2.2.1 Traditional Distributed Lock Managers. Before discussing RDMA-based distributed lock managers, we briefly review the architecture of a traditional distributed lock manager [25, 34]. Typically, each node runs a lock manager (LM) instance (or daemon), in charge of managing a *lock table* for the objects stored on that node. Each object (e.g., a tuple in the database) is associated with a *lock object* in the lock table of the node that it is stored on.[2] Before reading or modifying a tuple, a transaction must request a (shared or exclusive) lock on it. The LM instance running on the local node communicates the transaction's lock request to the LM instance running on the remote node hosting the object [25, 34]. The transaction can proceed only after the corresponding (i.e., remote) LM has granted the lock. The locks are released upon commit/abort in a similar fashion, by going through the remote LM. As discussed in Section 1, these traditional LMs are distributed but centralized, in the sense that each LM represents a single point of decision for granting the locks on its associated objects. Next, we discuss distributed and decentralized LMs.

2.2.2 RDMA-Based Distributed & Decentralized Lock Managers. Unlike traditional distributed lock managers, in RDMA-based DLMs,[3] a local LM (acting on behalf of a transaction) can directly access lock tables in remote nodes instead of going through the remote DLM instance.[4] While this improves performance in most cases, it also requires new RDMA-aware protocols for lock management [15, 18, 47, 62].

To the best of our knowledge, almost all RDMA-based DLMs use atomic verbs (instead of SEND/RECV) for two main reasons. First, the SEND/RECV verbs are avoided due to their channel semantics; the solution would be no different than traditional client/server-based solutions with CPU involvement. Second, the READ/WRITE verbs cannot be used alone due to their vulnerability to data races, which can jeopardize the consistency of the lock objects themselves. Consequently, previous proposals have all used CAS [18, 62], or combined CAS with FA atomic verbs [15, 47], depending on the data structure used for modeling their lock objects.

Lock Representation and Acquisition— Since RDMA atomic verbs operate on 64-bit values, all RDMA-based DLMs use 64-bit values to represent their lock objects, but with slight variations. For example, Devulapalli et al. [18] implement only exclusive locks and use the entire 64-bit value of a lock object to identify its *owner* (i.e., the transaction currently holding the lock). Others [15, 47] implement both shared and exclusive locks by dividing the 64-bit into two 32-bit regions (Figure 1). In these cases, the upper 32-bit region represents the state of an exclusive lock, and DLMs use CAS to change this value to record the current [15, 18] or last [47] exclusive lock owner of the object; the lower 32-bit region represents the state of shared locks, and DLMs use FA to manipulate this value to count the current number of shared lock owners.

[2] For simplicity of presentation, here we assume a single primary copy for each tuple.
[3] Note that we use DLM as an acronym for a Decentralized Lock Manager, rather than a Distributed Lock Manager.
[4] Some combine both architectures by using one-sided RDMA for lock acquisition but relying on additional messages (similar to traditional models) to address lock conflicts [47]. These protocols, however, suffer under contended workloads (see §6.2).

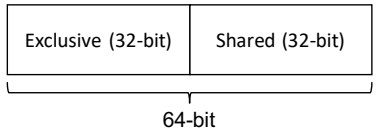


Figure 1: The 64-bit representation of a lock object L used by previous RDMA protocols [15, 47].

Advisory Locking— Note that RDMA-based DLMs typically use *advisory locking*, meaning participants cooperate and follow/obey a locking protocol without the lock manager enforcing it (i.e., *mandatory locking*). This is because one-sided atomic verbs interact with lock tables without involving local DLM instances.

Handling Lock Conflicts— Perhaps the most important aspect of any lock manager is how it handles lock conflicts. There are three possible scenarios of a lock conflict:

(a) **Shared → Exclusive**
(b) **Exclusive → Shared**
(c) **Exclusive → Exclusive**

For example, (a) occurs when a transaction attempts to acquire an exclusive lock on an object that others already have a shared lock on. To handle lock conflicts, DLMs typically use one or a combination of the following mechanisms:

(1) *Fail/retry:* a transaction simply fails on conflict and continues trying until the acquisition succeeds [15, 62].
(2) *Queue/notify:* lock requests with conflict are enqueued. Once the lock becomes available, a DLM instance or the current lock owner notifies the requester to proceed [18, 47].

Which of these mechanisms is used has important ramifications on the design and performance of a DLM, as discussed next.

3 DESIGN CHALLENGES

Since they lack a centralized queue, DLMs face several challenges:

(C1) Lock starvation caused by lack of global knowledge.
(C2) Fault tolerance in case of transaction failures.
(C3) Deadlocks due to high concurrency.

We discuss each of these challenges in the following section.

3.1 Lock Starvation

Without a central coordinator, DLMs must rely on their partial knowledge for lock acquisition and handling lock conflicts. Due to their lack of global knowledge, existing RDMA-based DLMs utilize `CAS` and `FA` with blind retries for lock acquisition, which makes them vulnerable to lock starvation.

Lock starvation occurs when a protocol allows newer requests to proceed before the earlier ones, causing the latter to wait indefinitely. The starved transactions might themselves hold locks on other tuples, thus causing other transactions to starve for those locks. Through this cascading lock-wait, lock starvation can cause severe performance degradation and significantly increase tail latencies. Existing DLMs allow for at least one or both of the following types of lock starvation (see Appendix A.1 for examples of both types of lock starvation):

(i) **Reader-Writer Starvation:** multiple readers holding shared locks starve a writer from acquiring an exclusive lock.
(ii) **Fast-Slow Starvation:** faster nodes starve slower nodes from acquiring a lock.

3.2 Fault Tolerance

DLMs must be able to handle transaction failures (e.g., due to application bugs/crashes, network loss, or node failures), whereby a transaction fails without releasing its acquired locks. As mentioned earlier, RDMA-based DLMs utilize one-sided atomic verbs that do not involve local DLM instances. This makes it difficult for the local DLM to detect and release the unreleased locks on behalf of the failed (remote) transaction. Under *advisory locking*, other transactions will wait indefinitely until the situation is somehow resolved. In several previous RDMA locking protocols [15, 18, 47], a local DLM does not have enough information on its lock table to handle transaction failures. Wei et al. [62] use a *lease* [24] as a fault tolerance mechanism that allows failed transactions to simply expire, allowing subsequent transactions to acquire their locks. However, their approach uses a lease only for shared locks, and cannot handle transactions that fail to release their exclusive locks.

3.3 Deadlocks

Deadlocks can happen between different nodes (and their transactions) in any distributed context. However, deadlock detection and resolution can become more difficult without global knowledge as the number of nodes increases. Furthermore, in some cases, the negative impact of deadlocks on performance can be more severe with faster RDMA networks. This is because one can process many more requests as network throughput increases using RDMA, and assuming a deadlock resolution takes a similar amount of time regardless of network speed, deadlocks can incur a relatively larger penalty on transaction throughput with faster networks.

In the next section, we present our algorithm that overcomes the three aforementioned challenges.

4 OUR ALGORITHM

In this section, we present `DSLR`, a fully decentralized algorithm for distributed lock management using fast RDMA networks. The high-level overview of how `DSLR` works is shown in Figure 2. Based on the challenges outlined in Section 3, we start by highlighting the primary design goals of our solution. Next, we describe `DSLR` in detail, including how it represents locks, handles locking/unlocking operations, and resolves lock conflicts.

4.1 Assumptions

In the following discussion, we rely on two assumptions. First, the system clock in each node is well-behaved, meaning none of them advance too quickly or too slowly, and the resolution of system clock in each node is at least ϵ, which is smaller than the maximum lease time that `DSLR` uses (i.e., 10 ms). This is similar to the assumption in [24], except that we do not require the clocks to be synchronized. Second, the lock manager on each node has prior knowledge of the location of data resources (and their corresponding lock objects) in the cluster. This can be achieved either by using an off-the-shelf directory service for the resources in the cluster (e.g., name server) [17, 20] or by peer-to-peer communications between the

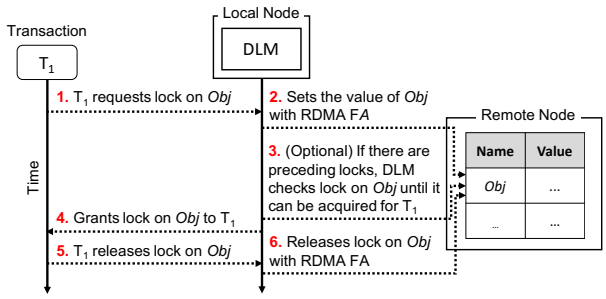


Figure 2: A high-level overview of lock acquisition and release in DSLR.

lock managers on-the-fly. The details of this process are, therefore, outside the scope of this paper.

4.2 Design Criteria

As discussed earlier in Section 3, a decentralized DLM faces several challenges, including lock starvation (C1), faults caused by transaction failures (C2), and deadlocks (C3). Next, we explain how our design decisions differ from those of previous DLMs and how they enable us to overcome the aforementioned challenges.

4.2.1 Representation of a Lock Object. As mentioned in Section 2.2.2, most RDMA-based DLMs split a 64-bit word into two 32-bit regions to represent shared and exclusive locks on an object. Unfortunately, algorithms using this representation rely on the use of CAS, which makes them vulnerable to lock starvation (C1). To solve this problem, we develop a new representation using four 16-bit regions instead of two 32-bit regions as part of our RDMA-based implementation of Lamport's bakery algorithm. We explain the details of our lock object representation in Section 4.4.

