Dissertation Submitted to Shanghai Jiao Tong University
for the Degree of Doctor

FAST DISTRIBUTED TRANSACTION PROCESSING USING RDMA AND NVM

Candidate:	Xingda Wei
Student ID:	017037910004
Supervisors:	Prof. Binyu Zang, Prof. Rong Chen, Prof. Haibo Chen
Academic Degree Applied for:	Ph.D. of Engineering
Speciality:	Computer Science
Affiliation:	School of software, SEIEE
Date of Defence:	May 24, 2021
Degree-Conferring-Institution:	Shanghai Jiao Tong University

FAST DISTRIBUTED TRANSACTION PROCESSING USING RDMA AND NVM

ABSTRACT

Fast distributed transaction processing is a key pillar in modern datacenter computing. However, distributed transaction processing is notoriously slow due to the cost of coordination among multiple nodes and logging for high availability and durability. The emergence of new hardware features, including fast network (RDMA), and fast storage-class memory (NVM) bring opportunities for reducing the cost of transaction processing. To fully utilize these hardware features, it is crucial to develop methodologies to bridge the semantic gap between distributed transactions and new hardware features, and abstractions for coordinating different hardware efficiently.

This thesis builds DrTM+X, a distributed transaction system that fully leverages the power of RDMA and NVM. DrTM+X is over one order of magnitude faster than distributed transaction systems without these hardware features. It also has a significant performance increase compared to state-of-the-art systems with RDMA and NVM.

The high performance of DrTM+X is enabled by three approaches. First, we conduct the first systematic study to summarize design guidelines to best utilize RDMA and NVM together, as well as abstractions for systems to transparently apply the summarized guidelines. We show with careful designs, transactions can fully leverage the high performance of RDMA and NVM. Second, we design *learned cache*, the first machine learning method to bridge the semantic gap between ordered data access of transactions and the hardware features of RDMA. It reduces the network roundtrips for ordered key-value access with RDMA from $O(\log N)$ to $O(1)$. Third, we design

hybrid schemes to tame the complexities of accelerating concurrency control with RDMA. We propose the first *phase-by-phase analysis* to select the best RDMA primitives for OCC, and DST to differentiate the processing of read-only transactions for higher concurrency. DST eliminates the performance bottleneck of traditional MVCC under RDMA.

Our thesis shows that fully leveraging new hardware features for transactions requires a careful system design. The techniques introduced in this thesis can also be applied broadly in various contexts. For example, our summarized hardware abstractions have benefited other RDMA-NVM systems, while DST helps to improve the performance and scalability of existing databases.

KEY WORDS: Distributed Transaction, Remote Direct Memory Access, Non-volatile Memory, Concurrency Control, Learned Index

Contents

Chapter 1 Introduction

Fast transaction processing is a key pillar for many systems, including but not limited to web service, stock exchange, and e-commerce. Due to the increasing amount of data volume, a common way to support transaction processing is to partition the data through many shards. This necessitates distributed transactions. Distributed transactions with serializability and high availability provide a powerful abstraction to programmers. They give them an illusion of a single machine that executes transactions with strong consistency and never fails.

"… only **4%** of wall-clock time
is spent on **useful** data processing …"

Michael Stonebraker,
"The Traditional RDBMS Wisdom is All Wrong"

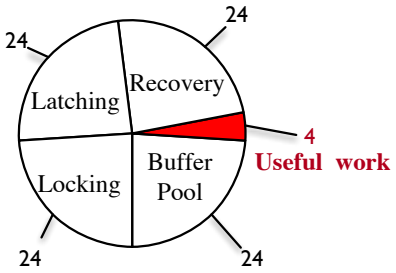


Figure 1–1 Costs of executing transactions. Reproduced from Michael Stonebraker [1].

Unfortunately, distributed transactions are notoriously slow, due to the cost of coordination among multiple nodes and logging for high availability and durability. Even single-node transaction processing has a high cost. Michael Stonebraker has once said that *"The traditional RDBMS wisdom is all wrong ... as only 4% of wall-clock time is spent on useful data processing, while the rest is occupied with buffer pools, locking, latching, recovery"* [1], illustrated in Figure 1–1. In distributed transactions, these costs are further magnified by slow remote node communications and providing stronger properties (i.e., high availability). For example, locking remote data in the distributed transaction has a much higher overhead than single-node systems because network latency is order-of-magnitude higher than local memory access even under fast networks [2]. Meanwhile, distributed transactions have to additionally synchronize multiple data replicas to achieve high availability.

People have been seeking to improve the performance of (distributed) transaction processing for decades [3-20]. Early works focused on improving the algorithmic properties in transaction protocols. For example, Rococo [19] proposes a new concurrency control method to avoid aborts in distributed transactions. Meanwhile, recent work has revealed that co-designing transactions with new hardware features can bring huge performance improvements [3, 7, 13, 17, 21]. For example, Silo is a centralized database that optimizes optimistic concurrency control (OCC) [22] with multi-core platforms. FaRM is a distributed transaction system that co-design OCC with advanced networking features. By carefully designing the system with modern hardware features, FaRM and Silo can achieve orders-of-magnitude better performance than traditional transaction systems.

This dissertation focuses on using the emerging datacenter hardware technologies, namely RDMA (Remote Direct Memory Access) and NVM (Non-volatile Memory) (§1.1), to reduce the costs in distributed transactions. RDMA is a fast networking feature widely adopted in modern datacenters. NVM is a fast storage-class memory. They both provide new hardware features that distributed transactions can use to reduce the execution costs.

A straightforward approach of applying new hardware features to distributed transactions—adopting the backward-compatible primitives provided by the new hardware—can clearly improve the performance of transactions. However, this approach fails to fully utilize the advanced features of the new hardware, and thus, results in inferior performance (e.g., see §3.2) Differently, we try to answer a natural question: *how can we build a fast distributed transaction system that can fully utilize the features of RDMA and NVM?* In the later section (§1.1), we will see that it's non-trivial to answer this question because first, distributed transaction processing has a semantic gap between new hardware features. Second, a careful system design is required to efficiently bridge heterogeneous hardware together.

1.1 Opportunities and challenges of using RDMA and NVM for distributed transactions

Based on the current technology trends, it seems that Moore's law would come to an end, and Dennard scaling would break down [23]. Hence, people can hardly improve the performance of transactions by simply switching to the next-generation CPU. Besides, it is more challenging for general-purpose CPUs to keep the step with rapid advances in fast

networking, especially in bandwidth [24]. Therefore, people are starting to co-designing systems with RDMA [2, 13, 17-18, 25-37], a fast networking feature with offloading capabilities; co-designing systems with NVM [38-50], a fast storage-class memory; and both [13, 18, 51-57].

Remote direct memory access (RDMA) is a high-bandwidth and low-latency networking feature. It provides datagram communication (*two-sided primitive*), together with offloading technology (*one-sided primitive*): the network card can directly access the memory of remote machines bypassing kernel and remote CPUs. Therefore, one-sided primitive has both high communication performance and high CPU utilization. Researchers from industry and academia have been using RDMA to boost the performance of distributed transactions, usually by orders of magnitudes [13, 17-18, 25]. Besides leveraging two-sided primitive to optimize the communications in the distributed transaction [25], one can also offload the transaction protocol with one-sided primitive [13, 17]. Hence, RDMA can possibly reduce the cost of remote locking shown in Figure 1–1.

Non-volatile memory (NVM) is a storage class memory. Compared to traditional storage like SSD, it can provide higher bandwidth and lower latency. Besides, it is also byte-addressable, which means that other devices (e.g., CPU or RDMA) can directly access it as DRAM. People can leverage NVM to reduce the buffering cost and logging cost required for recovery in transaction processing (Figure 1–1).

Challenges of adopting RDMA and NVM for distributed transactions. Though RDMA and NVM pose an optimistic direction for reducing the costs in distributed transactions, we face two key challenges in fully utilizing them for transactions:

- **Limited offloading capability.** Only using two-sided primitive—the backward compatible primitive of RDMA—cannot fully leverage the power of it. This is because CPU would first become the bottleneck (§3.2). However, one-sided RDMA only has limited offloading capabilities, i.e., the NIC only supports simple memory read/write. Therefore, simply offloading transactions to one-sided RDMA will cause network amplification [25], degrading the overall utilization of RDMA. This dilemma further opened a recent active debate over which RDMA primitive, namely one-sided or two-sided, is better suited for distributed transactions [13, 17, 25, 58]. This dissertation will present how to close this debate with a new phase-by-phase analysis of optimizing RDMA-enabled distributed transactions

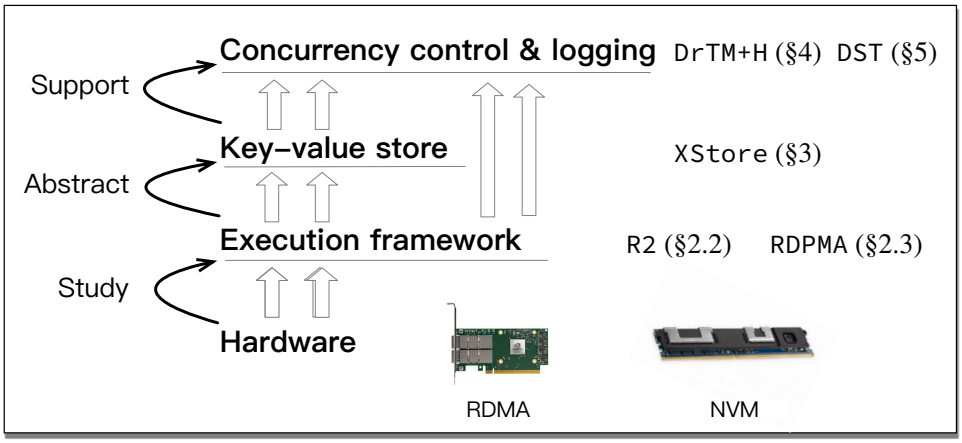


Figure 1–2 DrTM+X (§6): a bottom-up approach to build a fast distributed transaction processing system.

(§4). We will further show how to leverage machine learning-based methods to efficiently offload complex operations to one-sided RDMA (§3).

- **Uncoordinated heterogeneous hardware.** NVM has the appealing feature of byte-addressability, which means one-sided RDMA can directly access NVM. This observation allows us to achieve the best of both worlds: leveraging one-sided RDMA to bypass CPU and use NVM for fast persistent operations. Unfortunately, RDMA and NVM are designed in an "uncoordinated" manner, meaning they are not specifically designed for each other. As a result, RDMA-NVM systems may suffer from inferior performance when accessing NVM with RDMA. For example, our initial experiment demonstrates that the performance of *remote write* is far from the limits of NVM (§2.3.3) when using one-sided RDMA to write NVM directly. Hence, it is imperative to conduct a thorough study of the inferior performance and provide optimizations to efficiently coordinate RDMA and NVM together. Unfortunately, as production NVM is just recently publicly available, we lack such a study. This dissertation will first present a systematic study on how to best leveraging NVM with RDMA before optimizing distributed transactions with them (§2.3).

1.2 Thesis contributions

This thesis focuses on reducing the costs for executing distributed transactions using RDMA and NVM. To tackle the challenges mentioned in §1.1, we take a bottom-up approach, as illustrated in Figure 1–2. At the bottom, we first conduct a systematic study on how to efficiently use RDMA and how to effectively glue RDMA and NVM together. Based on the study, we build two execution frameworks (`R2` (§2.2) and `RDPMA` (§2.3)) to abstracted the optimizing details away from upper systems. By building other system components atop of `R2` and `RDPMA`, we can effectively prevent inefficiency caused by improper usage of hardware features. Going up, we split the transactional processing into several layers (e.g., concurrency control) and examined how each of them can be effectively designed with RDMA or NVM (§3, §4 and §5). Finally, we combined our proposed techniques into a single platform (§6) to provide fast distributed transactions. In summary, this thesis makes the following five contributions:

- **A systematic study on how to best use RDMA and NVM (§2).)** We conduct a systematic study to summarize optimization hints that the system designer can use to program with RDMA, and to exploit NVM with RDMA better. Specifically, we demonstrate how system configurations, NVM access patterns, and RDMA-aware optimizations affect the efficacy of systems that use RDMA and NVM. Based on the summarized hints, we build two systems to fully utilize RDMA and NVM and hide detailed optimizations. `R2` is a distributed execution framework designed with RDMA, and `RDPMA` is a remote persistent memory library designed for RDMA and NVM. These two systems form the basis of the systems presented in later chapters. Finally, we also demonstrate how to use `R2` and `RDPMA` to improve the performance of existing systems beyond distributed transactions. m
- **An RDMA-friendly ordered key-value store using a learned approach (§3).** Ordered key-value store is an important building block in distributed transactions, e.g., supporting secondary-index. Since querying the key-value entry over the network is costly, RDMA has gained considerable interest in network-attached ordered key-value stores. However, due to the limited abstraction provided by one-sided RDMA, traversing the remote tree-based index in ordered key-value stores with it becomes a critical obstacle, causing an order-of-magnitude slowdown and limited scalability due to multiple roundtrips. Using index cache with

conventional wisdom—caching partial data and traversing them locally—usually leads to limited effect because of unavoidable capacity misses, massive random accesses, and costly cache invalidations. On the other hand, the CPU would become the bottleneck for two-sided RDMA-based ordered key-value stores.
We argue that the machine learning (ML) model is a perfect cache structure for the tree-based index termed *learned cache*. Based on it, we design and implement `XStore`, an RDMA-based ordered key-value store with a new hybrid architecture that retains a tree-based index at the server to perform dynamic workloads (e.g., inserts) using two-sided RDMA and leverages a learned cache at the client to perform static workloads (e.g., gets and scans) using one-sided RDMA. The key idea is to decouple ML model retraining from index updating by maintaining a layer of indirection from logical to actual positions of key-value pairs, which allows a stale learned cache to continue predicting a correct position for a lookup key. Evaluations on micro and production workloads demonstrate the efficiency of `XStore`: it can outperform the state-of-the-art one-sided and two-sided-based ordered key-value stores by up to 5.9× (from 2.7×).

