SwitchTx: Scalable In-Network Coordination for Distributed Transaction Processing

Junru Li†, Youyou Lu‡*, Yiming Zhang♣, Qing Wang†, Zhuo Cheng♠, Keji Huang♠, Jiwu Shu‡
†{lijr19, q-wang18}@mails.tsinghua.edu.cn, ‡{luyouyou, shujw}@tsinghua.edu.cn
♣zhangyiming@xmu.edu.cn, ♠{chengzhuo, huangkeji}@huawei.com
†‡Tsinghua University, ♣NICEX Lab, Xiamen University, ♠Huawei Storage Product Line

ABSTRACT

Online-transaction-processing (OLTP) applications require the underlying storage system to guarantee consistency and serializability for distributed transactions involving large numbers of servers, which tends to introduce high coordination cost and cause low system performance. In-network coordination is a promising approach to alleviate this problem, which leverages programmable switches to move a piece of coordination functionality into the network. This paper presents a fast and scalable transaction processing system called SwitchTx. At the core of SwitchTx is a decentralized multi-switch in-network coordination mechanism, which leverages modern switches' programmability to reduce coordination cost while avoiding the central-switch-caused problems in the state-of-the-art Eris transaction processing system. SwitchTx abstracts various coordination tasks (e.g., locking, validating, and replicating) as in-switch gather-and-scatter (GaS) operations, and offloads coordination to a tree of switches for each transaction (instead of to a central switch for all transactions) where the client and the participants connect to the leaves. Moreover, to control the transaction traffic intelligently, SwitchTx reorders the coordination messages according to their semantics and redesigns the congestion control combined with admission control. Evaluation shows that SwitchTx outperforms current transaction processing systems in various workloads by up to 2.16× in throughput, 40.4% in latency, and 41.5% in lock time.

PVLDB Reference Format:
Junru Li, Youyou Lu, Yiming Zhang, Qing Wang, Zhuo Cheng, Keji Huang, Jiwu Shu. SwitchTx: Scalable In-Network Coordination for Distributed Transaction Processing. PVLDB, 15(11): 2881 - 2894, 2022.
doi:10.14778/3551793.3551838

1 INTRODUCTION

Transactions with consistency and serializability [1] provide a simple but powerful *abstraction* for programming and reasoning about distributed storage systems, where a single server never fails and always executes one transaction at a time in an order consistent with the real distributed execution. Fast and scalable in-memory transaction processing is the basis for many online-transaction-processing (OLTP) applications like web service, stock exchange, and e-commerce. A common way to support this large-scale transaction processing is partitioning data into shards spreading over servers with concurrency control [1–6]. Data partitioning necessitates distributed transaction processing, which tends to cause high coordination cost including network communication, locking/unlocking, data replication as well as aborts and retries.

There have been numerous studies for alleviating coordination cost in distributed transaction processing, e.g., by designing new concurrency control and replication protocols [7–12], optimizing for specific (independent) transactions [13–16], partitioning data more efficiently to reduce contention [17–21], and leveraging fast networks that bypass OS kernel [22–27]. However, these proposals essentially require heavy involvement of CPU cores in coordination and thus are inefficient in transaction processing.

Recent advances in programmable network hardware [28–31] provide new opportunities for *in-network coordination* by moving the coordination functionality into the network. Eris [14], a state-of-the-art transaction system, uses a *central* switch or middlebox to generate multiple sequence numbers for each independent transaction to reduce the coordination cost. Although effectively improving transaction performance in a small scale, the centralized sequencing mechanism (i) bounds the overall system throughput to the capability of a single switch, (ii) substantially increases the processing latency for the scenario where the single switch does not locate on the path from clients to servers, and (iii) limits the transaction types due to switch's hardware constraints. Further, the network stack only offers general-purpose traffic control that does not consider transaction semantics, thus resulting in requirement mismatches (i.e., packet processing order in the network and transaction processing order in database). and function redundancies (i.e., congestion control in the network and admission control in the database)

In this paper, we present an in-memory transaction processing system, SwitchTx. At the core of SwitchTx is a novel scalable in-network coordination mechanism. It leverages switches' programmability to reduce coordination (including concurrency control and replication) cost while avoiding the central-switch-caused problems. It also intelligently controls the network traffic (i.e., message processing order and flying message count) based on transaction semantics.

First, SwitchTx abstracts various coordination tasks as in-switch gather-and-scatter (GaS) operations, where switches *gather* the messages of a transaction phase, perform state transition of the state machine while meeting conditions, *scatter* messages to finish the current phase, and recycle the state machine for the next phase (or another transaction). In-switch GaS not only reduces the the communication length by half but also eliminates processing and queuing

*Youyou Lu is the corresponding author (luyouyou@tsinghua.edu.cn).

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551838

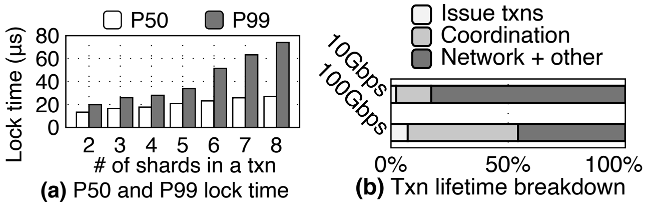

Figure 1: Coordination cost.

overhead in software. Second, different from Eris which relies on a central switch for sequencing all transactions, for each transaction SwitchTx offloads the coordination task to a tree of switches where the transaction's client and participants connect to the leaves. SwitchTx reduces round-trips of transaction processing by exploiting the locality of messages and has no constraint on the transaction types. Third, SwitchTx controls the network traffic (i.e., throughput pressure and processing order) intelligently. To control the processing order, SwitchTx leverages the processing queues in the network stack to reorder the concurrent messages from different transactions according to their semantics; To control the flying message count in the network, SwitchTx redesigns the admission control combined with the network congestion control.

To the best of our knowledge, we are the first to propose a scalable multi-switch in-network coordination mechanism for distributed transaction processing, which offloads *all* coordination functionality to multiple programmable switches and couples network traffic control with distributed transaction semantics. We have implemented a prototype of SwitchTx using Barefoot Tofino switches. SwitchTx supports optimistic concurrency control (OCC) and primary-backup replication. Evaluation with various benchmarks shows that SwitchTx outperforms current transaction processing systems by up to 2.16× in throughput, 40.4% in latency, and 41.5% in lock time.

2 BACKGROUND AND MOTIVATION

2.1 Distributed Transaction Processing

Large-scale transaction processing systems partition data into shards spreading over servers. This subsection briefly reviews data partitioning and coordination for distributed transactions.

Data partitioning. Each server manages an exclusive shard of the entire data set, and cores in a server manage the data in two different ways. (i) Each core is treated as a logical server; data of a server is further partitioned to cores (i.e., one-shard-per-core approach) [7, 14]. (ii) Cores share data of the server (i.e., one-shard-per-server approach), using lock or version synchronization to control concurrent data accesses [23, 32]. SwitchTx focuses on the one-shard-per-server approach and accelerates coordination among servers.

Coordination for distributed transactions. In a large-scale distributed transaction system, concurrency control (such as two-phase locking and optimistic concurrency control [4]) and replication protocols, usually induce high network coordination cost. Two-phase locking (2PL) uses locks and is suitable for high-contention workloads, but suffers from the deadlock problem [1]. In contrast, optimistic concurrency control (OCC) first executes the operations in the execute phase and then handles conflicts in the commit phase, which is more efficient for low-contention workloads. OCC is widely adopted in modern distributed transaction systems (including SwitchTx) because of its simplicity [23, 24, 33]. But OCC needs more network coordination, therefore, SwitchTx uses programmable switches to solve this problem.

In a system that uses OCC for concurrency control and primary-backup replication for availability, transactions are processed through five phases, namely, the *execute* phase, *lock* phase, *validate* phase, *commit backup* phase, and *commit primary* phase. For each transaction, (i) the client reads records without acquiring locks and buffers data in the write set into a private workspace in the execute phase; (ii) OCC detects write-write conflicts and read-write conflicts in the lock phase and validate phase, respectively. OCC acquires write locks during the lock phase, and in the validate phase, OCC guarantees that the read data is not changed since the execute phase; (iii) if there are no conflicts then the transaction enters the commit (backup/primary) phase, in which the transaction installs data atomically in the backup servers and primary servers.

2.2 Coordination Cost

Distributed transaction processing has massive cost for network coordination (e.g., multiple round trips), which is a performance killer. To illustrate the performance impact, we use a microbenchmark to evaluate FaSST [23], a state-of-the-art distributed transaction processing system. In this benchmark, we use 8 servers, each running 24 threads; we disable replication, where transactions do not need the commit backup phase; threads are symmetric: each of them both issues new transactions and handles network requests to participate in transactions issued by threads in other servers. Each transaction randomly reads and writes 8 records. By varying the number of servers involved in each transaction from 2 to 8, the throughput degrades from 7.1Mops to 3.1Mops (43.6%); the P99 tail latency increases from 60.9μs to 126.2μs (2.07×). Specifically, the coordination cost mainly includes the following two aspects.

