NetLock: Fast, Centralized Lock Management Using Programmable Switches

Zhuolong Yu
Johns Hopkins University

Yiwen Zhang
University of Michigan

Vladimir Braverman
Johns Hopkins University

Mosharaf Chowdhury
University of Michigan

Xin Jin
Johns Hopkins University

ABSTRACT

Lock managers are widely used by distributed systems. Traditional centralized lock managers can easily support policies between multiple users using global knowledge, but they suffer from low performance. In contrast, emerging decentralized approaches are faster but cannot provide flexible policy support. Furthermore, performance in both cases is limited by the server capability.

We present NetLock, a new centralized lock manager that co-designs servers and network switches to achieve high performance without sacrificing flexibility in policy support. The key idea of NetLock is to exploit the capability of emerging programmable switches to directly process lock requests in the switch data plane. Due to the limited switch memory, we design a memory management mechanism to seamlessly integrate the switch and server memory. To realize the locking functionality in the switch, we design a custom data plane module that efficiently pools multiple register arrays together to maximize memory utilization We have implemented a NetLock prototype with a Barefoot Tofino switch and a cluster of commodity servers. Evaluation results show that NetLock improves the throughput by 14.0–18.4×, and reduces the average and 99% latency by 4.7–20.3× and 10.4–18.7× over DSLR, a state-of-the-art RDMA-based solution, while providing flexible policy support.

CCS CONCEPTS

• **Networks → Programmable networks**; **Cloud computing**; **In-network processing**; *Data center networks.*

KEYWORDS

Lock Management, Programmable Switches, Centralized, Data plane

ACM Reference Format:
Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3387514.3405857

1 INTRODUCTION

As more and more enterprises move their workloads to the cloud, they are increasingly relying on databases provided by public cloud providers, such as Amazon Web Services [4], Microsoft Azure [8], and Google Cloud [7]. *Performance* and *policy support* are two important considerations for cloud databases. Specifically, cloud databases are expected to provide high performance for many tenants and enable rich policy support to accommodate tenant-specific performance and isolation requirements, such as starvation freedom, service differentiation, and performance isolation.

Lock managers are a critical building block of cloud databases. They are used by multiple concurrent transactions to mediate access to shared resources in order to achieve high-level transactional semantics such as serializability. With recent advancements that exploit fast RDMA networks and in-memory databases to significantly improve the performance of distributed transactions [18, 46] (i.e., decrease *think time*), the overhead of acquiring and releasing locks is now a major component in the end-to-end performance of cloud-based enterprise software [49].

Existing lock manager designs (both centralized and decentralized) face a *trade-off* between performance and policy support (Figure 1). The traditional centralized approach uses a server as a *central* point to grant locks [3, 23]. With the global view of all lock operations in the server, this approach can easily support various policies, such as starvation freedom and fairness [23, 24, 29, 48]. The drawback is that the lock server, especially its CPU, becomes the performance bottleneck as transaction throughput increases.

To mitigate the CPU bottleneck, recent decentralized solutions leverage fast RDMA networks to achieve high throughput and low latency [17, 40, 46, 49]. Clients acquire and release locks by updating the lock information on the lock server through RDMA, without involving the server's CPU. However, since the locking decisions are made by the clients in a decentralized manner, it is hard to support and enforce rich policies [49].

We present NetLock, a new approach to design and build lock managers that sidesteps the trade-off and achieves both high performance and rich policy support. We observe that compared to the actual data stored in a database, the lock information is only a small amount of metadata. Nonetheless, the metadata requires *high-speed, concurrent* accesses. Network switches are specifically designed and optimized for high-speed, concurrent data input-output workloads, making them a natural place to accelerate lock operations.

The key idea of NetLock is to leverage this observation and co-design switches and servers to build a fast, centralized lock manager. Switches provide orders-of-magnitude higher throughput and lower latency than servers. By using switches to process lock requests in the switch data plane, NetLock avoids the CPU

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405857

bottleneck of server-based centralized approaches, and achieves high performance. By using a centralized design, NetLock avoids the drawback of decentralized approaches and can support many essential policies.

Realizing this idea is challenging for at least two reasons. First, switches only have limited on-chip memory. Although the size of lock information is orders-of-magnitude smaller than that of the actual storage data, it can still exceed the switch memory size for large-scale cloud databases. While previous work [31] has proposed the idea of extending the switch memory with the server memory, it does not consider the characteristics of locking and does not provide a concrete solution for memory management. To address this challenge, we design a mechanism to seamlessly integrate the switch and server memory to store and process lock requests. NetLock only offloads the *popular* locks to the switch and leaves other locks to servers. We formulate the problem as an optimization problem and design an optimal algorithm for memory allocation.

Second, switches only have limited functionalities in the data plane and cannot process lock requests. Prior work [27] has shown how to build a key-value store in switches and solved the fault-tolerance problem, but a key-value store is not a fully functional lock manager that can support different types of locks and support policies. To address this challenge, we leverage the capability of emerging programmable switches to design a data plane module to implement necessary features required by NetLock. To maximize memory utilization and avoid memory fragmentation, we design a shared queue data structure to pool the register arrays in multiple data plane stages together and allocate it to the locks. Each lock owns an adjustable, continuous region in the shared queue to store its requests. We design custom match-action tables in the data plane to support both shared and exclusive locks with common policies.

NetLock is incrementally deployable and compatible with existing datacenter networks. It is well-suited for cloud providers that have dedicated racks for database services. It only needs to augment the Top-of-Rack (ToR) switches of these database racks with a custom data plane module for processing lock requests. Since the custom module is only invoked by lock messages, other packets are processed by switches as before. NetLock does not change other switches in the network, and it is compatible with existing routing protocols and network functions.

Recently there is a surge of interest in in-network computing. While it is arguable whether applications should be moved to the network and to what extent, NetLock takes a modest approach to make the network more application-aware. Assisting locking in the network is not a radical deviation from traditional network functionalities. We emphasize that the application (i.e., transaction processing) is still running on servers. NetLock provides locks with switches to resolve contentions and enforce policies for concurrent transactions, which is similar to using switch-based signals like Random Early Detection (RED) and Explicit Congestion Notification (ECN) to resolve congestion and enforce fairness for concurrent flows, but in a more application-aware way for databases. Furthermore, compared to changing all NICs and redesigning applications to leverage RDMA, replacing only the switch and transparently updating the lock manager provides a competitive alternative to high-performance database applications. NetLock can provide better performance and lower the cost by reducing the lock servers.

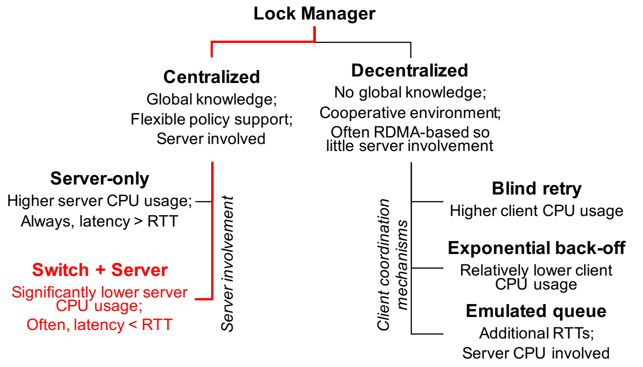


Figure 1: Design space for lock management.

In summary, we make the following contributions.

- We propose NetLock, a new centralized lock manager architecture that co-designs programmable switches and servers to achieve high performance and flexible policy support.
- We design a memory management mechanism to seamlessly integrate the switch and server memory, and a custom data plane module for switches to store and process lock requests.
- We implement a NetLock prototype on a Barefoot Tofino switch and commodity servers. Evaluation results show that NetLock improves transaction throughput by 14.0–18.4×, and reduces the average and 99% latency by 4.7–20.3× and 10.4–18.7× over the state-of-the-art DSLR, while providing flexible policy support.

2 BACKGROUND AND MOTIVATION

In this section, we first provide background on the design of lock managers. Then we motivate the usage of programmable switches to design lock managers, by identifying potential benefits and discussing its feasibility.

2.1 Background on Lock Management

Lock managers are used by distributed systems to mediate concurrent access to shared resources over the network, where locks are typically held in servers. There are two main approaches for accessing locks, i.e., centralized and decentralized, as shown in Figure 1.

Centralized lock management. A centralized lock manager uses a server as a *central* point to grant locks [3, 23]. Because the server has the global view of all lock requests and grant decisions, it can easily enforce policies to provide many strong and useful properties, such as starvation-freedom and fairness [23, 24, 29, 48].