- **A phase-by-phase approach to finding the optimal primitive choice for RDMA-enabled distributed transactions (§4).** There is currently an active debate on which RDMA primitive (i.e., one-sided or two-sided) is optimal for distributed transactions. Such a debate has led to a number of optimizations based on one RDMA primitive, which was shown with better performance than the other.
We perform a systematic comparison between different RDMA primitives with a combination of various optimizations using representative OLTP workloads. More specifically, we investigate the implementation of optimistic concurrency control (OCC) by comparing different RDMA primitives using a phase-by-phase approach with various transactions from TPC-C, SmallBank, and TPC-E. Our results show that no single primitive (one-sided or two-sided) wins over the other on all phases. We further conduct an end-to-end comparison of prior designs on the same codebase (`R2`) and find none of them is optimal.
Based on the above analysis, we build `DrTM+H`, a new hybrid distributed transaction system that always embraces the optimal RDMA primitives at each phase of transactional execution. Evaluations using popular OLTP workloads including TPC-C and SmallBank show that `DrTM+H` achieves over 7.3 and 90.4 million

transactions per second on a 16-node RDMA-capable cluster respectively. This number outperforms the pure one-sided and two-sided systems by up to 1.89X and 2.96X for TPC-C with over 49% and 65% latency reduction.

- **A scalable and general timestamp method to enable MVCC for distributed transactions (§5).** Multi-version concurrency control (MVCC) can unleash more concurrency for the read-only transaction, a common workload in modern data-center applications [59]. However, the traditional centralized timestamp scheme used to support MVCC may become a performance and scalability bottleneck, especially in fast RDMA-enabled transactions. We present DST, a *decentralized scalar timestamp* scheme to scale distributed transactions using multi-version concurrency control. DST is efficient in storage and network by being a scalar timestamp but requiring no centralized timestamp service for coordination. The key observation is that concurrency control (CC) protocols like OCC and 2PL already imply a serializable order among concurrent read-write transactions through conflicting database tuples. To this end, DST piggybacks on CC protocols to maintain the timestamp ordering with low cost and no new scalability bottleneck for read-write transactions. DST further provides snapshot reads with bounded staleness by using a hybrid scalar timestamp (physical clock and logical counter).
  DST is a general timestamp method, i.e., not specific to RDMA-enabled transactions. To demonstrate the generality of DST, we provide a general guideline for the integration of DST and further show the effectiveness by using three representative transactional systems (i.e., DrTM+H (§4), MySQL cluster [60], and Rococo [19]) with different CC protocols. Experimental results show that DST can achieve more than 95% of optimal performance (using Read Committed) without compromising correctness. With DST, DrTM+H achieves up to 1.8X higher peak throughput for TPC-E and outperforms other timestamp schemes by 6.3X for TPC-C. DST also leads up to 1.9X and 2.1X speedup on TPC-C for MySQL cluster and Rococo, respectively.

- **A fast distributed transaction processing system using RDMA and NVM (§6).** We propose DrTM+X, a fast distributed transaction processing system that incorporates all the above systems. Specifically, DrTM+X leverages XStore to support efficient ordered data accesses in distributed transactions, uses DrTM+H and DST

for concurrency control and equips a fast logging system for high availability and durability with RDPMA.

1.3 Thesis overview

This thesis is structured following the bottom-up approach of DrTM+X (Figure 1–2): In §2, we provide the necessary background on RDMA and NVM, our systematic study on them and how we build R2 and RDPMA based on the study. §3 presents XStore, the storage backend of DrTM+X. §4 provides necessary background on RDMA-enabled distributed transactions, and presents DrTM+H, an RDMA-enabled distributed transaction system with optimal RDMA primitive choices. In §5, we present DST, an efficient timestamp scheme to support MVCC over DrTM+H. Finally, §6 summarizes how we apply the systems in previous chapters for fast distributed transaction processing.

1.4 Open-source code

The systems and tools presented in this dissertation are available online:

- R2 (§2.2): https://github.com/wxdwfc/r2.git
- RDPMA (§2.3): https://github.com/SJTU-IPADS/librdpma
- XStore (§3): https://github.com/SJTU-IPADS/xstore
- DrTM+H (§4) and DrTM+X (§6): https://github.com/SJTU-IPADS/drtmh
- DST (§5): https://github.com/SJTU-IPADS/dst

Chapter 2 A Study of using RDMA and NVM

Designing distributed transactions with RDMA and NVM requires a clear understanding of how to best utilize these advanced hardware features. In this chapter, we conduct a thorough systematic study on how to fully exploit RDMA's high performance and how to efficiently glue RDMA and NVM together. We study and collect various RDMA or RDMA-NVM related optimizations—scattered from different sources—into one systematic study. We also propose new optimizations that address the limitations of the existing study. These optimizations are backed by an open-source set of tools (https://github.com/wxdwfc/r2.git and https://github.com/SJTU-IPADS/librdpma) for empirically evaluating their effects. We believe such a study can also benefit future system co-design with RDMA and NVM.

Based on the study, we built two systems with all the studied optimizations. `R2` is a high-performance execution framework designed with RDMA (§2.2). `RDPMA` is a library for efficient reading/writing NVM using RDMA (§2.3). It built upon `R2` by extending it with NVM. These two systems are the foundations of systems presented in the later chapters.

Roadmap. Since we use empirical analysis to study the effects of different optimizations, we will start with a brief overview of our evaluating clusters (§2.1). Then, we will present the design of `R2` (§2.2) and `RDPMA` (§2.3).

2.1 Evaluation clusters

Without explicit mention, we use two clusters for the evaluations throughout the dissertation, **VAL** and **R74V**. Table 2.1 summaries their hardware configurations.

VAL. It is a rack-scale RDMA-capable cluster with 16 machines. Each machine has with two 12-core Intel Xeon E5-2650 v4 processors, 128GB of RAM, and two ConnectX-4 MCX455A 100Gbps Infiniband NIC via PCIe 3.0 x16 connected to a Mellanox SB7890 100Gbps InfiniBand Switch. ConnectX-4 is a representive RDMA-capable NIC with high performance [58, 61]. Without explicit mention, we conduct all RDMA-related experiments on val.

Table 2–1 Measurement clusters.

Cluster	#Nodes	Descriptions
VAL	16	2 × Intel Xeon E5-2650 v4 (12 cores), 128GB DRAM, 2 × ConnectX-4 IB RNIC (100Gbps)
R74V	1	2 × Intel Xeon Gold 5215M (10 cores), 384GB DRAM, 2 x ConnectX-5 IB RNIC (100Gbps), 1x 1.5T NVM (12x Optane DIMM)
	6	2 × Intel Xeon E5-2650 v4 (12 cores), 128GB DRAM, 2 × ConnectX-4 IB RNIC (100Gbps)

R74V. This cluster is also a rack-scale RDMA-capable cluster. **R74V** has 7 machines in total, six of them have the same hardware configurations as the one in **VAL**. The left machine has equipped with Optane PM: it has two 10-core Intel Xeon Gold 5215M processors, 384GB DRAM and two ConnectX-5 MT27800 100Gbps Infiniband NIC. We attach six NVM DIMMs to each processor, allowing them to achieve the maximum (ideal) bandwidth of Optane PM (320Gbps for read and 100Gbps for write). All machines are connected to a Mellanox SB7890 100Gbps InfiniBand Switch.

2.2 R2: a high-performance execution framework with RDMA

R2 is the base execution framework used by later systems in this dissertation. It has integrated with most of the state-of-the-art RDMA-aware designs, as well as our proposed optimizations. This section presents its design and an evaluation of its performance. We start with a brief overview of RDMA.

2.2.1 Background on RDMA

RDMA is a fast networking feature with high throughput (e.g., 100Gbps bandwidth), low latency (e.g., 2μs), and low CPU overhead. Representative implementations of RDMA include InfiniBand (IB) and RDMA over Converged Ethernet (RoCE). RDMA is well-known for its one-sided primitives (READ/WRITE[①]): RDMA-capable NIC (RNIC) can directly read/write the server memory bypassing the server CPU. RDMA also provides two-sided primitives (SEND/RECV) that are similar to message passing.

QP and the programming model of RDMA. RDMA hosts use queue pair (QP) to issue RDMA requests, which has one *send queue* and one *completion queue*. Figure 2–1

① We may use READ/WRITE as one-sided RDMA READ/WRITE.

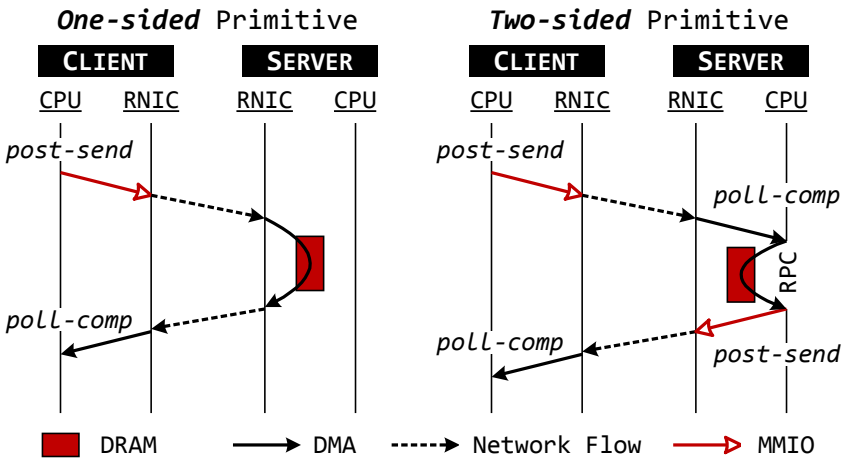


Figure 2–1 An overview of different RDMA primitives.

presents the workflow of these two primitives. To issue an RDMA request (e.g., one-sided RDMA READ), the host calls *post_send*, which uses memory-mapped IO (MMIO) to post the request to the *send queue*. If the host marks the request as *signaled*, then it can further obtain the completion event of the sent request, e.g., whether the payload of the READ has been fetched to the host, by polling the *completion queue* via *poll_comp*.

Table 2–2 Different transport modes of QP and supported operations. **RC**, **UC**, and **UD** stand for Reliable Connection, Unreliable Connection, and Unreliable Datagram, respectively.

	SEND/RECV	WRITE	READ/ATOMIC
RC	✓	✓	✓
UC	✓	✓	✗
UD	✓	✗	✗

It is worth noting that QPs have various transport modes, each supports different sets of primitives (see Table 2–2). The Reliable Connected (RC) mode supports all RDMA primitives, while the Unreliable Datagram (UD) mode only supports two-sided primitive (`SEND/RECV`). On the other hand, UD is connectionless so the application can use fewer UD QPs than RC QPs [25].

2.2.2 Basic design of R2

QP creation. When building `R2`, we found how the system creates QP can significantly affect the primitive performance posted by the QP. More specifically, the system creates QP from a context exposed by the user-space RDMA driver (`ibv_contenxt`). We use

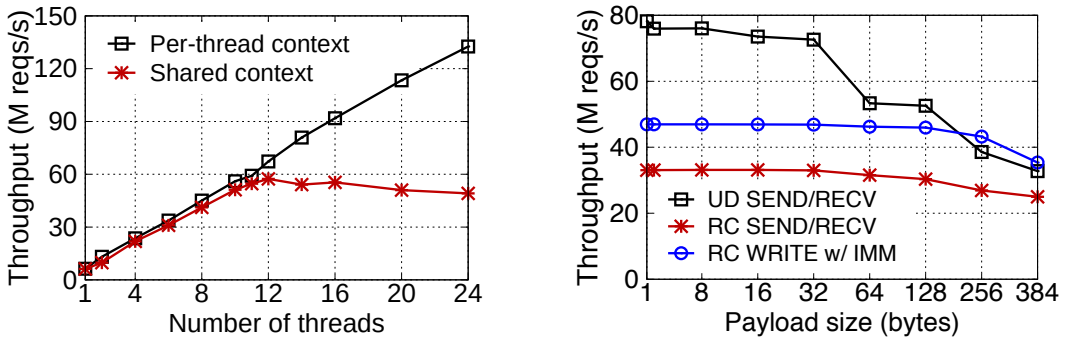


Figure 2–2 (a) The performance of RDMA `WRITE` using different QP creation strategies. (b) A comparison of different RDMA-enabled RPC implementations.

a dedicated context to create QPs for each thread; otherwise, there will be false synchronizations within the driver even each thread uses its own QP. The performance impact is shown in Figure2–2(a). The root cause is that each QP uses a pre-mapped buffer to send MMIOs to post requests while the buffer may be shared. The buffer is allocated from a context according to Mellanox's driver implementation, where each context has limited buffers. For example, the mlx4 driver [62] uses 7 dedicated buffers and 1 shared buffer. This means that if the context is used to create more than 8 QPs, then extra QPs have to share the same buffer. Even if each thread uses one exclusive QP, the throughput of a shared context drops by up to 63% with the increase of threads. The overhead comes from synchronizations on the shared MMIO buffer.

One-sided primitive. Each thread manages n RC QPs to connect to n machines. We use standard Verbs API to post a one-sided request to the QP corresponding to the machine. RDMA `WRITE` requests with payloads less than 64 bytes are inlined to improve throughput [30]. Note that we do not simply wait until the completion of the operation (§2.2.3): we execute other application requests or RPC functions for better utilizing CPU and network bandwidth.

Two-sided primitive. Unlike one-sided primitive which has a simple and straightforward implementation, there are many proposed RPC implementations (two-sided primitive) atop of RDMA [2, 25, 30, 51, 63-64]. They can be categorized into `SEND/RECV` verb based [25], RDMA `WRITE` based [2, 51, 63-64] and hybrid one [30].

We use `SEND/RECV` verbs over UD QP as our two-sided implementation in `R2` for

three reasons. First, in a symmetric setting, SEND/RECV verbs over UD has better performance than other implementations over RDMA, especially for transaction systems [25]. This is also confirmed in our experiment (see Figure 2–2(b)). Second, based on our studies of one-sided RDMA performance, *one-sided RDMA based RPC is unlikely to outperform UD based RPC* especially for small messages. The peak throughput of one-sided WRITE reaches 130M reqs/s when the payload size is smaller than or equal to 64 bytes (Figure 2–4). For an RPC communication, two RDMA WRITEs are required (one for send and one for reply). Thus, the peak throughput of RPC implemented by one-sided RDMA operations is about 65M reqs/s, lower than that of the implementation based on SEND/RECV over UD (79M reqs/s).

Discussions. SEND/RECV over UD does not provide a reliable connection channel. Therefore, it may be unfair to compare it to RC based two-sided implementations which have reliability guarantees. However, since RDMA network assumes a lossless link layer, UD has much higher reliability than expected [25]. Further, packet losses can be efficiently handled by transaction's protocol [25] or RPC layer [65].