4.2.2 RDMA Verbs. Previous RDMA-based DLMs rely heavily on CAS to change the value of a lock object as they acquire or release a lock. This causes lock starvation (C1) because, as demonstrated in Section 3.1, if the value of a lock object keeps changing, a DLM blindly retrying with CAS will continuously fail. Instead of using CAS, DSLR uses FA—which, unlike CAS, is guaranteed to succeed—to acquire and release locks with a single RDMA operation. We describe how we use FA and READ for lock acquisition and handling of lock conflicts in Section 4.5 and 4.6.

4.2.3 Handling Transaction Failures and Deadlocks. To the best of our knowledge, existing RDMA-based DLMs have largely overlooked the issue of transaction failures (C2) and deadlocks (C3). To handle transaction failures, we propose the use of a *lease* [24]. (Note that DrTM [62] also uses a lease for shared locks, but DSLR utilizes it specifically for determining transaction failures. Also, the lease expiration in DSLR is determined locally without the need for synchronized clocks.) We also employ a timeout-based approach to handle deadlocks, utilizing our lease implementation. In addition, we adopt a well-known technique from networking literature, called *random backoffs*, in our bakery algorithm (we explain this technique in Appendix A.3).

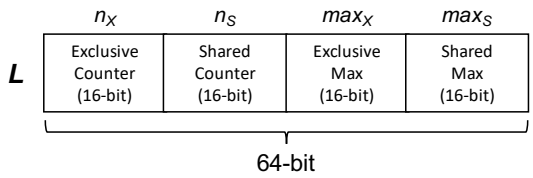


Figure 3: DSLR's 64-bit representation of a lock object L.

4.3 Lamport's Bakery Algorithm

Before introducing DSLR, we provide a brief background on Lamport's bakery algorithm [36]. Lamport's bakery algorithm is a mutual exclusion algorithm, designed to prevent multiple threads from concurrently accessing a shared resource. In his algorithm, Lamport essentially models a bakery, where each customer entering the bakery receives a ticket with a number that is monotonically increasing. This number is incremented each time a customer enters the bakery. In addition to the ticket numbers, there is also a *global counter* in the bakery, showing the ticket number of the current customer being served, and once the customer is done, this global counter is incremented by 1. The next customer who will be served by the bakery will be the one whose ticket number matches the current value of the global counter, and so on. In the subsequent sections, we describe how DSLR modifies this original bakery algorithm in an RDMA context with both shared and mutual (i.e., exclusive) locks. We formally prove DSLR's starvation-free behavior in Appendix A.2.

4.4 Lock Object Representation

Figure 3 shows a 64-bit representation of a lock object that DSLR uses in its RDMA-specific variant of the bakery algorithm, which takes the form of $\{n_X, n_S, max_X, max_S\}$. Using the bakery analogy from Lamport's original algorithm, the upper 32 bits, n_X and n_S, are equivalent to global counters, showing the largest ticket numbers of customers (in our case, transactions) that are currently being served for exclusive and shared locks, respectively. The lower 32 bits, max_X and max_S, are equivalent to the next ticket numbers that an incoming customer will receive for exclusive and shared locks, respectively. By simply incrementing max_X or max_S using FA and getting its original value, a transaction obtains a *ticket* with the current max numbers; then it only needs to wait until the corresponding *counter* value (i.e., n_X or n_S) becomes equal to the number on its obtained ticket.

Note that the original bakery algorithm assumes unbounded counters. However, we are restricted to a 16-bit space (i.e., a maximum of 65,535) to store the value of each counter. The challenge here—if we keep incrementing the values—is an overflow, making the state of in-flight transactions invalid. DSLR circumvents this problem by periodically resetting these counters before one can overflow (see §4.9). While doing so, it ensures that all other transactions will wait until the reset is properly done, abiding by the advisory locking rules of DSLR. We will explain how this process works in more detail in subsequent sections.

4.5 Lock Acquisition

Our algorithm uses a single FA operation to acquire a lock, or simply queue up for it by adding 1 to max_X or max_S without having to directly communicate with other nodes in the cluster. Algorithm 1 presents the corresponding pseudocode. Figure 4 demonstrates an

```
function AcquireLock(tid, L, mode)
    Inputs : tid: the id of the requesting transaction
             L: the requested lock object
             mode: lock mode (shared, exclusive)
    Global : ResetFrom[tid, L]: the value of L that tid needs
             to use for resetting L
             (accessible by all transactions in the local node)
    Consts : COUNT_MAX = 32,768
    Output : Success or Failure
 1  ResetFrom[tid, L] ← 0
 2  if mode = shared then
 3      prev ← FA (L, max_S, 1)
 4      if prev(max_S) ≥ COUNT_MAX OR
          prev(max_X) ≥ COUNT_MAX then
 5          FA (L, max_S, -1)
 6          Performs RandomBackoff
 7          if prev(n_S) or prev(n_X) has not changed for
              longer than twice the lease time since last
              failure then
 8              Reset L // Refer to Lines 11–19 in
                        // HandleConflict
 9          return Failure
10      else if prev(max_S) = COUNT_MAX-1 then
11          ResetFrom[tid, L] = {prev(max_X), COUNT_MAX,
                                 prev(max_X), COUNT_MAX}
12      if prev(n_X) = prev(max_X) then
13          return Success
14      else
15          return HandleConflict(tid, L, prev, mode)
16  else if mode = exclusive then
17      prev ← FA (L, max_X, 1)
18      if prev(max_S) ≥ COUNT_MAX OR
          prev(max_X) ≥ COUNT_MAX then
19          FA (L, max_X, -1)
20          Performs RandomBackoff
21          if prev(n_S) or prev(n_X) have not changed for
              longer than twice the lease time since
              last failure then
22              Reset L // Refer to Lines 11–19 in
                        // HandleConflict
23          return Failure
24      else if prev(max_X) = COUNT_MAX-1 then
25          ResetFrom[tid, L] = {COUNT_MAX, prev(max_S),
                                 COUNT_MAX, prev(max_S)}
26      if prev(n_X) = prev(max_X) AND
          prev(n_S) = prev(max_S) then
27          return Success
28      else
29          return HandleConflict(tid, L, prev, mode)
end function
```

Algorithm 1: The pseudocode for the *AcquireLock* function (see Table 1 for procedure definitions).

example, where a transaction with tid = 3 (i.e., T_3) wants to acquire an exclusive lock on a lock object L, and L at the time has the value

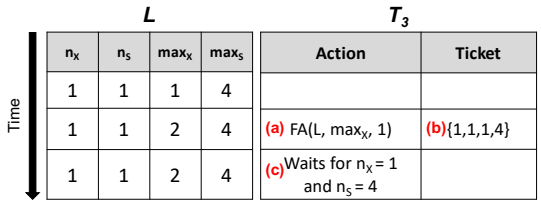


Figure 4: An example of a transaction T_3 acquiring an exclusive lock with DSLR on a lock object L.

Function	Description
val(segment)	represents the value of segment in the 64-bit value val. A segment is one of n_X, n_S, max_X, or max_S.
prev←CAS (L, current, new)	runs CAS on L, changing it from current to new only if the value of L is current. The returned value, prev, contains the original value of L before CAS.
prev←FA (L, segment, val)	runs FA on L, adding val to segment of L. The returned value, prev, contains the original value of L before FA. For example, FA (L, max_X, 1) adds 1 to max_X of L.
val←READ (L)	runs RDMA READ on L and returns its value.

Table 1: List of notations and procedures used by DSLR.

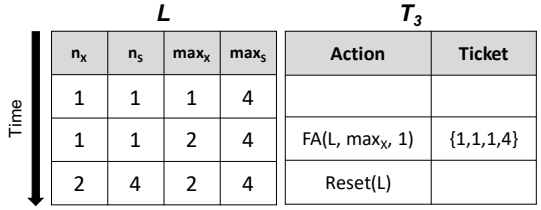


Figure 5: An example of a transaction T_3 resetting a lock object L to resolve a deadlock.

of {1, 1, 1, 4}. Then, T_3 needs to increment max_X of L by 1 to get a ticket with the next maximum number. This, **(a)** in Figure 4, will set L = {1, 1, 2, 4} and T_3 will have the ticket, **(b)** in Figure 4. For exclusive locks, T_3 needs to wait for both shared and exclusive locks preceding it on L. By comparing the values on its ticket and the current n_X and n_S on L, T_3 knows that there are currently $max_S - n_S$ = 3 transactions waiting or holding shared locks on L; thus, T_3 will wait for them, **(c)** in Figure 4. Here, the *HandleConflict* function is called subsequently (explained in the next section). Similarly, if T_3 wants to acquire a shared lock on L, it needs to increment max_S and wait until $prev(max_X) = n_X$.

