Fast and Scalable In-network Lock Management Using Lock Fission

Hanze Zhang, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University; Shanghai AI Laboratory; MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University;* Ke Cheng, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Rong Chen, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University; Shanghai AI Laboratory; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Haibo Chen, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China; Key Laboratory of System Software* (*Chinese Academy of Sciences*)

https://www.usenix.org/conference/osdi24/presentation/zhang-hanze

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Fast and Scalable In-network Lock Management Using Lock Fission

Hanze Zhang[1,2,4] Ke Cheng[1,3] Rong Chen[1,2,3] Haibo Chen[1,3,5]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University [2]Shanghai AI Laboratory
[3]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
[4]MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University
[5]Key Laboratory of System Software (Chinese Academy of Sciences)

Abstract

Distributed lock services are extensively utilized in distributed systems to serialize concurrent accesses to shared resources. The need for fast and scalable lock services has become more pronounced with decreasing task execution times and expanding dataset scales. However, traditional lock managers, reliant on server CPUs to handle lock requests, experience significant queuing delays in lock grant latency. Advanced network hardware (e.g. programmable switches) presents an avenue to manage locks without queuing delays due to their high packet processing power. Nevertheless, their constrained memory capacity restricts the manageable lock scale, thereby limiting their effect in large-scale workloads.

This paper presents FISSLOCK, a fast and scalable distributed lock service that exploits the programmable switch to improve (tail) latency and peak throughput for millions of locks. The key idea behind FISSLOCK is the concept of lock fission, which decouples lock management into grant decision and participant maintenance. FISSLOCK leverages the programmable switch to decide lock grants synchronously and relies on servers to maintain participants (i.e., holders and waiters) asynchronously. By using the programmable switch for routing, FISSLOCK enables on-demand fine-grained lock migration, thereby reducing the lock grant and release delays. FISSLOCK carefully designs and implements grant decision procedure on the programmable switch, supporting over one million locks. Evaluation using various benchmarks and a real-world application shows the efficiency of FISSLOCK. Compared to the state-of-the-art switch-based approach (NetLock), FISSLOCK cuts up to 79.1% (from 43.0%) of median lock grant time in the microbenchmark and improves transaction throughput for TATP and TPC-C by 1.76× and 2.28×, respectively.

1 Introduction

Distributed lock services are essential building blocks for coordinating concurrent access to shared resources in numerous distributed systems, such as OLTP databases [23, 62, 67], file systems [18, 51], and rich cloud-based systems [1, 22, 25, 52]. Modern distributed systems commonly rely on fine-grained locks to concurrently access near-billion-scale datasets, such as files and directories [57, 58, 61], database tuples [10, 12, 14], and knowledge graphs [2, 4, 15].

With the prevalence of affordable high-performance networks (e.g., RDMA) and high-capacity persistent memory (e.g., Intel Optane) in modern datacenters, it is not uncommon to see microsecond-scale execution time in distributed in-memory systems [20, 60, 66, 71–73, 78] (see Table 1). As a result, the overhead of granting locks (10–100 µs) becomes non-trivial (e.g., comparable to task execution time) and even dominates the end-to-end performance [62, 75, 76].

Distributed lock managers [6, 13, 30, 54, 55, 65, 76] are commonly designed in a centralized manner to handle lock requests and grant locks. This makes it easy to enable powerful features such as latency predictability [31, 38, 46], starvation freedom [36], and performance isolation [76]. Specifically, before and after accessing a set of objects, the corresponding locks must be acquired from and released to the lock manager, which acts as a central point for granting and managing locks. Traditional lock managers rely on commodity servers to serve lock requests, which imposes one network round-trip overhead in granting locks and often incurs significant queueing delay due to limited request processing throughput of the server CPUs.

To overcome these drawbacks, it has recently been proposed to use the programmable switch as a centralized lock manager to host part of locks [76], as it offers lower latency and higher throughput than servers for packet processing. Further, using the switch to handle lock requests halves the network overhead due to its central network location. However, due to the limited switch memory (typically just a few MB), only a small fraction of locks (e.g., less than 10,000 [76]) can be hosted on a programmable switch for large-scale workloads. This is mainly because the variable-size metadata of a lock—a set for holders and a queue for waiters—consumes several hundred bytes of switch memory. Moreover, the performance of existing switch-based approaches is heavily dependent on the workloads, which must be both highly skewed and predictable to achieve significant improvement. It is also difficult, or even impossible, to dynamically update the data plane model of a programmable switch for exchanging locks.

Key insight. The centralized lock manager can be divided into two phases: *synchronous* grant decision and *asynchronous* participant maintenance. Making a grant decision is based solely on the *fixed-size* metadata (lock mode),

Table 1: *Task execution time of distributed in-memory systems.*

Systems	Workload	Exec. Time
Txn. Processing [37, 73]	TPC-C / TATP	7 / 2.8 µs
File System [18, 51]	Read / Write / Mkdir	1 / 10 / 20 µs
Key-value Store [71, 78]	Search / Insert / Delete	8 / 15 / 12 µs
Online Trading [20]	Trade Execution	10 µs

while maintaining participants also requires the remaining *variable-size* metadata (holders and waiters).

Our approach. Armed with the above insight, we design FissLock, a *switch-centric* lock managing system that also leverages programmable switches but in a new way to offer significant performance improvement (both latency and throughput) and memory efficiency for millions of locks. The key idea is *lock fission*, which decouples grant decision and participant maintenance procedures of the lock manager and deploys the two parts on the programmable switch and commodity servers, respectively. Specifically, the switch acts as the *decider* that immediately makes a decision and replies to the requester if the lock is granted. Meanwhile, the lock request is further routed to the server, hosting the *agent* of the lock, for updating lock holders and waiters with rich semantics. The main advantages of lock fission are three-folds. First, it stores only a small, fixed-size lock mode in switch memory to accelerate millions of locks, which is two orders of magnitude larger than existing switch-based approaches. Second, it leverages high-speed, line-rate request processing of the switch to concurrently grant locks with lower latency and higher throughput than server-based approaches. Third, it delegates lock management tasks (i.e., maintaining holders and waiters) to the server of the lock holder, enhancing load balance and data locality for diverse workloads.

FissLock proposes the first design of the lock fission protocol, which splits the lock manager into a centralized, stationary decider on the switch, and migratable agents for each lock on the servers. The protocol introduces new workflows for acquiring and releasing locks, which allow for halfway responses from the switch when acquiring grantable locks and migrating agents among servers to exploit locality and balance lock management loads. FissLock further addresses the anomalies in the protocol caused by network exceptions using incarnation checks.

FissLock stores small fixed-size metadata (e.g., lock mode) for each lock in the switch memory. By carefully designing on-switch metadata structure, a single switch with a few MB of memory can host millions of locks. To implement the decider of the lock fission protocol on an ASIC-based programmable switch (e.g., Intel Tofino [16]), FissLock devises a 6-stage pipeline to process four types of packets in the protocol. Each stage employs one or more match-action units in the switch data plane to perform simple operations that a programmable switch can afford, such as metadata matching and updating, and packet destination selection.

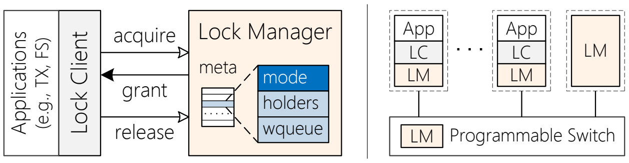

Fig. 1: *Distributed lock management.*

We implemented FissLock from scratch on a Tofino switch [16] and evaluated it using a microbenchmark, two transaction benchmarks, and a real-world application. Our experimental results show that FissLock cuts up to 79.1% (from 43.0%) of median lock grant time in the microbenchmark and improves the transaction throughput of TATP [12] and TPC-C [14] by 1.76× and 2.28×, respectively, compared to the state-of-the-art switch-based approach (NetLock [76]). We built a Redis-backed mobile banking application [11] with FissLock, which is one order of magnitude faster than Redis's official implementation (RedLock [5]).

Contributions. We summarize our contributions as follows:

- An in-depth analysis of performance issues in existing lock manager designs for modern distributed in-memory systems (§2).
- A new centralized lock management scheme, *lock fission*, which decouples grant decision and participant maintenance to embrace the best of both programmable switches and servers (§3).
- A switch-centric lock manager design that enables the lock fission protocol (§4) and implements grant decision for millions of locks on the programmable switch (§5).
- A prototype implementation and evaluation that demonstrates the efficacy of FissLock over state-of-the-art (§7).

2 Background

2.1 Distributed Lock Management

The distributed lock service commonly uses a centralized lock manager (LM) to handle all requests and grant the lock for various applications, such as transactions and file systems. As shown in the left part of Fig. 1, before reading or updating the protected data, applications—through the lock client (LC)—*acquire* the lock in shared or exclusive mode by sending the request to the lock manager, and waits until the lock manager *grants* the lock. After accessing the data, applications *release* the lock to the lock manager asynchronously. The lock manager maintains the metadata of locks (*meta*) identified by a unique lock ID. The metadata of each lock contains a mode of lock (*mode*), a set of lock holders (*holders*), and a queue for waiters (*wqueue*) [13]. The mode is a small, fixed-size flag (2 bits) that represents the current lock state (i.e., free, exclusive, or shared) and decides the grant of locks. Both holders and wqueue require large, variable-size data structures (up to hundreds of bytes) that represents the current lock participants and enables flexible locking policies (e.g., priority and fairness).