2.2.3 Optimizations review and passive ACK

Since RDMA is a well-explored recent years, many optimizations have been proposed in prior work to better leverage it [2, 30, 63, 66]. R2 has integrated most of these optimizations. We further propose a new optimization, *Passive ACK*, which improves RDMA primitives when the completion acknowledgement (ACK) of the request is not on the critical path of the application.

Coroutine (CO). Although the latency of RDMA operations reaches several microseconds, it is still higher than the execution time of many applications [25]. Thus, it is worth to use coroutines to further hide the network latency by sending multiple requests from different transactions in a pipelined fashion. FaSST [25] uses coroutine to improve the throughput of its RPC. FaRM [2, 13] optimizes both one-sided operations and RPCs using an event loop to schedule transactions with RDMA operations. We use a set of coroutines to execute application logic at each thread. Each coroutine yields after issuing some network requests (including both one-sided and two-sided ones), and they resume the execution until they receive the completions of one-sided requests (or the replies of

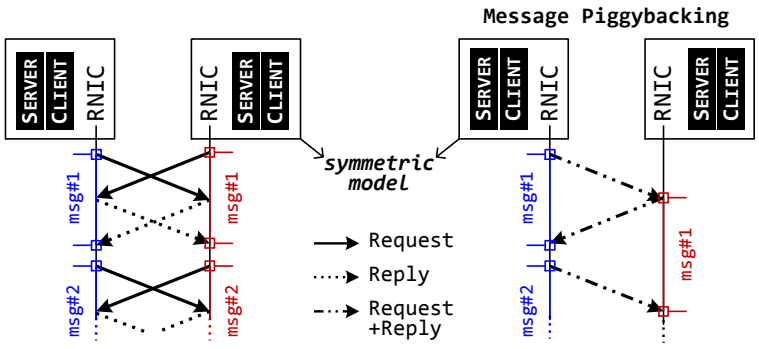


Figure 2–3 A sample of passive ACK for two-sided primitive.

two-sided RPCs). Typically, a small number of coroutines is sufficient for RDMA latency hiding (e.g., 8) [25].

R2 follows FaSST [25] by using coroutine from Boost C++ library to manage context switches between clients when issuing network requests. Boost coroutine is efficient in our experiments, which has very low overhead for context switch (about 20 ns).

Outstanding requests (OR). Even coroutine overlaps computation with I/O from different transactions, it is still important to send requests from one transaction in parallel. This further increases the utilization of RNICs and reduces the end-to-end latency of transactions, i.e., there is no need to wait for the completion of one request before issuing another one. For example, the read/write set of many OLTP transactions can be known in advance [67]. Therefore, it is possible to issue these reads and writes in parallel.

Doorbell batching (DB). There are several ways to issue multiple outstanding requests to RNIC. A common approach is to post several MMIOs corresponding to different requests. On the other hand, doorbell batching rings a doorbell to notify RNIC to fetch multiple requests by itself using DMA [66]. MMIO is costly which usually requires hundreds of cycles. Therefore, doorbell batching can reduce CPU overhead on the sender side and make a better usage of PCIe bandwidth, since it only requires one MMIO per batch to ring the doorbell.

One restriction of doorbell batching is that only requests from one QP can be fetched by the RNIC in a batched way. This means that different one-sided requests cannot be batched together if they are not sent to the same machine. Due to this limitation, doorbell batching is usually applied to two-sided implementation based on UD QP [25].

Passive ACK (PA). The performance can be further improved if the completion of requests (ACK) is done off the critical path of transactional execution. We achieve this by acknowledging the request passively.

For one-sided primitive, the request is marked as unsignaled, and then the completion of the request is confirmed passively after a successful polling of one subsequent signaled request. This avoids consuming RNIC's bandwidth.① For two-sided primitive, the optimization has the potential to double the throughput in a symmetric model by piggybacking the reply messages with the request messages. As shown in Figure 2–3, passive ACK can save half of the messages (replies).

It should be noted that not all of the completions can be acknowledged passively. For example, one-sided `READ` requires a completion event; otherwise, the application does not know whether the read is successful or not. Fortunately, as we will see in the later chapter (§4.1), in transactional execution, a transaction is considered to be committed when the log has been successfully written to all backups (see Figure 4–1). Hence the write-back request at the commit phase can be acknowledged passively.

2.2.4 A primitive-level performance analysis

In this section, we present the basic performance of different RDMA primitives, on `R2` including raw RDMA performance and the performance of micro-benchmarks. The micro-benchmarks simulate common transactional workloads. These experimental results serve as the guideline for using the appropriate primitives for transactions.

Experiment setup. We conduct the experiments on the **VAL** cluster. We run 24 worker threads (same as the number of available cores per machine) on each machine in our experiments. Each worker thread runs an event loop to execute transactions, handles RPC requests, and polls RDMA events. The events of RDMA including the completion of one-sided RDMA requests and the reception of RPC requests/replies.

RDMA raw performance. Prior work has shown that two-sided primitives have better performance and scalability than one-sided ones [25]. They draw this conclusion from from an old generation of RNIC (ConnectX-3). Further, they only show the poor scalability of one-sided primitive using small payloads (less than 32 bytes). We extend their

① Verbs from the same send queue are processed in a FIFO manner [68].

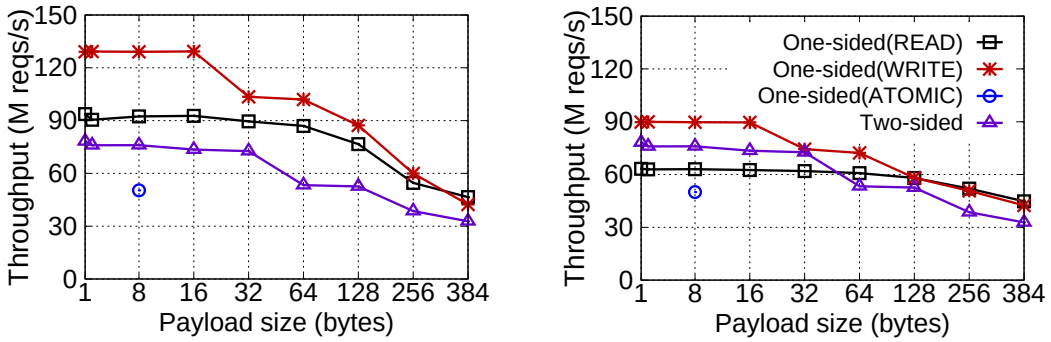


Figure 2–4 A comparison of one-sided and two-sided primitives using (a) a 16-node cluster and (b) an emulated 80-node connection setting. RDMA `ATOMIC` only supports the 8-byte payload.

evaluation [25] on raw RDMA performance to show that: one-sided primitives have better performance than two-sided ones using 16 nodes, as shown in Figure 2–4(a). More importantly, the scale of the cluster only affects *one-sided primitives with small payloads*. For example, with our emulated 80-node connection setting, one-sided primitives still outperform two-sided ones when data payloads are larger than 64 bytes.

Emulating massive RDMA connections. On our 16-node cluster, we create 5 RC QPs to connect to each machine at each worker. The number of QPs (5x16 QPs per thread) is sufficient to run in an 80-node cluster. We choose the QPs randomly to post upon issuing a request. Note that the total number of QPs (960 per NIC) has exceeded the total number of QPs that can be cached at RNIC.

Primitive evaluation. Figure 2–4(a) presents the evaluation results of the primitive analysis. For read operations, one-sided primitives (`READ`) outperform two-sided ones by up to 1.6X when payload size is below 64 bytes, and by up to 1.37X for larger payloads. For write operations, one-sided primitives (`WRITE`) outperform reads on small payloads but get a similar trend on large payloads (from 1.03X to 1.35X). Note that we do not incur memory copy overhead for two-sided primitives, as done in prior work [25], since adding such overhead will affect the performance of two-sided ones, especially for large messages.

Figure 2–4(b) further presents the results on an emulated 80-node connection setting. The performance of one-sided `READ` becomes slow-growing with the decrease of payloads from 128 bytes. This is because RNIC experiences QP cache misses at this time.① However, one-sided `READ`s can still outperform two-sided primitives when pay-

① We use PCIe counters (pmu-tools (https://github.com/andikleen/pmu-tools)) to measure QP cache misses.

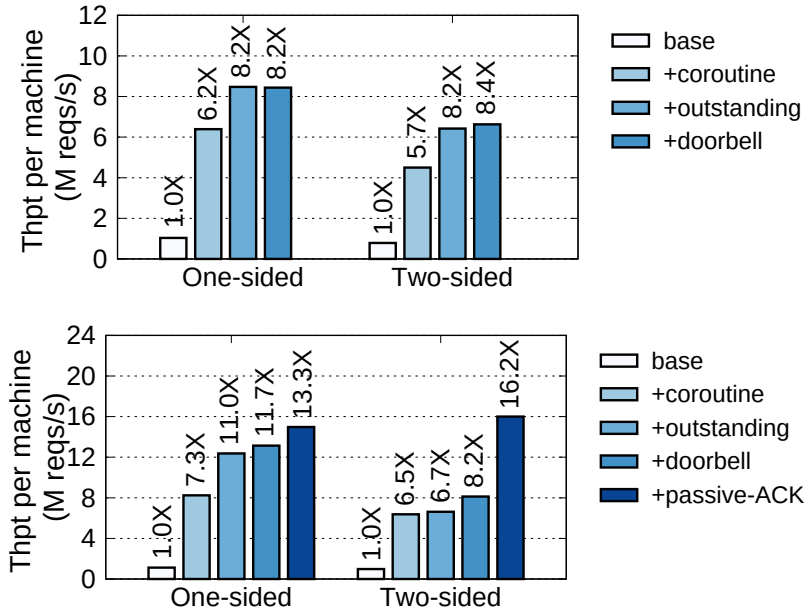


Figure 2–5 A comparison of one-sided and two-sided primitives for multiple-object (a) reads and (b) writes with 64-byte payloads.

loads are larger than 64 bytes. Because the cost of data transfer instead of QP cache misses dominates the performance for larger payloads.

A final takeaway is that, although one-sided `ATOMIC` is relatively slow [66], it can still achieve 48M reqs/s on each machine, which is much higher than the requirements of many workloads (e.g., `TPC-C`). Therefore, the performance will not be the main obstacle to leverage one-sided atomic primitives in transactional execution (e.g., distributed spinlock). We evaluate this approach in the transactional workload (§4.3.2).

Micro-benchmarks. Better performance in raw throughput does not always mean better performance in real applications. We use two micro-benchmarks to compare how different primitive performs under common transactional workloads, and how previous optimizations affect the performance (Figure 2–5).

Effects of optimizations. We first show how existing optimizations improve the performance of each primitive from Figure 2–5. Coroutine, outstanding requests and doorbell can be applied to both workloads.

Coroutine hides the latency and improves the performance of one-sided and two-sided by 7X and 6.46X, respectively. Adding outstanding requests by posting more requests per batch further improve the throughput due to better uses of RNIC's processing

capability.

Doorbell batching does not always improve the performance of one-sided primitive, but it constantly improves the throughput of two-sided ones. This is because doorbell batching can only apply to a single QP, which is suitable for UD-based two-sided implementation. On the contrary, one-sided requests are sent through multiple RC QPs, which reduces the chances of using doorbell batching. Further using doorbell batching requires bookkeeping the status of posted requests, which adds additional overhead.

2.2.5 Summary: offloading when completion is required

By enabling passive acknowledgement (PA), the performance of one-sided `WRITE` is further improved by 1.13X, while that of two-sided primitive is nearly doubled (1.96X) due to the reduction of half of the messages (for reply). This makes the only case where two-sided outperforms one-sided. Otherwise, one-sided primitive always has better performance than two-sided ones. This is consistent with the results in Figure 2–4. For example, multiple `READ`s can achieve peak throughput about 8.43M, which is close to the raw performance of one-sided `READ` (about 86.9M per machine).

2.2.6 Related work on RDMA-aware optimizations

FaRM [2] proposes a set of techniques to mitigate cache pressure of RNIC, including using huge page to reduce page entries stored in RNIC and sharing QPs between threads to reduce the connections. HERD [30] first discovers the benefits of using UD QPs for messaging to improve performance and scalability. A recent guideline paper of RDMA [66] describes several optimizations on better leveraging RDMA features, including using doorbell mechanism to post a batch of requests. It also studies how low-level factors (e.g., payload inlining) impact the overall performance. FaSST [25] argues that UD, though unreliable as its name, has high reliability in modern datacenters because RDMA assumes a lossless link layer. Hence, UD QP is well suited for two-sided primitives. Finally, LITE [63] proposes a kernel indirection layer for RDMA which improves the scalability and programmability of RDMA. Many of such optimizations can be used cumulatively to improve performance. We apply all of them in `R2` except for LITE. LITE requires modifying the kernel and it is also not designed for our scenario.

2.3 RDPMA: fast remote persistent memory with RDMA and NVM

Remote persistent memory is a key building block in distributed systems, which provides reading/writing over persistent memory through the network. This section presents the design of RDPMA, an efficient library to provide remote persistent memory using RDMA and NVM.

2.3.1 Introduction

People have been studying remote persistent memory with emerging hardware technologies like Remote Direct Memory Access (RDMA) and Non-Volatile Memory (NVM) for many years, including but not limited to databases [13, 18, 21, 25, 57-58], file systems [51-52, 69], key-value stores [2, 70], and distributed shared memory systems [53-54, 56]. These RDMA-NVM systems can either leverage NVM to persistently store the data or use it as DRAM to extend DRAM capacity.

Unfortunately, few systematic studies examined how to best leverage NVM with RDMA, since production NVM is only publicly available via Intel Optane DC persistent memory [71] (Optane PM①) until recently. Except for a few systems [56, 72], prior work either uses emulated NVM [52-53] or simply treats DRAM as NVM [13, 18, 51, 58]. Without such a study, system developers are unclear whether existing RDMA-NVM designs are efficient for Optane PM due to the following three reasons. First, emulating NVM with RDMA is particularly challenging because, to the best of our knowledge, most NVM emulators use CPU for the emulation [73]. However, RDMA may access NVM in a CPU-bypassing manner. Second, a recent study revealed that even the emulator could not faithfully simulate many Optane PM features [74]. Finally, a few systems evaluated with Optane PM do not consider its unique performance characteristics [56].