DSLR has an additional logic in place to reset segments of a lock object before they overflow, and to ensure that other transactions take their hands off while one is resetting the value (*Lines 4–9* and *Lines 18–23* in Algorithm 1 for shared and exclusive locks, respectively). This logic enables incoming transactions to reset the counters if necessary, hence preventing situations where they would wait indefinitely for other failed transactions to reset the counters. We describe this resetting procedure in Section 4.9.

```
function HandleConflict(tid, L, prev, mode)
    Inputs : tid: ID of the requesting transaction
             L: the requested lock object
             prev: the value of the lock object at time of FA
             mode: lock mode (shared, exclusive)
    Output : Success or Failure
1   while true do
2       val ← READ (L)
3       if prev(max_X) < val(n_X) or
                prev(max_S) < val(n_S) then
4           return Failure
5       if mode = shared then
6           if prev(max_X) = val(n_X) then
7               return Success
8       else if mode = exclusive then
9           if prev(max_X) = val(n_X) and
              prev(max_S) = val(n_S) then
10              return Success
11      if val(n_X) or val(n_S) have not changed
            for longer than twice the lease time then
12          if mode = shared then
13              reset_val ← {prev(max_X),
                prev(max_S) + 1, val(max_X), val(max_S)}
14          else if mode = exclusive then
15              reset_val ← {prev(max_X) + 1,
                prev(max_S), val(max_X), val(max_S)}
16          if CAS(L, val, reset_val) succeeds then
17              if reset_val(max_X) ≥ COUNT_MAX OR
                  reset_val(max_S) ≥ COUNT_MAX then
18                  Reset L to zero // Refer to Lines 6–7 in
                                    // Algorithm 3
19              return Failure
20      wait_count ← (prev(max_X) − val(n_X)) +
                     (prev(max_S) − val(n_S))
        Wait for (wait_count × ω) μs
end function
```

Algorithm 2: Pseudocode of the *HandleConflict* function (see Table 1 for procedure definitions).

4.6 Handling Lock Conflicts

In our algorithm, a lock conflict occurs when a transaction finds that the current *counter*s of L are less than the unique numbers on its assigned ticket, meaning there are other preceding transactions either holding or awaiting locks on the lock object L. DSLR determines this by examining the return value of FA (i.e., prev) and calling *HandleConflict* if there is a lock conflict. Algorithm 2 is the pseudocode for the *HandleConflict* function. Remember that prev is the value of L right before the execution of FA. Here, the algorithm continues polling the value of L until it is tid's turn to proceed with L by comparing the current *counter*s of L with the numbers of its own ticket (*Lines 5–7* for shared, *Lines 8–10* for exclusive locks in Algorithm 2). DSLR detects transaction failures and deadlocks when it still reads the same *counter* values even after twice the length of the proposed lease time has elapsed (*Lines 11–19*). This is determined locally by calculating the time elapsed since the last read from the same *counter* values. This function returns Failure only when transaction failures or deadlocks are detected by DSLR and the counters are already reset. In such a case, the transaction can retry by calling the *AcquireLock* function again. DSLR also avoids busy polling by waiting a certain amount of time proportional to the number of preceding tickets. Specifically, DSLR calculates the sum of the number of preceding exclusive and shared tickets (i.e., wait_count in *Line 20*). Then, it waits for this sum multiplied by a default wait time ω, which can be tuned based on the average RDMA latency of the target infrastructure (5 μs in our cluster). This technique is similar to the dynamic interval polling idea used in [55], which reduces network traffic by preventing unnecessary polling (we study the effectiveness of DSLR's dynamic interval polling in Appendix A.7). Next, we explain how DSLR handles such failures and deadlocks in the *HandleConflict* function.

4.7 Handling Failures and Deadlocks

When DSLR detects transaction failures or deadlocks by checking *counter* values for the duration of the proposed lease time, it calls the *HandleConflict* function to reset the *counter* values of L on behalf of tid (*Line 16*).

Note that, where there is a risk of an overflow (i.e., *counter* reaching COUNT_MAX), tid will also be responsible for resetting. After the reset, tid fails with the lock acquisition. It releases all locks acquired thus far and retries from the beginning. For example, suppose T_3 detects a deadlock and wants to reset L, as shown in Figure 5. T_3 basically resets L with CAS such that the next transaction can acquire a lock on L, and this resolves the deadlock. T_3 and other transactions that were waiting on L must retry after the reset. The beauty of this mechanism is that a deadlock will be resolved as long as any waiting transactions (say T_W) reset the *counter* of L, where transactions before T_W can simply retry while transactions after T_W can continue with acquiring locks on L. Note that DSLR detects and handles other types of failures, such as transaction aborts and node failures, using a different mechanism. Specifically, transaction aborts are handled in the same fashion as normal transactions; when a transaction is aborted, DSLR simply releases all its acquired locks. However, to detect node failures (which are less common) or a loss of RDMA connections between the nodes, DSLR relies on sending heartbeat messages regularly (10 seconds by default) between the nodes in the cluster and checking the event status of the message from the RDMA completion queues (CQs). We describe the details of the *ReleaseLock* function in the next section.

4.8 Lock Release

Releasing a lock object L with DSLR is as simple as incrementing n_X or n_S with FA, unless the lease has already expired. Algorithm 3 is the pseudocode for the *ReleaseLock* function. An extra procedure is only needed if the transaction unlocking the lock object happens to also be responsible for resetting the value of L in order to prevent overflows. This is determined by inspecting the value of ResetFrom[tid, L], which would have been set during *AcquireLock*, if tid is required to perform the resetting of L. In that case, tid will increment *counter*, even if the lease has expired, since it has to reset the value of L. Next, we explain how DSLR resets counters of a lock object to prevent overflows.

```
function ReleaseLock(tid, L, elapsed, mode)
    Inputs : tid: ID of the requesting transaction
             L: the requested lock object
             elapsed: the time elapsed since the lock acquisition
                      of L
             mode: lock mode (shared, exclusive)
    Global : ResetFrom[tid, L]: the value of L that tid needs
             to use for resetting L
             (accessible by all transactions in the local node)
    Output : Success
1   if (elapsed is less than the lease time) or
       (ResetFrom[tid, L] > 0) then
2       if mode = shared then
3           val ← FA (L, n_S, 1)
4       else if mode = exclusive then
5           val ← FA (L, n_X, 1)
6       if ResetFrom[tid, L] > 0 then
7           Repeat CAS(L, ResetFrom[tid, L], 0) until it
                succeeds
            ResetFrom[tid, L] ← 0
8   return Success
end function
```

Algorithm 3: Pseudocode of the *ReleaseLock* function (see Table 1 for procedure definitions).

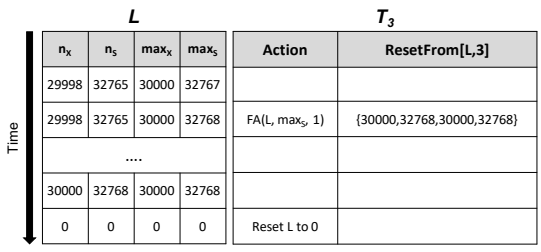


Figure 6: An example of a transaction T_3 resetting a lock object L to avoid overflow of counters.

4.9 Resetting Counters

In DSLR, we have a hard limit of COUNT_MAX, which is $2^{15} = 32,768$, for each 16-bit segment of a lock object L. In other words, DSLR only allows counters to increase until halfway through their available 16-bit space. This is identical to the commonly-used buffer overflow protection technique with a canary value to detect overflows [16]. In our case, DSLR uses the 16^{th} most significant bit as a canary bit to reset the value before it actually overflows.

For example, suppose T_3 wants to acquire an shared lock on L, as shown in Figure 6. After performing FA on L, T_3 receives prev = {29998, 32765, 30000, 32767}. Now, max_S of prev is 32,767, which is COUNT_MAX-1, meaning (max_S) of the object L has reached the limit COUNT_MAX. At this point, DSLR waits until T_3 and all preceding transactions are complete by setting ResetFrom[3, L] to {30000, 32768, 30000, 32768}. When T_3 releases its lock on L, DSLR resets the value of L to 0 from ResetFrom[3, L] with CAS. Note that once either max_X or max_S reaches COUNT_MAX, no other transactions can acquire the lock until it is reset. If a transaction detects such a case (i.e., *Line 3 and 17* in Algorithm 1), it reverses the previous FA by decrementing either max_X or max_S and performs a random backoff to help the resetting process. Our use of a random backoff ensures that the repeating CAS will eventually avoid other FAs and reset the counter without falling into an infinite loop. In fact, previous work [35] has formally shown that a network packet can avoid collisions with other packets (and be successfully transmitted over the network) with a bounded number of retries using a random backoff, assuming there is a finite number of nodes in the cluster. Similarly, the expected number of CAS retries to avoid other FAs, and successfully reset the counter, will also be bounded (see Appendix A.3 for details).

Note that the use of a 16-bit space means that, at least in theory, the value of a lock object L can still overflow if there are more than 32,767 transactions trying to acquire a lock on the **same object** at the **same time**. However, this situation is highly unlikely in practice with real-world applications. Nonetheless, we only utilize CAS for resetting counters and avoid redundant CAS calls, unlike previous approaches, since our first FA simultaneously acquires the lock successfully or enqueues for the lock in case of conflicts.

5 SUPPORTING ADDITIONAL CAPABILITIES

In this section, we explain how DSLR supports some additional features that are often needed by modern database systems.

5.1 Support for Long-Running Transactions

Mixed workloads are increasingly common [46, 50], also known as hybrid transactional/analytical processing (HTAP). The use of a fixed lease time can lead to penalizing long-running queries or transactions. To allow such transactions to complete before their lease expires, we use a *multi-slot leasing* mechanism to support varying lease times. Specifically, to request a longer lease, a transaction can add a number larger than 1 (say k) to the next ticket number (i.e., max_X or max_S). Here, *Line 3* of Algorithm 1 changes from FA (L, max_S, 1) to FA (L, max_S, k) for shared locks. *Line 17* changes similarly for exclusive locks. Our lease expiration logic is also changed accordingly, whereby each transaction determines its own expiration time based on its own ticket numbers rather than a fixed lease duration for all transactions. In other words, the lease expiration will be proportional to δ, where δ is the difference between the current and the next ticket numbers (i.e., $\delta = (\text{prev}(\text{max}_X) - \text{prev}(n_X)) + (\text{prev}(\text{max}_S) - \text{prev}(n_S))$). Similarly, the transaction releases its acquired lock by adding k instead of 1 to n_S or n_X (i.e., *Lines 3 and 5* of Algorithm 3). Note that DSLR can prevent unfairly long durations by imposing the maximum value of k that can be used by any transaction.