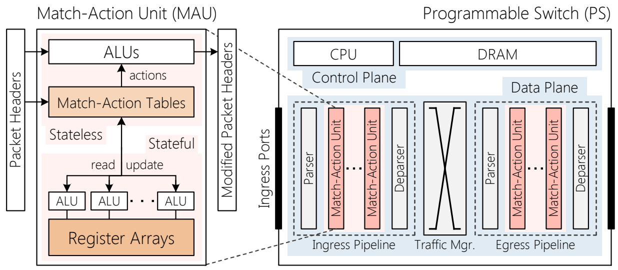

Fig. 2: *The architecture of programmable switches (PS) and the internal structure of match-action units (MAU).*

The lock manager can run on dedicated servers to avoid interference with applications [54, 55, 65] (called SrvLock). However, it may not scale well with fast-growing application workloads and datasets. Therefore, the lock manager can also be partitioned and co-located with applications to further exploit locality [6, 13, 30] (called ParLock), but it may suffer from load imbalance under skewed workloads. Recently, the programmable switch has been proposed to handle skewed workloads by managing a small fraction of hot locks (e.g., less than 10,000) directly on the switch (i.e., NetLock [76]), because it provides higher throughput than servers, halves the network latency, and saves server resources. However, due to the limited memory resources of programmable switches (a few MB), just a part of the workload can be accelerated, and the rest will be downgraded to server-based solutions. The right part of Fig. 1 illustrates the above three solutions for distributed lock management.

2.2 Programmable Switch (PSwitch)

Programmability is becoming a trend in modern network switch design, with support from major manufacturers like Cisco [3], Broadcom [33], and Intel [16]. Compared with commodity servers, programmable switches possess orders of magnitudes higher throughput (several billion packets per second) and lower delay (less than 1 µs) for packet processing [34]. Yet, only very limited memory resources (a few MB) are available. As shown in Fig. 2, modern programmable switches have a general-purpose CPU with DRAM (i.e., the *control plane*) attached to the switch ASIC (i.e., the *data plane*) via a PCIe bus. The control plane hosts a Linux-based operating system that manipulates the switch ASIC as a device. The data plane is programmable via P4 Language [19], which describes the logic of packet parsing, processing, and forwarding. Specifically, packet processing is realized as pipelines of *match-action units* (MAUs), which perform pre-defined packet modifications (*actions*) according to the value of specific fields in the packet header. Incoming packets can be either forwarded to a single egress port (*unicast*) or replicated to multiple egress ports (*multicast*). Each MAU reads and updates user-defined data stored in *register arrays* (hundreds of KB) once for each packet, and conditionally modifies packet header fields according to *match-action tables* (up to over 1 MB).

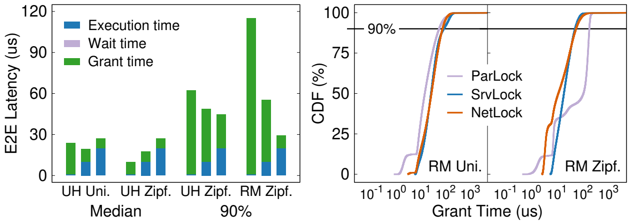

Fig. 3: *The end-to-end median and 90^{th} percentile latency breakdown with different task execution times (1 µs, 10 µs, and 20 µs) under update-heavy (UH) and read-mostly (RM) workloads (left), and the lock grant time distribution under read-mostly Uniform (Uni.) and Zipfian (Zipf.) workloads when using different LMs (right).* **Testbed:** *An 8-node cluster with an Intel Tofino switch.* **Workload:** *A microbenchmark with one million locks (see §7.1 for details).*

2.3 Performance Analysis

Grant time becomes increasingly important. The end-to-end latency of typical distributed tasks is mainly composed of execution time and lock acquire/release time. Lock release time is irrelevant as locks are typically released asynchronously. Lock acquire time consists of two parts—*wait time* and *grant time*, which denotes the duration that the request is suspended in wqueue and the rest, respectively. Nowadays, microsecond-scale execution time becomes common in distributed in-memory systems [18, 51, 60, 66, 71–73, 78]. Meanwhile, the rapidly increasing size of datasets (e.g., millions of objects per thread [29, 66]) dramatically enlarges the lock space, resulting in lower lock contention rate and less wait time. Consequently, grant time becomes non-negligible in the end-to-end task latency. As shown in Fig. 3 (left), grant time accounts for a significant portion of the median and tail end-to-end task latency under workloads with varied read-write ratio and skewness, while wait time is negligible because of the low lock contention.

Despite its importance, grant time has not drawn enough attention in existing lock manager designs, in which we observe three major performance issues (see Fig. 3 (right)).

Issue#1: Unstable latency. All existing approaches rely on server CPUs to process (partial or all) lock requests, introducing significant queueing delay that makes the grant time unstable. Since the server receives and processes packets in batches, the handling latency of requests is proportional to their positions in the batch. Due to the limited packet processing power of server CPUs, this results in non-trivial grant time variance, from a few µs to over 100 µs in our testbed. Note that requests handled by the switch do not exhibit apparent queueing delay because of line-rate processing.

Issue#2: Limited acceleration. Both ParLock and NetLock adopt fast-path request handling to accelerate a part of lock requests, but the portion of accelerated requests is rather limited. ParLock partitions the locks to handle some lock requests locally, which saves 1 RT as compared with SrvLock. However, the portion of locally handled requests is inversely proportional to the cluster size (e.g., 12.5% in our 8-node setup). NetLock manages hot locks on the programmable switch, halving the required round trips and eliminating queueing delay. However, due to limited switch memory capacity, the switch can only manage thousands of locks, which are insufficient for protecting million-scale datasets without exhibiting significant contention (see §7.7). When managing 1 million locks, even with workload profiling in advance, NetLock only accelerates 1% and 27% of grants under the Uniform and Zipfian workloads, respectively, as most requests are handled by the lock server instead of the switch.

Issue#3: Workload sensitivity. The performance of ParLock and NetLock are sensitive to workload attributes, which results in their dependence on prior knowledge of the workload. ParLock partitions locks statically, which incurs severe load imbalance problem under skewed workloads. In our experiment on the Zipfian read-mostly workload, 56.7% of requests are processed by one server, which throttles the LM and results in extremely high grant time. Due to the limited switch capacity, NetLock prefers skewed workloads and heavily relies on workload profiling for detecting hot locks. It falls back to SrvLock on occasions that the workload is uniform or has dynamic patterns (e.g., e-commerce).

3 Approach and Overview

System model and design goals. FissLock is a distributed lock management system that uses programmable switches to accelerate the processing of millions of locks across diverse workloads. It is designed for distributed in-memory systems that rely on a centralized lock service to coordinate concurrent access of microsecond-scale tasks to large-scale shared datasets. Unlike lock-based coordination services like Zookeeper [32] and Chubby [21], which aim for reliable but coarse-grained coordination, FissLock is not designed to achieve high availability. Instead, FissLock has three high-level design goals:

- **Efficiency**: Grant locks in single-digit microseconds to meet the common needs of microsecond-scale tasks.
- **Pervasiveness**: Unleash full-scale acceleration for million-scale locks, making it feasible for large-scale systems.
- **Robustness**: Ensure good yet stable performance for diverse or dynamic workloads without prior knowledge.

Key insight. The lock management can be divided into two phases: grant *decision* and participant *maintenance*. The decision phase determines whether the lock can be granted with regard to the current lock mode, while the maintenance phase manages lock participants (i.e., holders and waiters) accordingly. We recognize two key insights that motivate the split design of a centralized lock manager. First, decision making must be *synchronous* (i.e., executed before handling other requests) to ensure the correctness of lock semantics, while participant maintenance can be *asynchronous* to shorten the critical path of granting a lock. Second, decision making relies solely on the *fixed-size* metadata (shared or exclusive), while participant maintenance also requires the remaining *variable-size* metadata (holders and waiters).

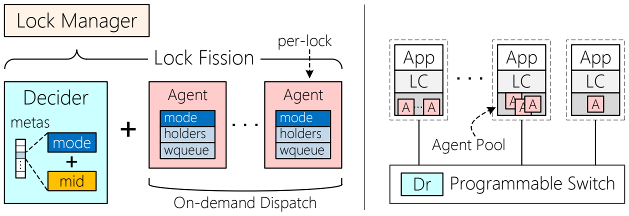

Fig. 4: *The lock fission scheme (left) and the system architecture of* FissLock *(right).*

Our approach. We propose *lock fission* as a technique that decouples decision and maintenance in terms of both functionality and metadata. Specifically, the lock manager is split into a centralized *decider* and multiple per-lock *agents* (see Fig. 4 (left)). The decider records each lock's mode and makes granting decisions for lock requests accordingly, while the agent stores and maintains the remaining lock metadata (e.g., holders and waiters) of the corresponding lock. The lock acquisition request is first sent to the decider, which makes decisions and replies instantly if the lock is granted. Simultaneously, the decider forwards the request to the responsible lock agent, which updates holders and waiters according to the decision. Release requests are also sent to the decider and forwarded to the agent. When the last holder releases, the agent grants the lock to the next holder or frees the lock. In both cases, the decider is notified in advance to update the lock mode.

Lock fission provides the opportunity to accustom workload attributes without prior knowledge. By recording the resident machine ID (*mid*) of each lock's agent, the decider is always able to route requests to the latest location of agents, which supports the dynamic migration of lock agents among machines. To balance the request handling load among machines, agents are *on-demand dispatched* to the lock holder's machine. When the lock is freed or transferred to waiters, the agent is deconstructed or migrated to the waiter's machine. The migration of agents also enables most release requests to be processed by local agents, thereby shortening the network path of lock release operations.

Lock fission meets the design goals by exploiting the *packet processing strengths* of programmable switches and the *high memory capacity* of commodity servers simultaneously. First, metadata for decision is small, fixed-size data, which enables decision making for million-scale locks with limited switch memory capacity. Second, since maintenance is decoupled with decision, servers are not involved in the lock granting critical path, which eliminates the queueing delay. Third, lock fission does not require any prior knowledge of the workload and resolves the load imbalance problem by dynamic lock agent migration.