First, coordination for distributed transactions not only induces high processing latency but also lengthens the lock time (i.e., the time between acquiring and releasing a lock) and version validation time (i.e., the time between execute phase and validate phase), leading to a high abort rate. We refer to these times as the *contention span* [20] of a transaction. To understand the impact on contention span from coordination, we evaluate the lock time in the low-contention workload, to exclude the interference from transaction abort. As shown in Figure 1(a), when the number of data shards involved in a transaction grows, the P50 lock time increases by 2.13×, and the P99 lock time increases by 3.97×.

Second, coordination tasks waste precious CPU cycles, even though they are simple and only include distributing and collecting small network messages. Figure 1(b) shows the latency breakdown of the transaction committing procedure; we observe that the software overheads for coordination are 15.0% and 47.1% under 10Gbps and 100Gbps network, respectively. We conclude that, with a faster network, the coordination cost of software designs is relatively heavier and leaves the high-speed network under-exploited.

2.3 Programmable Switches

Figure 2 shows the architecture of programmable switches. The switches provide flexible pipelines where users can design protocols

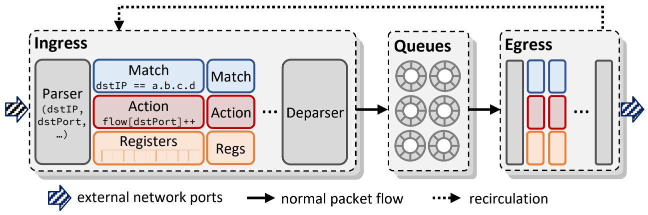

Figure 2: The architecture of programmable switches.

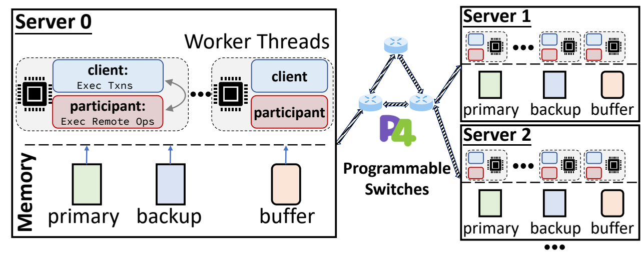

Figure 3: SwitchTx overview.

by programming *parser* and *match-action tables*. Applications use a switch control plane to configure match-action pairs in these tables. Programmable switches also have on-chip memory (*registers* arrays) which can be used to store information.

When a packet arrives at an ingress port, the switch parses the packet header and then applies match-action tables to this packet. If the packet *matches* a key in a *table*, the switch executes the corresponding *action* (e.g., modifying packet header, packet metadata, and register arrays). The packet might be dropped, transmitted to an egress port, or resubmitted to the ingress port. Finally, the egress applies its match-action tables to drop or forward the packet.

2.4 Challenges

The programmable switches provide opportunities to redesign distributed transaction coordination mechanisms. To reduce the coordination cost and exploit the resource of high-speed network, we need to address the following two challenges.

Multi-switch scalability. Eris [14] partitions data per core and introduces a centralized switch or middlebox as a sequencer to generate monotonically increasing IDs (i.e., sequence numbers) for transactions. Each core (i.e., logical server) executes transactions according to their sequence numbers. Unfortunately, the centralized in-switch sequencing mechanism can neither *scale out* to multiple switches nor *scale up* to multiple pipelines in a switch. Eris is not suitable for large-scale (e.g., cross-rack) transaction processing for the following three reasons. First, the centralized switch in Eris is a single-point performance bottleneck, which bounds the overall system throughput to the processing capability of the central switch. Second, all transactions must be routed to the centralized switch for sequencing, which prevents Eris from exploiting locality [23, 24, 34] and thus substantially increases the processing latency in a multi-switch/multi-rack system. Third, the header size of packets in Eris is proportional to the shard count; yet switch hardware supports to parse limited size header (up to 224 8-bit words); this constraint prevents Eris from supporting certain types of transactions such as queries of large ranges and aggregate processing (they both access many shards).

Semantic gap between transactions and network. Network traffic control (e.g., message processing order and flying message count) determines whether the network resources can be fully utilized. The inappropriate *processing order of messages* in the network stack might introduce extra aborts. For example, a lock operation must fail if it is processed before the unlock operation; the message of retrying transaction needs to have higher processing priority to reduce the tail latency. Further, the transaction processing system controls the number of concurrent transactions by *admission control* algorithms (a.k.a Multi-Programming Limit or MPL) to avoid excessive transaction aborts and retries, while the network stack controls the number of concurrent network messages by *congestion control* algorithms to avoid packet loss and retransmission. There are function redundancy and interference between the two control algorithms. Distributed transaction systems need to consider both of them, for example, allowing clients to issue more transactions, when the conflict is rare, can cause unnecessary latency increase due to network congestion.

3 DESIGN

We design SwitchTx with the following four goals.

Reduce coordination cost. Considering that switches are in the routing paths of distributed transaction messages, our first goal is to offload *all* coordination functionality to switches, so as to reduce interaction between servers, kill transaction latency, shorten contention spans, and save CPU cycles.

Avoid single-point bottleneck. Large-scale transaction processing systems may contain thousands of (or even more) servers, and the overall throughput far exceeds the capacity of any single switch. Our second goal is to utilize all switches in the network to parallelize the coordination of disjoint transactions.

Manipulate transaction traffic intelligently. Switches can monitor system status and apply software-defined protocols, providing opportunities to control transaction traffic. Our third goal is to reorder the messages by the transaction semantics and to co-design network congestion control with transaction admission control.

Minimize resource usage in switches. Programmable switches have limited on-chip memory and processing resources. Our fourth goal is to minimize the resource usage of switches and prevent switch memory from being exhausted.

3.1 SwitchTx Overview

SwitchTx is an in-memory transaction processing system which leverages the in-network coordination and transaction traffic control to accelerate distributed transaction processing. Figure 3 shows its end-to-end architecture.

SwitchTx divides data (based on primary keys) into many shards spreading over servers, and servers store data in memory. For high availability, data shards are replicated. Specifically, SwitchTx uses 2-way[1] primary-backup replication (i.e., a primary and a backup).

Each server has several worker threads, and they share the data of the server. The worker threads are in the symmetric model, where each one operates as a *client* and a *participant* at the same time.

[1] Our design is general for system with a higher replication factor.

Specifically, each worker thread is a client: it receives the external transaction requests from applications and then executes transactions (i.e., `reads/writes` data from the local shards and sends requests to `read/write` remote shards). Each worker thread is also a participant: it manages the data and responds to `read/write` requests from clients and coordination requests from switches.

Switches in the cluster have programmability, and they are responsible for the coordination (including concurrency control and replication) between participants. Specifically, SwitchTx uses optimistic concurrency control (OCC) protocol and 2-way primary-backup replication. To guarantee serializability, transactions read data and acquire lock only from the primary replicas of data shards. SwitchTx needs four synchronous phases (i.e., lock, validate, commit backup, and commit primary) to commit a cross-shard transaction.

In-network coordination. We observe that the coordination tasks are to synchronize the results from participants in the current phase and make the transaction enter the next phase. SwitchTx abstracts the coordination tasks as in-switch *gather-and-scatter* (GaS) operations. The switches gather the replies of results from participants in the current phase, perform state transition, and scatter messages to finish the current phase under certain conditions (i.e., phase failure or phase success). GaS makes transactions enter their next phases as quickly as possible. In SwitchTx, with the in-network coordination, the client is only involved in the execution phase, and the switches perform the coordination tasks in subsequent four phases.

To make the GaS operation scale out to a large scale (i.e., multiple racks with multiple switches) and further exploit the processing resource of all switches, SwitchTx generates a *tree topology* among switches for each transaction. The client and the participants connect to the leaves; the messages are gathered from the child switches to the root switch and are scattered reversely. We first introduce how SwitchTx realizes in-network coordination using one single switch in §3.2, and then detail how SwitchTx extends the single-switch design to multiple switches for scalability in §3.3.

Transaction traffic control. Further, we observe that there is a semantic gap between transaction semantic and general network protocol. To manipulate transaction traffic intelligently, we introduce new transaction traffic control algorithms in §3.4. SwitchTx reorders messages in a batch-based and priority-based manner, in the servers and switches respectively; SwitchTx monitors performance metrics and packet loss rate and applies dynamic transaction admission control according to them.

3.2 In-Switch Gather-and-Scatter

3.2.1 Gather-and-Scatter. We observe that once *all* participants complete the current phase successfully, the transaction enters the next phase, and as long as *any* participant completes with failures, the transaction aborts. To employ the switch as a coordinator, SwitchTx abstracts various coordination tasks as in-switch gather-and-scatter operations. From the perspective of the switch, any coordination task is to (i) *gather* a certain amount (all or one) of completion replies from one set of participants and (ii) *scatter* the corresponding phase transition messages to another set of participants.