Our initial experiments further demonstrate that RDMA-NVM systems suffer from inferior performance when they treat NVM as DRAM. For example, the performance of *remote write* over remote persistent memory is far from the limits of NVM (§2.3.3) after switching the memory used in a *remote write* benchmark from DRAM to NVM: 16B one-sided RDMA WRITE only achieves 29% of NVM's ideal write throughput. Hence, it is imperative to conduct a thorough study of the inferior performance and provide optimizations to mitigate the inefficiency.

① Since we exclusively study Optane PM in this dissertation, we use the terms NVM and Optane PM interchangeably throughout the presentation.

There have been valuable studies on how to efficiently use NVM [74] with CPU and how CPU cache may affect the efficiency of RDMA with NVM [72]. Yang *et al*. [74] provide optimizations for the CPU to best utilize Optane PM. We study their findings on RDMA-NVM systems and confirm the importance of their optimizations. Nevertheless, we found some of the optimizations are suboptimal when considering RDMA. We further present optimizations that are best suited for RDMA on NVM. Kalia *et al*. [72] is the most relevant work: we share the same goal of improving RDMA's performance with NVM. Specifically, they identify how CPU cache could hinder RDMA from fully utilizing NVM write bandwidth. We made a similar observation during our study. Besides, we also study other RDMA-NVM related factors, including inappropriate system configurations (§2.3.4.1) and application access patterns (§2.3.4.2).

Finally, existing systems use two network roundtrips to implement persistent write atop RDMA and NVM [75] because existing RDMA hardware is unaware of NVM. We argue that RDMA-NVM systems should consider broadly explored RDMA-aware optimizations [30, 66] to improve the persistent write performance with RDMA. With the help of known RDMA-aware optimizations, RDMA only needs one roundtrip for remote persistent write on the current hardware platforms (§2.3.4.3).

In this chapter, we conduct a thorough systematic study on how to best utilize NVM with RDMA to build remote persistent memory. Our focus is on *remote write*, i.e., the client issues write to the server NVM, either using one-sided or two-sided RDMA. The remote NVM read performance is close to that of DRAM (§2.3.3). Specifically, we made the following three contribution:

A summary of optimization hints (H1–H9) to best utilize NVM with RDMA (§2.3.4). We study and collect various RDMA-NVM related optimizations—scattered from different sources—into one systematic study. We also propose new optimizations (**H6–H8**) that address the limitations of the existing study. The summarized hints are categorized into system configuration advice (§2.3.4.1), access pattern advice (§2.3.4.2) and RDMA-aware advice (§2.3.4.3). We empirically demonstrate how these hints help to fully utilize NVM for different RDMA primitives, i.e., RDMA can attain close to NVM write bandwidth and processing power.

`RDPMA`: a well-tuned remote persistent memory library integrated with H1–H9 (§2.3.5). We use the summarized hints to build `RDPMA`, and use it to analyze and im-

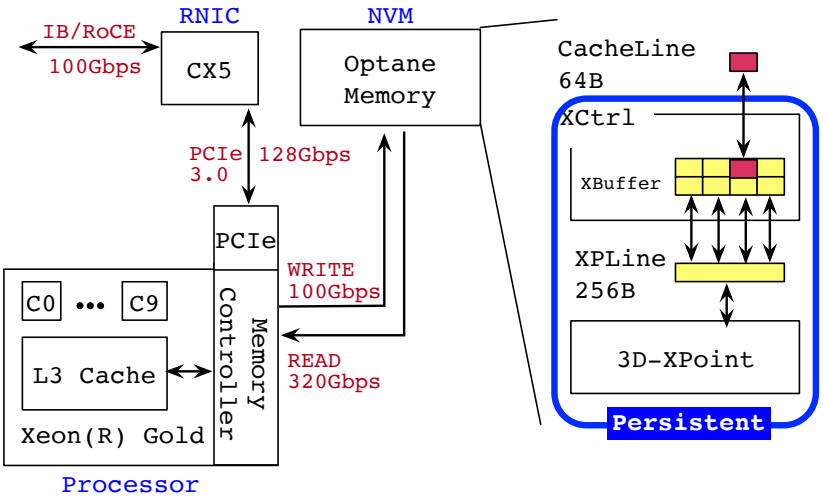


Figure 2–6 Hardware components of a node with NVM in an RDMA-capable cluster.

prove the design of an representive RDMA-NVM systems, Octopus [51]. Octopus is a distributed file system designed for RDMA and NVM. We find that there is still significant room for improvement in it because it is designed when no production is available. `RDPMA` helps to improve the I/O throughput of file data operations in Octopus by up to 2.4X (from 1.2X). The results strongly suggest we need further revisiting the design and implementations of existing RDMA-NVM systems, especially those not designed for Optane PM.

In later chapter (§6.3), we will also present how to use `RDPMA` to optimize the performance of durable distributed transactions.

2.3.2 Background on NVM

Figure 2–6 presents typical hardware components of a node with NVM in an RDMA-capable cluster. RDMA-capable NIC (RNIC) and NVM are attached to the processor, and they communicate with each other via the PCI Express (PCIe) bus.

2.3.2.1 Optane PM (NVM)

Intel Optane DC persistent memory [71] (Optane PM) is the first commercially available NVM. Besides a huge performance gain compared to traditional persistent storage (e.g., SSD), Optane PM also provides a DRAM-like memory interface. Thus, CPU can use `load` and `store`/`non-temporal store`① (`ntstore`) to read and write it, and

① `non-temporal store` has the same semantic as `store` except that it bypasses the CPU cache.

RNIC can access it through PCIe read/write transactions.

Our study relies on an in-depth look at how Optane PM handles reads/writes. The right half of Figure 2–6 presents an overview of its components. Data is stored in NVM DIMMs (3D XPoint), while XController (XCtrl) transforms the read/write requests from the processor/PCIe into read/write requests to 3D XPoint. XCtrl has two important features. First, it receives requests in cacheline (CLine) granularity (64B), while 3D XPoint stores data in XPLine granularity (256B). Such a difference in granularity may incur read-/write amplification. Second, in order to reduce write amplification, XCtrl has a small write-combining buffer (XBuffer) that merges adjacent cacheline writes into one XPLine write. Note that read requests also compete for XBuffer with write requests [74].

Persistent domain. Data is persistent once it reaches the node's persistent domain. On the current hardware platform, the persistent domain comprises the Optane PM and the processor's memory controller, as shown in Figure 2–6. Future hardware will further extend the persistent domain to the processor cache [76]. Nevertheless, the scope of the persistent domain is orthogonal to the results of this chapter. We describe its impact in §2.4 in more detail.

Optane PM counters. Optane PM provides various useful counters① that we use to analyze its behavior: `NReadReq` and `NWriteReq` record how many 64B read and write requests are received by XCtrl, while `NMediaRead` and `NMediaWrite` record how many bytes are read/written by 3D-XPoint Media. Based on these counters, we calculate the counter rates and use the counter rates to compute the read/write amplification of NVM: e.g., `NMediaWrite` *rate* / (`NWriteReq` *rate* ×64) measures the write amplification of Optane PM.

2.3.2.2 RDMA with NVM

Since NVM have the same interface as DRAM, RDMA can read and write it like DRAM (as shown in Figure 2–1). One-sided RDMA primitives communicate with Optane PM through PCIe read/write transactions, while two-sided RDMA uses server CPU to read/write Optane PM②. When different RDMA primitives access NVM, several fac-

① Measured via `ipmctl` [77].

② Although two-sided RDMA can use PCIe read/write transactions to write messages to Optane PM, we omit the discussion of such a case because its mechanism is the same as one-sided RDMA WRITE.

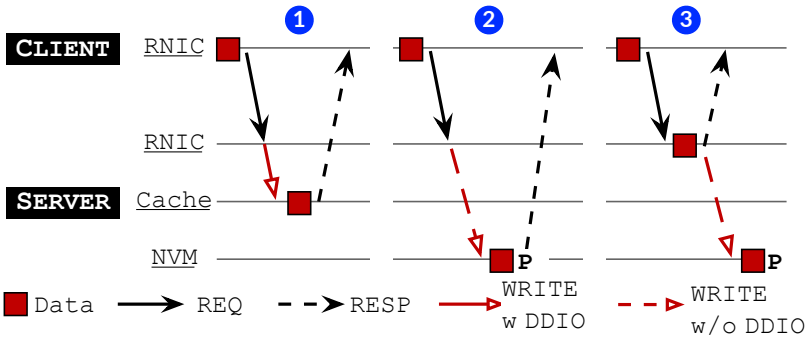


Figure 2–7 Three execution flows of using one-sided RDMA to write the server NVM. Note that without proper configurations, all three request flows are possible. The client uses the MMIO to post the WRITE request (**REQ**), and the client RNIC generates the response (**RESP**) via DMA. When DDIO (§2.3.2.2) is enabled, RNIC writes to the server's last level cache (LLC). Otherwise, RNIC directly writes to Optane PM. **P** denotes the persistent point of the data, i.e., when the client can ensure the WRITE is persistent.

tors may impact their efficacy:

Access granularity. RNIC, CPU and NVM (including XController and 3D XPoint) have different access granularities. Requests that do not match the device granularity cause extra read/write requests to the NVM. Hence, systems should carefully tune their NVM access patterns for different RDMA primitives (§2.3.4.2). Table 2–3 summarizes the access granularities of different devices.

Table 2–3 Access granularities of different hardware components.

	CPU	PCIe	XController	3D XPoint
Granularity	CLine	CLine	CLine	XPLine
Payload	64B	64B	64B	256B

DDIO. Data Direct I/O (DDIO) [78] aims to improve the server cache locality of the DMA-ed data, which allows the last level cache (i.e., L3 Cache) as the primary destination of the RNIC's DMA-ed data (e.g., one-sided RDMA WRITE and two-sided RDMA). However, it is not friendly to Optane PM (§2.3.4.1).

Persistence. Two-sided primitives can use extended CPU instructions (e.g., `clwb`) to ensure that the write to NVM is persistent. However, one-sided RDMA has no such instruction. Thus, one-sided RDMA-NVM WRITE is not persistent.

Figure 2–7 depicts three possible execution flows of one-sided RDMA-NVM WRITE on the current hardware platforms, only in the second case that the data is persistent (❷). In the first case (❶), the data is not persistent because it still resides in the volatile processor cache when the client receives the response. In the third case (❸), the data is cached at the RNIC's internal buffer when the client believes the write has finished. RNIC is not in the persistent domain (see Figure 2–6).

Implementing persistent one-sided RDMA WRITE over current hardware (i.e., guarantee to achieve ❷ in Figure 2–7) requires specific configurations and extra one-sided requests: we should first disable DDIO to bypass the processor cache and then send an extra one-sided RDMA READ to the same QP issued the WRITE [79] to flush the previously cached WRITEs. These two steps guarantee the WRITE is executed as the second case in Figure 2–7. However, a strawman implementation of this strategy uses two network roundtrips for a single write [75]. We describe optimizations for persistent WRITE in §2.3.4.3.

2.3.3 Methodology

This section describes the optimization target of `RDPMA`. `RDPMA` focuses on *remote write*. i.e., the client issues write requests to the server NVM using either one-sided or two-sided RDMA. For *remote read*, we find it has close performance to that of DRAM due to the asymmetric read/write performance feature (§2.3.2.1) of NVM.

To empirically analyze the read/write features of RDMA with NVM, we conduct a microbenchmark to evaluate the performance of *remote read* and *remote write* implemented by different RDMA primitives on **R74V** (§2.1). In this benchmark, each client sends read/write requests with different payloads to the server's NVM via RDMA, similar to prior work [2, 30, 58, 80]. The request addresses are chosen randomly. For one-sided RDMA, the client directly uses its primitives to implement *remote read* and *remote write*. For two-sided RDMA, the client sends messages to the server, and the server reads/writes NVM with `memcpy` after receiving the messages. We implement the benchmark on `R2`. Unless otherwise mentioned, we report the per-socket peak throughput or bandwidth of reading/writing NVM through RDMA.

Figure 2–8 and Figure 2–9 present the performance of *remote read* on DRAM and NVM for one-sided and two-sided RDMA, respectively. For large payloads (e.g., 2,048B), the read performance of NVM is close to that of DRAM for both one-

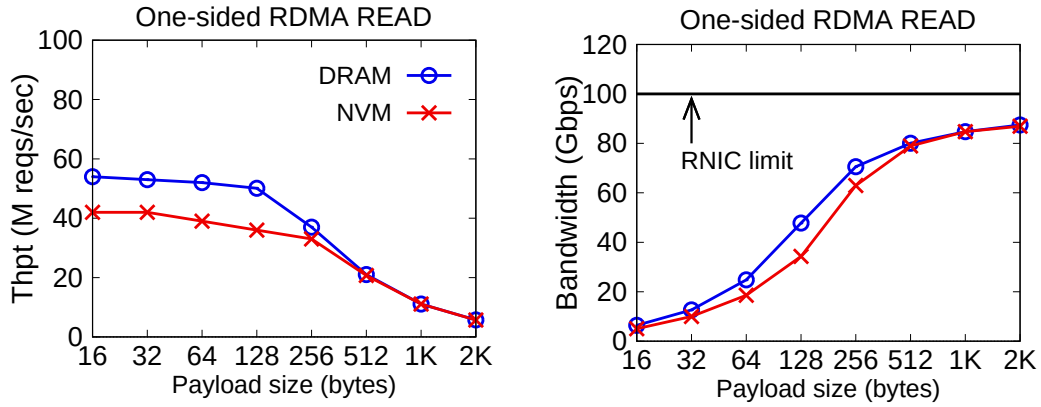


Figure 2–8 A comparison of one-sided RDMA READ performance on DRAM and NVM, (a) throughput and (b) aggregated bandwidth.