A centralized lock manager can be *distributed* across multiple servers, by having each server be responsible for a subset of lock objects. There is a distinction between distributed and decentralized. Centralized and decentralized approaches differ in how the decisions to grant locks are made, i.e., whether they are made by the central lock manager or by the clients in a decentralized manner. Both approaches can be made distributed to scale out.

The lock manager can either be co-located with the storage server that actually stores the objects or be in a separate server. In the former case, the lock manager daemon would consume the resources of the storage server, which can be otherwise used to process storage requests such as transactions. In the latter case, lock managers for multiple storage servers can be consolidated to a few dedicated servers.

Decentralized lock management. Centralized lock managers suffer from low performance, as the server CPUs become the bottleneck to handle a large number of lock requests from clients [49]. Decentralized lock managers often leverage fast RDMA networks to address the performance problem [17, 40, 46, 49]. A decentralized lock manager still has a designated server to maintain necessary information for each lock in a lock table, e.g., the current transaction ID that holds the lock and whether the lock is shared or exclusive. Different from centralized ones, a decentralized lock manager relies on clients to make decisions in a distributed manner. The lock table at the lock server is updated by the clients using RDMA verbs, such as SEND, RECV, READ, WRITE, CAS, and FA. This approach reduces CPU utilization at the lock server.

There are a few different strategies for the clients to acquire locks in this approach. The simplest one is *blind fail-and-retry*, where each client tries to acquire a lock independently, and retries after a timeout if not succeed [46]. This strategy has high client CPU usage, and can cause starvation and hence long tail latencies. *Exponential back-off* can be used to reduce the CPU usage, but it further increases latencies. More advanced ones use *distributed queues* to emulate centralized lock managers [17]. Such strategies, while avoiding starvation, incur extra network round-trips and lose the benefit of high performance. The most recent solution in this category, DSLR [49], adapts Lamport's bakery algorithm [32] to order lock requests and guarantees first-come-first-serve (FCFS) scheduling; this reduces starvation and achieves high throughput.

Decentralized lock managers typically use *advisory locking*, where clients *cooperate* and follow a distributed locking protocol. This is because the clients use RDMA verbs to interact with the lock table in the lock server without involving the server's CPU. It is different from *mandatory locking* used by centralized lock managers that can *enforce* a locking protocol, as the lock manager is solely making locking decisions. Besides the difficulty to enforce a protocol, decentralized lock managers cannot flexibly support various policies such as isolation, without significantly degrading performance using an expensive distributed protocol.

2.2 Exploiting Programmable Switches

Providing both high performance and policy support. Traditional server-based approaches make a *trade-off* between performance and policy support. Centralized approaches provide flexible policy support, but have low performance; decentralized approaches achieve the opposite. The goal of this paper is to design a solution that sidesteps the trade-off and provides both high performance and policy support. Our key idea is to design a centralized solution with fast switches, which can benefit from switches to achieve high performance while still providing flexible policy support as being a centralized approach. Moreover, since switches provide orders-of-magnitude higher throughput and lower latency than servers, this solution is even faster than decentralized, RDMA-based approaches. This is especially important for emerging fast transaction systems based on RDMA networks and in-memory storage [18, 46]. In these systems, the transactions themselves are executed in memory, and thus the execution cost is comparable to the locking and unlocking cost, meaning that the system needs to spend a considerable amount of server resources for lock managers as for the storage servers themselves. Leveraging switches to build faster lock managers can both improve the transaction performance and reduce the system cost.

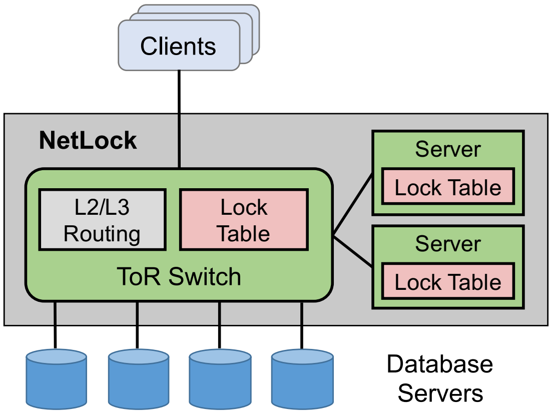


Figure 2: NetLock architecture.

Building lock managers with programmable switches. While traditional switches are fixed-function, emerging programmable switches, such as Barefoot Tofino [9], Broadcom Trident [5] and Cavium XPliant [1], make it feasible to design, build and deploy switch-based lock managers. Leveraging programmable switches provides orders-of-magnitude higher performance than FPGA-based (e.g., SmartNICs) or NPU-based solutions. While this paper focuses on programmable switches, the mechanisms designed for NetLock can also be applied to programmable NICs.

Programmable switches allow users to develop custom data plane modules, which can parse custom packet headers, perform user-defined actions, and access the switch on-chip memory for stateful operations [12, 13]. With this capability, we can program the switch data plane to parse lock information embedded in a custom header format, to perform lock and unlock actions, and to store the lock table in the switch on-chip memory.

3 NETLOCK ARCHITECTURE

In this section, we first give the design goals of NetLock, and then provide a system overview of NetLock.

3.1 Design Goals

NetLock is a fast, centralized lock manager. It is designed to meet the following goals.

- **High throughput.** State-of-the-art distributed transaction systems can process hundreds of millions of transactions per second (TPS) with a single rack [18, 30, 45], and each transaction can involve a few to tens of locks. To avoid being the performance bottleneck of fast distributed transaction systems, the lock manager should be able to process up to a few billion lock requests per second (RPS).
- **Low latency.** Given the tens of microseconds transaction latency enabled by fast networks and in-memory databases [18, 30, 45],

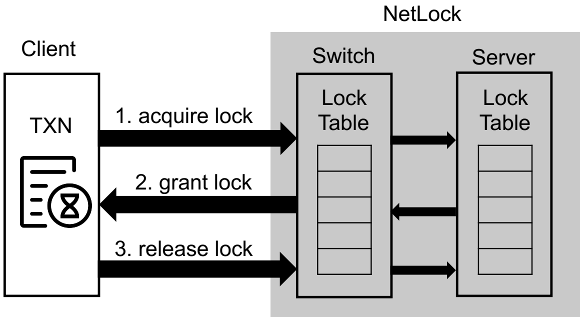


Figure 3: Lock request handling in NetLock. The switch directly processes most lock requests.

the lock manager should provide low latency to process lock requests, in the range of a few to tens of microseconds.

- **Policy support.** For a cloud environment, the lock manager should provide flexible policy support to accommodate tenant-specific requirements. Specifically, we consider common policies including starvation freedom, service differentiation, and performance isolation.

3.2 System Overview

A NetLock lock manager consists of one switch and multiple servers in the same rack (as shown in Figure 2), where the round-trip time (RTT) between machines within the same switch is typically single-digit microsecond. The switch is the ToR switch of a dedicated database rack that is specifically provisioned for database services, which is common in public clouds. Different database racks have their own NetLock instances. Besides adding a new data plane module for NetLock to the ToR switch, no other changes are made to the datacenter network. The ToR switch only invokes the NetLock module to process lock requests, and it processes other packets as usual. NetLock does not affect existing network functionalities.

At a high level, clients send lock requests to NetLock without knowing whether the requests will be processed by a switch or a server. Behind the scene, NetLock processes lock requests with a combination of switch and servers. It integrates the switch and server memory to store and process lock requests. When a lock request arrives at the switch, the switch checks whether it is responsible for the lock. If so, it invokes the data plane module to process the lock; otherwise, it forwards the lock requests to the server. The switch only stores and processes the requests on popular locks, while the lock servers are responsible for the requests on unpopular locks. The lock servers also buffer the requests on popular locks when the queues in the switch are overflowed.

4 NETLOCK DESIGN

In this section, we describe the design of NetLock that exploits programmable switches for fast, centralized lock management.

4.1 Lock Request Handling

As shown in Figure 3, to acquire a lock for a transaction, the client first sends a lock request to NetLock and waits for NetLock to grant the lock. NetLock directly processes most lock requests with the lock switch and only leaves a small portion to the lock servers. After the lock is granted, the client executes its transaction and sends a release notification to NetLock if the lock is no longer needed.