The GaS operation needs the following information: a message counter (**counter**), the number (**threshold**) of participants in the current phase (**gather_group**), and the participants in the next phase (**scatter_group**). Specifically, for an ongoing transaction, `counter` is 0 at the beginning. (i) In the gather step, when receiving a reply message from participants in `gather_group`, the switch increments `counter` by 1; The switch does not route messages (i.e., drops it) if `counter` is less than `threshold`. (ii) In the scatter step, once `counter` is equal to `threshold`, the switch multicasts the message to the participants in `scatter_group` to notify them for the next phase, then resets `counter` to 0.

Table 1: Cases in gather-and-scatter.

message type	threshold*	next phase	scatter_group
lock_ok	#W_P	Validate	R_P
validate_ok	#R_P	Commit backup	W_B
replicate_ok	#W_B	Commit primary	W_P
commit_ok	#W_P	-	unicast to client
fail	1	Unlock	W** & client
version_copy_ok***	#R_P	Validate & Read	R_P

* W/R: write/read; P/W: primary/backup; #: participant count.
** The `scatter_group` in `fail` messages excludes the message sender.
*** `version_copy_ok` is used for read-only transactions.

Table 1 lists all cases of the GaS operations in SwitchTx. For example, the `threshold` in the lock phase for `lock_ok` messages is the number of write participants[2] and the `scatter_group` is the primary replicas of read shards. It means that a transaction enters the validate phase once all locks on write participants are held.

We show two transaction examples using the GaS operations: a committed one and an aborted one.

<u>*Committed transactions.*</u> Figure 4 shows a committed read-modify-write transaction. It reads records from $shard_0$ and $shard_1$, modifies them by the user's logic, and writes records in $shard_1$ and $shard_2$. Figure 4(a) shows the basic procedure of transaction processing with in-server coordination. In the execution phase, the client reads records from primary replicas P_0 and P_2, and executes the transaction. And then, in the lock phase, it sends requests to P_1 and P_2 to acquire write locks; in the validate phase, it verifies that the versions of records in P_0 and P_2 are not changed. Finally, in the commit backup phase, it writes logs to B_1 and B_2; in the commit primary phase, it writes and unlocks the locked records in P_1 and P_2.

SwitchTx extends the basic procedure to offload coordination to the switch, as shown in Figure 4(b). The client sends the whole write data to the primary and backup replicas at the beginning of the lock phase so that the subsequent coordination phase does not need to involve the client. While in the original OCC, the keys in the write set are combined with messages in the lock phase, and the values are combined with messages in the commit backup/primary phase.

In the lock phase, the switch uses the `counter` to count the number of the `lock_ok` messages from P_1 and P_2, and it compares `counter` with the number of write participants (i.e., `threshold = 2`). It drops the first `lock_ok` message and multicasts the second one (i.e., the last one) as the validate requests to $\langle P_0, P_2\rangle$. In the validate phase, similarly, the switch waits for `validate_ok` messages from P_0 and P_1, and then multicasts the last one to $\langle B_1, B_2\rangle$. After that, in the commit backup phase, it waits for `replicate_ok` messages from B_1 and B_2, and then multicasts the last one to $\langle P_1, P_2\rangle$; in the commit primary phase, the switch waits for `commit_ok` messages

[2] They are primary replicas of write shards.

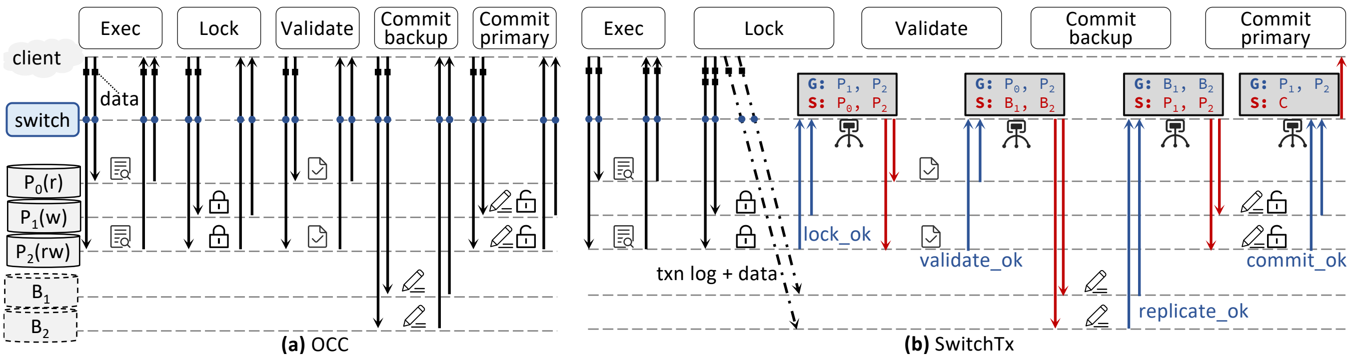

Figure 4: The lifetime of a committed transaction. G/S is gather/scatter operation. P_i/B_i is the primary/backup replica of the data $shard_i$; This example Txn reads data from $shard_{0,2}$ and writes data to $shard_{1,2}$.

from P_1 and P_2 and routes the last one to the client as a sign of transaction committed.

<u>*Aborted transactions.*</u> Figure 5 shows an aborted read-modify-write transaction example. The switch multicasts the first `fail` message (orange one in Figure 5) to write participants and the client. Therefore, write participants can release the locks of the transaction as quickly as possible. After receiving the first fail message, the switch sets an `aborted` flag for the transaction. The switch drops subsequent messages ((?) in Figure 5) of the transaction by checking the `aborted` flag. When clients need to reuse the resources which belong to the aborted transaction for new transactions, clients send an `init` message and reset the `aborted` flag.

Other details. We decouple the data and control messages, where data messages contain keys, values, and versions, and control messages contain the transaction states and the information for GaS operations. SwitchTx only needs to use the fixed-size control messages to coordinate the processing of transactions between participants. The format of control messages is depicted in Figure 6, which occupies a UDP source port to identify the SwitchTx protocol. Clients generate the `txn_id` using the triple ⟨`local_txn_id`, `server_id`, `thread_id`⟩, where `local_txn_id` is increased at the start of transactions. The switch uses the `txn_id` to identify the `counter` stored in switch registers.

To reduce switch resource usage, we minimize the transaction information stored in switches. The `threshold` and `scatter_group` are carried as metadata by the reply messages as shown in Figure 6.

Servers store not only the data shard but also the coordination information for flying transactions. Each transaction logs its coordination information (i.e., `txn_id`, keys in write set, and write shards, current phase) with data to its write shards' replicas. We will detail the design for server/switch failure in §3.5.

3.2.2 Switch Workflow. Figure 7 shows the workflow of the programmable switches in SwitchTx, including the following 6 steps.

At the beginning, Step ❶ handles the network disorder anomalies. Each transaction installs its `txn_id` in the switches to occupy and init its resources for GaS. Because when the new transaction begins, messages belonging to the aborted transaction, which uses the same GaS resource, might be still flying in the network (e.g., message (?) in Figure 5). SwitchTx uses registers to record the `txn_id` of the current transaction and drops the invalid messages of the already

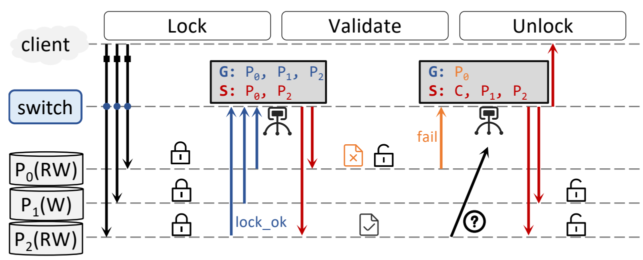

Figure 5: The lifetime of an aborted transaction. This example Txn reads data from $shard_{0,2}$ and writes data to $shard_{0,1,2}$.

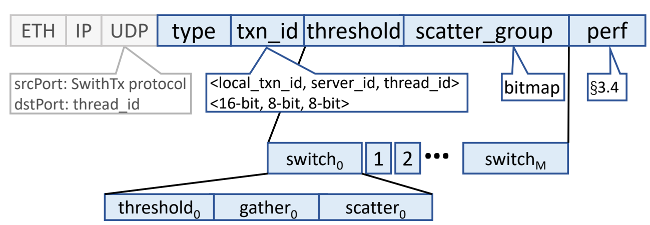

Figure 6: Control message format.

aborted transaction. Then, Steps ❷ and ❸ select the multicast fields in the messages for the multi-switch environment (§3.3). Afterward, Steps ❹ and ❺ maintain the `aborted` register for each transaction to multicast the first `fail` message and drop the subsequent messages. Further, a new transaction needs to send an `init` message at the beginning: it installs the new `txn_id` and clears the `aborted` flag of the last transaction. Finally, Step ❻ increments the `counter` register, compares the `counter` with the `threshold`, and decides the action (drop or multicast) to the message. If the `counter` is equal to the `threshold`, the switch resets the `counter` for the next phase. Furthermore, switches assign each message to queues of different priorities (§3.4.1) according to the message semantics.