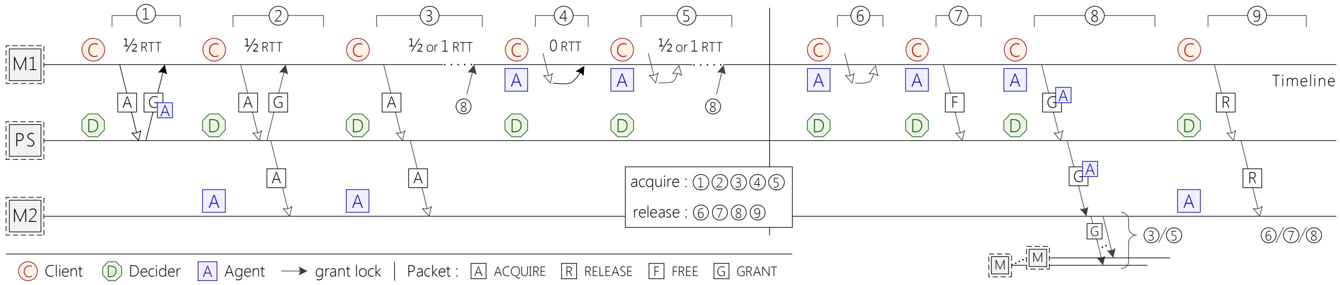

Fig. 5: *The lock acquisition and release workflow in the lock fission protocol.*

System overview. FissLock is a switch-centric lock management system that applies lock fission to enable efficient and centralized management of million-scale locks. As shown in Fig. 4 (right), FissLock is composed of lock clients (LC), a decider (Dr), and lock agents (A). Lock clients are libraries that encapsulate lock acquire/release requesting functionalities into APIs. The decider resides on the switch for accelerating lock grants and routing requests to agents. Each machine owns an agent pool that manages all agents on it. Lock requests forwarded from the decider are received by the agent pool, which finds the responsible agent for each request and hands over the request. The agent subsequently updates the lock metadata.

Applications acquire locks via the LC, which sends the request to the decider, or if the lock agent can be found locally, the lock agent. The decider makes decision and replies instantly, *multicasting* the request to the lock agent's resident machine at the same time. Packets arriving at machines are dispatched to the lock client (grant replies) and the agent pool (lock requests), which wakes up application tasks and calls agent functions to update the lock metadata respectively. When the lock is released, similar to the acquire case, the request is forwarded to or directly handed over to the lock agent. If the lock needs to be granted to the next holder, the agent is transferred along with the lock ownership.

4 FissLock

This section describes the lock fission protocol implemented by FissLock. Although the design principles of lock fission are independent of specific lock mechanisms used, we elaborate on the read-preferring design as an example, and briefly discuss the write-preferring design in §6.

4.1 Lock Operation Workflow

Fig. 5 illustrates the nine possible workflows (①–⑨) for acquiring and releasing locks in the lock fission protocol. The branches executed in each workflow are marked on the pseudocode presented in Fig. 6, Fig. 7, and Fig. 8.

Lock acquisition workflow (①–⑤). To acquire a lock, the lock client (C) sends an ACQUIRE request to the lock decider (D) (Line 5 in Fig. 6). The decider makes a lock granting decision by examining the mode of the lock (*meta.mode*) and the requested mode in the packet (*pkt.mode*). If the lock is free (①), the decider will update the lock mode and immediately grant a lock with an empty agent (A) to the client by returning a GRANT packet (Lines 2–6 in Fig. 8). After receiving the packet, the client calls the grant function of the agent pool to initialize the agent and add it to the pool (Lines 17–19 in Fig. 7). If the lock is being held and both modes are SHARED (②), the decider will still immediately grant the lock to the client and multicasts the ACQUIRE request to the agent (Lines 8–11 in Fig. 8). The agent will add the requester to holders later (Lines 2–4 in Fig. 7). Finally, if the lock cannot be granted immediately (③), the decider will forward the request to the agent, and the agent will append the requester to the wait queue (Line 6 in Fig. 7). In rare cases (④ and ⑤), the agent is on the same machine since another client on the machine is hosting the lock, such that the client will locally acquire the lock by calling the acquire function of the agent pool (Line 2 in Fig. 6). The agent can make a decision by itself—to grant (④) or to wait (⑤)—without consulting the decider. This is because the agent always has the latest lock mode and does not need to update the decider.

Lock release workflow (⑥–⑨). In the lock fission protocol, the agent (A) is always located on the same machine as the current holder (e.g., the first holder of the shared lock).

```
# lid|mid|tid: lock|machine|task ID
# mode: lock mode in {FREE=00, EXCLUSIVE=10, SHARED=11}

sslock_acquire(lid, mid, tid, mode)                    ①②③④⑤
1    if agents.find(lid) then    # local agent (rare)     ④⑤
2    |  if acquire(lid, mid, tid, mode) then
3    |  |  return    # local grant                          ④
4    else    # remote agent                               ①②③
5    |  net_send(ACQUIRE, {lid, mid, tid, mode})
     # wait for grant
6    pkt = net_recv(GRANT, lid, mid, tid)                ①②③⑤
7    if pkt.agent != nil then    # add agent
8    |  grant(lid, mid, tid, mode, pkt.agent)              ①③

sslock_release(lid, mid, tid)                           ⑥⑦⑧⑨
9    if agents.find(lid) then    # local agent             ⑥⑦⑧
10   |  release(lid, mid, tid)
11   else    # remote agent (rare)                            ⑨
12   |  net_send(RELEASE, {lid, mid, tid})
```

Fig. 6: *Pseudocode of* FissLock *client lib implementation.*

```
# agent:
#   mode: lock mode in {FREE=00, EXCLUSIVE=10, SHARED=11}
#   holders: a set of lock holders {mid, tid}
#   wqueue: a queue of waiters {mid, tid, mode}

acquire(lid, mid, tid, mode)                                ②③④⑤
1  agent = agents[lid]
2  if mode == SHARED && agent.mode == SHARED then           ②④
3      agent.holders.add({mid, tid})
4      return TRUE   # grant lock
5  else                                                     ③⑤
6      agent.wqueue.append({mid, tid, mode})
7      return FALSE   # wait for grant

release(lid, mid, tid)                                      ⑥⑦⑧
8  agent = agents[lid]
9  agent.holders.remove({mid, tid})
10 if agent.holders.empty() then                            ⑦⑧
11     agents.remove(lid)   # remove agent
12     if agent.wqueue.empty() then                         ⑦
13         net_send(FREE, {lid})   # free agent
14     else   # transfer agent and grant lock               ⑧
15         next = agent.wqueue.pop()
16         net_send(GRANT, {lid, next.mid, next.tid,
                               next.mode, agent})

grant(lid, mid, tid, mode, agent)                           ①③
17 agent.mode = mode
18 agent.holders.add({mid, tid})   # grant lock
19 agents.add(lid, agent)   # add agent
20 if agent.mode == SHARED then   # grant others
21     ..   # pop shared waiters and add to holders
22     ..   # send grant lock (w/o agent) to them
```

Fig. 7: *Pseudocode of* FISSLOCK *agent pool implementation.*

Therefore, when releasing a lock, its agent is highly probable to be local to the requester. The client requests the local agent to release a lock through calling the `release` function of the agent pool (Lines 9–10 in Fig. 6). If the lock is also held by other clients of the machine (⑥), the release completes immediately (Line 9 in Fig. 7). If there is no waiter (⑦), the agent pool will remove the local agent and send a `FREE` request to the decider (Lines 11–13 in Fig. 7). The decider will free the lock and drop the packet directly (Lines 14–15 in Fig. 8). If there are waiters (⑧), the lock with its agent will be transferred to the next holder, popping from the wait queue, by sending a `GRANT` packet to the decider (Lines 15–16 in Fig. 7). The decider updates the lock metadata and forwards the packet to the machine of the next holder (Lines 17–19 in Fig. 8). After receiving the packet, the client calls the `grant` function of the agent pool to maintain the agent and add it to the pool (Lines 17–19 in Fig. 7). Furthermore, if the lock could be shared with subsequent waiters, the client will pop them from the wait queue and add to the holders, sending a `GRANT` packet to each of them (Lines 20–22 in Fig. 7). If the client, without a local agent (e.g., one holder of a shared lock), releases the lock (⑨), it will send a `RELEASE` request to the decider (Line 12 in Fig. 6). The decider will then forward the request to the agent (Line 13 in Fig. 8), and the agent will call the `release` function (Lines 8–16 in Fig. 7) to release the lock, as in the above workflows (⑥–⑧).

```
# meta:
#   mode: lock mode in {FREE=00, EXCLUSIVE=10, SHARED=11}
#   mid: machine ID

process_acquire(pkt): # pkt: {lid, mid, tid, mode}          ①②③
1  meta = metas[pkt.lid]
2  if meta.mode == FREE then   # lock is free               ①
3      meta = {pkt.mode, pkt.mid}   # alloc agent
       # grant lock and assign (empty) agent
4      agent = {pkt.mode, {}, {}}
5      grant_pkt = pkt.append(agent)
6      forward_packet_to(GRANT, pkt.mid, grant_pkt)
7      return
8  if meta.mode == SHARED && pkt.mode == SHARED then        ②
       # grant lock
9      grant_pkt = pkt.append(nil)
10     forward_packet_to(GRANT, pkt.mid, grant_pkt)
   # acquire lock on agent
11 forward_packet_to(ACQUIRE, meta.mid, pkt)                ②③

process_release(pkt): # pkt: {lid, mid, tid}                ⑨
12 meta = metas[pkt.lid]
   # release lock on agent
13 forward_packet_to(RELEASE, meta.mid, pkt)

process_free(pkt): # pkt: {lid}                             ⑦
14 metas[pkt.lid] = {FREE, nil}   # free agent
15 drop_packet(pkt)

process_grant(pkt): # pkt: {lid, mid, tid, mode, agent}⑧
16 meta = metas[pkt.lid]
17 if pkt.agent != nil then   # transfer agent
18     meta = {pkt.mode, pkt.mid}
   # grant lock and assign agent
19 forward_packet_to(GRANT, pkt.mid, pkt)
```

Fig. 8: *Pseudocode of* FISSLOCK *decider implementation.*

4.2 Network Exceptions

The lock fission protocol splits the lock manager into a stationary decider on the switch and migratable agents for each lock on the servers. Since they are connected via the network, network exceptions including lost, out-of-order, and delayed packets may cause some anomalies. FISSLOCK addresses these anomalies by retransmission, rerouting, and incarnation checks, respectively. As requests for different locks do not interfere with each other, we only consider out-of-order and delayed packets pertaining to the same lock.