Algorithm 1 ProcessLockRequest(req)

```
 1: if $req.lock \in switch.locks()$ then
 2:     if $req.type == acquire$ then
 3:         if $switch.CanGrant(req)$ then
 4:             Grant $req.lock$ to $req.client$
 5:         else if $switch.CanQueue(req)$ then
 6:             Queue $req$ at switch
 7:         else
 8:             Forward $req$ to server
 9:     else
10:         Release $req.lock$, and grant it to pending requests
11: else
12:     Forward $req$ to server
```

Algorithm 1 shows the pseudocode of the switch. Since the switch is the ToR switch of the database rack and is on the path for a request to reach the lock servers, the switch can always process the request first. If the switch is responsible for the corresponding lock object (line 1), it checks the lock availability and policy. If the lock can be granted, the switch directly responds to the client (line 3-4). If the lock cannot be granted immediately, the switch queues the request if it has enough memory (line 5-6). If the switch is not responsible for the lock object or does not have sufficient memory, it forwards the request to the lock server based on the destination IP (line 8 and 12). The locks are partitioned between the lock servers. The client obtains the partitioning information from an off-the-shelf directory service in datacenters [20, 25], and sets the destination IP to that of the server responsible for the lock. After the client releases the lock, NetLock can further grant the lock to other requests (line 10). The performance benefit of NetLock comes from that most requests can be directly processed by the switch, without the need to visit a lock server.

One-RTT transactions. In the basic mode, a client gets a grant from NetLock (taking 0.5 RTT by the lock switch or 1 RTT by the lock server) and then issues another request to fetch the data from a database server (taking 1 RTT) to finish the transaction, which takes 1.5–2 RTTs in total. Some recent distributed transaction systems (e.g., DrTM [46], FARM [19] and FaSST [30]) combine lock acquisition and data fetching in a single request to a database server, and thus are able to finish a transaction in 1 RTT. NetLock can apply the same idea to achieve one-RTT transactions. Specifically, after a lock is granted, instead of replying to the client, NetLock forwards the request to the corresponding database server to fetch the item, making lock acquisition and data fetching in one RTT. More importantly, unlike existing solutions (e.g., DrTM, FARM and FaSST) that rely on fail-and-retry which may lead to low throughput and high latency, all requests to the database servers can *successfully* fetch data, because the locks have already been granted by NetLock. This is critical under high-contention scenarios to reduce overhead at both clients and database servers, and achieve high throughput and low latency. For locks not in the switch, the lock server is combined with the database server as existing solutions to achieve one-RTT transactions. For requests with payloads such as writes, the switch forwards the data if the lock can be granted, and drops the data, otherwise. Some transactions that involve read-modify-write

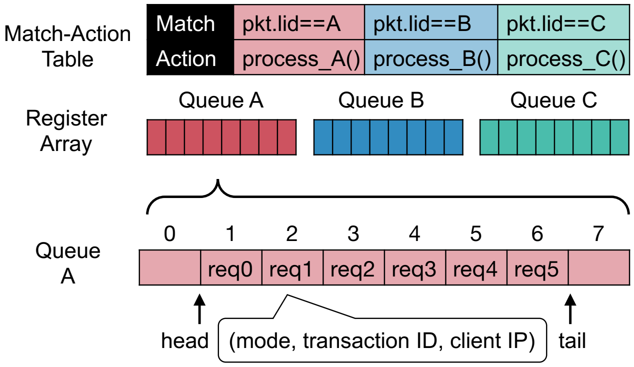


Figure 4: Basic data plane design for lock management.

operations cannot fundamentally be done in one RTT because the client has to do some compute and the current design does not push compute to the lock and database servers. In addition to its high performance, NetLock also supports flexible policies that cannot be implemented by existing decentralized solutions.

4.2 Switch Data Plane

Programmable switches expose stateful on-chip memory as register arrays to store user-defined data. NetLock leverages register arrays to store and process lock requests in the switch. Figure 4 shows a basic data plane design. The design allocates one array for each lock to queue its requests. A special UDP destination port is reserved for NetLock. A lock request contains several fields: action type (acquire/release), lock ID, lock mode, transaction ID, and client IP. The match-action table maps a lock ID (i.e., *lid*) to its corresponding register array, and the action in the table performs operations on the register array to grant and release locks.

Because register arrays can only be accessed based on a given index, they do not natively support queue operations such as enqueue and dequeue. We implement *circular queues* based on register arrays to support necessary operations for NetLock. Specifically, we allocate extra registers to keep the head and tail pointers. The pointers are looped back to the beginning when they reach the end of the array. For example, queue A in Figure 4 has six queued requests, and the head and tail are index 1 and 6, respectively.

Each slot in a queue stores three important pieces of information, i.e., mode, transaction ID, and client IP. Mode indicates whether the request is for a shared or exclusive lock. Transaction ID identifies which transaction the lock is requested for. Client IP stores the IP address from which the lock request is sent. The IP address is used by the switch when it generates a notification to grant the lock to the client. Additional metadata such as timestamp and tenant ID can also be stored together.

Optimize switch memory layout. Because the memory for each register array is pre-allocated and the size is fixed after the data plane program is compiled and loaded into the switch, the basic design cannot flexibly change the queue size at runtime. When the workload changes, the set of locks in the switch and the size of each queue would need to change according to the memory allocation algorithm to maximize the performance. Allocating a large queue to accommodate the maximum possible contentions for each lock is undesirable because it would cause memory fragmentation and result in low memory utilization, especially given that the switch on-chip memory is limited.

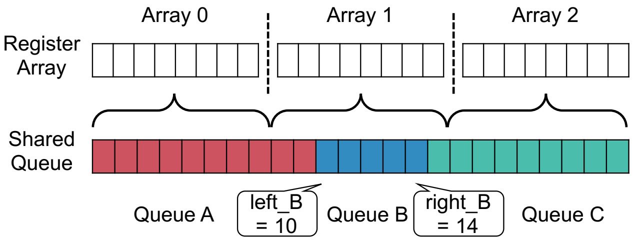


Figure 5: Combine multiple register arrays to a shared queue for locks with different queue sizes.

To address this problem, we design a *shared queue* to pool multiple register arrays together and enable the queue size to be dynamically adjusted at runtime (Figure 5). Instead of statically binding each register array to a lock, we combine these arrays together to build a large queue shared by all the locks. Accessing a slot in the shared queue with an index can be mapped to accessing the register arrays by appropriately setting the index, e.g., accessing slot 10 in the shared queue can be mapped to accessing slot 10-8=2 in array 1. Each lock is allocated with a continuous region in the shared queue to store its requests. We allocate extra registers to store the boundaries of each queue, e.g., 10 and 14 for queue B. Since the boundaries are stored in registers, they can be modified at runtime. Another benefit of this design is that the individual register arrays do not have to be in the same stage, which allows NetLock to pool memory from multiple stages together to build a large queue that exceeds the memory limit of a single stage.

Handle shared and exclusive locks. The shared queue design solves the storage problem of how to store the requests, but it does not solve the computation problem of how to process them. The challenge comes from the limitation that the data plane can only perform one read/write operation to a register array when it processes a packet.

This limitation brings two issues. First, when a lock release notification arrives at the switch, the switch dequeues the corresponding request from the queue, and the lock could be granted to the next request in the queue. This requires two operations: one is to dequeue the head, and the other is to read the new head. Second, when a request to acquire a shared lock is granted, if the following requests in the queue are also for a shared lock, then these requests can also be granted. This requires multiple read operations until an exclusive lock request or the end of the queue. We leverage a feature called *resubmit* available in programmable switches to overcome the limitation. The resubmit feature allows the switch data plane to resubmit the packet to the beginning of the packet processing pipeline, so that the packet can go through and be processed by the pipeline again, obviating the need to send *another* packet to the switch from servers. Note that the use of resubmit here does not cause extra overhead, because the servers in the traditional server-based lock managers also need to send a packet to grant each shared lock to the corresponding client. Figure 6 illustrates how to handle the four cases for shared and exclusive locks.

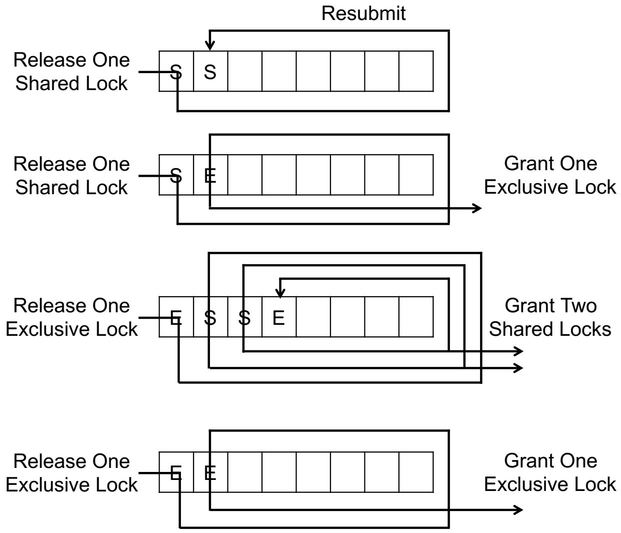


Figure 6: Handle shared and exclusive locks.