Switches use match-action tables to implement the above steps. The processing in match-action tables is sequential. There are no out-of-order issues either between different messages or between match-action tables of the same message.

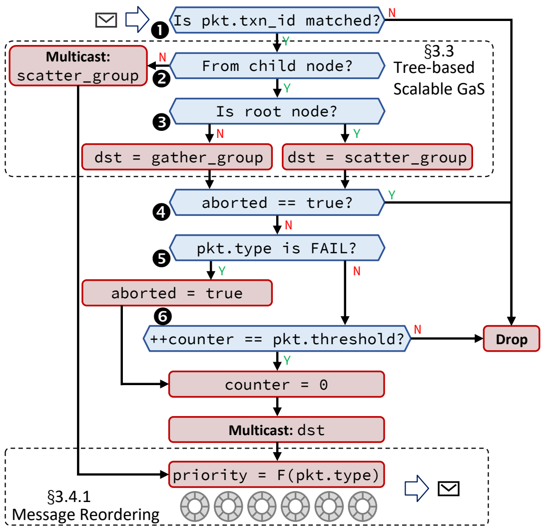

Figure 7: Switch workflow. The `aborted` and `counter` are two registers indexed by the hash value of `pkt.txn_id`.

3.2.3 GaS for Special Transaction Types. SwitchTx also optimizes the read-only and blind-write transactions; SwitchTx supports single-shard transactions without extra coordination overhead.

Read-only and blind-write transactions. Since SwitchTx is based on the single-version protocol and the update is applied in place, read-only transactions need to check the version to guarantee serializability. Read-only transactions need two phases: the version copy phase and validate phase. In the version copy phase, the participants copy the read-set versions to the local buffer, and then send `version_copy_ok` messages to use GaS operation to synchronize that all participants have copied read-set versions. In the validate phase, participants reply `fail` / `read-set data` to the client if the versions are changed/unchanged; the client retries this transaction when receiving a `fail` message. Blind-write transactions are a special case of general transactions without the execute phase and the version check phase.

Single-shard transactions. For the transactions which read/write data only from a single shard, SwitchTx has no coordination overhead. Before sending `lock_ok` messages to the switches in the lock phase, the participant checks whether the validate phase only needs itself; if the participant is the only one in the validate phase, it direct validates the transaction without coordination.

3.3 Tree-based Scalable GaS

The coordination in SwitchTx is decentralized because different transactions can use different switches to perform GaS operations, which is the foundation of scalability. Different from Eris which relies on a central switch for sequencing, SwitchTx is scalable without the single-point bottleneck. Gather-and-scatter focuses on intra-transaction coordination, and thus the states of each transaction in the switch are not shared with other transactions.

To further exploit the benefit of in-network coordination in a large-scale system with multi switches, SwitchTx introduces a tree-based gather-and-scatter design. This design can also support the multiple

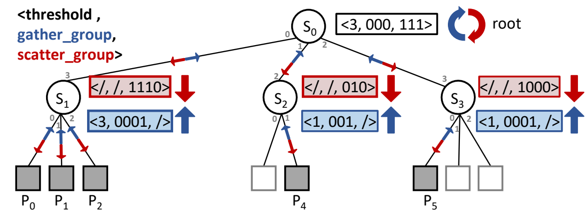

Figure 8: Scalability for multi-switch environment. Task: gathering messages from P_0, P_1, P_2, P_4 and P_5, and then scattering the last one to these primary replicas.

pipelines in a single switch. We reasonably assume that all switches are programmable and SwitchTx knows the network topology.

Before committing transactions the client chooses a group of switches with a tree-topology network as a *coordination tree*, which connects all participants and the client. The switch in the highest layer in the group is the root of the tree. In the gather step, the non-root switches gather the messages from their child nodes (switches or servers) and send the results to their parent nodes. The root switch gathers enough messages and begins to scatter the requests to the participants in the next phase. Then, in the scatter step, the non-root switches multicast the requests from the root switch. For example, as shown in Figure 8, the participants include five servers among all racks; therefore, the coordination tree includes all four switches. On the one hand, once child switches (i.e., S_1, S_2, and S_3) gathers enough messages, they route their last message to the root switch (i.e., S_0). S_0 treats the other three switches as participants of the gather step. On the other hand, S_0 multicasts the result to the three child switches, and then the child switches multicasts the result to the five servers directly.

To implement it, we extend the control message format (Figure 6). Each switch has a part of fields in the control message including `threshold`, `gather_group`, and `scatter_group`. In the gather step, for the child switch, the `gather_group` is the port of its father switch; then it routes the last message to its father switch after gathering enough messages. For the root switch, the `gather_group` is `0`; and it enters the scatter step. In the scatter step, the root switch initiates a multicast along the coordination tree from the root to the servers; each non-root switch multicasts the message according to the corresponding `scatter_group` in the messages.

The coordination tree is decided by the participants in each transaction. Clients maintain the selective trees for different transactions (according to the locations of participants). They issue new transactions by selecting these coordination trees in a round-robin order. The topology in SwitchTx is static, and to support dynamic topology, we need a reconfiguration service (i.e., zookeeper). Specifically, when a switch is added/removed, all servers should reach a consensus on the new topology.

3.4 Transaction Traffic Control

According to these semantics, we introduce a semantic-aware message reordering mechanism and redesign the admission control to control the concurrent transaction count.

3.4.1 Message Reordering in SwitchTx. We observe that making transactions enter (e.g. lock) into the contention span late and exit (e.g., unlock) from it early can reduce the contention span. We achieve this by using priorities for different requests. The principle for assigning priority is that the request that ends the contention span has a higher priority to be processed. In SwitchTx, we classify network messages into three priority levels.

- Highest priority: ❶ messages for lock releasing, i.e., `replicate_ok` and `fail` messages, ❷ messages for validating the data versions, i.e., `lock_ok` and `version_copy_ok` messages, and ❸ all messages of retrying transactions.
- Lowest priority: the messages which are out of contention spans, i.e., read requests in the execute phase and `commit_ok` messages.
- Medium priority: other messages.

SwitchTx leverages both switches and servers to reorder messages.

Priority queues in switches. SwitchTx leverages the existing priority queues in switches. Switches support priority-based scheduling by leveraging multiple queues, each of which manages messages with a specific priority. Queues with higher priorities are serviced before those with lower ones, which ensures that messages with a higher priority are processed and transmitted earlier.

Batch-based reordering in servers. Each thread polls messages in its queue to form a batch and sorts them by their priorities before processing. Since this batching mechanism does not wait for a timeout or count threshold, it does not introduce extra latency.

3.4.2 Admission Control in SwitchTx.

We use `MPL` to represent the maximum number of parallel requests allowed in the whole system. The `MPL` of a thread represents the maximum number of requests a thread can issue at the same time, and the server's `MPL` is the sum of threads' `MPL`. SwitchTx dynamically adjusts the `MPL` of each thread, considering the two aspects: global performance metrics and individual network conditions. The global performance metrics include throughput, tail latency, and abort rate. Further, the workload characteristics (e.g., skewness, transaction types) are time-varying, leading to dynamic performance metrics and network conditions. Therefore, the admission control in SwitchTx aims to combine itself with network congestion control and be adaptive to time-varying workload characteristics.

We designed an epoch-based `MPL` update strategy. We divide time into continuous epochs (e.g., 100ms). At the end of an epoch, each thread collects the global performance metrics and records the packet loss rate for the current epoch. We adjust `MPL` in an additive increase/multiplicative decrease manner. Using throughput as the performance metric in Listing 1, the `MPL` keeps increasing until throughput drops or packets are lost. If there is packet loss, the `MPL` is reset to 1, since slow reduction can not immediately drain the queue to relieve network pressure.

SwitchTx uses control messages to carry performance metrics. The sender takes the local performance metrics and writes them into the control message, and the receiver updates the sender's performance and calculates the latest global performance.

```
/* Init: restart=true, MPL=1, slow_start=MAXMPL/2 */
void admission_control(epoch: i, throughput: T){
  if (packet_loss(i)){
    slow_start = max(MPL / 2, 1);
    restart = true, MPL = 1;              // reset
  }
  /* Slow start, additive increase */
  else if (restart){
    restart = false;
    MPL = (MPL < slow_start) ? MPL * 2 : MPL + 1;
  }
  else if (T_i > αT_{i-1} && !packet_loss(i)) // α=0.9
    MPL = (MPL < slow_start) ? MPL * 2 : MPL + 1;
  /* Multiplicative decrease */
  else
    restart = true, MPL /= 2;

  if (MPL == 0) MPL = 1;
}
```

Listing 1: The logic of admission control in SwitchTx.

Out-of-order packets. When the client receives an abort message, it immediately reclaims the resources on switches and the server's buffer for the new transaction. Switches and servers use the `txn_id` to filter and drop the in-flight (i.e., out-of-order) messages of the aborted transactions.