Lost packets. FISSLOCK uses different approaches to handle the loss of packets initially sent by the switch and servers.

Server-initiated packets. FISSLOCK addresses the loss of packets initially sent by servers through TCP-based retransmission. When a server receives a packet, it sends an acknowledgement (ACK) to the origin of the packet. The switch forwards ACKs without updating on-switch metadata. Specifically, the destination of `FREE` packets (⑦) is the switch instead of servers, in which case the switch sends the packet back as an ACK. Servers monitor the arrival of ACKs for each packet regularly and retransmit packets that are not ACKed within a certain time frame.

In cases where the ACK is lost or delayed, the retransmis-

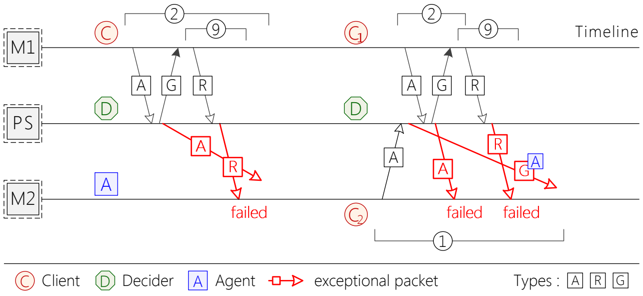

Fig. 9: *Two anomalies in the protocol due to out-of-order packets.*

sion mechanism can cause packet duplication, which is handled by servers through TCP. To prevent duplicated packets from corrupting on-switch metadata, the switch maintains a sequence number for each server to signify the number of processed packets. Servers track the number of packets sent to the switch, excluding retransmitted ones, and embed this number into all packets. If the incoming packet's number is not larger than the on-switch number, indicating that the packet is a duplicate, the switch does not update the lock metadata when processing the packet.

Switch-initiated packets. In workflows ① and ②, the lock request is granted by the switch instead of servers. Therefore, the GRANT packets in these workflows are considered switch-initiated. To avoid the complexities of monitoring ACK arrivals and executing retransmissions on the switch, FISSLOCK addresses the loss of switch-initiated packets in an alternative way—by setting a fixed timeout for lock acquisition operations. If an acquisition operation times out due to the absence of the GRANT packet, the client releases the lock (⑨) and retries the acquisition operation later. When processing the RELEASE packet, the switch frees the lock and drops the packet if it originates from the agent's server (①). If the RELEASE packet arrives at the agent, the agent removes the client from holders (②) or, if the client is not in holders (③), the wait queue.

Out-of-order packets. Packets with dependencies arriving out of order may lead to two anomalies. If ACQUIRE and RELEASE requests from the same requester are reordered (Fig. 9 (left)), the agent pool will fail to process the RELEASE request because the requester is not yet the holder or waiter of the lock. Further, the lock will never be released after granting it to the requester. If the ACQUIRE (and RELEASE) packet arrives before the agent is granted (Fig. 9 (right)), the agent pool will fail to process the ACQUIRE (and RELEASE) request because the agent does not exist. To avoid extra on-switch design, FISSLOCK resolves these anomalies by simply sending failed requests back to the decider. The decider then routes the request to the agent again, correcting the order.

Delayed packets. The ACQUIRE packet, which is immediately granted by the decider (②), may lead to anomalies when it arrives after the agent has been transferred (⑧) or freed (⑦). Note that the lock mode and the ACQUIRE packet must be both shared, as the lock is immediately granted. As shown in Fig. 10, the decider will incorrectly transfer the lock with its agent to the next holder (⑧) or grant the freed lock to a new client (①). The agent pool will fail to process the delayed packet.

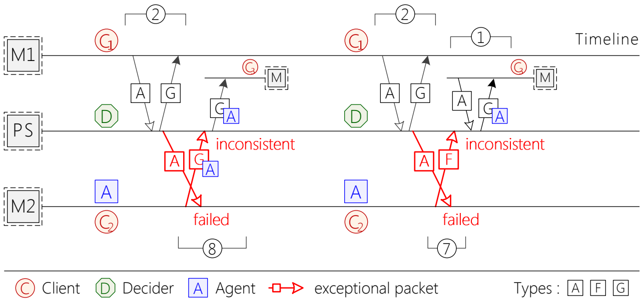

Fig. 10: *Two anomalies in the protocol due to delayed packets.*

FISSLOCK uses incarnation checks [68] to detect anomalies on the decider. Both the decider and the agent have a per-lock *incarnation* that is initially zero and is incremented when receiving a shared ACQUIRE request (②). The agent pool will include the expected incarnation in the GRANT and FREE packets. When the decider receives these packets and the lock is shared, it will check if its own incarnation matches the incarnation in the packet. If they match, the decider resets the incarnation and handles the request as normal. Otherwise, it refuses the packets, and the agent pool restores the agent to continue handling lock requests. Moreover, the failed ACQUIRE request will also be sent back to the decider, which will then route the request to the agent again.

4.3 Protocol Correctness

The lock fission protocol introduces two changes to the traditional reader-writer lock design: decoupling the decision process from the lock manager and allowing lock agents to migrate among servers. We argue the correctness of our lock fission protocol by showing that these changes preserve the reader-writer property [24, 53]. Specifically, we prove that the following two invariants always hold.

Invariant 1 (reader-writer exclusion): *Locks held in exclusive mode do not have any other holders; locks held in shared mode do not have any exclusive holders.*

In the traditional reader-writer lock design, all acquisition requests are processed by the lock manager, which maintains Invariant 1. In the lock fission protocol, requests are either processed by the local agent or sent to the decider. Both of them grant or suspend lock requests following the same criteria as traditional lock managers. Hence, Invariant 1 holds as long as the local agent and the decider always have consistent lock mode, and the lock mode correctly reflects the number of holders, which we prove as follows.

Lemma 1: *If the local agent exists, it has the same lock mode as the decider.*

Proof: The lock mode in the decider becomes shared or exclusive when a free lock is acquired for the first time (①) or the lock is transferred to a shared or exclusive waiter (⑧). In both cases, the agent is carried by the `GRANT` packet, so the local agent does not exist until the `GRANT` packet is received by the requester's machine. Moreover, the lock mode in the carried agent, which subsequently becomes the local agent, is identical to the updated lock mode in the decider. Before the local agent is removed, the decider only receives `ACQUIRE` and `RELEASE` packets, which do not change the lock mode. Hence, the lock mode in the decider remains consistent with the lock mode in the local agent.

Lemma 2: *The lock mode in the decider always correctly reflects the number of holders.*

Proof: We enumerate all possible lock mode transitions to prove Lemma 2. Initially, each lock is free and has no holders. The lock mode transits from free to shared or exclusive only when the lock is granted to a requester, which guarantees that the lock has at least one holder. If the lock mode is exclusive, subsequent `ACQUIRE` packets will not be granted. Hence, exclusive locks have at most one holder. The lock mode transits back to free only when receiving `FREE` packets whose incarnation matches the decider's incarnation, which indicates that all holders have released the lock. Similarly, the lock mode transits between shared and exclusive when receiving `GRANT` packets that have matched incarnation. In this case, all former holders have released the lock, and the destination of the `GRANT` packet becomes the new holder.

Invariant 2 (finite wait): *Waiters of the lock will be granted in finite time.*

Traditional lock managers decide to suspend a lock request and add the requester to the wait queue simultaneously. However, in the lock fission protocol, these two operations are decoupled and executed by different entities, i.e., the decider and the agent. We show that Invariant 2 still holds in the decoupled setup, which allows the agent to migrate among servers, by proving Lemma 3 and 4.

Lemma 3: *Lock acquisition requests that are suspended will eventually be added to the wait queue.*

Proof: Lock acquisition requests may be suspended by the agent locally (⑤) or remotely (③). In the former case (⑤), the local agent directly adds the requester to the wait queue when suspending the request. In the latter case (③), the decider, after deciding not to grant the lock, forwards the `ACQUIRE` request to the agent, which adds it to the wait queue. If the agent is not present due to packet reordering, the request is sent back to the decider. The decider then forwards it to the agent's latest location, guaranteeing that it will eventually be recorded in the agent's wait queue.

Lemma 4: *Waiters recorded in the wait queue will eventually be granted.*

Proof: All holders of a lock will eventually release it, either locally (⑥) or remotely (⑨). Therefore, the number of holders will eventually become zero, unless the lock is continuously granted to new shared holders (②), which FissLock averts by adopting a starvation prevention mechanism (see §6). When the number of holders reaches zero, at least one waiter is granted and becomes the new holder (⑧). Thus, according to induction, all waiters will eventually be granted.

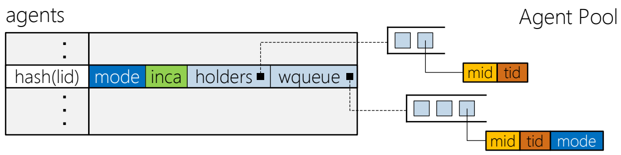

Fig. 11: *The main structures in the agent pool.*

5 Design

5.1 On-server Agent Pool

Data structures. As shown in Fig. 11, the agent pool uses a hash table to store granted agents, which is shared by all clients on the same machine. Each agent maintains complete lock metadata, including mode, incarnation, holders, and wqueue. The holders is an unordered set of current lock holders (*mid*, *tid*), and the wqueue is a FIFO queue of pending lock requests (*mid*, *tid*, *mode*). The incarnation (*inca*) is used to handle delayed packets.

Agent operations. The agent pool adds an agent when receiving a `GRANT` packet with agent information (Lines 17–19 in Fig. 7) and removes an agent when the lock is freed or granted to the next holder (Line 11 in Fig. 7). Furthermore, if the agent pool fails to process packets due to network exceptions (see §4.2), it sends such packets back to the decider. For incarnation checking, the agent pool includes the expected incarnation in the `GRANT` and `FREE` packets and restores the agent if the decider refuses these packets.