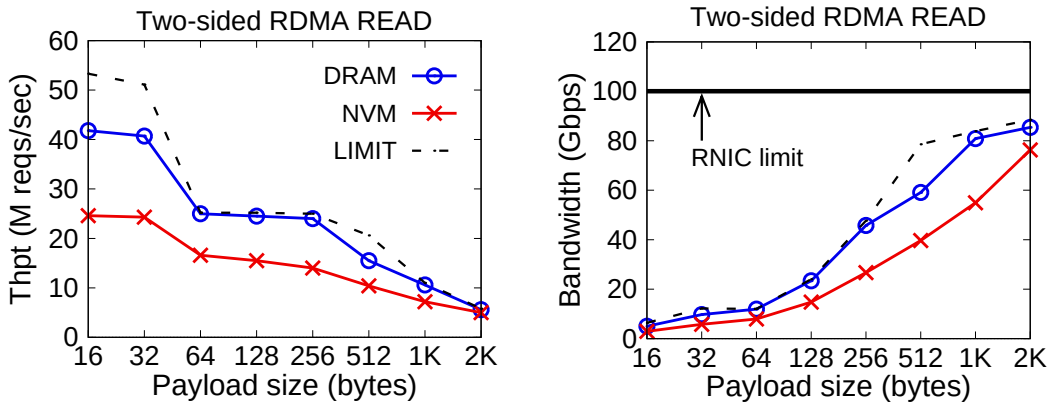


Figure 2–9 A comparison of two-sided RDMA READ performance on DRAM and NVM, (a) throughput and (b) aggregated bandwidth. LIMIT is measured when the server directly returns to the client without reading the payload.

sided and two-sided primitives (99% and 90%). Reading NVM can hardly become a bottleneck in RDMA-NVM systems since NVM has a much higher read bandwidth than RNIC (320Gbps vs. 100Gbps). Note that for small reads (e.g., 16B), two-sided RDMA READ still suffers from obvious throughput degradation: it only achieves 59% of the DRAM read throughput. This is because CPU has much a higher read latency when reading NVM compared to DRAM (271ns vs. 82ns)①. In contrast, increased NVM read latency has negligible impact on one-sided RDMA READ since the PCIe latency is the dominant factor (~1000ns [82]).

① Measured by Intel Memory Latency Checker (MLC) [81].

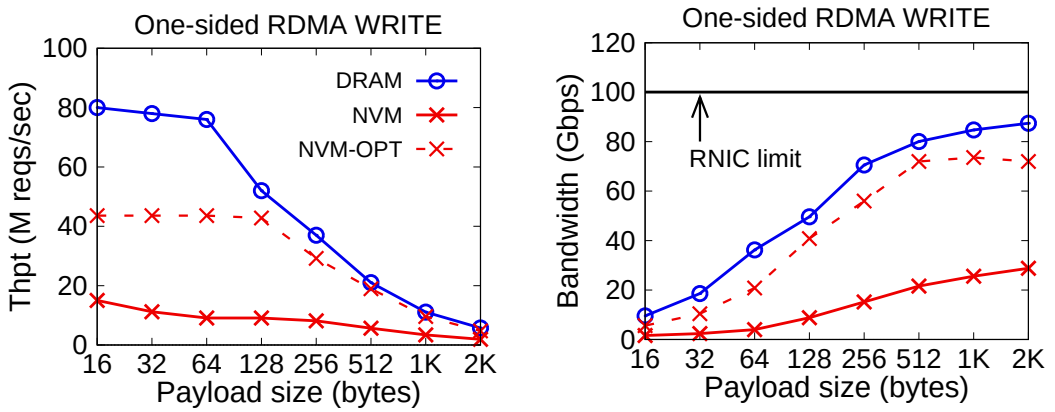


Figure 2–10 A comparison of one-sided RDMA WRITE performance on DRAM and NVM, (a) throughput and (b) aggregated bandwidth. NVM-OPT applies the optimizations from §2.3.4.

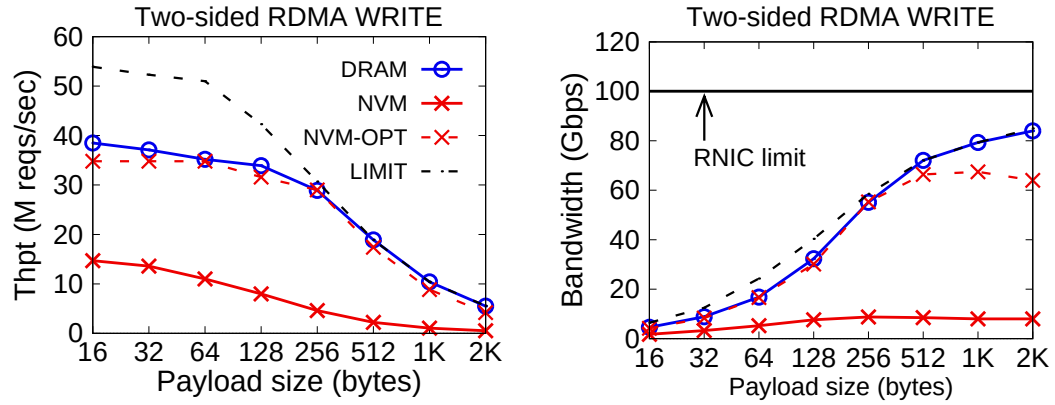


Figure 2–11 A comparison of two-sided RDMA WRITE performance on DRAM and NVM, (a) throughput and (b) aggregated bandwidth. LIMIT is measured when the server directly returns to the client without writing the payload. NVM-OPT applies the optimizations from §2.3.4.

Unlike *remote read*, the performance of *remote write* is much slower than that of DRAM, as shown in Figure 2–10 and Figure 2–11. More importantly, these results are not optimal because the measured performance is far from the theoretical limit of NVM or RDMA. For example, one-sided RDMA WRITE can achieve only 29% of the NVM peak write throughput (15M vs. 52M reqs/sec①) for small 16B writes. Further, one-sided and two-sided RDMA saturate only 38% and 12.5% of the NVM's peak write bandwidth for large 2,048B writes, respectively. Therefore, there is significant room for improvement.

① We estimate the NVM peak write throughput by dividing its peak write bandwidth (100Gbps) with the size of XPLine (256B).

Table 2–4 A summary of design advice, optimization hints, and whether the hints can apply to a specific RDMA primitive. ✓ indicates a positive optimization effect, and "–" means the hint does not target the case. **DB** and **OR** states for optimizations doorbell batching and outstanding request discussed in §2.2.3, respectively.

Optimization hints	One-sided	Two-sided
H1. Avoid cross-socket NVM accesses	✓	✓
H2. Limit concurrent access to a single NVM DIMM for two-sided	-	✓
H3. Disable DDIO; if DDIO must be enabled, use two-sided RDMA	✓	-
H4. Use `ntstore` instead of `store` for large writes	-	✓
H5. Use XPLine granularity (256B) for writes	✓	✓
H6. Use PCIe DW granularity (64B) for small writes (i.e., < XPLine)	✓	-
H7. Use cacheline granularity (64B) with `ntstore` for small writes	-	✓
H8. Use less atomic operations on NVM	✓	✓
H9. Use OR with DB for persistent WRITE	✓	-

The approach of `RDPMA`. To understand why *remote write* has inferior performance, we conduct a systematic study to summarize various performance-relevant factors for RDMA to write NVM. Inspired by a recent CPU-specific NVM study [74], we mainly follow two directions. First, we investigate what system configurations can affect RDMA's efficiency with NVM (§2.3.4.1). Second, we study which access patterns from RDMA are friendly to NVM (§2.3.4.2). For each direction, we empirically study whether known NVM optimizations are necessary or optimal for RDMA. The results show that some setups are not necessary, while some optimizations are sub-optimal for RDMA. To this end, we present new optimizations by fully considering NVM characteristics with RDMA. Finally, we use existing RDMA optimizations to improve the persistent write of one-sided RDMA atop of NVM (§2.3.4.3).

A preview of `RDPMA`. Figure 2–10 and Figure 2–11 present the optimized version of one-sided and two-sided RDMA NVM write with `RDPMA` (*NVM-opt*). After applying all the optimizations, they have significantly better performance and achieve close to the NVM limit. For example, one-sided 16B RDMA NVM WRITE achieves 45M reqs/sec, 87% of the ideal peak throughput of NVM write.

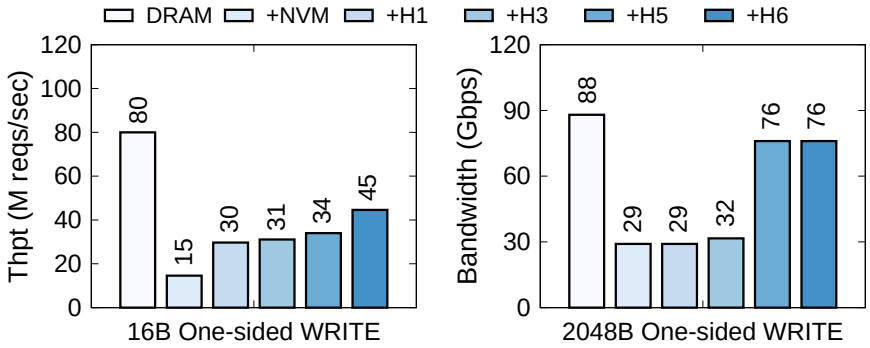


Figure 2–12 Factor analysis of the optimizations used to improve NVM write performance atop of one-sided RDMA WRITE under (a) small payload (16B) and (b) large payload (2,048B). We do not include **H8** as it does not target remote write. Note that the error bars are small.

2.3.4 Design advice for RDMA-NVM systems

This section summarizes design advice and optimization hints for high-performance RDMA-NVM systems, as shown in Table 2–4. We present both new optimizations (e.g., Hint 6, **H6**) and brief descriptions of known optimizations. Further, we show that some known optimizations that only consider CPU accessing NVM are sub-optimal for RDMA (e.g., **H5**). Among these optimizations, **H1–H8** applies to systems that use Optane PM as volatile storage and persistent storage, while **H9** only targets systems that use Optane PM as persistent storage.

We make two assumptions about the hardware components. First, the RNIC is a PCIe-based device, which holds for most existing RNICs [66]. Second, the NVM is Optane PM [71], the first (and only) commercially available NVM device. Unless otherwise stated, we use the same *remote write* microbenchmark in §2.3.3 for the study.

2.3.4.1 Configuration advice

A prior CPU-specific NVM study [74] has provided valuable configuration setups (e.g., NUMA setup) for the CPU to better utilize NVM. We summarize these setups in **H1** and **H2**. A natural question to answer is: do RDMA-NVM systems require the same setups? Our study reveals that first, RDMA-NVM systems should also consider **H1**. Meanwhile, one-sided RDMA does not necessarily require **H2** as the CPU. Finally, RDMA introduces a new configuration option, **H3**. Succinctly, the configuration advice for RDMA-NVM systems is the following three optimization hints:

H1. Avoid cross-socket NVM accesses;

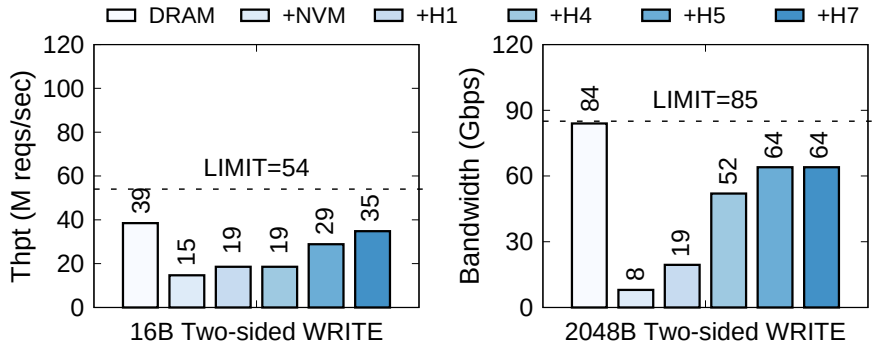


Figure 2–13 Factor analysis of the optimizations used to improve NVM write performance atop of two-sided RDMA for (a) small payload (16B) and (b) large payload (2,048B). LIMIT is measured as the server directly returns to the client without writing the payload. We do not include **H8** as it does not target remote write. Note that the error bars are small.

H2. Limit concurrent access to a single NVM DIMM for two-sided RDMA;

H3. Disable DDIO; If DDIO must be enabled, use two-sided RDMA for large NVM writes;

Hint H1. Yang *et al*. [74] found that the NVM write bandwidth of a socket could be halved from other sockets. Since an RNIC leverages its attached socket to access NVM from another socket (see Figure 2–6), slow cross-socket NVM accesses also impact RDMA-NVM systems. Figure 2–13 and Figure 2–12 illustrate this: removing cross-socket access improves the baseline two-sided and one-sided RDMA write performance by up to 2.4X (8Gbps vs. 19Gbps) and 2X (15M vs. 30M reqs/sec), respectively. Thus, RDMA-NVM systems should also avoid cross-socket NVM accesses.

Apply H1. Readers may wonder whether **H1** is feasible in real systems. One strategy to apply **H1** is first attaching one RNIC for each socket, and then treating each socket as a logical node in a cluster. Such a setup is common in RDMA-capable systems [13, 17, 83-85], and it naturally avoids cross-socket NVM access. Furthermore, even if there are insufficient RNICs for each socket to have a dedicated RNIC, one can adopt the techniques in IOctopus [86] to apply **H1**. Using IOctopus, one RNIC can simultaneously communicate with multiple sockets. Hence, RDMA can directly access the NVM bypassing the attached socket.

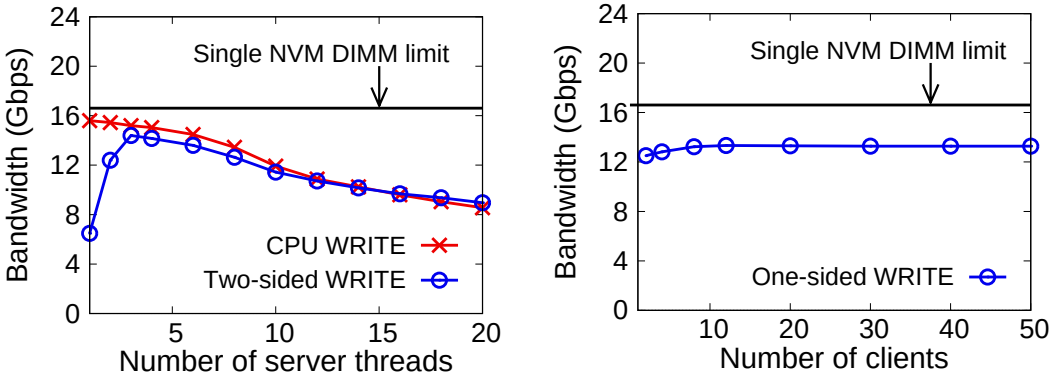


Figure 2–14 Effects of concurrent accesses to a single NVM DIMM for (a) two-sided RDMA and (b) one-sided RDMA WRITE using a 2048B payload. The write bandwidth limit of a single NVM DIMM is about 16Gbps (one-sixth of the evaluating Optane PM). Note that we have enabled all other optimizations for both RDMA primitives in this experiment.