With this multi-slot leasing, a long-running transaction is effectively obtaining multiple tickets with consecutive numbers, while other transactions infer its lease expiration time based on the number of outstanding tickets shown on their own tickets. The maximum lease time possible will be $\phi \times \omega$, where ϕ is the remaining ticket numbers in L and ω is the default wait time. This means we can always tune ω to accommodate longer transactions, even when ϕ is small (i.e., there are few remaining tickets). Therefore, multi-slot leasing practically eliminates a hard limit on how long a transaction can remain in the system, thereby allowing long-running transactions to run successfully. We study the effectiveness of this technique in Section 6.5.

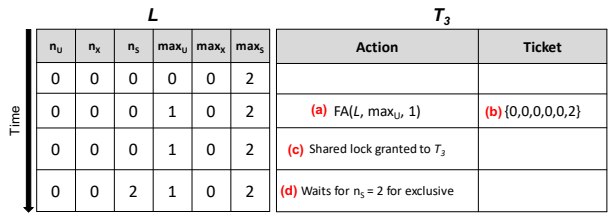


Figure 7: An example of a transaction T_3 acquiring an update lock on a lock object L.

5.2 Lock Upgrades

Many modern databases support lock upgrades. For example, a SQL statement "SELECT ... FOR UPDATE" would acquire shared locks on its target rows to read them first, and then later upgrade those shared locks to exclusive ones after draining the existing shared locks (held by other transactions) so that it can update those rows.

DSLR supports lock upgrades by implementing a third type of locks, i.e., *update* locks. An update lock is similar to an exclusive lock, except that a transaction can acquire an update lock even when other transactions already have shared locks on that object. In the presence of those shared locks, the transaction with the update lock (say T_U) can only read the object. Once all other shared locks on that object are released, T_U is finally allowed to write to the object. Other shared and exclusive lock requests that arrive after an update lock has been granted must wait for the update lock to be released.

To implement update locks, we simply introduce two new ticket counters, n_U and max_U. This means we must divide our 64-bit lock object L between six counters (rather than four). For example, suppose the transaction T_3 wants to acquire an update lock on a lock object L, as shown in Figure 7. Following the same lock acquisition procedure in Section 4.5, T_3 takes its ticket by adding 1 to max_U **((a)** in Figure 7). Even though there are already two other transactions with shared locks on L (**(b)** in Figure 7), T_3 is still granted a shared lock (**(c)** in Figure 7). Once the other two transactions release their shared locks, T_3 is granted an exclusive lock on L (**(d)** in Figure 7).

6 EVALUATION

In this section, we empirically evaluate our proposed algorithm, DSLR, and compare it with other RDMA-based approaches to distributed locking. Our experiments aim to answer several questions:

(i) How does DSLR's performance compare against that of existing algorithms? (§6.2)

(ii) How does DSLR scale as the number of lock managers increases? (§6.3)

(iii) How does DSLR's performance compare against that of queue-based locking in the presence of long-running reads? (§6.4)

(iv) How does DSLR support long-running reads effectively with its multi-slot leasing mechanism? (§6.5)

Additional experiments are deferred to Appendix A.

6.1 Experiment Setup

Hardware— For our experiments, we borrowed a cluster of 32 *r320* nodes from the Apt cluster (part of NSF CloudLab infrastructure for scientific research [2]), each equipped with an Intel Xeon E5-2450 processor with 8 (2.1Ghz) cores, 16GB of Registered DIMMs running at 1600Mhz, and running Ubuntu 16.04 with Mellanox OFED driver 4.1-1.0.2.0. The nodes were connected with ConnectX-3 (1x 56 Gbps InfiniBand ports) via Mellanox MX354A FDR CX3 adapters. The network consisted of two core and seven edge switches (Mellanox SX6036G), where each edge switch was connected with 28 nodes and connected to both core switches with a 3.5:1 blocking factor.

Baselines— For a comparative study of DSLR, we implemented the following previous RDMA-based, distributed locking protocols:

(1) **Traditional** is traditional distributed locking, whereby each node is in charge of managing the locks for its own objects [25, 34]. Although distributed, this approach is still centralized, since each LM instance is a central point of decision for granting locks on the set of objects assigned to it. That is, to acquire a lock on an object, a transaction must communicate with the LM instance in charge of that node. This mechanism uses two-sided RDMA SEND/RECV verbs with queues.

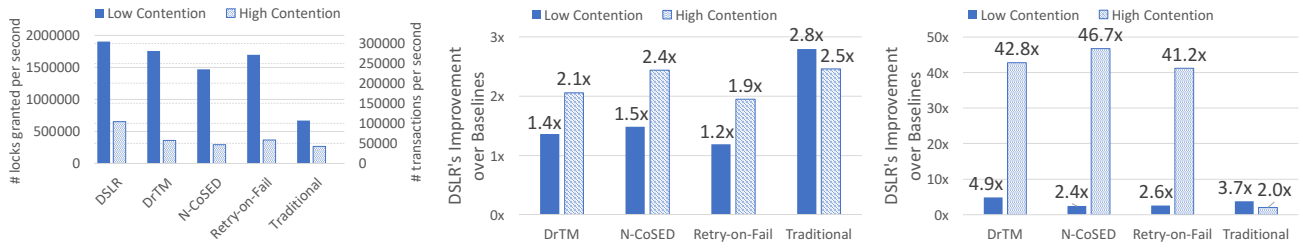
(2) **DrTM** [62] is a decentralized algorithm, which uses CAS for acquiring both exclusive and shared locks. This protocol implements a *lease* for shared locks, providing a time period for a node to hold the lock. In case of lock conflicts, exclusive locks are retried with CAS, and shared locks are retried if the lease has expired. In our experiment, we follow the guidelines provided in their paper for specifying the lease duration.

(3) **Retry-on-Fail** [15] is another decentralized algorithm, which uses CAS for exclusive and FA for shared lock acquisition. Their protocol simply retries in all cases of lock conflicts. Although this work is not published, their approach represents an important design choice (i.e., always retry), which merits an empirical evaluation in our experiments. (We refer to this protocol as Retry-on-Fail, as it was not named in their report.)

(4) **N-CoSED** [47] uses CAS for exclusive and FA for shared lock acquisition. While decentralized, it still tries to obtain global knowledge by relying on distributed queues and extra 'lock request/grant' messages between the DLMs upon lock conflicts.

Implementation— We implemented all baselines in C/C++ (none of them had a readily available implementation). For RDMA, we used *libibverbs* and *librdmacm* libraries with OpenMPI 1.10.2 [6]. Since none of the baselines had a mechanism to handle deadlocks, we also implemented DSLR's timeout-based approach for all of them. For each experiment, we varied the timeout parameter (i.e., maximum number of retries) for each baseline, and only reported their best results in this paper.

Servers & Clients— In each experiment, we used one set of machines to serve as LMs and a separate set as clients. Each client machine relied on four worker threads to continuously generate transactions by calling stored procedures on one of the nodes. The LM on that node would then acquire the locks on behalf of the transaction, either locally or on remote nodes. For remote locks, a CLM would contact other CLMs in the cluster, whereas a DLM would acquire remote locks directly. Once all the locks requested by a transaction were acquired, the transaction committed after a *think*



(a) Throughput (b) Average latency (c) 99.9% latency

Figure 8: Performance comparison of different distributed lock managers under TPC-C (low and high contention).

time of γ and released all its locks. Unless specified otherwise, we used $\gamma = 0$. The data on each node was entirely cached in memory, while the redo logs were written to disk. Each experiment ran for 5 minutes, which was sufficiently large to observe the steady-state performance of our in-memory prototype.

Workloads— We experimented with two workloads, the well-known TPC-C benchmark and our own microbenchmark (described in Appendix A.4). For TPC-C, we used two settings: a *low-contention* setting with 10 warehouses per node, and a *high-contention* setting, with one warehouse per node. Each LM instance had a lock table with a lock object for every tuple of its local warehouse(s). Each transaction requested a number of shared or exclusive locks, depending on its type. We used the same proportion of different transaction types as the original TPC-C specification.

6.2 Locking Performance For TPC-C

We evaluated the performance of DSLR and all the other baselines by running TPC-C in both low (10 warehouses per node) and high (1 warehouse per node) contention settings. We used a cluster of 16 nodes, each with an LM instance, and we used the remaining 16 machines to generate transactions (see Section 6.1). We measured the throughput, average latency, and tail (i.e., 99.9%) latency of the TPC-C transactions under each locking algorithm. As shown in Figure 8a, under high contention, our algorithm achieved 1.8–2.5x higher throughput than all other baselines. Under low contention, however, DSLR's throughput was still 2.8x higher than Traditional, but was only 1.1–1.3x higher than the other DLMs. This was expected, as all algorithms essentially perform the same operation to acquire an uncontended lock: they all use a single RDMA atomic operation (except for Traditional, which still has to use two SEND/RECV operations). Note that SEND/RECV and atomic operations have similar latencies, and the slower performance of Traditional is due to its use of two RDMA operations instead of one.