5.2 On-Switch Lock Decider

Data structures. The metadata of all locks (*metas*) is stored in register arrays of MAUs (RA in Fig. 12). When lock packets pass through the MAU, predefined actions read and update lock metadata via ALUs attached to the RA. To guarantee line-rate packet processing, each RA can only be accessed once per packet, and the ALU is allowed to perform a few simple arithmetic operations. Therefore, the lock decider functionality and metadata must be split into multiple MAUs, creating a pipeline. Each MAU stores a piece of metadata and performs the corresponding logic. All RAs are indexed by the lock ID (*lid*).

FissLock carefully selects the register size of RA for minimal memory consumption. The allowed register size includes 1 bit and a specific amount of bytes (1, 2, 4, or 8). An intuitive design is to store all of the lock *mode*, machine ID (*mid*), and incarnation (*inca*) into 1-byte RAs, while it wastes 6 bits for each lock mode. Instead, FissLock stores the lock mode with two 1-bit RAs (*free RA* and *r/w RA* in

Table 2: *Packet types used in* FISSLOCK.

Type	Contents
ACQUIRE	lock ID and mode to be acquired, machine ID and task ID of the requester
RELEASE	lock ID to be released, machine ID and task ID of the requester
FREE	lock ID to be freed, INCA, and lock mode before free
GRANT	lock ID to be granted to machine ID and task ID, INCA, and agent of the lock (optional)

Fig. 12), indicating whether the lock is held and whether the lock is shared, respectively. Both *inca* and *mid* use 1-byte RAs (*inca RA* and *mid RA* in Fig. 12). Although 8 bits are still over-sufficient, there are unfortunately no smaller register units in the RA to store them. Packing multiple *mid*s (or *inca*s) in one register is not feasible due to memory accessing restrictions. In summary, FISSLOCK compresses the switch memory consumption of each lock to 18 bits, which is two orders of magnitudes smaller than prior work [76].

Using 1-byte registers for *mid* and *inca*, a single MAU is not enough for the million-scale lock amount. Therefore, FISSLOCK splits them into multiple MAUs, each storing a fixed range of locks. For these MAUs, the lock ID is translated into a MAU *index* and an *offset* within the MAU (*MAU-selection* stage in Fig. 12).

Packet processing pipeline. The decider is realized as a 6-stage pipeline that processes four types of packets in FISSLOCK (see Table 2), where all packets share the same header format but use different header fields. Each stage checks the metadata in the packet or loaded from former stages for selecting proper actions to execute. Actions leverage the ALUs attached to RAs to read, update, and write back the metadata simultaneously. FISSLOCK organizes the MAU order to ensure that all metadata is loaded from RAs before being used by subsequent stages. Only packet types marked in Fig. 12 are processed in corresponding stages. The logic of each stage is described as follows.[1]

MAU-selection stage translates the lock ID into the MAU *index* and the *offset* in the MAU in Fig. 12 for Check stage and MID stage. In Check stage and MID stage, MAUs other than the indexed MAU are skipped.

Check stage checks and updates the incarnation. ACQUIRE packets for a shared lock increment the incarnation by 1. GRANT packets with agent and FREE packets reset the incarnation if the current lock mode is exclusive[2] or the incarnation in the packet is matched, i.e., there are no delayed packets. Otherwise, the packet is marked as invalid and will be skipped by the rest stages except for Destination stage.

Free stage updates the free register in RA and loads its original value to the FREE flag in Fig. 12. ACQUIRE and FREE packets set the free register to 0 (held) and 1 (free) despite its original value, respectively, as 1-bit RAs do not support conditional updates with regard to the original register value.

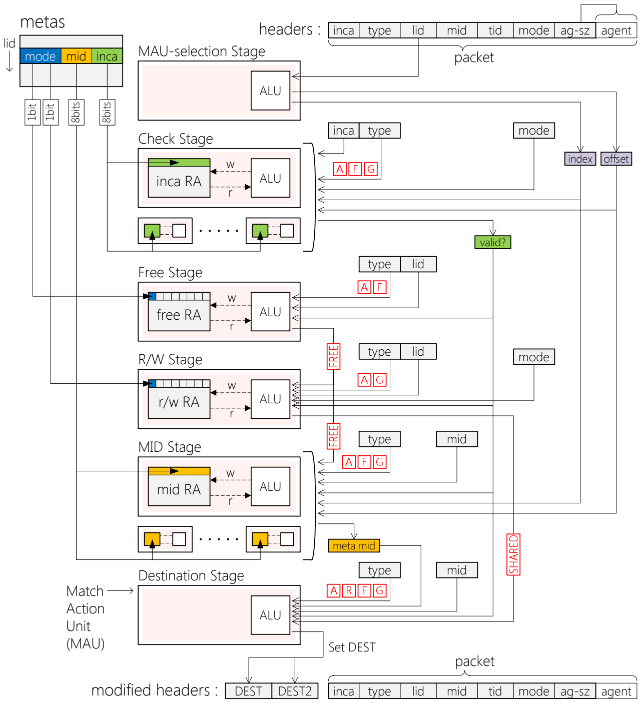

Fig. 12: *The metadata of locks stored in register array of MAUs and packet processing pipeline (data flow) in programmable switch.*

R/W stage updates the r/w register in RA. GRANT packets update the r/w register to the next holder's mode. ACQUIRE packets only update the r/w register to the requester's mode when the lock is free (Line 3 in Fig. 8), i.e., the FREE flag is 1. This stage sets the SHARED flag in Fig. 12 that indicates whether both the *pkt.mode* and the r/w register are SHARED, which determines packet destinations later.

MID stage loads and updates *meta.mid* in mid RA.ACQUIRE and GRANT packets store the *pkt.mid* into mid RA as the new agent's location when the lock is free (Line 3 in Fig. 8) and when carrying the agent (Lines 17–18 in Fig. 8), respectively. FREE packets reset *meta.mid* to 0 (Line 14 in Fig. 8).

Destination stage routes packets to correct egress ports. The destination is controlled by two fields defined by the switch (*DEST* and *DEST2* in Fig. 12), which specify the ports that the packet should be replicated and forwarded to. Both fields can be null for dropping the corresponding packet replica. Packets specify their destination in DEST and the multicasted packet's destination in DEST2. Programmed egress pipelines update the type of multicasted packets to GRANT.

ACQUIRE packets are forwarded to the *pkt.mid* when the lock is free (Line 6 in Fig. 8), *meta.mid* when the request is suspended (Line 11 in Fig. 8), and both *mids* when a shared lock is granted to a shared requester (Lines 10–11 in Fig. 8).

[1] Stages for identifying lock packets and supporting retransmissions are not included for convenience.

[2] The mode field of packet header is used to transfer the current lock mode.

RELEASE packets are always forwarded to *meta.mid* (Line 13 in Fig. 8). GRANT and FREE packets are forwarded to *pkt.mid* (Line 19 in Fig. 8) and dropped (Line 15 in Fig. 8), respectively, unless marked as invalid in Check stage, in which case they are forwarded back to *meta.mid*.

5.3 Failure Handling

Failure model. FISSLOCK can tolerate individual and simultaneous switch and server failures, but it does not guarantee availability (without data replication). Switch and server failures are detected by a reliable external coordinator using heartbeats. On failed servers, the granted locks are considered expired, and pending lock acquire operations are considered aborted. FISSLOCK assumes that applications will handle lock expiration and aborted operations, such as manually aborting ongoing transactions. The availability of FISSLOCK can be achieved through existing methods that replicate the switch data plane to backup switches [39], which are orthogonal to our work.

Failure recovery. Switch and server failures in FISSLOCK may result in the loss of network packets (see Table 2), switch states (*metas*), lock agent states (*agents*), and lock client states (granted and pending lock requests). When a failure is detected by the coordinator, FISSLOCK will restart the failed switch if necessary and perform three steps sequentially to recover the aforementioned states.

- **Server data aggregation (S1)**. All surviving servers pause lock operations and submit lock agent states and lock client states to the coordinator.
- **Server data recovery (S2)**. All surviving servers recover the lost and inconsistent states by referring to the aggregated states from the coordinator.
- **Switch data recovery (S3)**. The switch recovers its states by referring to lock agent states from all surviving servers.

5.4 Scale to Multiple Racks

FISSLOCK can be scaled out by partitioning locks to each ToR switch. Each switch only handles requests for the locks it manages and routes other requests to responsible racks. Each machine has a global *mid*, which is translated by the switch to an egress port for the next-hop switch or the machine. In the current implementation of FISSLOCK, these translation rules are statically predetermined, which disables on-demand scaling. However, on-demand scaling could potentially be achieved by updating these rules through the switch control plane. Although requests for remote-rack locks may experience higher network latency, they still have a stable grant time without any queueing delay. The imbalance of loads among switches is not a significant concern, as switches have orders-of-magnitude higher packet processing speed than servers. Even under heavy loads, servers would reach saturation before a hotspot switch does.

6 Implementation

We implemented FISSLOCK from scratch using roughly 1,200 lines of P4 code and 5,000 lines of C++ code.[3] DPDK is used for packet sending and receiving.

Non-linear lock IDs. The lock ID can be sparse and inefficient for indexing RAs. In this case, FISSLOCK maps lock IDs to linear RA indexes with an RPC daemon. Lock clients cache the mapping and embeds the RA index in packets.

Lock scales. FISSLOCK supports efficient management of over 1 million on-switch locks. In our experiments on TPC-C, we use them to protect billion-scale data by protecting a range of data objects with each lock (see §7.4). To support out-of-range locks that exceed the switch capacity, FISSLOCK adopts ParLock as a fallback, i.e., these locks are handled by on-server LMs, and the switch forwards their requests to the server (see §7.6).