Hint H2. Another observation from Yang *et al.* [74] is that the CPU fails to scale up when writing to a single NVM DIMM. This phenomenon also affects two-sided RDMA because it uses CPU to write to the NVM. As shown in Figure 2–14(a), compared to using four threads at the server to handle writes, two-sided RDMA-NVM write bandwidth drops 37% when using 20 threads. Note that to avoid interference from other factors, we have enabled all other optimizations hints in this experiment. Therefore, system designers should also reduce concurrent access to a single NVM DIMM for two-sided RDMA. In practice, designers can control which DIMM to access by selecting the appropriate NVM addresses [74]. For example, assuming the starting address of the NVM is 4KB-aligned, and the NVM is interleaved on 6 DIMMS, the first and seventh 4KB is on the first DIMM, and the second 4KB is on the second DIMM, etc.

Does one-sided RDMA suffer from the same issue? To quantify this, we further conduct an experiment to measure the concurrent write performance of one-sided RDMA-NVM WRITE. In this benchmark, we increase the number of clients that send concurrent one-sided RDMA WRITE to a single NVM DIMM, and measure their aggregated bandwidth. Figure 2–14(b) presents the results: one-sided RDMA WRITE scales well with the increased number of concurrent requests. This implies that one-sided RDMA is more robust when concurrently accessing a single NVM DIMM.

To the best of our knowledge, it remains unknown why the CPU cannot scale up when accessing a single NVM DIMM. Yang *et al.* [74] suspected that the high NVM

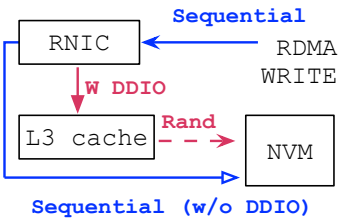


Figure 2–15 DDIO changes the sequential accesses of RNIC into random accesses.

access latency may cause head-of-line blocking effects at the processor. Similarly, we suspect that the RNIC can scale well for one-sided RDMA because the increased latency of NVM access is not that significant compared to PCIe latency.

Hint H3. One important RDMA-specific configuration is whether to enable DDIO, which controls the destination of one-sided RDMA WRITE (§2.3.2.2). A prior study has revealed that DDIO has a huge performance impact on one-sided RDMA-NVM WRITE for large payloads [72]; we also made a similar observation during the study. Consequently, **H3** is an important configuration setup for RDMA-NVM systems.

Figure 2–16(a) shows the effects of DDIO on one-sided RDMA-NVM WRITE. Since large RDMA-NVM WRITE is more sensitive to DDIO, we use a bulk write benchmark where a single client issues a sufficient large payload to measure the peak bandwidth of WRITE. With DDIO enabled, we observe that WRITE can only reach half of the NVM peak bandwidth (42Gbps vs. 100Gbps). On the other hand, the WRITE can achieve close to NVM peak bandwidth with DDIO disabled.

Ideally, the client should saturate the RNIC(NVM) bandwidth in this benchmark. However, it fails because DDIO changes the sequential writes from RNIC to random writes to NVM. Figure 2–15 illustrates this: the RNIC first sequentially writes the data into the cache, after that the cache randomly evicts the data to the NVM. Random writes cannot saturate NVM bandwidth because NVM has less chance to merge adjacent writes to avoid write amplification (see §2.3.2.1).

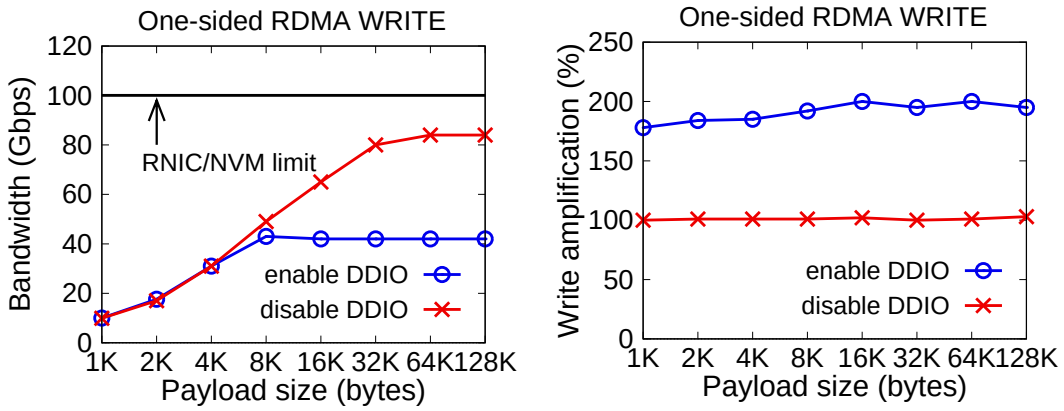


Figure 2–16 (a) Effects of DDIO to bulk one-sided RDMA-NVM WRITE, (b) write amplification analysis of one-sided RDMA-NVM WRITE. Note that we have enabled all other optimizations in this experiment.

Measuring the write amplification of DDIO. To quantify the effect of DDIO, Figure 2–16(b) analyzes the write amplification of NVM① with different configurations. We can see that enabling DDIO incurs a roughly 2X write amplification to NVM WRITE, which explains why the bandwidth is halved for a large payload (e.g., more than 64KB).

Implications for RDMA-NVM systems. If the systems must use one-sided RDMA WRITE to saturate the NVM bandwidth, we recommend considering **H3** to turn off the DDIO first. The system can statically enable/disable DDIO via the BIOS setups [52] or dynamically adjust the bits in Integrated I/O (IIO) Configuration Registers [87-89] at runtime. We follow prior work [89] to use configuration registers to configure DDIO at runtime.

Limitations of disabling DDIO. On the current hardware platform, the DDIO configuration affects all the devices on a processor. Thus, server CPU will have a poorer cache locality for DMA-ed data (e.g., messages in two-sided primitives) with DDIO disabled, resulting in degraded two-sided RDMA performance. For example, we measured a 57% peak throughput drop (54M vs. 23M reqs/sec) of two-sided RDMA primitives after disabling DDIO. Therefore, the current RDMA-NVM systems will make a trade-off

① We measure the write amplification of DDIO via NVM counters. §2.3.2.1 describes the measurements in more detail.

between the bandwidth of one-sided RDMA NVM WRITE and the performance of two-sided RDMA. Considering the limitations of disabling DDIO, we extend **H3** as follows:

H3 (extended). If DDIO must be enabled, use two-sided RDMA for large NVM writes;

Two-sided RDMA can leverage the CPU at the server to fully utilize NVM [74]. Hence, RDMA-NVM systems can adopt a hybrid approach for implementing NVM write based on the request payload size.

2.3.4.2 Access pattern advice

Tuning systems NVM access pattern according to the Optane PM's features is critical to NVM-aware systems. Known optimizations for CPU include choosing the appropriate CPU instructions and using the proper access granularity [74]. We summarize these hints in **H4–H5**:

H4. Use `ntstore` instead of `store` for large writes;

H5. Use XPLine granularity for writes;

Both hints can also benefit RDMA-NVM systems. However, we found **H5** is not optimal when considering RDMA, especially for small writes, because it incurs huge network amplification. For example, applying **H5** only improves the performance of 16B one-sided RDMA NVM WRITE by 1.1X (31M vs. 34M reqs/sec), which remains far from NVM's ideal processing rate (52M reqs/sec). In this case, **H5** will incur 16X (256B vs. 16B) network amplification to one-sided RDMA WRITE. Since moving 256B data to DRAM over RDMA is even slower than NVM ideal processing rate (37M vs. 52M reqs/sec, as shown in Figure 2–10), the network would first become the bottleneck for small writes.

To this end, we found the key for small writes to fully utilize NVM is to *avoid sending unnecessary read requests to the NVM*. As we have mentioned in §2.3.2.1, NVM read requests compete for **XCtrl** processing power with NVM write requests. Hence, unnecessary read requests would drastically reduce the NVM write throughput. This fact allows using a smaller access granularity to saturate NVM's processing rate with RDMA.

CPU and RNIC generate an extra read request to NVM if the payload of the write request does not fit their access granularities (i.e., cacheline and PCIe DW). They execute such a write request in a read-modify-write pattern to avoid overwriting the original

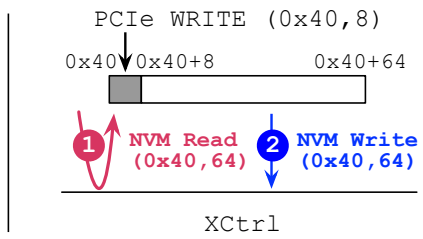


Figure 2–17 An example of PCIe partial write: PCIe sends two NVM requests when writing 8B at address 0x40.

content. Thus, using the CPU/RNIC access granularity is sufficient to prevent sending unnecessary reads from the device to NVM. Based on this observation, we first propose **H6–H7** to complement **H5** for small writes. Further, since the read-modify-write pattern is not friendly to NVM, we also propose **H8** to suggest systems use fewer such operations. Specifically, we propose the following hints to complement **H4–H5**:

- **H6**. For one-sided RDMA, use PCIe data word (64B) granularity when the payload is smaller than XPLine;
- **H7**. For two-sided RDMA, use cacheline granularity (64B) with `ntstore` when the payload is smaller than XPLine;
- **H8**. Use less atomic operations on NVM;

Hint H6. PCIe issues write in a read-modify-write pattern with *PCIe partial-write*. Figure 2–17 shows a concrete example where the RNIC uses PCIe to write 8B at 0x40. It will first send a read request to the NVM to read the data word at 0x40 (❶). Then, it overwrites the entire data word according to the write request (❷).

Figure 2–12(a) presents the optimized performance of **H6** to one-sided RDMA WRITE: it improves the performance of 16B WRITE to 87% of the NVM's peak write throughput (45M vs. 52M reqs/sec). The result is 1.3X faster than applying **H5** thanks to the reduced network amplification. Figure 2–18(a) further examines how eliminating PCIe partial write helps to prevent sending read requests to the NVM. We apply **H6** by first aligning the written address to PCIe DW (+1. Align to PCIe DW), and then padding the payload size (+2. Pad to PCIe DW) to a multiple of PCIe DW. As we can see, after applying both steps, one-sided RDMA WRITE does not issue a read request to the NVM. Consequently, small WRITEs (e.g., no larger than 64B) can reach close to the

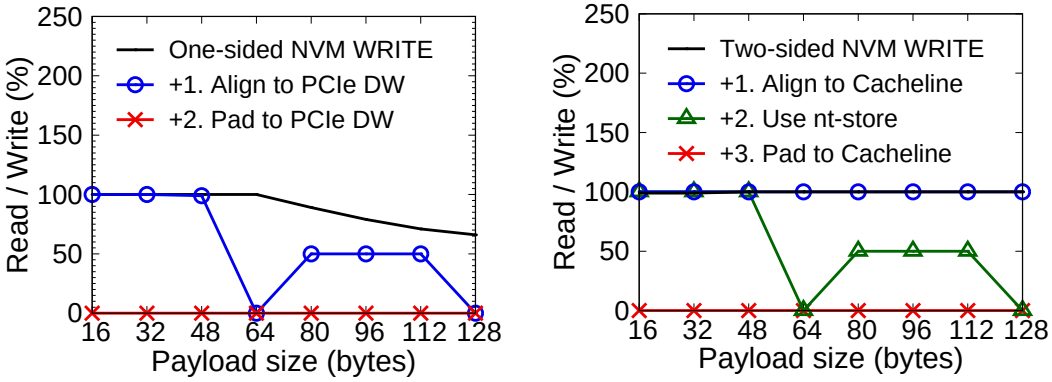


Figure 2–18 The ratio of extra NVM read per write of (a) one-sided RDMA WRITE and (b) two-sided RDMA on the remote write benchmark. Aligning and padding the write payload to device access granularity (e.g., PCIe DW) is sufficient to avoid unnecessary NVM reads for one-sided RDMA WRITE. On the other hand, two-sided RDMA further needs to use `ntstore`.

NVM processing limit.

We should mention that reducing the PCIe partial write may also benefit one-sided RDMA-DRAM WRITE. However, our experiments show that it may even have a negative effect on DRAM WRITE. For example, the 16B DRAM WRITE has a 25% performance degradation on our testbed after applying **H6**.

Hint H7. Similar with **H6**, **H7** suggests how to prevent read-modify-write for two-sided RDMA. As shown in Figure 2–12, it improves the 16B two-sided RDMA-NVM WRITE performance by 1.8X (19M vs. 35M reqs/sec). To apply **H7**, two-sided RDMA should use `ntstore` together with use cacheline granularity (+1. Align to Cacheline and +3. Pad to Cacheline, as shown in Figure 2–18(b)). This is because, the CPU would pre-fetch the cacheline using NVM read with `store`. As shown in Figure 2–18(b), two-sided RDMA-NVM WRITE always achieves a 100% read/write ratio even after using the proper access granularity.

Hint H8. A lesson learned from **H7–H8** is that the read-modify-write access pattern is not friendly to NVM. Atomic operations (e.g., one-sided RDMA ATOMIC compare and swap) naturally follow a read-modify-write pattern. Worse, the designer cannot apply prior hints to optimize atomic operations. For instance, one-sided RDMA ATOMICs cannot apply **H6** since they use a fixed 8B granularity. Consequently, we suggest using

fewer atomics on NVM for RDMA. Note that we are not suggesting disabling atomics, but *moving the data for atomic operations (e.g., spinlock) from NVM to DRAM whenever possible*.

Discussion of H6–H8. Although applying **H6** and **H7** may waste NVM storage for storing the padding, while adopting **H8** could change the persistent semantic of atomic data, we believe **H6–H8** is actionable in real systems. This is because there are many scenarios in RDMA-NVM systems that can use **H6-H8** without wasting storage or changing the application semantics. For instance, existing RDMA-NVM enabled databases [2, 13, 17, 25, 58] do not require the lock to be persistent. Thus, we can safely move their lock metadata from NVM to DRAM. Furthermore, distributed logging [2, 13] naturally uses padding to accommodate future logs. Thus, applying **H6** to logging does not introduce additional storage overhead. Finally, as we will present in §2.3.5, **H6–H8** can have huge performance improvements for existing systems.

2.3.4.3 RDMA-aware advice

We conclude our study of building efficient persistent remote memory with RDMA and NVM by discussing how known RDMA-aware optimizations can mitigate the inefficiency of implementing persistent write atop existing hardware platforms. As we have mentioned in the introduction, a strawman approach to implementing persistent write using one-sided RDMA requires two network roundtrips: the first WRITE attempts to store the data to NVM, while the second READ ensures that the written data is flushed to the persistent domain (e.g., Optane PM). Fortunately, it is possible to leverage well-known RDMA-aware optimizations to avoid the additional network roundtrip of READ. **H9** summarizes this fact:

H9. Enable outstanding request with doorbell batching for one-sided persistent RDMA WRITE.