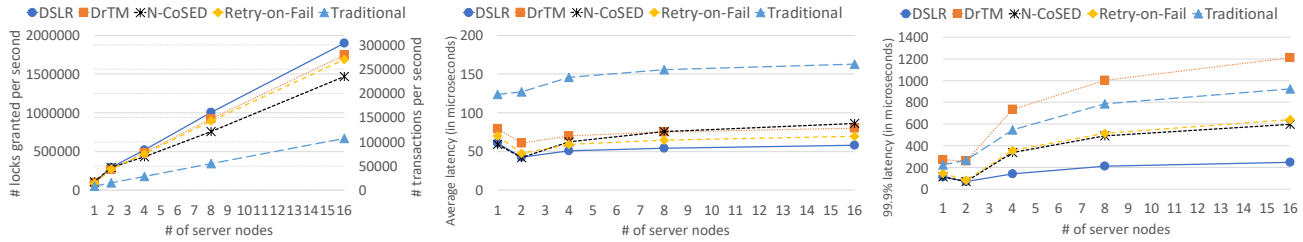
For the same reason, average latencies were also similar for all DLMs under low contention, but were much lower than Traditional's average latency (again, due to the latter's use of two SEND/RECV verbs instead of a single operation). Figure 8b reports the ratio of each baseline's average transaction latency to that of DSLR (i.e., DSLR's speedup). Here, under low contention, DSLR's average latency was 1.2–1.5x lower than other DLMs but 2.8x lower than Traditional. For high contention, however, DSLR's average latency was nearly half of the other techniques, i.e., 1.9–2.5x. This was mainly due to DSLR's utilization of one-sided READ, which is much faster than CAS operations (and blind retries) used by other DLMs in case of lock conflicts.

DSLR's most dramatic improvement was reflected in its tail latencies. As shown in Figure 8c, the 99.9 percentile transaction latencies were significantly lower under DSLR than all other baselines: 2.4–4.9x under low contention and up to 46.7x under high contention. This considerable difference underscores the important role of starvation and lack of fairness in causing extremely poor tail performance. Here, Traditional, despite its lower throughput and higher average latency, behaved more gracefully in terms of tail latencies, compared to the other baselines. This was due to Traditional's global knowledge, allowing it to successfully prevent starvation and ensure fairness even in the face of high-contention scenarios. DSLR, on the other hand, achieved the best of both worlds: its decentralized nature allowed for higher throughput, while maintaining sufficient global knowledge allowed it to prevent starvation (and thereby higher tail latencies). The other DLMs that lacked any global knowledge–thus, any mechanism for preventing starvation—skyrocketed in their tail latencies. When compared with Traditional that did not have the issue of lock starvation, the tail latency of DSLR was still about 2x lower. This was because Traditional, as a queue-based lock manager, still had to use two pairs of SEND/RECV's for each lock/unlock request, one for sending lock/unlock request and another for receiving the response of the lock/unlock request, whereas DSLR only needed a single RDMA operation (i.e., FA) for both locking and unlocking. Overall, the results demonstrate that DSLR is quite robust against lock starvation scenarios and performs better than other baselines in general.

6.3 Scalability of DSLR

We studied the scalability of DSLR compared to other distributed lock managers. We repeated the experiment with increasing numbers of machines, from 2 to 32. For N machines, $N/2$ were server nodes (and hence lock managers) and the remaining $N/2$ machines generated client transactions. We used the *low-contention* TPC-C setting (10 warehouses per node).

As shown in Figure 9, the throughput of all distributed LMs scaled almost linearly (e.g., DSLR achieving 14.5x scalability with 16x additional nodes) as the ratio of the number of server nodes to that of worker threads were constant. N-CoSED and Traditional, which share common characteristics of using queues and SEND/RECV verbs, scaled worse than others in terms of throughput, due to the network congestion caused by their extra messaging. Retry-on-Fail and DrTM scaled better than these queue-based algorithms, as they did not experience as much lock starvation under *low-contention*. However, DrTM showed the worst performance in terms of its tail latency. This was due to its use of lease, as exclusive locks were forced to wait on shared locks until their lease time expired. Overall, DSLR



(a) Throughput (b) Average latency (c) 99.9% latency

Figure 9: Scalability of different distributed locking algorithms with increasing number of nodes.

demonstrated a better throughput than other baselines. For average and 99.9% tail latencies, the performance of DSLR remained consistent and was more robust than other baselines even as the number of nodes increased, again thanks to its starvation-free behavior.

6.4 Performance with Long-Running Reads

The main advantages of a first-come-first-serve (FCFS) policy are its simplicity, fairness, and starvation-free behavior. However, this also means that an FCFS policy cannot reorder the requests. This can be a drawback in situations where reordering the transactions might improve performance [41], e.g., when there are long-running reads in the system.

To consider such scenarios, we implemented an additional baseline, called **Traditional_RO**, which is similar to Traditional (i.e., queue-based) except that it supports transaction reordering. Specifically, Traditional_RO allows readers ahead of the writers, as long as there is already a shared lock held on the object. To avoid starvation of the writes, we also limited the maximum number of shared locks that can bypass an exclusive lock to 10. This is similar to the strategy proposed in [41], except that we used the number of locks instead of their timestamp to ensure better performance for Traditional_RO. Here, we modified the *high-contention* TPC-C setting described in Section 6.2, as follows: we submitted a long-running read transaction with probability γ and a transaction from the original TPC-C workload with probability $1-\gamma$. We varied γ exponentially between 0.001% to 1%. Long-running read transactions required a table-level shared lock on *Customer* table in order to perform a table scan.

Figure 10 reports the results for DSLR versus Traditional and Traditional_RO. As expected, the throughput dropped significantly for all lock managers, as soon as long-running reads were introduced, even at 0.001%. The performance of DSLR and Traditional_RO became similar at the ratio of 0.01%, with Traditional_RO starting to perform better at the 0.1% ratio. Traditional_RO's performance was about 1.2–1.4x better than that of DSLR between the ratio of 0.1% and 1%. This is because it began to leverage transaction reordering with enough long-running read transactions. By allowing other table scans and also short reads from *NewOrder* and *OrderStatus* transactions ahead of other writes (i.e., *Payment* and *Delivery* transactions), Traditional_RO achieved a better performance overall. However, at 1%, the entire system was brought to a halt (only around 1,400 transactions per second for Traditional_RO), compared to when there were no long-running reads (around 110,000 transactions per second for DSLR). This was due to the extreme degree of contention caused by the long-running reads. The experiment demonstrated that queue-based lock managers can benefit from transaction reordering in the presence of long-running reads. However, long-running reads by nature hurt the performance of transactional databases significantly, and there must be a large portion of such long-running reads in the overall workload for transaction reordering to achieve a better performance than DSLR. More importantly, when a shared lock is granted, the lock manager typically does not know when the requesting transaction will release its locks. In other words, the remaining runtime of transaction is not known to the database in general, e.g., the currently held shared lock might be short-lived while the newly arrived one might be long-running. This is perhaps why, to the best of our knowledge, most major databases do not use transaction reordering.

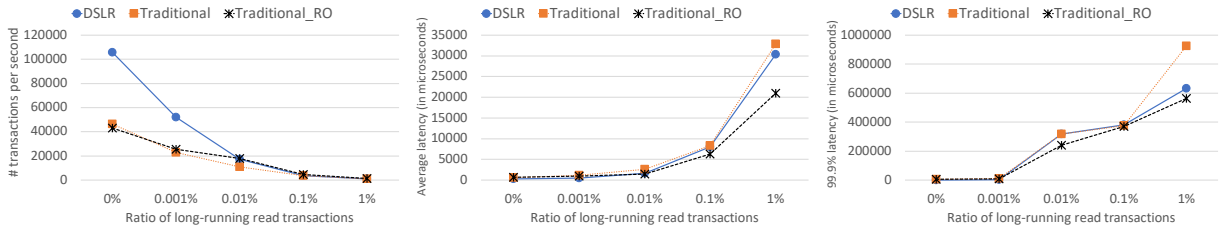
6.5 Effectiveness of Our Multi-Slot Leasing

We studied the effectiveness of our multi-slot leasing mechanism (Section 5.1), a technique designed for accommodating long-running transactions. We used the modified TPC-C workload described in Section 6.4. For this experiment, we varied the ratio of long-running read transactions from 0.05 to 0.5. We ran DSLR, once with a fixed lease time (i.e., 10 ms) and once with multi-slot leasing. As shown in Figure 11, multi-slot leasing led to a better throughput with zero transaction aborts, implying successful execution of the long-running transactions. On the other hand, under DSLR with a fixed lease time, more than half of the transactions aborted. This was caused not only by those long-running transactions that were aborted, but also by other transactions that were blocked by such transactions and eventually timed out and were aborted as well. This confirms that long-running transactions can cause cascading aborts under a fixed lease setting. The experiment therefore shows the effectiveness of our multi-slot leasing mechanism.

7 RELATED WORK

The rise of fast networks has motivated the redesign of distributed systems in general, and databases in particular. While there has been much work on using RDMA for analytics or general data processing [11, 19, 48, 54], here we focus on more relevant lines of work, namely those on distributed lock management and transaction processing. We also discuss other techniques for reducing tail latencies as well the distinction between coordination-free and decentralized protocols.

Distributed Lock Management— Devulapalli et al. [18] propose a distributed queue-based locking using RDMA operations. In their design, each client has its own FIFO (first-in-first-out) queue of waiting clients, to which it will pass the ownership of the current lock. Unlike our algorithm, it requires extra communications using CAS



(a) Throughput (b) Average latency (c) 99.9% latency

Figure 10: Performance comparison between DSLR and queue-based (i.e., two-sided) lock managers with/without transaction reordering, under a modified TPC-C with long-running reads.