- **Shared → Shared**. When a shared lock is released, the switch dequeues the head, and uses resubmit to check the new head. If the new head is a shared lock request, the processing stops, because the shared lock has already been granted with the old head when it entered the queue.
- **Shared → Exclusive**. This case differs from the first case on that the new head is an exclusive lock request, which has not been granted yet. As such, after the shared lock is released, the lock becomes available, and the switch sends a notification to the client to grant the lock.
- **Exclusive → Shared**. When an exclusive lock is released, the packet is resubmitted to grant the next lock request in the queue. The resubmit action is repeated by multiple times until an exclusive request or the end of the queue.
- **Exclusive → Exclusive**. When an exclusive lock is released and the next request is also exclusive, the next request is granted. Because the lock is exclusive and cannot be shared, the switch does not need to resubmit it again.

Algorithm 2 shows the pseudocode of the switch that covers the above four cases. If the request is to acquire a lock, it is enqueued (line 1-2). The request is directly granted if the queue is empty, or if all requests in the queue are shared and the request is also shared (line 3-5). If the request is to release a lock, the current head in the queue is removed, and the lock is resubmitted to grant the following request (line 7-12). For case "shared → shared", no further processing is needed. For case "shared → exclusive" and "exclusive → exclusive", the new head is granted the lock (line 15-16). For case "exclusive → shared", multiple subsequent shared locks are granted (line 17-27). The nuance in the lock processing is that when there are multiple transactions holding a shared lock, these transactions may not release their locks in the order that the requests are enqueued. Because the switch can only release locks at the head of the queue, it does not check the transaction ID when releasing locks. This design does not affect the correctness, because only one transaction can hold an exclusive lock, and the operations for releasing shared locks are commutative.

Algorithm 2 SwitchDataPlane(pkt)

```
1:  if $pkt.op == acquire$ then
2:      $queue.enqueue(pkt)$
3:      if $queue.is\_empty()$ or
4:       ($queue.is\_shared()$ and $pkt.mode == shared$) then
5:          $grant\_lock(pkt.tid, pkt.cip)$
6:  else
7:      if $meta.flag == 0$ then
8:          $(mode, tid, cip) \leftarrow queue.dequeue()$
9:          $meta.flag \leftarrow 1$
10:         $meta.mode \leftarrow mode$
11:         $meta.pointer \leftarrow queue.head()$
12:         $resubmit()$
13:     else if $meta.flag == 1$ then
14:         $(mode, tid, cip) \leftarrow queue[meta.pointer]$
15:         if $mode == exclusive$ then
16:             $grant\_lock(tid, cip)$
17:         else if $meta.mode == exclusive$ then
18:             $grant\_lock(tid, cip)$
19:             $meta.pointer \leftarrow meta.pointer.next()$
20:             $meta.flag \leftarrow 2$
21:             $resubmit()$
22:     else
23:         $(mode, tid, cip) \leftarrow queue[meta.pointer]$
24:         if $mode == shared$ then
25:             $grant\_lock(tid, cip)$
26:             $meta.pointer \leftarrow meta.pointer.next()$
27:             $resubmit()$
```

Pipeline layout. A switch may have several pipelines, and the pipelines do not share state. In NetLock, the lock tables and their register arrays are placed in the *egress* pipes that *connect to their corresponding lock servers.* This placement avoids unnecessary recirculation across pipelines. Specifically, when a request arrives, it is sent to the egress pipe that either owns the lock or connects to a lock server that has the lock. If the request is granted, it is mirrored to the upstream port to the client or the database server to finish the transaction (Section 4.1). Otherwise, it is enqueued either at the egress pipe or in a lock server.

4.3 Switch-Server Memory Management

Since the switch on-chip memory is limited, NetLock co-designs the switch and servers and stores only the popular locks to the switch memory. The switch control plane is responsible for creating and deleting locks, and assigning memory for locks between the switch and lock servers. The key challenge in memory allocation is that it requires us to consider the contentions from multiple requests to the same lock. When a lock is granted to a client, other requests are queued in the switch and occupy memory space until the lock is released.

Memory allocation mechanism. We first analyze the amount of switch memory required to support a certain throughput. Let the rate of lock requests to object i be r_i. Let the maximum contention for object i be c_i, which means that there are at most c_i concurrent requests for object i. We assume c_i is known based on the knowledge of how many clients may need this lock, and we use a counter to measure r_i. Let the queue size for object i in the switch be s_i. If $s_i \geq c_i$, then the switch can guarantee to process all requests for object i, without queueing requests in the server. The memory allocation is to decide which locks to assign to the switch, and for each assigned lock, how much switch memory to allocate for it.

Algorithm 3 MemoryAllocation(locks)

```
Sort locks by $r_i/c_i$ in decreasing order
for lock $i$ in $locks$ do
    $s_i \leftarrow min(switch.available, c_i)$
    $switch.available \leftarrow switch.available - s_i$
    Allocate $s_i$ for lock $i$ in switch memory
Allocate remaining locks to the servers
```

Let the switch memory size be S. We formulate the problem as the following optimization problem.

$$maximize \sum_i r_i s_i / c_i \tag{1}$$

$$s.t. \sum_i s_i \leq S \tag{2}$$

$$s_i \leq c_i \tag{3}$$

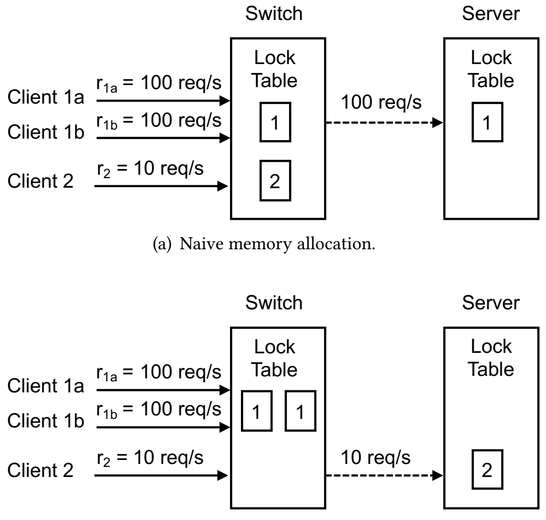
The goal is to process as many lock requests in the switch as possible, reducing the number of servers we need for NetLock. For object i, because in the worse case the lock requests for i always achieve the maximum contention c_i, only a portion (s_i/c_i) of lock requests can be queued at the switch, and the other portion ($1-s_i/c_i$) have to be sent to the server. Therefore, the optimization objective, which is the request rate the switch can guarantee to process, is $\sum_i r_i s_i/c_i$. The constraint is that the total memory allocated to the locks cannot exceed the switch memory size S, i.e., $\sum_i s_i \leq S$. The switch does not need to allocate more than c_i memory slots to object i, thus we have $s_i \leq c_i$.

This problem is similar to the fractional knapsack problem, which can be solved with an optimal solution in polynomial time. Algorithm 3 shows the pseudocode. Specifically, the value of allocating one slot to object i in the switch is r_i/c_i. To maximize the objective, the algorithm allocates the switch memory based on the decreasing order of r_i/c_i.

The rate r_i and contention c_i for each lock are obtained by measuring the workload. NetLock maintains two counters to track r_i and c_i for each lock respectively, and updates the memory allocation based on Algorithm 3 when the workload changes. During the update, NetLock first drains the requests of the locks that are to be swapped out from the switch, and then allocates the switch memory to more popular locks. Note that, for inserting a new lock object, the new lock queue is first added to a lock server, and then would be moved to the switch if the lock becomes popular.