Specifically, outstanding request [30] allows us using the completion of READ as the completion of the WRITE, as long as the two requests are sent to the same QP. Since the READ to persist the WRITE must be post to the same QP as the WRITE (§2.3.2.2), we no longer need to wait for the first WRITE to complete. Thus, this optimization reduces the wait time of the first network roundtrip. Applying outstanding request to persistent WRITE is correct because first, later READ flushes previously WRITE [79], and RNIC

processes requests from the same QP in a FIFO order [68].

Based on outstanding request, doorbell batching [66] further allows us to send the READ and WRITE in one request using the more CPU and bandwidth efficient DMA, reducing the latency of posting RDMA requests.

On our testbed, a single one-sided RDMA request takes 2μs. Thus, a strawman implementation of *remote persistent write* uses 4μs. After applying **H9**, one-sided *remote persistent write* takes 3μs latency to finish.

2.3.5 RDPMA and improved system design

In this section, we first present how we use the results of our study to implement RDPMA (§2.3.5.1), an efficient remote persistent memory library. Then, we use RDPMA to optimize existing RDMA-NVM systems (§2.3.5.2).

2.3.5.1 RDPMA

RDPMA provides simple yet straightforward remote persistent read/write interfaces. We build it on R2 to fully leverage its RDMA-aware optimizations (§2.2). RDPMA further transparently adopts most our summarized hints (§2.3.4) to better leverage NVM with RDMA. We don't incorporate all the hints since several hints depend on the application semantic (e.g., **H1**). The upper system (e.g., DrTM+X) can manually add these optimizations (§6.3).

Specifically, RDPMA provides the following interfaces:

Alloc. The system must use Alloc to allocate the memory for reading/writing. It returns a fat pointer encoding the memory addresses and payload. For safety reasons, RDPMA does not support reading/writing NVM with arbitrary pointers. This is because, RDPMA heavily applies aligning/padding to achieve better performance (e.g., **H6** in §2.3.4). If the user passes a pointer that points to an improperly aligned/padded memory, RDPMA would corrupt the memory content.

Read. Given a fat pointer allocated from Alloc, Read transfers its content to the caller. The caller can specific which RDMA primitive to use. Since reading NVM is efficient with RDMA, RDPMA does not apply any specific optimization to the basic implementation of different RDMA primitives (§2.3.3).

Write. It overwrites the original content pointed by the fat pointer returned from Al-

`loc`. Like `Read`, the caller can choose which RDMA primitive to use. For each RDMA primitive, `RDPMA` adopts the relative optimizations summarized in Table 2–4.

`PWrite`. Compared to `Write`, `PWrite` additionally provides persistency, i.e., when the call returns, `RDPMA` guarantees the written data must reaches the persistent domain (§2.3.2.1). For one-sided primitive, `RDPMA` optimizes one-sided based `Write` with **H9**. For two-sided primitive, `RDPMA` adds `sfence` after a two-sided based `Write`.

2.3.5.2 Improved system design: Octopus

Existing or future RDMA-NVM systems can benefit from our summarized optimization hints (§2.3.4) and `RDPMA`. In this section, we present how we use these hints to improve the performance of an representative open-source RDMA-NVM systems, Octopus [51]. Octopus is a distributed file system designed for RDMA and NVM. It is designed when no production NVM is available.

Overview. Octopus uses a distributed NVM pool to store the file system metadata and its file data blocks. It achieves high throughput and bandwidth for reading/write file data through *Client-Active Data I/O*: the client directly read/write a file's data block using one-sided RDMA READ/WRITE. Besides *Client-Active Data I/O*, Octopus also leverages RDMA-enabled distributed transactions to update the filesystem metadata. Its transactional protocol use one-sided RDMA ATOMICs to coordinate conflicting metadata operations.

Optimizations. We focus on improving Octopus's *Client-Active Data I/O* because the distributed transaction is not supported in its current public available codebase①. Nevertheless, we believe our findings can also improve distributed transaction performance in Octopus, e.g., using **H8** to improve its lock performance. Specifically, we port Octopus to use `RDPMA` for reading/writing files that are also allocated with `RDPMA`.

2.3.5.3 Evaluation

We follow the Data I/O benchmark in the Octopus paper [51] for the evaluation. In this benchmark, each client writes a fixed payload to a random location in a randomly

① https://github.com/thustorage/octopus

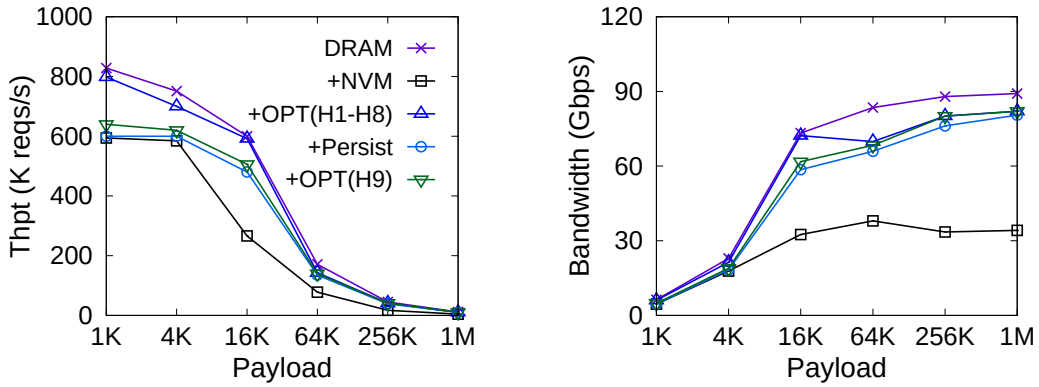


Figure 2–19 Data I/O (a) throughput and (b) bandwidth of Octopus (Multiple Clients).

chosen file. The client first uses two-sided RDMA to query the file metadata (e.g., data block addresses). Then, it writes the data payload with one-sided RDMA WRITE.

Comparing targets. In Figure 2–19, **DRAM** is the vanilla Octopus using DRAM to emulate the NVM pool. **+NVM** uses Optane PM as NVM pool, and **+OPT(H1-H8)** applies our optimizations to **+NVM** with `RDPMA`. **+Persist** further adopts an existing approach [75] to support synchronous durable file write atop pf **+OPT(H1-H8)**. Finally, **+OPT(H9)** leverages **H9** to reduce network roundtrips for persistence of **+Persist**.

Performance. As shown in Figure 2–19, **+OPT(H1-H8)** improve **+NVM** by up to 2.4X (from 1.2X), mainly due to applying **H3**. Without **H3**, Octopus's client-active I/O cannot fully saturate NVM's write bandwidth because it uses one-sided RDMA WRITE with DDIO enabled to write to the NVM. Further, **+OPT(H9)** outperforms **+Persist** by 1.06X (from 1.02X), thanks to the reduced RDMA roundtrips for persistent write.

2.3.6 Summary

This section conducts a systematic study on better leveraging RDMA and NVM to build efficient remote persistent memory. Based on the study, we build `RDPMA`; a remote persistent memory library integrated with the studied optimizations. Evaluation of `RDPMA` on existing RDMA-NVM systems confirms its benefits. We believe our summarized hints as well as experiences in applying them to existing systems can benefit future systems developers when designing systems with RDMA and NVM.

2.3.7 Related work on NVM

RDMA-NVM systems. RDPMA continues the line of research of using RDMA and NVM to improve the performance and reliability of distributed systems [13, 18, 51-57]. Kashyap *et al*. [90] explores the trade-offs of using different methods to ensure NVM write persistence with RDMA. They conduct their experiments on emulated NVM. The study in this section instead focuses on Optane PM. Orion [52] is a distributed file system designed for RDMA and NVM. It does not consider RDMA-ware optimizations (i.e., **H9**) and chooses two-sided RDMA for the persistent write. We believe RDPMA can provide better design decision for it. Hotpot [54] uses RDMA and NVM to build a distributed persistent shared memory. AsymNVM [56] proposes an asymmetric architecture to use NVM with RDMA. Though AsymNVM is evaluated with Optane PM, it does not consider the performance features of Optane PM. Hence, we believe RDPMA can improve its performance on Optane PM.

NVM-aware systems. Except for RDMA-NVM systems, researchers are building NVM-aware systems for decades, including but not limited to file systems [38-40], NVM-aware data structures [41-42], key-value stores [43], NVM-aware JVM [44-45], and transactions on NVM [46-50]. Like RDMA-NVM systems, most of them use emulated NVM since there is no commercially available NVM at that time. We hope the study in this section can further inspire future research on revisiting previous NVM-aware systems on Optane PM.

2.4 Discussion and future trends

Generality of the study. Our study focuses on specific RNIC (Mellanox ConnectX-4, Mellanox ConnectX-5) and NVM (Intel Optane DC Persistent Memory), while other hardware devices may yield different results. Nevertheless, we believe these RNICs are representative RNICs, as recent generations of Mellanox RNICs (e.g., Connect-IB) all share the same architecture. Moreover, Optane PM is the only commercially available NVM. Finally, we also provide open-source tools that the developers can use to examine their design choices under different hardware configurations.

Next-generation NVM. The next-generation of NVM not only has a better performance but also provides a larger scope of persistent domain. First, it will have a 25%

higher bandwidth [91]. Second, it will include the processor cache in its persistent domain [76]. This feature is desirable for one-sided RDMA since one-sided RDMA no longer depends on DDIO for WRITE to be persistent (§2.3.2.2). Consequently, the designer does not need to make a trade-off between one-sided persistence and two-sided RDMA performance (§2.3.4).

On the other hand, the new features of next-generation NVM are unlikely to change the advice of our study. First, the primary focus of our study is *how to avoid NVM write becoming the bottleneck* in RDMA-NVM systems (§2.3.3), even when NVM write has a comparable performance with RDMA (see Figure 2–6). Thus, the 25% bandwidth improvement of the next-generation NVM is insufficient to twist the performance comparisons between RDMA and NVM, since future generations of RDMA will have much higher bandwidth. For example, RNIC with 200Gbps bandwidth has already been commercially available [92], which is 2X higher than our evaluated RNIC. Besides performance, the enhanced functionality of next-generation NVM, i.e., putting cache in the persistent domain, is also unlikely to address the current performance issue caused by the cache. This is because the random cache eviction is still not suitable for NVM. Meanwhile, an extra one-sided RDMA READ is still required to ensure persistence as long as the RNIC is not re-designed.

Suggestions to hardware designers. There are proposals to extend RDMA to cooperate with NVM, e.g., Talpey *et al*. [93] proposed to add a *one-sided commit* primitive to support one-side persistent RDMA WRITE. Nevertheless, our study reveals that existing proposals are insufficient: *the hardware designers not only need to consider hardware extensions to support more functionality, but also should consider extensions for better performance*. For instance, adding an RDMA-version of `ntstore`, e.g., one-sided *non-temporal* RDMA WRITE that allows the WRITE to bypass the cache—can greatly improve the flexibility in configuring DDIO for RDMA-NVM systems (§2.3.4).

2.5 Conclusion

Designing high-performance systems with RDMA and NVM requires a clear understanding of these hardware features. This chapter provides a systematic study on how to efficiently use RDMA and how to best utilize NVM with RDMA. Based on the study, we built two systems. `R2` is an execution framework designed with RDMA features, while

RDPMA is an efficient remote persistent memory library built upon RDMA and NVM. Integrating R2 and RDPMA to existing systems can lead to up to 2.2X better performance. Over the later chapters, we will discuss how DrTM+H, XStore and DrTM+X build upon R2 and RDPMA.

Chapter 3 Learned cache for RDMA-based Ordered Key-value Store

This chapter presents the design of `XStore`, an in-memory network-attached key-value store designed for RDMA. Network-attached in-memory key-value stores have become the foundation of modern databases [13, 26, 94]. RDMA (Remote Direct Memory Access) has recently generated considerable interests in these key-value stores (aka RDMA-based KVs) in both academia [17, 30, 95] and industry [2, 26, 96], as it enables direct access to the memory of remote machines with low latency and CPU/kernel bypassing. However, leveraging RDMA to ordered key-value stores encounters a significant obstacle—traversing the tree-based index with one-sided RDMA primitives is costly and complex. This is because it usually requires multiple network round trips (e.g., $O(\log N)$) and rapidly saturates bandwidth.

Many recent academic and industrial efforts [13, 28, 97] therefore proposed *index caching* to reduce RDMA operations. Yet, the conventional wisdom on implementing cache—replicating partial data and accessing them locally—does not work well with the tree-based index, and the drawbacks are amplified by maintaining the *tree-based* cache with RDMA primitives. First, the tree-based index can be large, so that the cache would suffer from unavoidable capacity misses. Second, the cache would aggravate random memory accesses and further increase the end-to-end latency. Third, updating the tree-based index may recursively invalidate the cache and cause false invalidation due to path sharing.

Inspired by recent research [98]—using machine learning (ML) models as an alternative index structure, we propose to leverage ML models as the (client-side) RDMA-based cache for the (server-side) tree-based index, termed *learned cache*. Specifically, the client uses learned cache to predict a small range of positions for a lookup key and then fetches them using one RDMA READ. After that, the client uses a local search (e.g., scanning) to find the actual position and fetches the value using another RDMA READ. Although using ML models as the index seems efficient (a few floating/int operations) and cheap (a small memory footprint) for static workloads (e.g., gets), it is also notoriously slow (frequently retraining ML models) and costly (keeping data in order) for dynamic workloads (e.g., inserts).

The key contribution described in this chapter is XStore, and fast RDMA-based ordered key-value store using our proposed learned cache. To address the challenges imposed by dynamic workloads, we propose a *hybrid* architecture that retains a tree-based index at the server to perform dynamic workloads (e.g., inserts) and leverages a learned cache at the client to perform static workloads (e.g., gets and scans). The hybrid architecture not only provides separate and appropriate execution paths for both workloads, but also simplifies the mechanism to guarantee the correctness of concurrent local and remote operations.