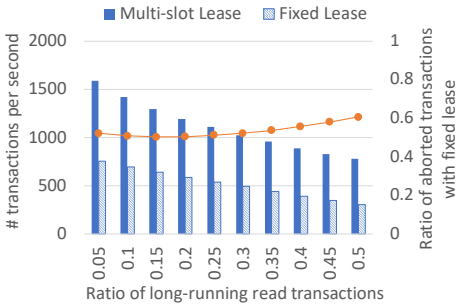


Figure 11: Throughput with fixed vs. multi-slot leasing under the modified TPC-C workload.

among clients in order to enqueue for, and pass the ownership of, a lock. Furthermore, they only support exclusive locks. N-CoSED [47] is another RDMA-based distributed locking, where every node uses CAS to directly place a lock onto the lock server and exchanges extra "lock request/grant" messages in case of lock conflicts. N-CoSED is similar to [18], as each node maintains the list (i.e., a queue) of other nodes waiting for each lock. Chung et al. [15] discuss an alternate and simpler approach by retrying CAS repeatedly until the operation is successful for exclusive locks, and continuously checking the exclusive portion of a lock object until it becomes zero for shared locks (i.e., Retry-on-Fail in §6). Their mechanism is simple and decentralized, yet faces the starvation problem when obtaining exclusive locks on popular objects that have many repeated, continuous reads. In other words, readers starve writers in their model.

RDMA-based Transaction Processing— NAM-DB uses one-sided RDMA read/write and atomic operations to reduce extra communications required by a traditional two-phase commit [12]. However, they only provide snapshot isolation, whereas our proposed distributed lock manager guarantees serializability. Wei et al. design an in-memory transaction processing system, called DrTM, that exploits advanced hardware features such as RDMA and Hardware Transactional Memory (HTM) [62]. DrTM uses CAS to acquire exclusive locks and simply aborts and retries in the case of lock conflicts. FaSST [32] is another system, which utilizes remote procedure calls (RPCs) with two-sided RDMA datagrams to process distributed in-memory transactions. In FaSST, locking is done with CAS, relying on aborts and retries upon failure (very similar to [62]). Li et al. propose an abstraction of remote memory as a lightweight file API using RDMA [38], while Dragojević et al. propose distributed platform with strict serializability by leveraging RDMA and non-volatile DRAM [19]. HERD [31] is a key-value store that makes an unconventional decision to use RDMA writes coupled with polling for communication rather than RDMA reads. They achieve higher throughput by using Unreliable Connection (UC), which unlike Reliable Connection (RC), does not send acknowledgement (ACK/NAK) packets. Their approach is, however, inapplicable to our setting, as all RDMA-based DLMs (including ours) rely heavily on atomic verbs to avoid data races, and RDMA atomic verbs are only available with RC. (Data races can happen with other RDMA verbs.)

Reducing Tail Latencies— A key advantage of DSLR is drastically reducing tail latencies by eliminating starvation. There are other approaches for reducing tail latencies, such as variance-aware transaction scheduling [26, 27], automated explanation [64] or diagnosis [43, 44] of lock-contention problems, redundant computations [21, 60], and choosing indices that are robust against workload changes [45]. All of these approaches are orthogonal to DSLR.

Coordination-free Systems— Bailis et al. [10] show that preserving consistency without coordination is possible when concurrent transactions satisfy a property called *invariant confluence*. Our decentralized algorithm improves the concurrency of distributed systems by allowing a faster coordination in lock management without imposing any extra conditions. In other words, our approach is much more general and does not require that the transactions satisfy invariant confluence.

8 CONCLUSION

In this paper, we presented DSLR, an RDMA-based, fully decentralized distributed lock manager that provides a fast and efficient locking mechanism. While existing RDMA-based distributed lock managers abandon the benefits of global knowledge altogether for decentralization, DSLR takes a different approach by adapting Lamport's bakery algorithm and leveraging the characteristics of FA verbs to sidestep the performance drawbacks of the previous CAS-based protocols that suffered from lock starvation and blind retries. DSLR also utilizes the notion of a lease to detect deadlocks and resolve them via its advisory locking rules. Our experiments demonstrate that DSLR results in higher throughput and dramatically lower tail latencies than any existing RDMA-based DLM.

Our future plan is to integrate DSLR into an existing, open-source distributed database, and study new recovery algorithms enabled by a fully decentralized and RDMA-based locking service.

A APPENDIX

Our appendix provides supplementary experiments for interested readers. Appendix A.1 provides examples for different types of lock starvation. Appendix A.2 provides a formal proof of DSLR's starvation freedom. Appendix A.3 describes the exponential random backoff technique that DSLR utilizes. Appendix A.4 studies the performance of DSLR under varying degrees of skew and contention. Appendix A.5 demonstrates the effectiveness of the random backoff strategy with DSLR. Appendix A.6 shows the impact of lease times on DSLR's ability to handle transaction failures. Appendix A.7 demonstrates the efficiency of dynamic internal polling with DSLR.

A.1 Examples of Lock Starvation

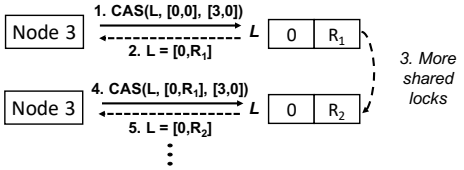


Figure 12: An example of a reader-writer starvation.

In this section, we provide concrete examples of reader-writer and fast-slow starvations discussed in Section 3.1. Figure 12 demonstrates an example of a reader-writer starvation. Here, the function CAS (L, *current*, *new*) changes the value of the lock object L to *new* only if the current value of L is *current*, and the function returns the actual value of L that it obtains from the comparison (same as how RDMA CAS works). Suppose node 3 (acting on behalf of a transaction) tries to acquire an exclusive lock on a lock object L by setting its exclusive portion to 3 with CAS, assuming there are currently no locks on L. Unfortunately, there are already R_1 shared locks on L; hence, CAS fails and node 3 retries CAS with the value of R_1 for the shared portion of L. However, these additional CAS calls can still fail if a stream of new shared locks arrive between consecutive CAS calls, as depicted in Figure 12.

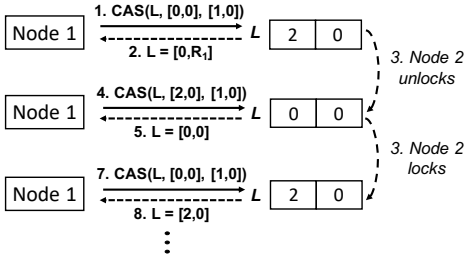


Figure 13: An example of a fast-slow starvation.

Similarly, Figure 13 illustrates an example of a fast-slow starvation. In this case, node 1 continues trying to acquire an exclusive lock on a lock object L; however, it always fails because node 2 can always acquire and release locks on L faster than node 1.

A.2 Proof of DSLR's Starvation Freedom

In this section, we provide a formal proof that DSLR is starvation-free.

Definition 1. **Lock starvation.** A lock starvation is a situation where a transaction is indefinitely unable to acquire a lock on a desired object, due to a locking protocol that allows subsequent transactions to acquire the lock before the current one.

Lemma 2. *If a transaction T_i arrives at time i and receives a ticket $t_i = \{t_i(X), t_i(S)\}$ for a lock object* L*, for any subsequent transaction T_j that arrives later than T_i (i.e., $j > i$) for* L*, its ticket t_j has a value that satisfies the following two conditions:*

$$t_j(X) > t_i(X) \quad OR \quad t_j(S) > t_i(S) \tag{1}$$

and

$$t_j(X) \not< t_i(X) \quad AND \quad t_j(S) \not< t_i(S) \tag{2}$$

Proof. Transaction T_i increments one of the next ticket numbers of L (i.e., $\mathtt{L}(\mathtt{max_S})$ for shared or $\mathtt{L}(\mathtt{max_X})$ for exclusive) by 1 with FA. Due to FA being an atomic operation, the next transaction (i.e., T_{i+1}) will also increment one of the next ticket numbers by 1, receiving the ticket $t_{i+1} = \{t_i(X) + 1, t_i(S)\}$ or $\{t_i(X), t_i(S) + 1\}$. Since all subsequent transactions follow the same protocol in DSLR, the ticket numbers will increase monotonically. This guarantees that any subsequent transaction T_j will receive a ticket t_j with one of its ticket numbers larger than that of t_i (Eq. 1). At the same time, the shared and exclusive ticket numbers of t_j are both no less than the ticket numbers of t_i (Eq. 2). □

Theorem 3. DSLR *is starvation-free.*

Proof. Suppose DSLR's locking protocol is not starvation-free. Let T_i be the transaction with the minimum ticket $t_i = \{t_i(X), t_i(S)\}$ that is being starved from acquiring a lock on L. All transactions arriving after T_i will get larger ticket numbers than t_i according to Lemma 2. These later transactions cannot proceed to acquire the lock ahead of T_i, until T_i increments the global counters of L upon releasing its lock, as shown in Algorithm 1 and 3. Eventually, all transactions with ticket numbers smaller than t_i will acquire and release the lock, incrementing the global counters of L, which would allow T_i to acquire the lock. Even if any of the prior transactions fail to increment the global counters of L (e.g., due to deadlocks), it will be detected by T_i or other subsequent transactions based on the lease expiration logic. In such a case, the values of L will be reset, as described in Section 4.7. This allows T_i to retry and eventually acquire the lock. Therefore, T_i can acquire a lock L, which is a contradiction. □

A.3 Exponential Random Backoff

Exponential random backoff is a widely-used technique in networking literature (e.g., they are used in the IEEE 802.11 protocol [13]) to resolve network collisions. The idea is to progressively wait longer between retransmission of data in case of a network collision. We discovered that adopting the exponential random backoff idea in DSLR is quite effective at resetting the value of a lock object for overflow prevention, transaction failure, and deadlock.