Theorem 1. *The memory allocation algorithm (Algorithm 3) is optimal for the optimization problem (1-3).*

Proof. We consider the situation where $\sum_i c_i > S$; otherwise, there is enough memory for all the locks. Let there be n locks in total. Without loss of generality, let $\frac{r_1}{c_1} > \frac{r_2}{c_2} > ... > \frac{r_n}{c_n}$. Algorithm 3 allocates as much memory as possible ($min(switch.available, c_i)$) for locks sorted by r_i/c_i. Assume this is not the optimal strategy. Let the optimal strategy be $s_1^*, s_2^*, ..., s_n^*$. Because $\sum_i c_i > S$, there exists at least one lock i such that $s_i^* < c_i$. Let the lock with the smallest ID be j, i.e., for any $i < j, s_i^* = c_i$, and $s_j^* < c_j$. If for any $k > j, s_k^* = 0$, the optimal strategy would be the same as Algorithm 3. Therefore, there exists at least one lock k such that $k > j$ and $s_k^* > 0$. Let $s_j' = s_j^* + 1$ and $s_k' = s_k^* - 1$. Because $\frac{r_j}{c_j} > \frac{r_k}{c_k}$, we have $\sum_i r_i s_i'/c_i > \sum_i r_i s_i^*/c_i$. This contradicts that the allocation $s_1^*, s_2^*, ..., s_n^*$ is optimal. So Algorithm 3 is optimal. □



(a) Naive memory allocation.

(b) Optimal memory allocation.

Figure 7: By allocating two slots in the switch to lock 1, the optimal allocation can process all lock requests to lock 1 in the switch, minimizing the server load.

Example. Figure 7 illustrates the key idea of the algorithm. There are two concurrent clients that acquire exclusive locks for object 1 with a rate of 100 requests per second each. The queue needs two slots to accommodate the contentions from the two clients. There is only one client that acquires exclusive locks for object 2, with a rate of 10 requests per second. The queue only needs one slot for one client. Suppose the switch memory only has two slots. The allocation in Figure 7(a) allocates one slot to each lock object. Since the switch cannot queue requests for two clients for object 1, in the worse case where the clients are highly synchronized, half of the requests are sent to the server. On the other hand, the optimal allocation in Figure 7(b) allocates two slots to object 1, minimizing the load on the server.

Performance guarantee. Since servers have plenty of memory to queue requests, servers are CPU-bounded, and the bottleneck is on the number of requests that can be processed by a server per second. Let the workload be $W = \{(r_i, c_i)\}$, and the solution to the optimization problem be $S = \{(s_i)\}$. Let r_s and r_e be the request rates that can be supported by a switch and a server, respectively. We assume that the switch is not the bottleneck, i.e., $r_s \geq \sum_i r_i$, so the switch is always able to support the request rate $\sum_i r_i s_i/c_i$. This assumption is reasonable, because if $r_s < \sum_i r_i$, then the ToR switch is congested. In such a case, not all lock requests can even be received by the database rack in the first place, and the workload would not be meaningful. Since the switch can process the request rate $\sum_i r_i s_i/c_i$, it requires $\lceil(\sum_i r_i - \sum_i r_i s_i/c_i)/r_e\rceil$ servers to serve the remaining request rate. In other words, with one switch and $\lceil(\sum_i r_i - \sum_i r_i s_i/c_i)/r_e\rceil$ servers, NetLock guarantees to support the workload $W = \{(r_i, c_i)\}$.

Handling overflowed requests. It is possible that the queues in the switch can be overflowed, because the switch cannot allocate enough memory for the last object it handles or the estimation of maximum contention for an object is inaccurate. For lock i, we denote its switch queue as $q_1[i]$, and its server queue as $q_2[i]$. When $q_1[i]$ is full, the switch forwards the overflowed requests to the server. The overflowed requests are only buffered in $q_2[i]$ in the server, not processed. Note that this is different from the locks that are not allocated to the switch and only have queues in the servers—the requests of those locks are both buffered and processed by the servers. The switch puts a mark on the packets to distinguish between these two cases.

As both $q_1[i]$ and $q_2[i]$ may contain requests, we need to ensure that the requests are processed as they would in a single queue. To achieve this, the requests are only granted and dequeued by $q_1[i]$ in the switch, and new requests are only enqueued at $q_2[i]$ in the server. When $q_1[i]$ becomes empty, the switch sends a notification to the server, and the server pushes some requests from $q_2[i]$ to $q_1[i]$. The number of requests that can be pushed is no bigger than the number of available slots in $q_1[i]$ to ensure $q_1[i]$ is not overflowed. When $q_2[i]$ becomes empty and $q_1[i]$ is not full, NetLock enters the normal mode, i.e., new requests can directly be enqueued at $q_1[i]$ in the switch. Because $q_2[i]$ is empty, enqueueing at $q_1[i]$ would ensure the same order as a single queue.

Moving locks between the switch and lock servers. When the popularity of a lock changes, the lock will be moved from the switch to a lock server or from its lock server to the switch. When moving a lock, NetLock pauses enqueuing new requests of this lock and waits until the queue is empty to ensure consistency. Memory fragmentation caused by moving locks between the switch and lock servers would reduce the memory that can be actually used to store lock requests. The memory layout on the switch is periodically reorganized to alleviate memory fragmentation.

4.4 Policy Support

NetLock is a centralized lock manager that can support and enforce policies. We consider the following three representative policies.

Starvation-freedom. Decentralized lock managers use partial information to grant locks, which can easily lead to lock starvation. Lock starvation happens when the lock manager allows later lock requests to acquire a lock before earlier lock requests, making some requests wait indefinitely to get the lock. Lock starvation is typically avoided by using a first-come-first-serve (FCFS) policy. The FCFS policy stores lock requests in a first-in-first-out (FIFO) queue, and always grants locks to the head of the queue. This policy is natively supported by the circular queue we design for the switch data plane. With this, NetLock supports request (lock) level starvation-freedom. Note that, there can still be starvation if some transactions do not complete because of deadlock, which is discussed in Section 4.5.

Service differentiation with priorities. It is challenging to support priority-based policies in the switch, as a register array can only be accessed once when processing a packet and a priority queue cannot be directly implemented with a register array. We leverage the multi-stage structure of the switch data plane to support priorities. Specifically, we allocate one queue in each stage for one priority. Since the packet is processed stage by stage, the high-priority requests in earlier stages are granted first. The request processing with priorities in the switch data plane follows Algorithm 2 with some tweaks. For a lock request with i-th priority, it is directly granted if all queues are empty, or if there is no exclusive lock request holding the lock or queued in the same or higher priority queues and the request itself is also for a shared lock. After the lock is released, NetLock will first grant the lock to the queue with the highest priority. Note that a priority can have a large queue spanning multiple stages to expand its queue size. The limitation of this solution is that the number of priorities is limited to the number of stages, which is usually 10-20 in today's switches. This limitation can be alleviated by approximation, e.g., grouping multiple fine-grained priorities into a single coarse-grained priority. Moreover, only high-priority requests need to be processed in the switch. Low-priority requests do not need fast processing, and can always be offloaded to the lock servers.

Performance isolation with per-tenant quota. Cloud databases often have multiple tenants and need to enforce fairness between them. Without a centralized lock manager, a tenant can generate requests and acquire locks at a faster rate than another tenant, and thus occupies most of the resources. While an FCFS policy can avoid starvation of the slower tenant, it cannot enforce the tenants to stay within their shares. It requires the lock manager to use rate limiters to enforce per-tenant quota. Rate limiters can be implemented in the switch data plane with either meters that can automatically throttle a tenant, or counters that count the tenants' requests and compare with their quotas.

4.5 Practical Issues

Switch memory size. We examine whether the switch memory is sufficient for a lock manager from two aspects.

Think time. The think time affects the maximum *turnover* rate of a memory slot. Let T be the duration of a request occupying a slot, which includes the round trip time of sending the grant and release messages and that of executing the transaction (i.e., think time). A slot can be reused by $1/T$ times per second (i.e., the *turnover* rate), providing a throughput of $1/T$ RPS. With S slots, the switch can achieve S/T RPS. Given fast networks and low-latency transactions, T can be a few tens of microseconds. As a switch has tens of MB memory, 100K slots with 20B slot size only consume 2 MB memory, which is a small portion of the total memory. Assuming $T = 20\ \mu s$ and $S = 100K$, the switch can support $S/T = 5$ BRPS, which is sufficient for the database servers the same rack. On the other hand, if $T = 1\ ms$, the switch needs 1M slots to achieve $S/T = 1$ BRPS.

Memory allocation. The memory allocation mechanism affects the utilization of the switch memory. It determines whether the switch can achieve the maximum rate S/T. If the switch memory is allocated to unpopular locks, the switch would only process a small portion of the total locks. Even when a memory slot is available, it may not be used to process a new request for its lock as there are no pending requests for this unpopular lock. If the memory slots are empty for half of the time, then the switch needs to double its memory slots in order to achieve the maximum rate. NetLock uses an optimal knapsack algorithm to efficiently allocate switch memory to popular locks to maximize the memory utilization. This handles skewed workload distributions. For uniform workload distributions, we combine multiple locks into one *coarse-grained* lock to increase the memory utilization.