Packet loss. SwitchTx transmits data messages via RDMA in reliable connection (RC) transport. Therefore, the data messages (especially the lock and write data) are consistent in the participants unless the server fails. The connections for control messages are not reliable, and the switch or server may drop messages due to buffer overflow. Clients use timeout as the signal of packet loss. When a transaction triggers a timeout, the client recycles resources in switches and uses switches to drop all control messages of this transaction, Then, the client sends abort messages to remote participants to abort and roll back the transaction.

Switch failure. Because SwitchTx only stores the coordination information instead of data in switches. When a switch fails, clients just keep aborting and retrying the blocked transactions as in the packet loss case until the restart of the failed switch.

Server failure. Each server acts as both a participant and a client (i.e., coordinator). We discuss participant recovery and coordinator recovery separately.

- Participant recovery. SwitchTx uses primary-backup replication for data. The switches do not store data, so the participant recovery algorithm is similar to prior work [23, 35]. SwitchTx can select a backup participant as the new primary participant.
- Coordinator recovery. Because coordinators' states can be recovered from the transaction states in participants, SwitchTx can directly discard all transaction states belonging to the failed coordinator. Switches first recycle resources used by the failed coordinator and drop the subsequent messages of this coordinator. SwitchTx uses a new server to aggregate the coordinator's transaction states in participants (all primaries and backups); then, SwitchTx determines that a transaction has been committed if and only if its state meets: 1) the state in one of the primary participants is committed, or 2) the states in all backups of all participants are committed. SwitchTx then finishes the committed transactions and rolls back the changes of other transactions.

3.5 Fault Tolerance

The network may have problems such as packet loss and out-of-order issues; servers and switches may fail. We first describe how to guarantee consistent states with out-of-order issues, then describe how to handle packet loss, node failure, and switch failure.

3.6 Discussion

3.6.1 ACID. We discuss the ACID properties [36] in SwitchTx.

- Atomicity: SwitchTx uses the redo logs in backups to guarantee the atomicity of transactions in presence of server failures.
- Consistency: users need to add their constraints in transaction logic, and data remains consistent after executing a transaction.
- Isolation: SwitchTx provides serializable isolation via lock for `write` and version validation for `read`. Serialization point is after the switch gathers all `validate_ok` messages
- Durability: in the event of server failures, SwitchTx provides availability via replication without guaranteeing durability.

3.6.2 Generality. Compared to deterministic OLTP systems [7, 15, 16], SwitchTx focuses on accelerating general transactions, which allow unknown read/write sets, dependencies between write and read, and user aborts. Further, the techniques can improve not only the OCC and 2PC protocols but also other CC or replication protocols:

- The synchronization among multiple machines is common in distributed concurrency control (e.g. Chiller [20]) and commitment protocols (e.g., EasyCommit [37]), and it can be offloaded to switches using the in-switch GaS operation.
- The message reordering and admission control in SwitchTx are general. Message reordering allows database to assign protocol-specific priority to each packet, and admission control can adjust MPL based on different performance metrics.
- The switches in SwitchTx do not store the data. Our techniques do not affect the original recovery algorithm of node failure.

3.6.3 Practicality. We discuss the practicality in data centers:

- Switch topologies: SwitchTx can support environments with a mix of normal switches and programmable switches, and the percentage of programmable switches only affects the performance.
- Memory usage in switches: The memory in a switch is limited to about 15 MB, which needs to store the match-action tables and registers. For match-action tables, message types in SwitchTx and other normal route (e.g., ip and udp) rules are small. For registers, each GaS slot uses 5 bytes (2 bytes for high 16-bit `txn_id` register, 1 byte for `aborted` register, and 2 bytes for `counter` register). Assuming that the fastest transaction latency is $10\mu s$, switches can support $\frac{2^{16}}{10\mu s}$=6.1G *txns/s* parallel transactions, which cost $2^{16} \times 5B$=$0.3125MB$ registers.
- Network isolation in switches: We only add the GaS and reordering function for packets of SwitchTx protocol; function for other network protocols is unmodified. SwitchTx can share switch resources with other applications fairly via multi-tenancy studies on programmable network [38].
- Cost: the current price of Barefoot Tofino switch (programmable) and Mellanox typical switch are $220 and $93.75 per port, respectively. Data centers already deploy programmable switches, such as Alibaba [39] and Facebook [40].

4 IMPLEMENTATION

We use RoCE (RDMA over Converged Ethernet) as the network stack of servers for high-performance communication.

Network routing. We use a general RPC framework [22, 41, 42] to transmit data messages, which is based on RDMA *WRITE_WITH_IMM* (a two-sided verb) with the reliable connection (RC) transport. The switches route the data messages by the IP protocol directly. Our control messages use the RDMA *RAW PACKET* [43] based on the UDP protocol. The switches can identity control messages using a preserved UDP source port. The coordination logic and the mapping from the bitmap (i.e., `gather_group` and `scatter_group`) to the switch physical port are preloaded to the switches.

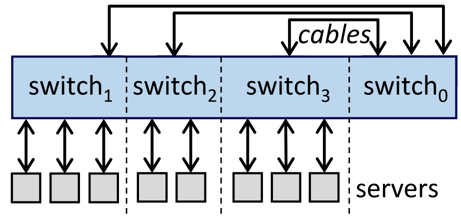

Figure 9: The multi-switch topology.

Packet steering. For simplicity, each server own the same number of threads. For data messages, each thread establishes a QP (queue pair) connection with a thread in other remote servers. For control messages, the UDP destination port is used to indicate the different threads, and each thread in a server is responsible for a fixed port. We use the NIC steering mechanism to dispatch the control messages.

5 EVALUATION

We evaluate SwitchTx under various workloads, seeking to answer the following questions:

- How do the different techniques employed in SwitchTx contribute to overall performance (§5.2)?
- How does the in-switch GaS perform compared to the in-server one? What are the benefits/overheads of admission control (§5.3)?
- How SwitchTx scale with the number of threads under both single-switch and multi-switch environments (§5.4)?
- How do the characteristics of workloads affect SwitchTx (§5.5)?
- How does SwitchTx perform compared to deterministic OLTP systems (§5.6)?

5.1 Experimental Setup

Experimental environment. SwitchTx is based on the symmetric model where each thread acts as both participant and client. We use 8 servers and 24 threads per server unless specified. Each server has two 12-cores Xeon E5-2650 v4 2.20GHz CPU nodes, and is equipped with a 100Gbps Mellanox ConnectX-5 NIC and 128GB memory. Regarding the partition scheme among servers, we use consistent hashing [44]. The storage behind SwitchTx is an in-memory key-value system; its index is cuckoo hash[45]. We use 2-way primary-backup replication for all evaluations. We use a Barefoot Tofino Wedge 100BF-32X switch [46] to simulate four independent virtual switches in Figure 9, where the physical switch connects itself with three cables. The connection between switches must go through the cables and the resources on the physical switch are partitioned into the four virtual switches. The environment of this configuration is similar to the real multi-rack environment. The network topology is the same as Figure 8.

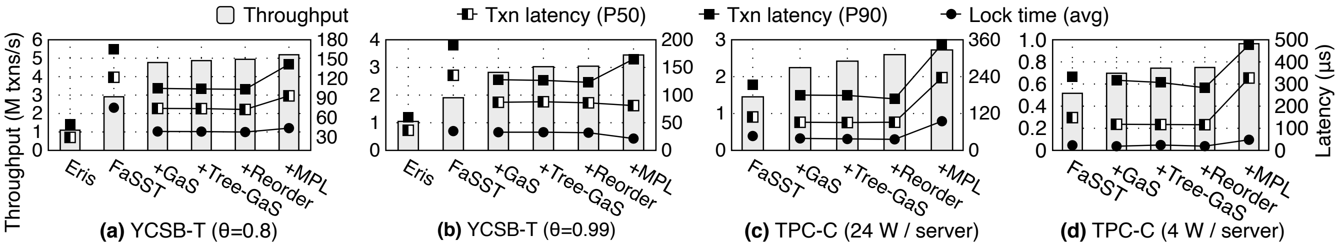

Figure 10: Overall performance. (a) YCSB-T with θ=0.8 and (c) TPC-C with 24 warehouses per server are the low-contention workloads; (b) YCSB-T with θ=0.99 and (d) TPC-C with 4 warehouses per server are the high-contention workloads.