In particular, we utilize a truncated binary exponential random backoff (BEB). When our algorithm detects a possible overflow or transaction failure/deadlock from the value of a lock object or its expired lease, the lock manager waits W microseconds before retrying the lock request, where W is drawn uniformly at random from the following interval:

$$[0, \min(R \times 2^{c-1}, L)]$$

where R is a default backoff time (10 microseconds in our experiments) and c is the number of consecutive deadlocks/timeouts for

the current lock object. We also impose a maximum value L to prevent an unlucky node from waiting too long, even when c is large (L=10,000 μs =10 ms by default).

A.4 Performance Study with Microbenchmark

For finer control over the workload parameters, we used our microbenchmark in a series of experiments to study the impact of various parameters on DSLR and other baselines. Here, each DLM instance hosted 100 million lock objects. Each transaction consisted of a single lock request, where the target object and the lock type (i.e., shared or exclusive) were both chosen randomly. Each time, we chose a shared lock with probability ρ and an exclusive one otherwise. We used $\rho = 0.5$, unless specified otherwise. To emulate a skewed distribution, the objects were chosen according to a power law distribution with exponent α. We varied α from 1.5 (less skewed) to 3 (more skewed).

First, we varied the exponent of the power law distribution α, while keeping the ratio of shared locks at 50% (i.e., $\rho = 0.5$). Figure 14 shows the throughput, average latencies, and tail latencies of DSLR versus other baselines. The larger the exponent, the more skewed the distribution, and the higher the contention (i.e., fewer popular objects). As contention rose, the throughput dropped for all algorithms. Compared to baselines, DSLR was more robust against increased contention (i.e., a moderate decrease in throughput). DSLR also delivered the lowest average and 99.9% tail latencies across all contention levels.

Among other baselines, N-CoSED and DrTM were affected the most by the increase in access skew (and thereby higher contention). N-CoSED suffered from having to use both CAS with blind retries and extra messaging with SEND/RECV. The reason for DrTM's lower performance was slightly different. Due to its reliance on a lease, DrTM forces transactions to wait for an existing lease to expire before acquiring an exclusive lock whenever the object has an existing shared lock (and lease). This severely impacted performance, whenever there were exclusive locks waiting for shared locks. Note that DSLR uses a lease differently, i.e., only to determine transaction failures. To confirm this, we ran another experiment where we varied the ratio of shared locks while fixing α to be 2. The results are shown in Figure 15.

Due to the aforementioned reasons, the performance of DrTM dropped significantly as soon as there were shared locks (since exclusive locks started to wait for the leases from shared locks). However, its performance became better than all other baselines when the workload became read-only (i.e., $\rho = 1$). This is because, in that setting, there were no more exclusive locks to wait on existing leases. Also, with no exclusive locks, DrTM required less number of RDMA operations than other protocols as DrTM only uses 1 CAS for locking and no operations are required for unlocking shared locks. Except for this special case (i.e., read-only workload), DSLR demonstrated better overall performance than other baselines across all configurations that we tried.

A.5 Effectiveness of Random Backoff

To study the effectiveness of the random backoff strategy in DSLR, we conducted additional experiments with TPC-C. We used the setup with much higher contention than the *high-contention* setting in Section 6.2. Out of 32 nodes, we had a single dedicated node, which hosted one warehouse, that served all lock requests, and used the remaining 31 nodes to generate lock requests. We varied the maximum backoff time (i.e., L in Appendix A.3) from 10 to 10,000 μs, while fixing the default backoff time (i.e., R) to 10 μs.

Figure 16 shows the throughput of DSLR with different maximum backoff times. Throughput was very low as the setup had an extremely high contention. There was a clear difference in throughputs between the maximum backoff of 100 μs and 1,000 μs. Results showed that DSLR experienced a very hard time in resetting counters with a maximum backoff time of less than 1,000 μs. As soon as the counters of objects reached COUNT_MAX, the throughput plummeted as DSLR kept failing to reset counters.

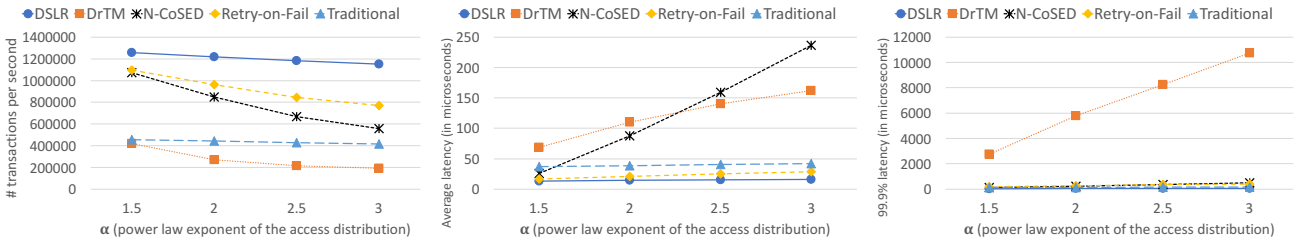
Once DSLR was provided with enough time to reset counters on lock objects (i.e., $L \geq$ 1,000 μs), its throughput was stabilized with DSLR successfully resetting counters as required. This experiment shows that the random backoff strategy is effective for resetting counters in DSLR even under extremely high contention settings.

A.6 Lease Time and Transaction Failures

We studied the impact of lease times on DSLR's ability to handle transaction failures. Transactions may fail due to client or application error, network problems, or even node failures. Again, we used TPC-C in a high contention setting, but intentionally made a transaction fail, without releasing its acquired locks properly, with probability p. We experimented with p=0.1%, 1% and 10%. We also varied DSLR's lease time from 10μs to 10,000μs and used the fixed lease time for this experiment. Figure 17 and 18 show the throughput and transaction abort rates under these different configurations. When the lease times were 10μs and 50μs, transaction abort rates were high and throughputs were low, confirming that too short of a lease time can lead to unnecessary aborts and hurt performance. However, too long of a lease time was more problematic (i.e., $\geq$ 500 μs), since it forced transactions that could have been aborted and retried to wait unnecessarily long. This had a cascading effect on other transactions, which in turn waited for objects held by those transactions. This resulted in a decreased throughput. The optimal lease time in this experiment was around 100 μs. However, note that the failure rate of transactions in real-world (and especially in rack-scale computing) is typically much lower than what we used in this experiment. In general, the lease time should be chosen conservatively in order to avoid premature aborts. This is why DSLR uses a lease time of 10 milliseconds by default. Together with the previous experiment in Section 6.5, this experiment shows that the lease time should be tuned according to the workload and the environment for an optimal performance in face of transaction failures.

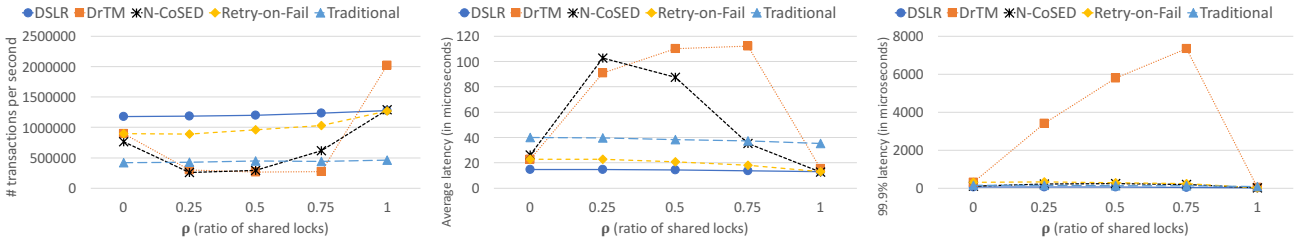
A.7 DSLR with Dynamic Interval Polling

DSLR uses READ for polling. To prevent DSLR from congesting the network with excessive READ operations, especially in high-contention scenarios, we use a dynamic interval polling (see §4.6 for details) rather than a naïve busy polling. To study the effectiveness of this technique, we evaluated DSLR with dynamic versus naïve busy polling. For finer control over the contention level, we modified the



(a) Throughput (b) Average latency (c) 99.9% latency

Figure 14: Performance of different algorithms under the microbenchmark with varying degrees of skew (α).



(a) Throughput (b) Average latency (c) 99.9% latency

Figure 15: Performance of different algorithms under the microbenchmark with varying degrees of contention.

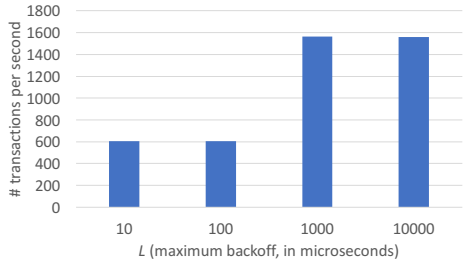


Figure 16: Throughput with different maximum backoff times under an extremely high contention.

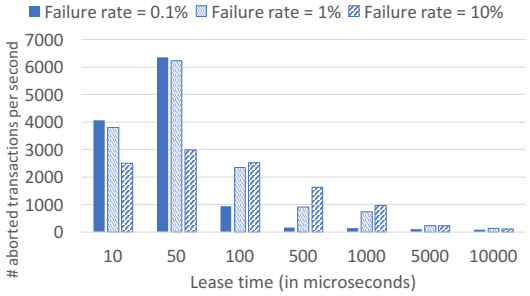


Figure 18: Impact of lease time on abort rate.