In summary, the think time determines the maximum *turnover* rate of a memory slot and thus the maximum throughput the switch can support with a given amount of memory, and the memory allocation mechanism determines whether the system can achieve the maximum *turnover* rate. Experimental results in Section 6.4 illustrate the relationship.

Scalability. We focus on rack-scale database systems in this paper. Based on the above analysis on switch memory size and the experimental results in Section 6.4, the memory of one switch is sufficient for most rack-scale workloads, and the ToR switch can be naturally used as the lock switch. In the cases where more memory is needed, additional lock switches can be attached to the rack as specialized accelerators for lock processing. For large-scale database systems that span multiple racks, each rack runs an instance of NetLock to handle the lock requests of its own rack.

Failure handling. We describe how to handle different types of failures in NetLock.

- *Transaction failure.* Transaction failures can be caused by network loss, application crashes, and client failures. When a transaction fails without releasing its acquired locks, other transactions that request for the same locks cannot proceed. NetLock uses a common mechanism, leasing [21], to handle transaction failures. It stores a timestamp together with each lock, and a transaction expires after its lease. The switch control plane periodically polls the data plane to clear expired transactions.
- *Deadlock.* Deadlocks are caused by multiple transactions waiting for locks held by others, and no transaction can make progress. It is resolved in the same way as for transaction failures. Clients retry when the leases expire until they succeed. In addition, deadlocks can be avoided if priority-based policies are employed.
- *NetLock failure.* When a lock server fails, the locks allocated to this server is assigned to another lock server. Clients resubmit their requests to the new server, and the server waits for the leases to expire before granting the locks. A switch failure is handled in the same way by assigning the locks to a backup switch. After the original switch restarts, the lock requests are queued into the original switch. When releasing a lock, we only grant locks from the backup switch until the queue in the backup switch gets empty. After all the queues in the backup switch get empty, the backup switch is no longer useful. When the switch restarts, it also synchronizes its states with the lock servers and waits for the overflowed requests that are buffered at the lock servers to drain before the switch starts processing new requests on the corresponding locks. The unpopular locks stored in lock servers are not affected by switch failures.

5 IMPLEMENTATION

We have implemented a prototype of NetLock, including the lock switch, the lock server, and the client.

The lock switch is implemented with 1704 lines of code in P4, and is compiled to Barefoot Tofino ASIC [9]. The lock table has a shared queue with a total of 100K slots. With 20B slot size, it only consumes 2MB, which is a small portion of tens of MB on-chip memory. The switch control plane is implemented with 750 lines of code in Python, which allocates the memory in the shared queue to different locks.

The lock server is implemented with 2807 lines of code in C. It handles lock requests that cannot be directly processed by the lock switch. To maximize the efficiency of multi-core processing and improve the performance, it uses Intel DPDK [2], and leverages Receive Side Scaling (RSS) to partition the lock requests between cores and dispatch the lock requests to the appropriate RX queues by the NIC for each core. With these optimizations, a lock server can achieve up to 18 MRPS with a 40G NIC in our testbed.

The client is implemented with 3176 lines of code in C. It is used to generate lock requests to measure the performance in the experiments. It also uses Intel DPDK and RSS to optimize performance, and one client server can generate up to 18 MRPS with a 40G NIC in our testbed.

6 EVALUATION

6.1 Methodology

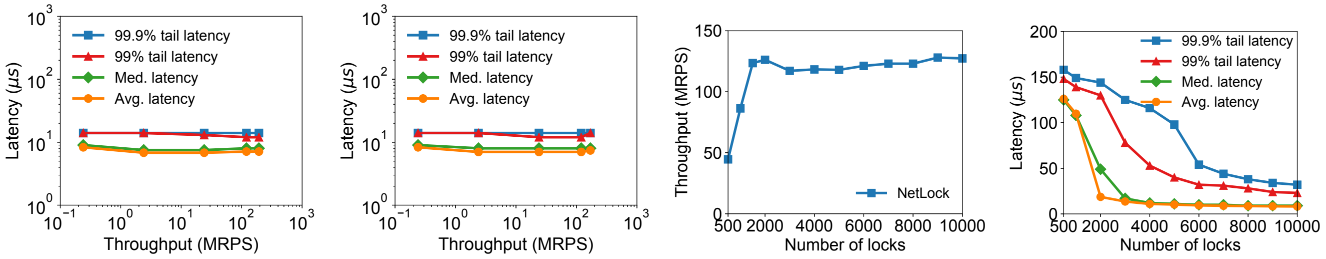
Testbed. The experiments of NetLock are conducted on our testbed consisting of one 6.5 Tbps Barefoot Tofino switch and 12 servers. Each server has an 8-core CPU (Intel Xeon E5-2620 @ 2.1GHz) and one 40G NIC (Intel XL710).

Comparison. We compare NetLock with the state-of-the-art lock manager DSLR [49] and DrTM [46]. Since DSLR and DrTM require RDMA, the experiments on DSLR and DrTM are conducted in the Apt cluster of CloudLab [6]. The configuration is comparable to our own testbed. Each server is equipped with an 8-core CPU (Intel Xeon E5-2450 @ 2.1GHz) and a 56G RDMA NIC (Mellanox ConnectX-3). We also compare NetLock with a recently proposed switch-based solution NetChain [27]. NetChain is not a fully functional lock manager, as it only supports exclusive locks. Therefore, requests for shared locks are treated as exclusive locks. NetChain handles concurrent requests with client-side retry. Since NetChain only stores items in the switch, we adapt the lock granularity based on the switch memory size and the number of locks, so that NetChain can handle all the requests in the switch. We emphasize that DSLR, DrTM and NetChain do not support flexible policies.

Workloads. We use two workloads. The first workload is a microbenchmark, which simply generates lock requests to a set of locks. It is useful to measure the basic performance of lock processing. The second workload is TPC-C [10]. It generates transactions based on TPC-C, and each transaction contains a set of lock requests. It is useful to measure the application-level performance. We use two settings for TPC-C, which is the same as DSLR: a low-contention setting with ten warehouses per node, and a high-contention setting with one warehouse per node. We use throughput, in terms of lock requests granted per second (RPS) and transactions per second (TPS), and latency as the evaluation metrics.

6.2 Microbenchmark

We use microbenchmark experiments to measure the basic throughput and latency of the lock switch to process lock requests. We cover both shared and exclusive locks.



(a) Shared locks.

(b) Exclusive locks w/o contention.

(c) Exclusive locks w/ contention.

(d) Exclusive locks w/ contention.

Figure 8: Microbenchmark results of switch performance on handling lock requests.

Shared locks. We first evaluate the performance for shared locks. We use all 12 servers in the testbed to generate requests to the lock switch. Since the requests are for shared locks, there are no contentions and the locks can be directly granted. Figure 8(a) shows the relationship between latency and throughput. The median (average) latency is 8 μs (7.1 μs), and the 99% (99.9%) latency is 12 μs (14 μs). We emphasize that the latency is dominated by the processing latency at the client software and NIC; the processing latency at the switch is under 1 μs. The latency is not affected by the throughput, because even we use all 12 servers to generate requests, they can still not saturate the switch. The switch can handle the lock request at line rate, and the Barefoot Tofino switch used in the experiment is able to process more than 4 billion packets per second.

Exclusive locks. We then evaluate the performance for exclusive locks. Similar to the previous experiment, we use 12 servers to generate requests for exclusive locks. To measure the baseline performance, the requests are sent to different locks and there are no contentions. Figure 8(b) shows the results, which are similar to those for shared locks. This is because in both cases, the requests are directly granted by the switch and processed at line rate.

To show the impact of contention on exclusive locks, we let the servers send lock requests to the same set of locks, and vary the number of locks in the set. The level of contention decreases as the number of locks increases. Figure 8(c) shows the impact of contention on the throughput. Under high contention (i.e., when the number of locks is small), the throughput is very limited. This is because the requests for the same lock have to be processed one by one, even though the switch still has spare capacity. The throughput increases as the contention decreases. Under low contention, the throughput is maximized by the speed of the 12 servers to generate lock requests. Figure 8(d) shows the latency results. The latency is more than 100 μs under high contention, and decreases to a few μs under low contention.