Workloads. We use the following two benchmarks:
YCSB-T. We modify YCSB [47] according to prior work [48]. Each server maintains a single table, where each record contains 24 columns. Each column is a key-value pair, which is the access granularity, including an 8-byte key and a 16-byte value. The key consists of the `server_id`, `record_id`, and `column_id`. Each transaction reads/writes 8 key-value pairs, and the write operation is read-modify-write. To focus on evaluating distributed transactions, the 8 keys are distributed on 8 servers by assigning the `server_id`. The `record_id` is generated by Zipf distribution; the `column_id` is randomly generated.
TPC-C. The TPC-C [49] benchmark simulates a wholesale supplier with five types of transactions. We use *new-order* and *payment* transactions, which are the two most dominant transactions (88%) and are the primary source of conflicts [23, 48, 50]. To focus on evaluating distributed transactions, all the items in new-order transactions are from remote warehouses. We evaluate SwitchTx under workloads with different distributed transaction configurations in §5.5.2.
Competitors. We compare SwitchTx with FaSST and Eris:
FaSST [23] is an OCC-based and RDMA-optimized distributed transaction system with server-based coordination.
Eris [14] uses a central programmable switch for sequencing to optimize independent deterministic transactions. we implement the sequencer in Eris on the programmable switch based on the open-source server-based sequencer [51]. Because Eris needs an extra transaction chipping algorithm to convert the transactions in TPC-C (they are not deterministic transactions) into multiple deterministic transactions, we evaluate Eris only under YCSB-T.

For a fair comparison, we implement FaSST [52], Eris [51] with the same network stack as SwitchTx. SwitchTx uses dynamic admission control to control the total concurrent transactions, and other systems use static admission control (`MPL` = 2 for each thread in YCSB-T, and `MPL` = 1 in TPC-C).

5.2 Overall Performance

To analyze the performance of SwitchTx, we apply each technique one by one. We evaluate the performance under YCSB-T (Figure 10.(a) and (b)) and TPC-C (Figure 10.(c) and (d)) with different conflict levels. YCSB-T changes the conflict levels by varying the Zipf θ parameter; TPC-C changes the conflict levels by varying the number of warehouses per server. The **+GaS** is equivalent to the configuration where only the core switch is a programmable switch. In Figure 10, **+GaS** represents that SwitchTx uses a single switch (i.e., $switch_0$ in Figure 9) to offload gather-and-scatter operations; **+Tree-GaS** represents that a tree of switches in the cluster executes the gather-and-scatter operation. **+Reorder** and **+MPL** (i.e., admission control) are two design parts of addressing the semantic gap between transactions and network: reordering network messages via the transaction semantic, and co-designing transaction admission control with network congestion control, respectively.

5.2.1 Throughput. By comparing the throughput between Eris, FaSST, and SwitchTx, we make the following three observations:

First, compared to FaSST, SwitchTx improves the throughput by up to 1.81× and 1.87× under YCSB-T and TPC-C, respectively. The throughput improvement from all four techniques: 1) In-switch GaS saves CPU cycles of coordination, which can be used to initiate and process more transactions; it also reduces conflicts between transactions by shortening the contention span; it improves the throughput by 1.63×, 1.48×, 1.54×, and 1.35× under the four benchmarks, respectively. 2) The tree-based GaS reduces more network pressure than GaS; and it uses a closer switch to handle GaS operations, further shortening the contention span; it improves the throughput by 2.1%~7.8%. 3) The semantic-aware message reordering provides a small improvement in throughput (up to 7.2%). 4) The dynamic admission control adjusts `MPL` based on the throughput to make full use of system resources; it improves the throughput by 5.0%, 13.4%, 4.8%, and 28.6% under the four benchmarks, respectively.

Second, under high-contention workloads (Figure 10.(b) and (d)), dynamic admission control brings more performance improvement compared to the improvement under low-contention workloads (Figure 10.(a) and (c)). This is because, in high-contention workloads, the dynamic `MPL` mitigates performance degradation due to abort.

Third, the throughput of Eris is only 37.7% (55.0%) and 21.1% (30.3%) of FaSST and SwitchTx under YCSB-T with θ=0.8 (0.99). This is because SwitchTx and FaSST can use more CPU resources. SwitchTx and FaSST use 192 threads, and Eris uses up to 40 threads for the limitation of the centralized sequencer. The limitation is that Eris needs to store the sequence number for each shard in the packet header (4 bytes per shard in our evaluation). However, the programmable switch can only parse 40 sequence numbers. Therefore, Eris can only support 40 shards in our evaluation (evaluation uses 15 shards in Eris's paper). Eris is suitable for the scenario where each thread is powerful but the number of threads is small, and we will show SwitchTx throughput with the same number of threads in the scalability evaluation (§5.4).

5.2.2 Latency. Figure 10 also shows the P50 and P99 end-to-end latency. We make the following four observations:

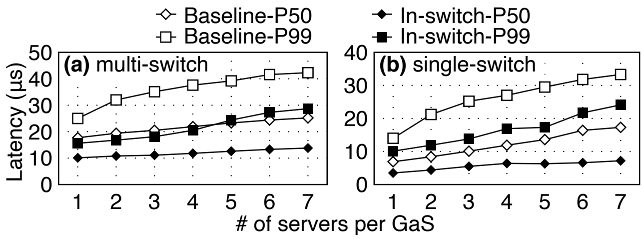

Figure 11: P50 and P99 tail latency of GaS operations. 24 threads per server; one GaS operation per thread.

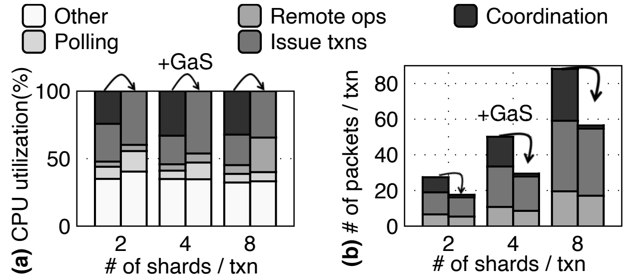

Figure 12: Saved CPU resources and packets from GaS.

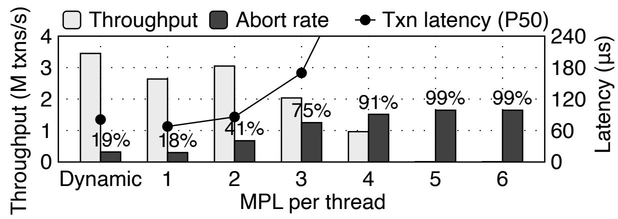

Figure 13: Dynamic vs. static admission control under YCSB-T. Total $MPL \times 24 \times 8$ **concurrent txns. Zipf** θ**=0.99.**

First, compared to FaSST, SwitchTx reduces the latency, due to the first three technologies: +GaS, +Tree-GaS, +Reorder. In-switch GaS operations (i.e., +GaS and +Tree-GaS) reduce the number of network hops on the critical path of transaction committing. The semantic-aware message reordering gives priority to messages of retrying transactions, which reduces the P99 latency by up to 7.7%.

Second, in-switch GaS designs (i.e., +GaS and +Tree-GaS) reduce the median latency by up to 40.4% and 21.6% under YCSB-T and TPC-C, respectively. This is because TPC-C has longer execute phases than the YCSB-T. The optimizations are not evident, leading to less latency reduction than YCSB-T.

Third, the admission control using throughput as the performance metric brings higher latency. Under TPC-C, the latency of SwitchTx is even worse than that of FaSST. This is because the dynamic admission control in SwitchTx is aimed at optimizing throughput; when throughput has room for improvement, it constantly increases `MPL`, resulting in high network pressure and high latency.

Fourth, Eris has low latency under YCSB workloads. Due to the in-switch centralized sequencer, Eris only needs 1 round trip to commit an independent transaction. However, the latency in Eris is longer than the latency of a network round trip ($4\mu s$); this is because Eris has to wait for the completion of requests with smaller serial numbers, instead of processing requests once it receives them.

5.2.3 Lock Time. Figure 10 also shows the average lock time. The lock time is the time between lock acquisition and lock release. The lock time is shorter under high-contention configurations for both YCSB-T (Figure 10.(b)) and TPC-C (Figure 10.(d)). This is because more transactions are aborted and retrying, which leads to earlier lock releasing. In this evaluation, we only focus on the lock time in the low-contention benchmark, because the lock time in this benchmark can represent the contention span of transactions. In YCSB-T with θ=0.8 (Figure 10.(a)), which has few inter-transaction conflicts, thanks to the in-network coordination (+GaS), SwitchTx reduces the lock time by 41.5%.

According to the overall performance evaluation, we can conclude that: 1) in-network coordination (+GaS and +Tree-GaS) in SwitchTx alleviates the coordination cost of distributed transactions, leading to higher throughput and lower latency; 2) the semantic-aware message reordering (+Reorder) reduces the tail latency; 3) throughput-optimized dynamic admission control (+MPL) fully exploits the throughput at the expense of latency; 4) Eris has lower latency, but its throughput is limited by the scalability of the centralized sequencer.

5.2.4 Recovery Time of Switch Failure. We stop and then restart the switch to emulate the switch failure. SwitchTx needs 32 seconds to reconfigure the switch and re-build the RC connection between servers. All servers keep aborting and retrying the blocked transactions until the switch recovers.