Read/Write preference. FISSLOCK is implemented as read-preferring since it is common. For a write-preferring design, the switch requires an additional 1-bit state *ww* (write-waiter) to indicate the existence of exclusive waiters. When encountering an exclusive waiter, the decider sets *ww* to true, so subsequent shared requests are not granted even if all holders are shared. The server-side write-preferring implementation is identical to server-based lock managers [24, 53].

Policy support. In FISSLOCK, the on-server lock agents are tasked with enabling various lock policies (e.g., fairness), as they determine the next holders when the current holders release the lock. For example, FISSLOCK ensures first-come-first-served fairness by using a FIFO wait queue, which prioritizes waiters that are enqueued earlier.

Starvation prevention. In the read-preferring design, FISSLOCK uses an additional 1-bit state per lock on the switch to prevent readers from starving writers. This state is periodically set by the agent if there exists a writer in the wait queue and is cleared when all current holders release the lock. In the write-preferring design, readers are not starved. When the lock is held by a writer, all incoming lock requests are appended to the agent's FIFO wait queue. This guarantees that writers are not granted ahead of preceding readers.

Deadlocks. FISSLOCK offers a lock aborting mechanism to assist applications resolve deadlocks. Specifically, it sets a local timeout for each lock request and aborts pending lock requests that have not been granted after the timeout.

7 Evaluation

7.1 Experimental Setup

Testbed. The experiments were conducted on a cluster consisting of four machines, each has two 12-core Intel CPUs, 128 GB of RAM, and two ConnectX-5 100 Gbps NICs. All

[3] The source code of FISSLOCK is available at https://github.com/SJTU-IPADS/fisslock.

Table 3: *Workload description.*

Microbenchmark			TATP			TPC-C		
Wkld	**Type**	**Ratio**	**Txn**	**Type**	**Ratio**	**Txn**	**Type**	**Ratio**
UH	R	50%	GS	R	35%	NEW	RW	45%
	W	50%	GD	R	10%	PAY	RW	43%
RM	R	90%	GA	R	35%	DLY	RW	4%
	W	10%	US	W	2%	OS	RO	4%
RO	R	100%	UL	W	14%	SL	RO	4%
	W	0%	IF	W	2%			
			DF	W	2%			

NICs are connected to a Top-of-Rack (ToR) wedge100BF-32x programmable switch, which is equipped with an Intel Tofino ASIC [16]. Each machine hosts two logical nodes, each has one CPU connected with one NIC used by all threads running on it. For each node, we assign 10 cores to application threads for issuing lock requests and receiving grant replies, and 1 core to FissLock's agent pool for maintaining lock metadata. Incoming packets are triaged utilizing DPDK Flow Director and Receive Side Scaling. To further adjust the degree of concurrency, we use coroutines in each application thread to simulate multiple clients.

Comparing targets. We compare FissLock with the state-of-the-art centralized lock manager NetLock [76] and two traditional server-based lock managers, namely SrvLock and ParLock. We re-implemented NetLock following its open-sourced artifact [9], which is not compatible with our programmable switch. The maximum number of locks that NetLock manages on the switch is determined by the scripts in its artifact. Due to the lack of open-source artifacts, we hand-crafted ParLock following the specification of popular commercial systems [6, 13], and used the lock server implementation of NetLock as SrvLock. ParLock uses the same allocation of CPU cores as FissLock, while NetLock and SrvLock use one node as the dedicated lock server and the other seven nodes as clients. For fairness, all systems are assigned the same total amount of CPU cores, i.e., 8 cores as lock managers and 80 cores as lock clients.

Workloads. We use one microbenchmark to evaluate lock granting performance, and two transaction benchmarks to study the impact on accelerating transaction execution (see Table 3). We trace all lock requests during a pre-execution phase for NetLock to profile the workload. To maintain fairness, we evaluate all lock managers using the same lock request traces and report the actual execution performance, like prior work [75, 76].

Microbenchmark. We built a microbenchmark to emulate the typical use of locks in modern distributed in-memory systems, where shared (resp. exclusive) locks are acquired before and released after data reads (resp. updates) to serialize these operations. To study the lock granting performance of lock managers under different read-write ratios and workload skewness, the microbenchmark includes three representative workloads: update-heavy (UH), read-mostly (RM), and read-only (RO). Each workload has both Uniform (Uni.) and Zipfian (Zipf.) lock request distributions.

Transaction benchmarks. TATP [12] and TPC-C [14] are evaluated to study the impact of lock managers on the end-to-end performance of distributed in-memory systems. TATP represents low-locality[4] and read-intensive (80% of transactions are read-only) workload, while TPC-C represents high-locality (~90% of transactions only access local tables [14, 37]) and write-intensive (8% of transactions are read-only) workload. We use the default population size (100,000 subscribers) for TATP and adopt the same per-client warehouse number as prior work [29, 66] for TPC-C. We protect a range of records with each lock to control the total amount of locks in the benchmark. We run 160 clients for TATP and 1,200 clients for TPC-C, all clients issue lock requests synchronously. We adopt the two-phase locking (2PL) protocol when executing transactions. After acquiring all locks required by each transaction, we delay around 2 µs for TATP and 10 µs for TPC-C to simulate the execution time in Table 1. Transactions executed over 10 ms are aborted to avoid deadlocks.

7.2 Memory Consumption

We first study the switch memory usage of FissLock, which limits the maximum number of locks a programmable switch can host. Following the internal structure of programmable switch ASICs (see Fig. 2), two factors collectively impact the limitation: the number of MAUs and the memory capacity of each MAU. Let N be the number of locks in the system, C be the per-MAU memory capacity in bits, then the amount of MAUs required M can be described as:

$$M = \left\lceil \frac{N}{C} \right\rceil \times 2 + \left\lceil \frac{N \times 8}{C} \right\rceil \times 2 + 4$$

where 4 MAUs are occupied by the control and computing logic of the lock decider, and others denote MAUs for lock mode (*mode*), machine ID (*mid*), and incarnation (*inca*). The switch ASIC in our testbed has 12 MAUs at the ingress pipeline, each providing about 560 KB stateful storage (i.e., register array). Assuming that all MAUs are used by FissLock and following an optimal allocation scheme (i.e., 2 MAUs for *mode*, 3 MAUs for *mid*, and 3 MAUs for *inca*), the maximum number of locks that can be hosted is 1.68 million. In contrast, NetLock can only manage a few thousand locks on the same switch ASIC throughout our experiments.

7.3 Lock Granting Performance

We study the lock granting performance of FissLock and baselines through the grant time distribution (Fig. 13) and lock request throughput (Fig. 14 (left)) in the microbenchmark. All experiments use 160 clients and 1 million locks.

[4] Like prior work [29, 37], we do not improve the locality by deliberately partitioning TATP tables.

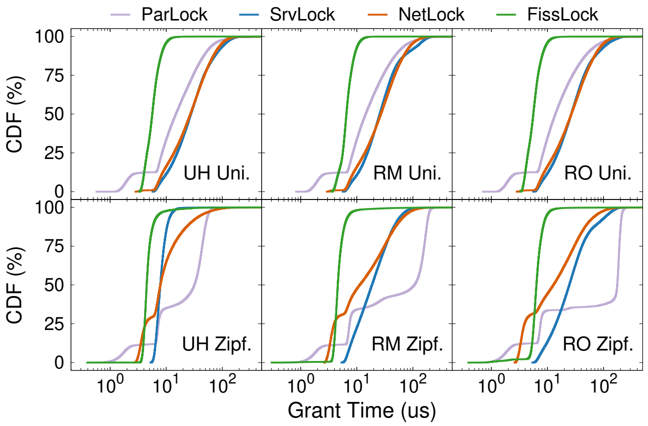

Fig. 13: *The CDF of lock grant time in the microbenchmark when using different lock managers.*

Overall, FissLock achieves low and stable grant time under all workloads. Compared with baselines, it cuts down the median grant time by up to 79.5% (SrvLock), 79.1% (NetLock), and 96.4% (ParLock), and the 90^{th} percentile grant time by up to 89.7% (SrvLock), 88.9% (NetLock), and 96.5% (ParLock). Consequently, FissLock outperforms baselines on lock request throughput by up to 4.08× (ParLock), 4.79× (SrvLock), and 4.99× (NetLock). The performance gain of FissLock mainly comes from three aspects: (1) the elimination of queueing delay by switch-based grant deciding, (2) the pervasiveness of acceleration by lock fission, and (3) the automatic load balancing by dynamic agent migration.

Uniform workloads. FissLock mainly benefits from (1) and (2) under Uniform workloads. The queueing delay dominates the grant time of all three baseline systems, which grows to up to 84.5 µs (SrvLock), 76.2 µs (NetLock), and 54.8 µs (ParLock) at 90^{th} percentile. Oppositely, FissLock controls the 90^{th} percentile grant time of all workloads under 9.42µs by eliminating queueing delay. NetLock falls back to SrvLock under Uniform workloads because only ∼1% of requests are handled by the switch, while FissLock accelerates all requests. ParLock handles around 12.5% of requests locally, which alleviates the remote LM's load and helps it slightly outperform SrvLock and NetLock.

Zipfian workloads. FissLock benefits from all three aspects under Zipfian workloads. Even with workload profiling, NetLock accelerates merely ∼27% of requests under Zipfian workloads. Although these requests have slightly (0–2µs) lower grant time than FissLock because FissLock devotes additional time to skim through the agent pool, the other 73% of requests are still handled by the server and have similar performance to SrvLock. Oppositely, FissLock makes grant decisions on the switch for all lock requests. ParLock suffers from severe load imbalance under Zipfian workloads, which significantly increases its grant time. Meanwhile, FissLock balances the load among servers and minimizes the impact of workload skewness, achieving up to 96.4% lower grant time than ParLock. The R/W lock contention becomes the main restriction of FissLock's throughput under Zipfian workloads and significantly reduces the lead of FissLock.