Comparison with lock server. We also compare the performance of a lock switch with a lock server. We use 10 servers to generate requests, and the workloads are similar to the previous experiments: shared locks, exclusive locks without contention, and exclusive locks with contention (5000 locks). The lock server is implemented with the same functionality and is configured with a different number of cores (1~8) in this experiment. Figure 9 shows the throughput of a lock switch and a lock server. The lock switch outperforms the lock server by 7× as the lock server easily gets saturated by a large number of requests. We emphasize that the lock switch is not saturated by the ten clients in this experiment. The performance gap would be even larger if there are more clients sending requests: the switch can process a few billion requests per second and can potentially replace hundreds of servers for the same functionality.

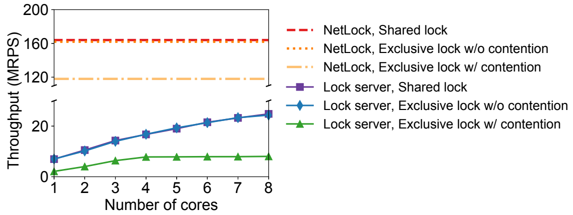
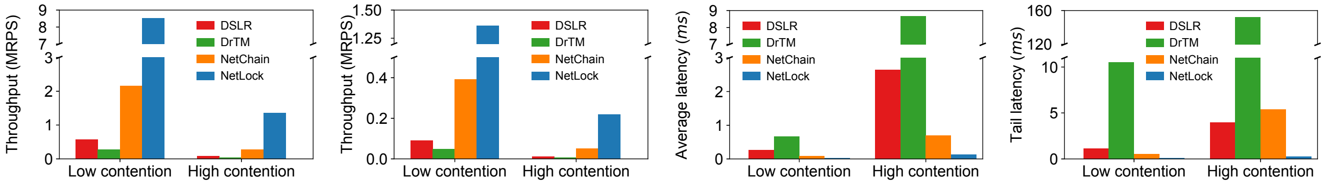


Figure 9: Comparison between a lock switch and a lock server with various number of cores. Ten servers are used to generate requests. The lock switch is not saturated. The lock switch can support a few billion requests per second.

6.3 Benefits of NetLock

We show the benefits of NetLock on its performance improvement and flexible policy support. The experiments use the TPC-C workload to show application-level performance.

Performance improvement over DSLR, DrTM and NetChain. We show the performance improvement of NetLock over the state-of-the-art solutions DSLR, DrTM and NetChain. We show two scenarios, and each is conducted under two TPC-C workload settings (high-contention and low-contention). Figure 10 shows the throughput and latency of the first scenario, where we use ten machines as clients to generate requests, and two machines as lock servers that run NetLock, DSLR or DrTM; NetChain only uses the switch, and does not use any servers for lock processing. Because NetChain treats both shared and exclusive locks as exclusive locks, it has many fail-and-retry operations which degrade its performance. With the co-design of the switch and lock servers, NetLock avoids a large number of fail-and-retry operations caused by contentions compared to NetChain. The clients only need to retry when there is a packet loss or deadlock. By offloading using a fast switch to process most requests and avoiding most of retries, NetLock improves the transaction throughput by 14.9× (28.6×, 3.5×) and 18.4× (33.5×, 4.4×) in low and high contention settings respectively compared with DSLR (DrTM, NetChain). Besides throughput, NetLock also reduces both the average and tail latencies, by up to 20.3× (66.8×, 5.4×) and 18.4× (653.9×, 23.1×) respectively compared with



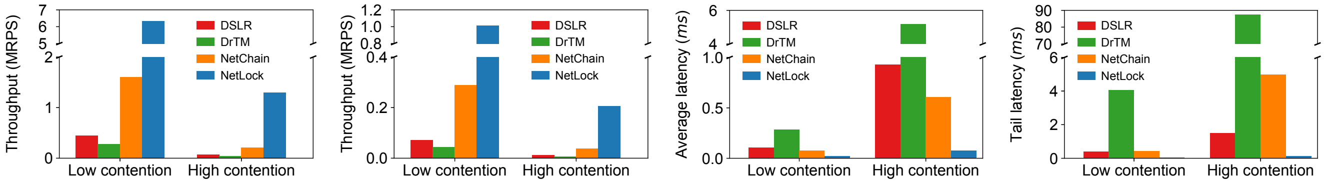
(a) Lock Throughput.

(b) Transaction Throughput.

(c) Average latency.

(d) Tail latency.

Figure 10: System comparison under TPC-C with ten clients and two lock servers.



(a) Lock Throughput.

(b) Transaction Throughput.

(c) Average latency.

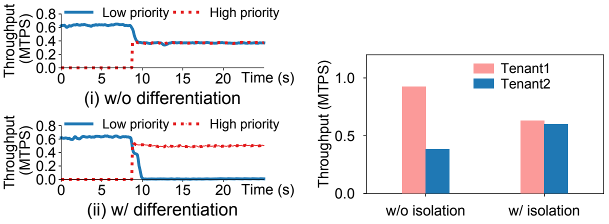
(d) Tail latency.

Figure 11: System comparison under TPC-C with six clients and six lock servers.

DSLR (DrTM, NetChain). Figure 11 shows the results of the second scenario, where we use six machines as clients and six machines as lock servers for NetLock, DrTM and DSLR, and NetChain only uses the switch for lock processing. While in this scenario the lock servers are less loaded than they are in the previous scenario, NetLock still achieves significant improvement. Compared to DSLR (DrTM, NetChain), it improves the transaction throughput by up to 17.5× (33.1×, 5.5×), and reduces the average and tail latency by up to 11.8× (65.6×, 7.7×) and 10.5× (602.8×, 34.4×) respectively.

Policy support. Besides performance, another benefit of NetLock is its flexible policy support. The default policy is starvation-freedom which helps reduce tail latency and is shown in the previous experiment. Here we show the other two representative policies mentioned in Section 4.4. Figure 12(a) shows how NetLock provides service differentiation with priorities. There are two tenants with five clients each. Without service differentiation, both tenants have similar performance when the high-priority tenant begins to send requests. With service differentiation, the high-priority tenant is prioritized over the low-priority tenant.

Figure 12(b) shows how NetLock enforces performance isolation. Different from the service differentiation experiment, we assign seven clients to tenant 1 and three clients to tenant 2. Because tenant 1 has more clients to generate requests at a faster rate than tenant 2, when there is no performance isolation, tenant 1 starves tenant 2 and achieves higher throughput. With performance isolation, each tenant can only obtain the tenant's own share, which is half of the resources here, and two tenants achieve similar performance.



(a) Service differentiation.

(b) Performance isolation.

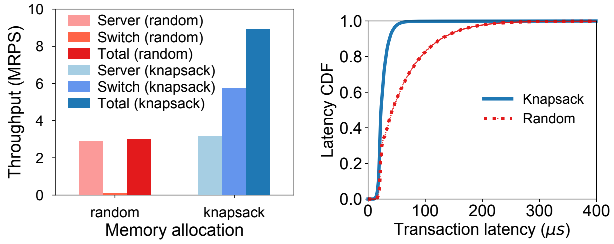
Figure 12: Policy support of NetLock.

6.4 Memory Management

We evaluate the efficiency of the memory allocation algorithm and the impact of the switch memory size on system performance. The experiments are conducted with ten clients and two lock servers under TPC-C workload (ten warehouses per node).

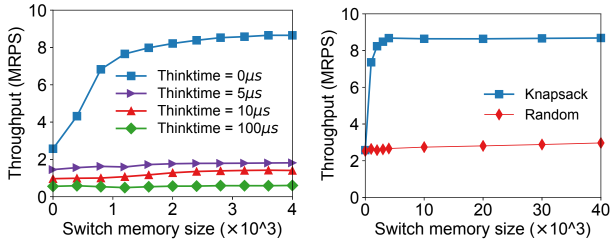
Memory allocation. NetLock uses an optimal knapsack algorithm to efficiently pack popular locks into limited switch memory to maximize system performance. We compare it with a strawman algorithm that randomly divides locks between the switch and the servers. Figure 13(a) shows the lock request throughput and its breakdown on the lock switch and the servers. Because the random approach does not allocate the switch memory to the popular locks, the switch only processes a small number of lock requests. On the other hand, NetLock efficiently utilizes the limited switch memory to process as many requests as possible, and improves the total throughput by 2.95×. Figure 13(b) shows the latency CDF of the two algorithms. Because the random approach processes most lock requests in the lock servers, it incurs high latency, especially at the tail. In comparison, because of the efficient memory allocation, NetLock processes many requests directly in the switch and significantly reduces the transaction latency.