5.3 In-Depth Analysis

5.3.1 Latency of In-Switch GaS. We evaluate the tail latency of GaS operations for in-server implementation and in-switch implementation. We use 8 servers, each with 24 threads; each thread issues a GaS operation, and each GaS operation completes the synchronization among `n` (from 2 to 8) servers, similar to barrier operation in MPI. Figure 11 shows the latency with the increasing number of servers involved in the GaS operations. We make the following two observations. First, due to the overhead of cross-rack communication, the latency of the multi-switch settings is higher than that of single-switch settings. Compared with the baseline (i.e., in-server) GaS operation, in-switch GaS operation reduces P99 latency up to 48.4% and 45.2% in multi-switch and single-switch settings respectively. The extra latency of cross-rack communication for in-switch GaS is less. Second, as `n` grows, the latency increases, since GaS operation is affected by the slowest messages, and the more involved servers in a GaS operation, its latency is more likely to be affected by the network quality fluctuations.

5.3.2 CPU Saving of In-Switch GaS. Figure 12 shows the CPU utilization and the packets of different functions in the systems without/with in-switch GaS. We make the following two observations. First, in the baseline, each server costs 24.2%~33.0% of CPU resources for coordination. SwitchTx uses them to perform other operations (i.e., issuing new transactions and processing data operations as participants), which can achieve higher throughput. Second, in the baseline, as the number of shards involved in each transaction

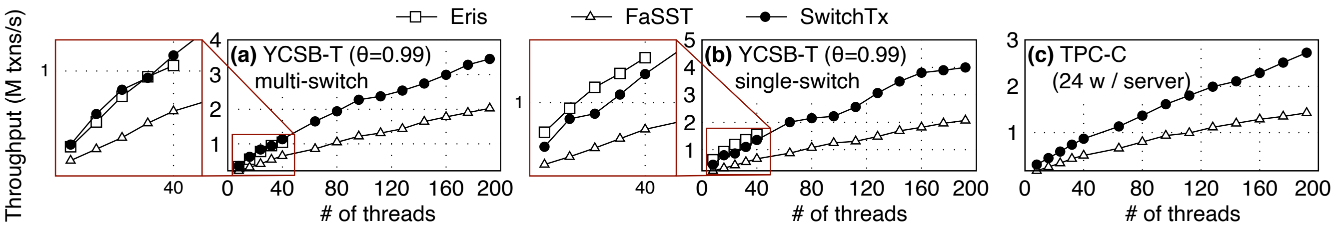

Figure 14: Throughput scalability under YCSB-T and TPC-C. Servers in (a) and (c) locate in three racks (multi-switch); all servers in (b) locate in a single rack (single-switch).

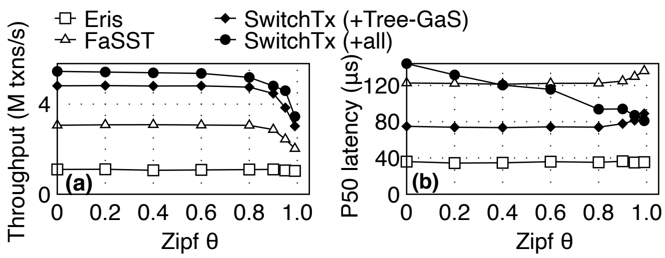

Figure 15: YCSB-T with varing Zipf θ.

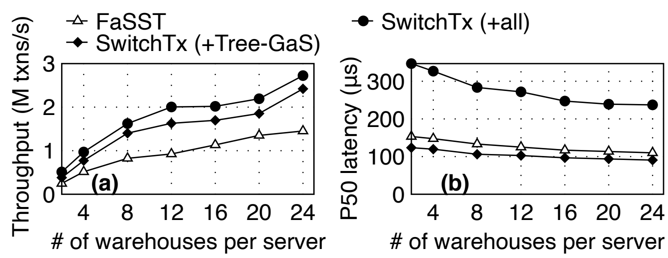

Figure 16: TPC-C with varing # of warehouses.

increases, the average number of coordination packets per transaction in the server increases from 8.4% to 29.1%. With in-switch GaS, the server only needs to process about 1.6 packets for the coordination of each transaction. Therefore, in-switch GaS brings higher performance improvement for transactions with more participants.

5.3.3 Admission Control. We evaluate the benefits of our dynamic admission control. To compare dynamic admission control with the static admission control, we evaluate the throughput and latency of SwitchTx with static `MPL`, as shown in Figure 13. We observe that the performance is the highest when `MPL`=2 in static admission control, but SwitchTx with dynamic one still boosts throughput by 13.4% and has 4.3% less latency. This is because the dynamic admission control changes the `MPL` over time based on real-time performance.

5.4 Scalability

We use 8 servers and increase the threads in each server to evaluate the scalability under YCSB-T (Figure 14.(a)(b)) and TPC-C (Figure 14.(c)). In this evaluation, (a) and (c) use the multi-switch topology; (b) uses the single-switch topology.

Scalability of FaSST and SwitchTx. The throughput of FaSST and SwitchTx are scalable as the thread count increases. This is because they do not have a centralized component for different transactions.

Scalability of Eris. As shown in Figure 14.(a), Eris performs worse than SwitchTx under multi-switch even at the small scale. this is because its centralized sequencer becomes the performance killer. As shown in Figure 14.(b), when all servers in the same rack, Eris performs better than SwitchTx at this small scale. However, the number of threads in each server of Eris is 5 (total 40 threads) due to the limited header size in programmable switches. Note that our throughput of 8 servers is up to 5.18Mops, which is larger than the software sequencer capacity (i.e., 1.61Mops using the dedicated servers) in the Eris paper. In SwitchTx, the throughput does not reach the switch bottleneck.

Further, the scalability of Eris is not linear when the number of threads is from 8 to 40. This is because 1) Eris needs the switch's coordination, even if the keys of a transaction are on the same server (but on the different threads); 2) Eris needs to process requests in the order of sequence numbers, and as the concurrent request count increases, the network out-of-order problem blocks the threads in Eris, which also damages the performance.

We also emulate a switch using a server. Because Eris needs to guarantee the atomicity of sequencer vector update, which is difficult to be scaled to multiple threads, the simulated switch uses only one thread. When there are 30 threads in the system with YCSB-T workload, the throughput of FaSST, Eris and SwitchTx are 0.24Mops/s, 0.29Mops/s, and 0.4Mops/s, respectively. This is because Eris's packets require more processing resources (i.e., multiple operations on the sequencer vector) than the other two systems.

5.5 Sensitivity Analysis

5.5.1 Varying the Contention Level of Workloads. Figure 15 shows the throughput and P50 (median) latency under YCSB-T with varying Zipf θ. Figure 16 shows the performance under TPC-C with varying warehouse count. We have the following two observations:

First, compared to FaSST, SwitchTx improves the throughput by 1.67×~2.16× under different contention levels. This is not only because SwitchTx saves CPU resources, but also because SwitchTx reduces conflicts by shortening the contention span.

Second, compared to FaSST, SwitchTx (+Tree-GaS) reduces the latency by 18.9%~39.5% under different contention levels. SwitchTx (+all), which uses the dynamic admission control, trades the latency for higher throughput. In YCSB-T (Figure 15.(b)), the latency in SwitchTx (+all) decreases when the Zipf θ increases. Because

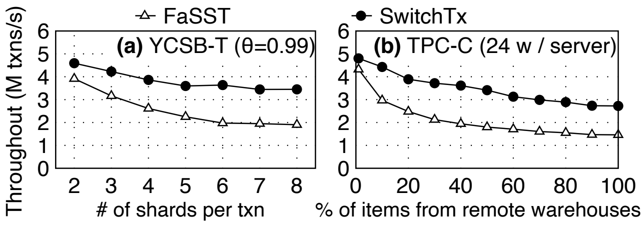

Figure 17: Throughput with varing % of distributed txns.

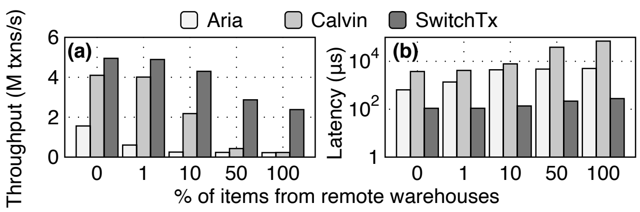

Figure 18: TPC-C with other CC protocols.

the `MPL` for YCSB-T in SwitchTx (+all) decreases when the contention level becomes high. In TPC-C (Figure 16.(b)), the latency in SwitchTx (+all) is always higher than the latency in SwitchTx (+Tree-GaS) and FaSST. This is because the transactions in TPC-C have a longer execution phase, which means higher retry overhead. SwitchTx (+all) has more retrying transactions for the higher `MPL`.

5.5.2 Varying the Percentage of Distributed Transactions. We design SwitchTx for distributed transactions. SwitchTx has no performance improvement for single-partition transactions. We evaluate the throughput under workloads with different percentages of distributed transactions. In YCSB-T (Figure 17.(a)), as the number of partitions involved in each transaction increases, compared to FaSST, the throughput improvement of SwitchTx is from 1.17× to 1.81×. In TPC-C (Figure 17.(b)). The percentage of items from remote warehouses in new-order transactions affects the percentage of distributed transactions. With the increasing percentages, the throughput improvement of SwitchTx is from 1.11× to 1.86×.