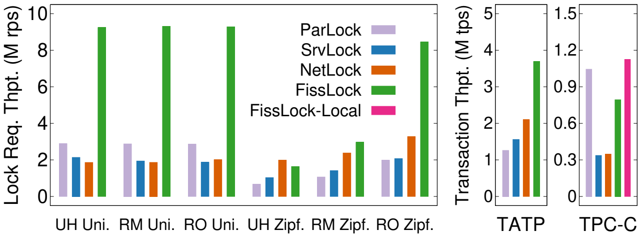

Fig. 14: *The lock request throughput in the microbenchmark, and the transaction throughput on TATP and TPC-C with different lock managers.*

7.4 Distributed Transaction Performance

We study the impact of lock managers on the end-to-end latency (Fig. 15) and throughput (Fig. 14 (right)) of transaction execution with TATP and TPC-C benchmarks. We show the latency distribution of each type of transaction individually.

TATP. TATP is a read-dominated workload containing 7 short transactions. Overall, FissLock outperforms baseline systems by 2.93× (ParLock), 2.37× (SrvLock), and 1.76× (NetLock) on transaction throughput. Since most transactions acquire only one or two locks, there were no deadlocks throughout the test. When executing transactions that only acquire one lock, FissLock exhibits similar performance to the microbenchmark case, i.e., the latency remains low for 99% of GS (17.5 µs), GA (15.4 µs), and UL (27.7 µs) transactions. For multi-lock transactions (GD, US, IF, and DF), the latency of FissLock is proportional to the average amount of locks acquired when executing the transaction. ParLock suffers from severe load imbalance when executing GS, US, UL, IF, and DF transactions, which drags down its throughput and results in lower queueing delay in GD and GA. Moreover, the effect of ParLock's local fast-path is less apparent in multi-lock transactions as the possibility of local grants is powered. NetLock has similar performance to SrvLock because the switch LM has limited acceleration proportion (17% in total), even if all rows are selected in a non-uniform manner, because the on-switch lock proportion is too restricted (0.09%). Queueing delay dominates the transaction latency of both systems.

TPC-C. TPC-C is a write-dominated workload containing 5 types of complicated transactions, where 90% of transactions solely acquire local locks. Given this workload locality information, ParLock can serve most requests with local lock servers, achieving superior performance. To show that FissLock can also benefit from this workload-aware optimization, we additionally evaluate FissLock-Local, which follows ParLock's method of co-locating locks with the clients acquiring them. Among lock managers that do not exploit

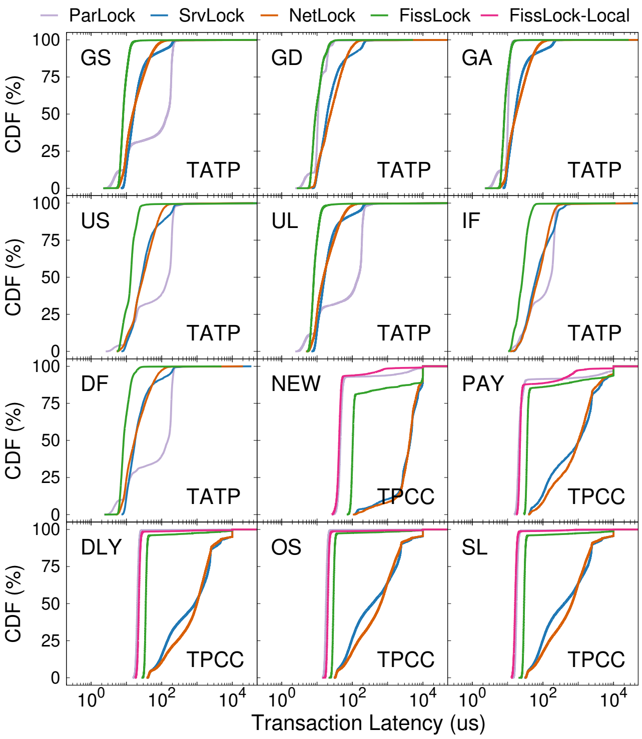

Fig. 15: *The CDF of transaction latency on TATP and TPC-C workloads when using different LMs.*

workload locality, FissLock outperforms SrvLock and NetLock by 2.36× and 2.28× on transaction throughput, respectively. Both FissLock-Local and ParLock handle over 90% of requests locally, where they have similar performance. However, FissLock-Local still has 1.08× higher throughput than ParLock because of better remote lock requesting performance. We analyze the latency of write-intensive and read-only transactions separately as follows.

Write-intensive transactions (NEW and PAY). NEW and PAY acquire 14 and 4 locks on average. Even when acquiring almost all locks remotely, FissLock still achieves fairly low and stable latency for 81% of NEW (< 125.5 µs) and 85% of PAY (< 44.5 µs) transactions. The latency of other transactions is dominated by the wait time due to lock contention. NetLock falls back to SrvLock because only 6% of requests to 0.3% of locks are handled by the switch, due to sparse data accesses in large datasets [29, 66]. Both LMs have over an order-of-magnitude higher transaction latency than FissLock because of queueing delay. ParLock and FissLock-Local have 40%–60% lower transaction latency than FissLock due to local lock request handling. FissLock-Local exhibits a shorter tail than ParLock because of higher remote lock acquire performance.

Read-only transactions (DLY, OS, and SL). All read-only transactions are local, which explains the identical performance of ParLock and FissLock-Local. The latency turning point of FissLock appears later (96.5%, 97.7%, and 96.2% for DLY, OS, and SL) because of lower wait time when acquiring shared locks. The performance of NetLock and SrvLock is similar to the write-intensive case as the queueing delay instead of wait time dominates the latency.

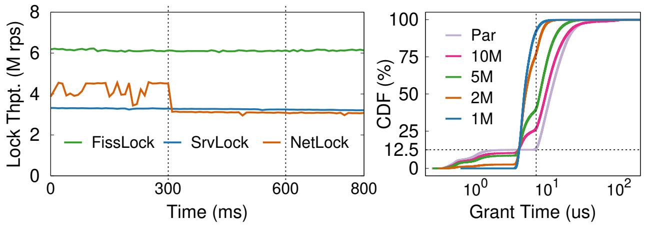

Fig. 16: *The timeline of lock request throughput on a dynamic workload (left), and the CDF of lock grant time with various lock scales when using a uniform RM workload (right).*

7.5 Dynamic Workload

To study the robustness of lock managers under dynamic workloads, we alter the hotspot of 1 million locks every 300 ms (i.e., 2,500 hot locks). In this workload, half of the requests target the hotspot, while the other half are evenly distributed to the remaining locks. As shown in Fig. 16 (left), FissLock achieves a consistently high throughput of over 6 million requests per second (M rps) regardless of hotspot changes, thanks to its pervasive acceleration. In contrast, NetLock's throughput fluctuates between 3.44 M and 4.58 M rps, as the switch only handles half of the requests. When the hotspot changes at 300 ms, the throughput instantly drops to around 3 M rps, close to SrvLock (its fallback), and remains low until the hotspot returns.[5]

7.6 Lock Scales

We further evaluate the lock granting performance of FissLock as the number of locks increases, using a uniform RM workload. We also report the result of ParLock, the fallback approach of FissLock, with 10M locks as a reference. As shown in Fig. 16 (right), as expected, the performance of FissLock gradually approaches that of ParLock as the number of locks increases. We found that FissLock still achieves 39.9% lower 20th percentile and 9.9% lower 90th percentile grant time compared to ParLock for 10M locks by shipping the load of around 10% requests to the switch and thereby relieving server CPUs. At 0–12.5 percentiles, workloads with fewer locks perform worse because almost all requests for on-switch locks are handled by the switch, while around 12.5% of requests for out-of-range locks are handled locally.

7.7 Lock Granularity

To justify the necessity of using fine-grained locks for large-scale datasets, we conduct an experiment that varies the number of locks used to protect accesses to 10 million objects. These objects are evenly distributed among the locks. As shown in Fig. 17 (left), when using coarse-grained locks,

[5] Due to space limitations, we omit this part in Fig. 16 (left).

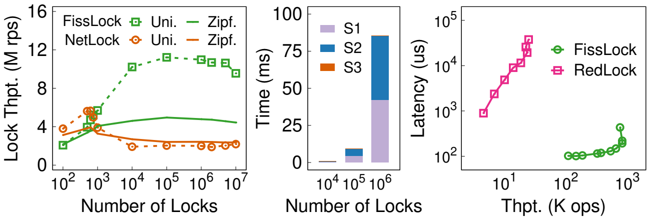

Fig. 17: *The throughput of* FISSLOCK *and NetLock using different number of locks for a 10-million dataset (left), the recovery time breakdown for* FISSLOCK *(mid), and the performance of mobile banking application using* FISSLOCK *and RedLock (right).*

the lock granting throughput of FISSLOCK is significantly dragged down (up to 5.40×) by lock contention in a uniform RM workload. NetLock achieves the peak throughput when using around 1,000 locks due to limited switch memory. However, the peak throughput is only 5.65 M rps due to severe lock contention. In contrast, FISSLOCK unleashes full-scale acceleration with over one million fine-grained locks, thereby resulting in 1.99× (Uniform) and 1.26× (Zipfian) higher peak throughput than NetLock.

7.8 Failure Recovery

To study the performance of failure recovery, we manually inject a simultaneous failure of one switch and one server into the experiment used in §7.7. As shown in Fig. 17 (mid), the recovery time mainly comes from aggregating states of surviving servers (S1) and repairing them (S2), and is proportional to the number of locks due to scanning the metadata (e.g., granted requests) of all locks. The recovery time for switch states (S3) is trivial, as it only involves held locks.

7.9 Application: Mobile Banking

We build a mobile banking application that supports common banking operations like balance checking (BC) and funds transferring (FT) [7, 8]. The application server uses FISSLOCK following the 2PL protocol for transactions and uses Redis-backed in-memory store [11]. It combines Redis asynchronous APIs and coroutines to hide network latency and maximize throughput when serving massive clients. We use Redis's official implementation of distributed locks (RedLock [5]) as the baseline and use a mixed workload containing 90% BC and 10% FT operations, which reflects the user behavior that checks balance much more frequent than transferring funds [11]. We initialize the bank with 1 million accounts. By adjusting the number of clients that issue operations, we compare accumulative throughput of all clients and median latency of the application using FISSLOCK and RedLock. As shown in Fig. 17 (right), the operation throughput when using RedLock peaks at 24.8 K ops due to lock contention. Conversely, when using FISSLOCK, the operation throughput scales to 825.4 K ops because FISSLOCK grants and transfers locks faster. Additionally, FISSLOCK cuts down the median latency of banking operations by at least one order of magnitude.