Switch memory size. As discussed in Section 4.5, the impact of switch memory size on the system performance depends on the think time and the memory allocation mechanism. Figure 14(a) shows the impact of memory size on throughput under different think times. The think time determines the maximum *turnover* rate of a memory slot, which limits the maximum throughput the switch can support with a given amount of memory. From the figure, we can see that when the think time is zero, the throughput quickly grows up with more memory slots and achieves 8.64 MRPS at the maximum. As the think time increases, the throughput is smaller and also grows more slowly. When the think time is 100 μs, the system can only achieve 0.60 MRPS because the memory in the



(a) Throughput. (b) Latency.

Figure 13: Impact of memory allocation mechanisms.



(a) Think time. (b) Memory allocation.

Figure 14: Impact of memory size under different memory allocation mechanisms and think times.

switch is not efficiently utilized. Thus, NetLock is more suitable for low-latency transactions.

Figure 14(b) shows the impact of memory size on throughput under different memory allocation mechanisms. Because the knapsack algorithm used by NetLock can efficiently utilize switch memory, the throughput increases quickly with more memory slots, and reaches the maximum throughput of 8.61 MRPS with 3000 slots. We emphasize that the maximum throughput is bottlenecked by the speed of generating requests from the clients and the intrinsic contentions between the transactions, not the switch. On the other hand, because the random algorithm allocates the switch memory to a random set of locks, it utilizes the switch memory poorly. As a result, more memory slots does not help improve the transaction throughput of the system under the inefficient memory allocation algorithm. Under this workload, NetLock can achieve significant improvement with 5×10^3 memory slots (160KB), which is only a small fraction of the switch memory (tens of MB).

6.5 Failure Handling

We finally evaluate how NetLock handles failures. We manually stop the switch to inject a switch failure, and then reactivate the switch. Figure 15 shows the throughput time series. At time 10 s, we let the NetLock switch stop processing any packets. The system throughput drops to zero immediately upon the switch "failure". Then we reactivate the switch to process lock requests. The switch retains none of its former state or register values. During the switch failure, the client keeps retrying and requesting locks for their transactions. Upon reactivation, some lock requests of a transaction can be processed by the new (reactivated) switch while others may be lost. NetLock uses leasing to handle this situation. After reactivation, the system throughput returns to the pre-failure level instantly. NetChain can be applied to chain several NetLock switches to further reduce the temporary downtime.

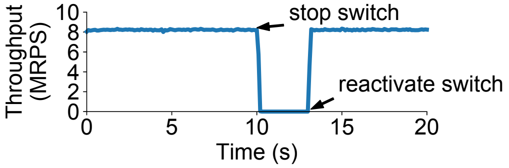


Figure 15: Failure handling result.

7 RELATED WORK

Lock management. Today's centralized lock managers are implemented on servers [3, 23, 24, 29, 48]. While they are flexible to support various policies, they suffer from limited performance. Recent work has exploited decentralized lock managers for high performance [17, 40, 46, 49]. These decentralized solutions achieve high performance at the cost of limited policy support. Compared to them, NetLock is a centralized lock manager that provides both high performance and the flexibility to support rich policies.

Fast distributed transactions. There is a long line of research on fast distributed transaction systems [11, 14, 19, 30, 34, 39, 44, 45, 47, 50, 51]. These systems use a variety of techniques to improve performance, from designing new transaction algorithms and protocols, to exploiting new hardware capabilities like RDMA and hardware transactional memory. NetLock can be used as a fast lock manager to improve general transactions without any modifications to transaction protocols.

In-network processing. Recently there have been many efforts exploiting programmable switches for distributed systems, such as key-value stores [28, 36–38], coordination and consensus [15, 16, 27, 35, 41, 52], network telemetry [22, 26], machine learning [42, 43], and query processing [33]. Kim *et al.* [31] proposes to extend switch memory with server memory using RDMA. NetLock provides a new solution for lock management, does not rely on RDMA, and includes an optimal memory allocation algorithm to integrate switch and server memory for the lock manager.

8 CONCLUSION

We present NetLock, a new centralized lock management architecture that co-designs programmable switches and servers to simultaneously achieve high performance and rich policy support. NetLock provides orders-of-magnitude higher throughput than existing systems with microsecond-level latency, and supports many commonly-used policies on performance and isolation. With the end of Moore's law, we believe NetLock exemplifies a new generation of systems that leverage network programmability to extend the boundary of networking to IO-intensive workloads.

Ethics. This work does not raise any ethical issues.

Acknowledgments. We thank our shepherd Kun Tan and the anonymous reviewers for their valuable feedback on this paper. This work is supported in part by NSF grants CCF-1629397, CRII-1755646, CNS-1813487, CNS-1845853, and CCF-1918757, a Facebook Communications & Networking Research Award, and a Google Faculty Research Award.

REFERENCES

[1] 2018. Cavium XPliant. https://www.cavium.com/.
[2] 2018. Intel Data Plane Development Kit (DPDK). http://dpdk.org/.
[3] 2018. Teradata: Business Analytics, Hybrid Cloud & Consulting. http://www.teradata.com/.
[4] 2019. Amazon Web Services. https://aws.amazon.com/.
[5] 2019. Broadcom Ethernet Switches and Switch Fabric Devices. https://www.broadcom.com/products/ethernet-connectivity/switching.
[6] 2019. CloudLab. https://www.cloudlab.us.
[7] 2019. Google Cloud. https://cloud.google.com/.
[8] 2019. Microsoft Azure. https://azure.microsoft.com/.
[9] 2020. Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.
[10] 2020. TPC-C. http://www.tpc.org/tpcc/.
[11] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Coordination avoidance in database systems. In *Proceedings of the VLDB Endowment*.
[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM CCR* (2014).
[13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*.
[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally distributed database. In *USENIX OSDI*.
[15] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos made switch-y. *SIGCOMM CCR* (2016).
[16] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at network speed. In *ACM SOSR*.
[17] Ananth Devulapalli and Pete Wyckoff. 2005. Distributed queue-based locking using advanced network features. In *IEEE ICPP*.
[18] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *ACM SOSP*.
[19] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *ACM SOSP*.
[20] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor Von Laszewski, Warren Smith, and Steven Tuecke. 1997. A directory service for configuring high-performance distributed computations. In *IEEE HPDC*.
[21] Cary Gray and David Cheriton. 1989. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *ACM SOSP*.
[22] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*.
[23] Andrew B Hastings. 1990. Distributed lock management in a transaction processing environment. In *Symposium on Reliable Distributed Systems*.
[24] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas F Wenisch. 2017. A top-down approach to achieving performance predictability in database systems. In *ACM SIGMOD*.
[25] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*.
[26] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. QPipe: Quantiles Sketch Fully in the Data Plane. In *ACM CoNEXT*.
[27] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI*.
[28] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP*.
[29] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. 2008. Deadlock immunity: Enabling systems to defend against deadlocks. In *USENIX OSDI*.
[30] 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*.
[31] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. 2018. Generic External Memory for Switch Data Planes. In *ACM SIGCOMM HotNets Workshop*.
[32] Leslie Lamport. 1974. A new solution of Dijkstra's concurrent programming problem. *CACM* (1974).
[33] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. 2019. The Case for Network Accelerated Query Processing.. In *CIDR*.
[34] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *ACM SOSP*.
[35] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R.K. Ports. 2016. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*.
[36] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be Fast, Cheap and in Control with SwitchKV. In *USENIX NSDI*.
[37] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *ACM ASPLOS*.
[38] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019.. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *USENIX FAST*.
[39] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions.. In *USENIX OSDI*.
[40] 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 *IEEE CCGrid*.
[41] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *USENIX NSDI*.
[42] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *ACM SIGCOMM HotNets Workshop*.
[43] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701* (2019).
[44] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *ACM SIGMOD*.
[45] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *USENIX OSDI*.
[46] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *ACM SOSP*.
[47] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. 2014. Salt: Combining ACID and BASE in a Distributed Database.. In *USENIX OSDI*.
[48] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. In *Proceedings of the VLDB Endowment*.
[49] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed Lock Management with RDMA: Decentralization without Starvation. In *ACM SIGMOD*.
[50] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The end of a myth: Distributed transactions can scale. In *Proceedings of the VLDB Endowment*.
[51] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. 2013. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *ACM SOSP*.
[52] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. In *Proceedings of the VLDB Endowment*.