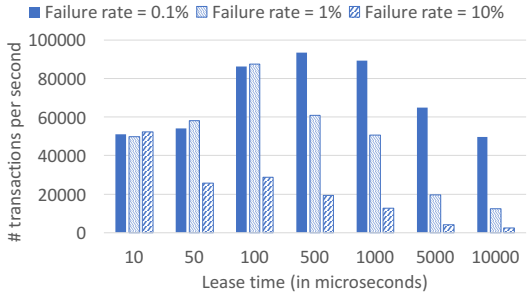


Figure 17: Impact of lease time on throughput.

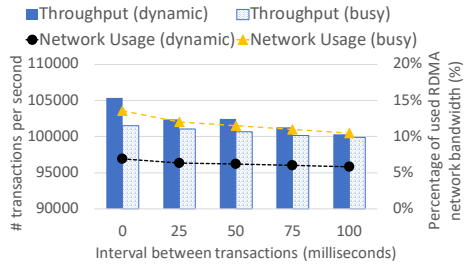


Figure 19: Transaction throughput and % of the RDMA network capacity used by DSLR with dynamic vs. busy polling.

high-contention TPC-C setting (described in Section 6.2) by introducing a delay between sending consecutive transactions from 0 (more contention) to 100 (less contention) milliseconds.

Figure 19 shows DSLR's transaction throughput against the percentage of RDMA capacity used in these scenarios. We measured the maximum RDMA capacity (i.e., the maximum number of RDMA operations the cluster can perform) of the network using Mellanox's Perftest package [9]. Even with busy polling, DSLR used less than 15% of the available capacity. DSLR's utilization of the network was considerably reduced with the use of our dynamic interval polling, to a little over 5%. Interestingly, DSLR was even able to achieve a slightly better throughput with dynamic polling, due to the reduction of unnecessary READ operations. This experiment shows that dynamic interval polling enables DSLR to retain its locking performance while reducing unnecessary READs.

ACKNOWLEDGMENTS

This work is in part supported by National Science Foundation (grants 1629397 and 1553169). The authors are grateful to MICDE and CloudLab for sharing their infrastructure, to Zhenhua Liu and Xiao Sun for their comments, to Anuj Kalia for his insight on RDMA atomic operations, and to Morgan Lovay for editing this manuscript.

REFERENCES

[1] 2016. RDMA over Converged Ethernet. http://www.roceinitiative.org/. (2016).
[2] 2017. APT. https://www.aptlab.net/. (2017).
[3] 2017. Druid | Interactive Analytics at Scale. http://druid.io/. (2017).
[4] 2017. InfiniBand Architecture Specification, Release 1.3. https://cw.infinibandta.org/document/dl/7859. (2017).
[5] 2017. InfiniBand Trade Association. http://www.infinibandta.org. (2017).
[6] 2017. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/. (2017).
[7] 2017. RDMA - iWARP. http://www.chelsio.com/nic/rdma-iwarp/. (2017).
[8] 2017. Teradata: Business Analytics, Hybrid Cloud & Consulting. http://www.teradata.com/. (2017).
[9] 2018. Perftest Package | Mellanox Interconnect Community. https://community.mellanox.com/docs/DOC-2802. (2018).
[10] Peter Bailis et al. 2014. Coordination avoidance in database systems. *PVLDB*.
[11] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-scale in-memory join processing using RDMA. In *SIGMOD*.
[12] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The end of slow networks: it's time for a redesign. *PVLDB*.
[13] Pablo Brenner. 1997. A technical tutorial on the IEEE 802.11 protocol. *BreezeCom Wireless Communications* (1997).
[14] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *USENIX OSDI*.
[15] Yeounoh Chung and Erfan Zamanian. 2015. Using RDMA for Lock Management. *arXiv preprint arXiv:1507.03274* (2015).
[16] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*.
[17] Peter B Danzig, Katia Obraczka, and Anant Kumar. 1992. An analysis of wide-area name server traffic: a study of the Internet Domain Name System. *ACM SIGCOMM Computer Communication Review* (1992).
[18] Ananth Devulapalli and Pete Wyckoff. 2005. Distributed queue-based locking using advanced network features. In *ICPP*.
[19] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: fast remote memory. In *USENIX NSDI*.
[20] Steven Fitzgerald et al. 1997. A directory service for configuring high-performance distributed computations. In *High Performance Distributed Computing, 1997. Proceedings. The Sixth IEEE International Symposium on*.
[21] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. 2015. Reducing latency via redundant requests: Exact analysis. *ACM SIGMETRICS Performance Evaluation Review* (2015).
[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *ACM SIGOPS operating systems review*.
[23] Kishore Gopalakrishna et al. 2012. Untangling cluster management with Helix. In *Proceedings of the Third ACM Symposium on Cloud Computing*.
[24] Cary Gray and David Cheriton. 1989. *Leases: An efficient fault-tolerant mechanism for distributed file cache consistency*.
[25] Andrew B Hastings. 1990. Distributed lock management in a transaction processing environment. In *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*.
[26] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas Wenisch. 2017. A Top-Down Approach to Achieving Performance Predictability in Database Systems. In *SIGMOD*.
[27] Jiamin Huang, Barzan Mozafari, and Thomas Wenisch. 2017. Statistical Analysis of Latency Through Semantic Profiling. In *EuroSys*.
[28] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX ATC*.
[29] Prasad Jayanti, King Tan, Gregory Friedland, and Amir Katz. 2001. Bounding LamportâĂŹs bakery algorithm. In *International Conference on Current Trends in Theory and Practice of Computer Science*.
[30] Horatiu Jula et al. 2008. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks.. In *OSDI*.
[31] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*.
[32] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *USENIX OSDI*.
[33] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. 2012. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*.
[34] Nancy P Kronenberg, Henry M Levy, and William D Strecker. 1986. VAXcluster: a closely-coupled distributed system. *ACM Transactions on Computer Systems*.
[35] Byung-Jae Kwak, Nah-Oak Song, and Leonard E Miller. 2005. Performance analysis of exponential backoff. *IEEE/ACM Transactions on Networking*.
[36] Leslie Lamport. 1974. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* (1974).
[37] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* (2001).
[38] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R Narasayya. 2016. Accelerating relational databases by leveraging remote memory and rdma. In *SIGMOD*.
[39] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*.
[40] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. 2014. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*.
[41] Gang Luo, Jeffrey F Naughton, Curt J Ellmann, and Michael W Watzke. 2010. Transaction reordering. *Data & Knowledge Engineering* (2010).
[42] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC*.
[43] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and resource modeling in highly-concurrent OLTP workloads. In *SIGMOD*.
[44] Barzan Mozafari, Carlo Curino, and Samuel Madden. 2013. DBSeer: Resource and Performance Prediction for Building a Next Generation Database Cloud. In *CIDR*.
[45] Barzan Mozafari, Eugene Zhen Ye Goh, and Dong Young Yoon. 2015. CliffGuard: A Principled Framework for Finding Robust Database Designs. In *SIGMOD*.
[46] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. 2017. SnappyData: A Unified Cluster for Streaming, Transactions, and Interactive Analytics. In *CIDR*.
[47] Sundeep Narravula, A Marnidala, Abhinav Vishnu, Karthikeyan Vaidyanathan, and Dhabaleswar K Panda. 2007. High performance distributed lock management services using network-based remote atomic operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid*.
[48] Jacob Nelson et al. 2015. Latency-tolerant software distributed shared memory. In *USENIX ATC*.
[49] Ravi Rajwar and James R Goodman. 2002. Transactional lock-free execution of lock-based programs. In *ACM SIGOPS Operating Systems Review*.
[50] Jags Ramnarayan, Barzan Mozafari, Sudhir Menon, Sumedh Wale, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra, and Kishor Bachhav. 2016. SnappyData: A hybrid transactional analytical store built on Spark. In *SIGMOD*.
[51] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *USENIX OSDI*.
[52] Ro Recio, P Culley, D Garcia, J Hilland, and B Metzler. 2005. *An RDMA protocol specification*. Technical Report. IETF Internet-draft draft-ietf-rddp-rdmap-03. txt (work in progress).
[53] Kun Ren, Alexander Thomson, and Daniel J Abadi. 2015. VLL: a lock manager redesign for main memory database systems. *The VLDB Journal* (2015).
[54] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-speed query processing over high-speed networks. *PVLDB*.
[55] Dongin Shin et al. 2013. Dynamic Interval Polling and Pipelined Post I/O Processing for Low-Latency Storage Class Memory.. In *HotStorage*.
[56] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*.
[57] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* (2013).
[58] Gadi Taubenfeld. 2004. The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms. *Distributed Computing* (2004).
[59] Boyu Tian, Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas Wenisch. 2018. Contention-aware lock scheduling for transactional databases. *PVLDB* (2018).
[60] Ashish Vulimiri, Oliver Michel, P Godfrey, and Scott Shenker. 2012. More is less: reducing latency via redundancy. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*.
[61] Carl A Waldspurger. 1995. Lottery and stride scheduling: Flexible proportional-share resource management. (1995).
[62] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*.
[63] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. *PVLDB*.
[64] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *SIGMOD*.
[65] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. *HotCloud* (2010).
[66] Lei Zhang, Yu Chen, Yaozu Dong, and Chao Liu. 2012. Lock-Visor: An efficient transitory co-scheduling for MP guest. In *ICPP*.