8 Related Work

Distributed lock management. There have been many efforts to investigate distributed lock management which are classified into two categories, centralized LM [6, 13, 21, 30, 54, 55, 65, 76] and decentralized LM [28, 56, 73, 75]. Centralized LMs are widely used because they enable rich properties such as latency predictability [31, 38, 46], starvation freedom [36], and performance isolation [76]. Decentralized LMs leverage one-sided RDMA primitives to bypass the CPU bottleneck of lock managers [28, 56, 73, 75], which offers better performance but loses support to the properties above. Prior work [76] uses programmable switches to host part of locks, achieving desired performance without sacrificing centralized properties. However, it assumes that the workload is highly skewed and predictable. Differently, FISSLOCK can accelerate million-scale locks for diverse workloads without prior knowledge.

In-network optimization. The emergence of programmable switches [3, 16, 33] inspires numerous in-network designs for distributed systems, including distributed cache [35, 42, 47, 49, 50], consensus and concurrency control [26, 27, 34, 43, 44, 59, 76], machine learning [17, 40, 48, 63, 64, 77], task scheduling [69, 74], and distributed data coherence [41, 45, 70]. These systems primarily leverage the stronger packet processing power and shorter network round trip of switches to achieve higher performance for a portion of workloads. NetLock [76], the system most relevant to FISSLOCK, implements a full lock manager on the switch to handle requests on hot locks. However, due to limited switch memory, the on-switch lock manager can only optimize thousands of locks. In contrast, FISSLOCK achieves consistent performance improvement for millions of locks via lock fission.

9 Conclusion

This paper presents FISSLOCK, a switch-centric lock service that enables lock fission scheme to provide microsecond-scale lock grant time for millions of locks. The concept of lock fission—decoupling the locking process to align with the characteristics of heterogeneous hardware—could be applied to other contexts. For instance, the locking process can be decoupled differently in various heterogeneous environments, such as disaggregated memory. We leave the investigation of these to future work.

10 Acknowledgement

We sincerely thank the OSDI reviewers and our shepherd for their insightful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (No. 62272291, 61925206, 6213201), the HighTech Support Program from STCSM (No. 22511106200), as well as research grants from Huawei Technologies and Shanghai Artificial Intelligence Laboratory. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

[1] Amazon ElastiCache for Redis. https://aws.amazon.com/elasticache/redis/.

[2] Bio2RDF: Linked Data for the Life Science. http://bio2rdf.org/.

[3] Cisco nexus 34180yc and 3464c programmable switches data sheet. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/datasheet-c78-740836.html.

[4] DBpedia's SPARQL Benchmark. http://aksw.org/Projects/DBPSB.

[5] Distributed Locks with Redis. https://redis.io/docs/manual/patterns/distributed-locks/.

[6] Druid | interactive analytics at scale. https://druid.apache.org/.

[7] Mifos-Mobile Android Application for MifosX. https://github.com/openMF/mifos-mobile.

[8] Mobile Banking App - Flutter. https://github.com/sangvaleap/app-flutter-mobile-banking.

[9] NetLock: Fast, Centralized Lock Management Using Programmable Switches. https://github.com/netx-repo/NetLock/.

[10] Oltpbench. https://github.com/oltpbenchmark/oltpbench/.

[11] Redis Enterprise for Mobile Banking. https://redis.com/solutions/use-cases/redis-enterprise-for-mobile-banking/.

[12] Tatp. https://tatpbenchmark.sourceforge.net/.

[13] Teradata: Business analytics, hybrid cloud & consulting. http://www.teradata.com/.

[14] Tpc-c. http://www.tpc.org/tpcc/.

[15] YAGO: A High-Quality Knowledge Base. http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago.

[16] A. Agrawal and C. Kim. Intel tofino2 – a 12.9tbps p4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–32, Los Alamitos, CA, USA, aug 2020. IEEE Computer Society.

[17] Daniel Amir, Tegan Wilson, Vishal Shrivastav, Hakim Weatherspoon, and Robert Kleinberg. Poster: Scalability and congestion control in oblivious reconfigurable networks. In Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz, editors, *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 1138–1140. ACM, 2023.

[18] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, 2020.

[19] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44(3):87–95, 2014.

[20] Andrew Brook. Evolution and practice: Low-latency distributed applications in finance: The finance industry has unique demands for low-latency distributed systems. *Queue*, 13(4):40–53, apr 2015.

[21] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 335–350, USA, 2006. USENIX Association.

[22] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2477–2489, New York, NY, USA, 2021. Association for Computing Machinery.

[23] 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. Spanner: Google's Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[24] Pierre-Jacques Courtois, F. Heymans, and David Lorge Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14:667–668, 1971.

[25] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 727–743, USA, 2018. USENIX Association.

[26] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738, 2020.

[27] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[28] A. Devulapalli and P. Wyckoff. Distributed queue-based locking using advanced network features. In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 408–415, 2005.

[29] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[30] A.B. Hastings. Distributed lock management in a transaction processing environment. In *Proceedings Ninth Symposium on Reliable Distributed Systems*, pages 22–31, 1990.

[31] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas F. Wenisch. A top-down approach to achieving performance predictability in database systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 745–758, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX-ATC'10, page 11, USA, 2010. USENIX Association.

[33] Broadcom Inc. Trident3-x7 / bcm56870 series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series.

[34] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 35–49, USA, 2018. USENIX Association.

[35] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 121–136. ACM, 2017.

[36] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 295–308, USA, 2008. USENIX Association.

[37] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[38] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[39] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: enabling fault-tolerant stateful in-switch applications. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 223–244. ACM, 2021.

[40] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M. Swift. ATP: in-network aggregation for multi-tenant learning. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 741–761. USENIX Association, 2021.

[41] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: in-network memory management for disaggregated data centers. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 488–504. ACM, 2021.

[42] Jason Lei and Vishal Shrivastav. Seer: Enabling future-aware online caching in networked systems. In Laurent Vanbever and Irene Zhang, editors, *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, pages 635–649. USENIX Association, 2024.

[43] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 104–120, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 467–483. USENIX Association, 2016.

[45] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 387–406. USENIX Association, 2020.

[46] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[47] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, Santa Clara, CA, March 2016. USENIX Association.

[48] Zhaoyi Li, Jiawei Huang, Yijun Li, Aikun Xu, Shengwen Zhou, Jingling Liu, and Jianxin Wang. A2TP: aggregator-aware in-network aggregation for multi-tenant learning. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, pages 639–653. ACM, 2023.

[49] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 795–809. ACM, 2017.

[50] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, Boston, MA, February 2019. USENIX Association.

[51] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, 2017. USENIX Association.

[52] Yunus Ma, Siphrey Xie, Henry Zhong, Leon Lee, and King Lv. Hiengine: How to architect a cloud-native memory-optimized database engine. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2177–2190, New York, NY, USA, 2022. Association for Computing Machinery.

[53] John Mellor-Crummey and Michael Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. volume 26, pages 106–113, 07 1991.

[54] C. Mohan and Inderpal Narang. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, page 193–207, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[55] Mohindra and Devarakonda. Distributed token management in calypso file system. In *Proceedings of the 1994 6th IEEE Symposium on Parallel and Distributed Processing*, SPDP '94, page 290–297, USA, 1994. IEEE Computer Society.

[56] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High performance distributed lock management services using network-based remote atomic operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 583–590, 2007.

[57] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, Santa Clara, CA, February 2017. USENIX Association.

[58] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.

[59] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, May 2015. USENIX Association.

[60] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 401–417, USA, 2016. USENIX Association.

[61] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248, 2014.

[62] Kun Ren, Alexander Thomson, and Daniel J. Abadi. Vll: A lock manager redesign for main memory database systems. *The VLDB Journal*, 24(5):681–705, oct 2015.

[63] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 150–156, New York, NY, USA, 2017. Association for Computing Machinery.

[64] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[65] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, page 19–es, USA, 2002. USENIX Association.

[66] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 433–448, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. Contention-aware lock scheduling for transactional databases. *Proc. VLDB Endow.*, 11(5):648–662, jan 2018.

[68] R. Kent Treiber. *Systems Programming: Coping with Parallelism*. Research Report RJ 5118 (53162). International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[69] Sreeharsha Udayashankar, Ashraf Abdel-Hadi, Ali Mashtizadeh, and Samer Al-Kiswany. Draconis: Network-accelerated scheduling for microsecond-scale workloads. pages 333–348, 04 2024.

[70] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with in-network cache coherence. In Marcos K. Aguilera and Gala Yadgar, editors, *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 277–292. USENIX Association, 2021.

[71] Xingda Wei, Rong Chen, and Haibo Chen. Fast rdma-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[72] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. KRCORE: A microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 121–136, Carlsbad, CA, July 2022. USENIX Association.

[73] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 87–104, New York, NY, USA, 2015. Association for Computing Machinery.

[74] Parham Yassini, Khaled M. Diab, Saeed Mahloujifar, and Mohamed Hefeeda. Horus: Granular in-network task scheduler for cloud datacenters. In Laurent Vanbever and Irene Zhang, editors, *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, pages 1–22. USENIX Association, 2024.

[75] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed lock management with rdma: Decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1571–1586, New York, NY, USA, 2018. Association for Computing Machinery.

[76] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 126–138. ACM, 2020.

[77] Changgang Zheng, Benjamin Rienecker, and Noa Zilberman. QCMP: load balancing via in-network reinforcement learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing, FIRA@SIGCOMM 2023, New York, NY, USA, 10 September 2023*, pages 35–40. ACM, 2023.

[78] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 15–29. USENIX Association, July 2021.