5.6 Comparison with Other CC protocols

We compare SwitchTx with two state-of-the-art deterministic OLTP systems. We use the open-source codes [53] of Aria [16].

Calvin [7] is a classical deterministic OLTP system that orders transactions and acquires the locks before executing transactions.

Aria [16] is a state-of-the-art deterministic OLTP system which allows read/write sets unknown before transaction execution.

Figure 18 shows the throughput and median latency under TPC-C. We increase the percentage of remote items in New-Order to increase the percentage of cross-shard transactions. We observe that the performance of Aria and Calvin degrades significantly as the number of cross-shard transactions increases. When all items are from remote warehouses, the throughput of SwitchTx is 10.2× and 10.6× higher than the throughput of Aria and Calvin, respectively. This is because the cross-shard transactions in Aria and Calvin cause blocking and need a lot of network communication. Further, Aria and Calvin have much higher latency than SwitchTx due to batching.

6 RELATED WORK

Distributed Transaction System. We discuss the related work in the field of distributed transaction systems.

Admission control in the transaction system. Industry-level databases usually use the static admission control to control the number of concurrent transactions. Further, some studies design adaptive admission control [54, 55] to handle dynamic workloads. Cicada [56] adjusts the backoff of retry transactions to reduce contention. SwitchTx explores the relationship between congestion control and admission control, to co-design them for distributed transaction systems.

Distributed transaction system under fast network. Some studies [5, 24, 27] use RDMA to design distributed transaction systems. FaSST [23] leverages scalable RDMA primitives to improve performance. DrTM [32, 57] combines RDMA and hardware transactional memory. NAM-DB [58] focuses on scalability in the RDMA-based distributed transaction system. These variant distributed transaction systems are orthogonal to SwitchTx, and we can apply our scalable GaS to them and make these systems perform better.

In-network transaction scheduling. AlNiCo [59] uses FPGA-based SmartNICs to schedule transaction requests, reducing contention between multiple CPU cores. Jepsen et al. [60, 61] introduce batch-based transaction reordering, grouping, and steering techniques in programmable switches to amortize the transaction overhead and reduce contention. They focus on single-server transactions and can not support distributed transactions.

In-switch cache system. Some studies leverage the memory in the switch to build the cache layer(e.g., NetCache [62], DistCache [63], and P4DB [64]) for distributed systems. NOCC [65] uses switches to cache versions and keys of data. Different from them, SwitchTx offloads the coordination task to a tree of switches instead of storing the data in switches. Using memory in switches to cache the data or locks will further exploit the performance but would complicate the crash consistency design; we leave it as our future work.

In-Network Aggregation Accelerator. Some studies [66–69] use programmable switches to accelerate aggregation work in AI training system. Different from the above studies, our proposed GaS abstraction aims to aggregate coordination metadata (rather than data). HovercRaft++ [70] is a replication system based on Raft using the switch to gather ACK messages from followers. Different from it, SwitchTx aims to accelerate all phases in the concurrency control and replication protocols and aims to be scalable to all switches.

7 CONCLUSION

This paper presents SwitchTx, a fast and scalable transaction processing system. SwitchTx introduces in-network gather-and-scatter operations to mitigate coordination between servers. Moreover, SwitchTx utilizes multiple switches to parallelize disjoint transactions and avoid the single-point bottleneck. SwitchTx realizes transaction traffic control to fully utilize network resources.

ACKNOWLEDGEMENTS

We sincerely thank the reviewers for their valuable feedback. We also thank Minhui Xie, Zhe Yang, Jing Wang, and Jian Gao for their suggestions. This work is funded by the National Natural Science Foundation of China (Grant No.62022051, 61832011, 61872376), Huawei,and the CCF-Huawei Innovation Research Plan.

REFERENCES

[1] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[2] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[3] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[4] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[5] 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*, pages 54–70, 2015.

[6] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. Reducing database locking contention through multi-version concurrency. *Proc. VLDB Endow.*, 7(13):1331–1342, aug 2014.

[7] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[8] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, 2014.

[9] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 279–294, 2015.

[10] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment*, 12(2):169–182, 2018.

[11] Robin Rehrmann, Carsten Binnig, Alexander Böhm, Kihong Kim, and Wolfgang Lehner. Sharing opportunities for oltp workloads in different isolation levels. *Proceedings of the VLDB Endowment*, 13(10):1696–1708, 2020.

[12] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[13] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[14] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120, 2017.

[15] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication. In *EDBT*, pages 73–84, 2020.

[16] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, jul 2020.

[17] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An efficient metadata service for Large-Scale distributed filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, Santa Clara, CA, February 2022. USENIX Association.

[18] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. Schism: a workload-driven approach to database replication and partitioning. 2010.

[19] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[20] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 511–526, 2020.

[21] Thamir M. Qadah and Mohammad Sadoghi. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 13–25, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An rdma-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 773–785, USA, 2017. USENIX Association.

[23] 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, 2016.

[24] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[25] Chao Wang, Kezhao Huang, and Xuehai Qian. A comprehensive evaluation of rdma-enabled concurrency control protocols. *CoRR*, abs/2002.12664, 2020.

[26] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. Th-dpms: Design and implementation of an rdma-enabled distributed persistent memory storage system. *ACM Trans. Storage*, 16(4), oct 2020.

[27] Masoud Hemmatpour, Bartolomeo Montrucchio, Maurizio Rebaudengo, and Mohammad Sadoghi. Analyzing in-memory nosql landscape. *IEEE Transactions on Knowledge and Data Engineering*, 34(4):1628–1643, 2022.

[28] Dan RK Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 209–215, 2019.

[29] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on load distribution and the role of programmable switches. *ACM SIGCOMM Computer Communication Review*, 49(1):18–23, 2019.

[30] Theophilus A Benson. In-network compute: Considered armed and dangerous. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 216–224, 2019.

[31] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with in-network cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292, 2021.

[32] 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*, pages 87–104, 2015.

[33] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, 2018.

[34] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 145–161, 2021.

[35] 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.

[36] Mohammad Sadoghi and Spyros Blanas. Transaction processing on modern hardware. *Synthesis Lectures on Data Management*, 14(2):1–138, 2019.

[37] Suyash Gupta and Mohammad Sadoghi. Easycommit: A non-blocking two-phase commit protocol. In *EDBT*, pages 157–168, 2018.

[38] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan RK Ports, and Aurojit Panda. Multitenancy for fast and programmable networks in the cloud. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[39] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 194–206, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: Building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 342–356, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–14, 2019.

[42] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. Octopus+: An rdma-enabled distributed persistent memory file system. *ACM Trans. Storage*, 17(3), aug 2021.

[43] Mellanox. Raw packet. https://community.mellanox.com/s/article/raw-ethernet-programming--basic-introduction---code-example, 2020.

[44] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[45] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[46] Barefoot Technologies Corporation. Barefoot tofino https://barefootnetworks.com/products/brief-tofino/, 2018.
[47] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
[48] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *Proceedings of the VLDB Endowment*, 13(5):629–642, 2020.
[49] Standard Specification. TPC BENCHMARK C. 1994.
[50] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.
[51] Jialin Li, Ellis Michael, and Dan RK Ports. Implementation of the Eris protocol with the software sequencer. https://github.com/nicklijl/simbricks-nopaxos, 2021.
[52] Anuj Kalia, Michael Kaminsky, and David G Andersen. Implementation of FaSST. https://github.com/efficient/fasst, 2017.
[53] Yi Lu, Yu Xiangyao, Lei Cao, and Madden Samuel. Implementation of the Aria. https://github.com/luyi0619/aria, 2021.
[54] Donghui Wang, Peng Cai, Weining Qian, and Aoying Zhou. Discriminative admission control for shared-everything database under mixed oltp workloads. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 780–791. IEEE, 2021.
[55] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich Nahum, and Adam Wierman. How to determine a good multi-programming level for external scheduling. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 60–60. IEEE, 2006.
[56] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multicore in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35, 2017.
[57] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–17, 2016.
[58] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transactions can scale. *arXiv preprint arXiv:1607.00655*, 2016.
[59] Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, and Jiwu Shu. AlNiCo: SmartNIC-accelerated contention-aware request scheduling for transaction processing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 951–966, Carlsbad, CA, July 2022. USENIX Association.
[60] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. In-network support for transaction triaging. 2021.
[61] Theo Jepsen. *Building blocks for leveraging in-network computing*. PhD thesis, Università della Svizzera italiana, 2020.
[62] 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*, pages 121–136, 2017.
[63] 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, 2019.
[64] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. P4db - the case for in-network oltp. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1375–1389, New York, NY, USA, 2022. Association for Computing Machinery.
[65] Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. Infinite resources for optimistic concurrency control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 26–32, 2018.
[66] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 279–291, 2019.
[67] 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 *NSDI*, pages 741–761, 2021.
[68] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.
[69] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L Wolf. Netagg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 249–262, 2014.
[70] Marios Kogias and Edouard Bugnion. Hovercraft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. Technical report, 2020.