We have implemented XStore by extending a concurrent $B^{+}tree$ [3] on R2 (§2.2). Evaluations using the YCSB benchmarks [99] with two synthetic and one real-world [100] datasets, as well as two production workloads from Nutanix [101] show that a single XStore server can achieve over 80 million read-only requests per second. This number outperforms state-of-the-art RDMA-based ordered key-value stores (i.e., DrTM-Tree [18], Cell [28], and eRPC+Masstree [65]) by up to 5.9× (from 3.7×). For workloads with inserts, XStore still provides up to 3.5× (from 2.7×) throughput speedup, achieving 53M reqs/s. The learned cache also reduces client-side memory usage significantly and further provides an efficient memory-performance tradeoff. For example, it can save 99% memory at the cost of 20% peak throughput, compared to caching the whole index. In §6.1 we will further evaluates the effectiveness of XStore in distributed transactions.

3.1 Background of RDMA-based ordered key-value store

This section present the environment XStore targets. XStore focuses on in-memory key-value (KV) stores that adopt the client-server model (network-attached) [30, 95, 102-103] and range index structures (tree-backed) [28, 97, 104]. The server hosts both key-value pairs and indexes in main memory and handles requests from multiple clients concurrently. The client interacts with the server through a library that provides basic key-value interfaces, including GET(K), UPDATE(K,V), SCAN(K,N)①, INSERT(K,V), and DELETE(K), as well as more complex operations built atop them.

Figure 3–1 presents two design choices for RDMA-based ordered key-value store.

Server-centric design (S-RKV) [17, 25, 65]. An obvious design is to take a traditional KV store and reimplement the communication layer (e.g., RPC) using RDMA primitives.

① SCAN(K,N) provides a form of range query that retrieves first (up to) N key-value pairs, where their keys are larger than or equal to K.

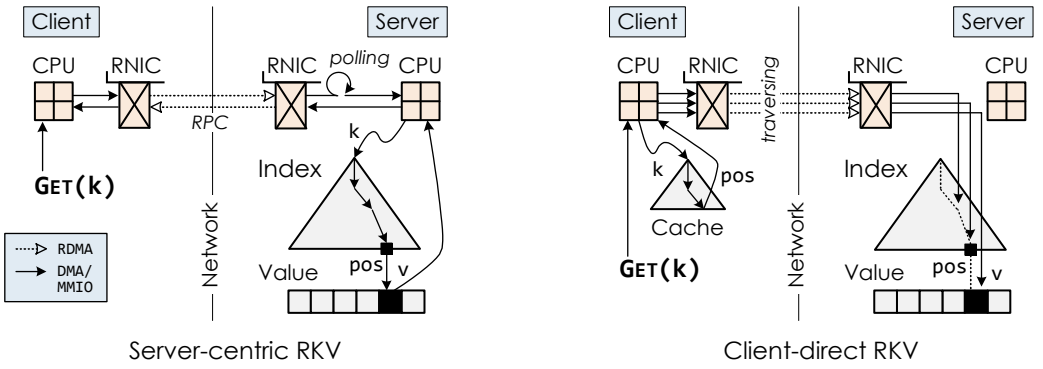


Figure 3–1 The architecture of RDMA-based key-value stores: (a) server-centric RKV and (b) client-direct RKV.

As shown in Figure 3–1a, the clients ship their requests to the server via RDMA network using one round trip for each; the server traverses the tree-based index and performs the request locally. The *server-centric* design allows access to the server-side store with only two RDMA operations (one for sending and one for receiving), no matter how complex the index structures are, thereby avoiding multiple round trips and message size amplification [25]. However, this design exploits only high performance (low latency and high bandwidth) but not CPU efficiency (remote CPU bypassing) of RDMA network at the server, which limits the scalability of these KV stores with the increase of clients.

Client-direct design (C-RKV) [13, 28, 97]. The adoption of RDMA makes it practical to allow clients to access data hosted on the server directly, thereby permitting an alternative (*client-direct*) design that relaxes the burden on server CPUs. To simplify the mechanism for consistency, this design is restricted to read-only requests (i.e., GET and SCAN) in most systems [2, 28, 95]. The choice is also motivated by the read-dominated nature of real applications [20, 105]. As shown in Figure 3–1b, the clients use one-sided RDMA operations to traverse the tree-based index and fetch the value directly for read-only requests; the server still needs to perform the rest of requests (i.e., UPDATE, INSERT, and DELETE) locally. The *client-direct* design can shift the CPU load on the server to the clients, which would alleviate the bottleneck (from CPU to network), especially on high-bandwidth networks (e.g., 100Gbps). However, it may consume extra network round trips for traversing the tree-based index due to the lack of richness of RDMA primitives, causing an order-of-magnitude slowdown (e.g., 11× slowdown in Figure 3–3a). For example, recent work [28, 97] uses RDMA READs to traverse remote B^+tree index and invariably incurs multiple network round trips ($O(\log N)$ [106]).

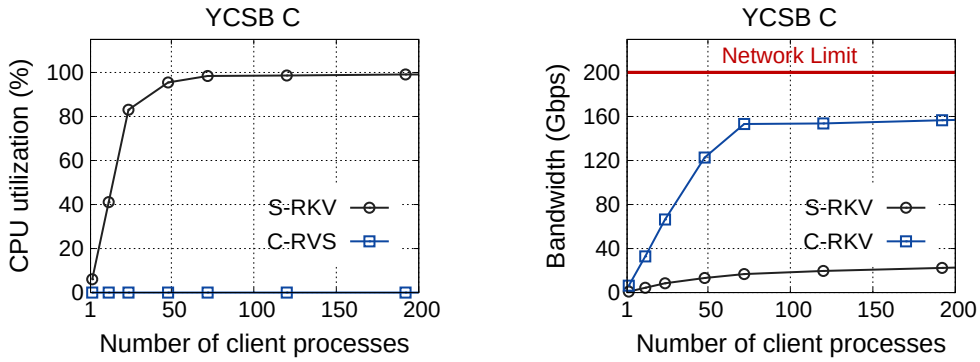


Figure 3–2 A comparison of server-side (a) CPU and (b) network bandwidth utilization for state-of-the-art server-centric (**S-RKV**) and client-direct (**C-RKV**) RDMA-based KV stores. **Workload**: YCSB C (100% read), using 100M keys with a uniform distribution. **Testbed**: The server has two 12-core CPUs and two 100Gbps RNICs.

Recently, *index caching* has been proposed to reduce network round trips for index traversal by RDMA-based systems [13, 17, 28, 107-108], namely, the client caches the server-side index locally. It aims at reducing RDMA READs for fetching the position of the value (aka *lookup*), instead of caching the value directly.① Thus, an *optimal* result with index caching only needs two RDMA operations per request (one for lookup and one for read).

3.2 Analysis of RDMA-based ordered key-value stores

In this section, we analyze existing RDMA-based ordered KVS presented in §3.1.

CPU is the primary scalability bottleneck in the server-centric design. Figure 3–2 compares the hardware resource utilization (CPU and network bandwidth) between S-RKV and C-RKV with the increase of clients. For S-RKV, the server rapidly saturates all CPUs (24 cores) but just consumes 11% of network bandwidth. It implies that CPU first becomes the performance bottleneck and limits the scalability with the increase of clients, especially when deploying fast networks. This also runs counter to the recent trend of building servers in modern datacenters with CPU-bypassing networks [13, 63, 109]. As shown in Figure 3–3a, S-RKV reaches the peak throughput of around 24M

① Considering RDMA performance degradation with increasing payload size [30], the client will only cache internal nodes [28, 108] and not directly fetch a batch of keys and (inline) values to avoid bandwidth amplification [28, 106].

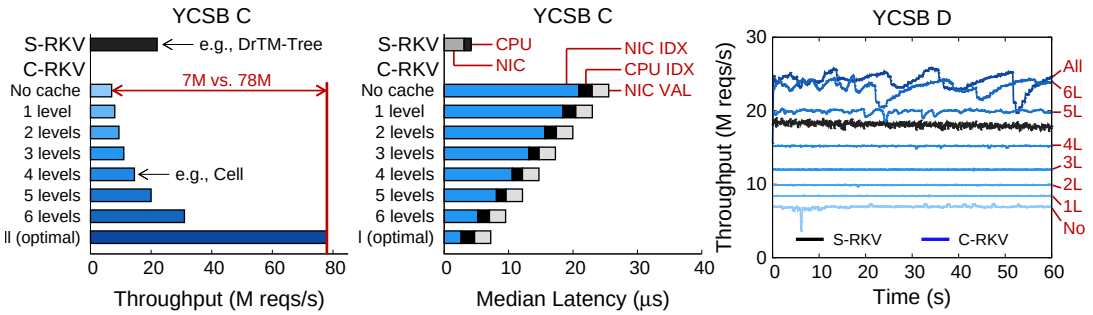


Figure 3–3 A comparison of server-side (a) peak throughput, (b) end-to-end median latency at low load, and (c) throughput timeline for state-of-the-art server-centric (**S-RKV**) and client-direct (**C-RKV**) RDMA-based KV stores. **Workload**: YCSB C (100% read) and YCSB D (95% read and 5% insert), using 100M keys with a uniform distribution. **Testbed**: The server has two 12-core CPUs and two 100Gbps RNICs.

reqs/s. Traversing tree-based index occupies most of CPU time, as it involves massive random memory accesses. On our testbed, we measured that one CPU core can perform 43 million 64-byte random reads per second at full speed. Thus, each core can only process up to 1.8M reqs/s for traversing a (8-level) B^+tree with 100M keys, even putting other CPU and network costs aside.

Costly RDMA-based traversal is the key obstacle in the client-direct design. C-RKV allows the client to traverse the server-side index directly by using one-sided RDMA READs, which can thoroughly bypass server CPUs (see Figure 3–2a). However, RDMA-based index traversal usually requires multiple network round trips (e.g., $O(\log N)$ for tree-based index) and saturates the network bandwidth quickly. As shown in Figure 3–3a, RDMA-based traversal limits the peak throughput of C-RKV to 7 million requests per second, even much lower than that of S-RKV. Using index caching at the clients can reduce RDMA operations by traversing index nodes locally. On our testbed, the throughput of C-RKV with index caching, similar to state-of-the-art design (Cell [28]), peaks at 14.5M reqs/s, as each request takes 4 RDMA READs (down from 8) for traversal.

Tree is not a proper structure for RDMA-based index cache. To our knowledge, existing RDMA-based index caches use *homogeneous structures* to store *partial* index nodes, similar to the conventional design. For example, each client replicates tree nodes and traverses them locally before accessing the tree-based index hosted on the server [13, 28, 97].

First, the tree-based index can be large [25, 110-111], and the traversal demands

multiple random accesses from the root to the leaf node. Thus, each client can only cache nodes near the root (e.g., top four levels [28]) to minimize thrashing and maximize hits [13, 28]. Yet, the index cache still suffers from *unavoidable capacity misses* (bottom node levels). In Figure 3–3a, for a read-only workload, the effect of RDMA-based caching for tree-based index is dominated by inner node levels cached. The *optimal* throughput (a *whole-index* cache) reaches 78M reqs/s using one RDMA READ for each traversal (fetch the position of the value), 3.3× better than S-RKV.

Second, traversing tree-based index is a memory-intensive but low-compute operation. The *homogeneous* index cache can just alter the type of memory accesses (i.e., remote and local), instead of reducing the number of memory accesses ($O(\log N)$). Hence, despite the index cache, traversing tree-based index would still incur *massive random accesses* and suffer from CPU cache misses, TLB misses, and RNIC's page translation cache misses. As shown in Figure 3–3b, even caching the whole index, the end-to-end latency of C-RKV is still 80% higher than S-RKV, and the CPU cost on index cache (CPU_IDX) occupies close to 30%.

Third, updates to the tree-based index (i.e., inserts and deletes) might propagate the changes from the leaf level to the root node, so that the index updates would probably invalidate the cache *recursively* [97] and cause *false* invalidations (path sharing). It would result in frequent cache misses and RDMA READs to retrieve updated index nodes. Worse yet, the more tree nodes cached, the more performance degrades. Further, preserving traversal consistency for dynamic workloads demands sophisticated detection schemes (e.g., fence keys [112-113]) and incurs additional overhead. As shown in Figure 3–3c, the *optimal* throughput significantly drops to 25M reqs/s with severe performance fluctuations, just because of 5% inserts.

3.3 An overview of `XStore`

Opportunity: ML Models. `XStore` is motivated by an attractive observation from the learned index [98]—a range index (e.g., B^+tree) that finds the position of a given key inside a sorted array approximates the cumulative distribution function (CDF) of the keys in the index. As shown in Figure 3–4, suppose the values have been sorted according to the lookup keys, the CDF (the red curve) is a mapping from the (sorted) keys to the (sorted) positions of their values, namely CDF(K) returns the *actual* position of the value corresponding to K. Prior work [98] proposes to approximate the shape of a CDF using

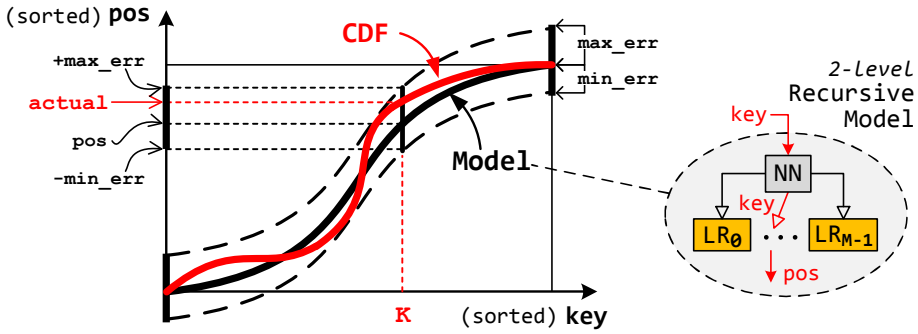


Figure 3–4 An example of using ML models to predict the position within a sorted array for a given key.

machine learning (ML) models, like neural nets (NN) and linear regression (LR), since they are able to learn a wide variety of distributions. As an alternative range index, the ML model is trained with every key to record the worst over- and under-prediction of a position (i.e., min- and max-error). In Figure 3–4, given a lookup key (K), the model (the black curve) can predict a position (pos) with a min- and max-error (min_err and max_err), and a local search (e.g., scanning) around the prediction is used to get the *actual* position. To further reduce the prediction error, a hierarchy of simple models (e.g., recursive-model index [98]) is used to partition the key space, where the model at level L picks the model at level L+1 based on the key.

Our approach: Learned Cache. The key idea behind `XStore` is to leverage machine learning (ML) models as (client-side) RDMA-based cache for the (server-side) tree-based index, termed "***learned cache***". The unique features of machine learning models can fundamentally overcome the drawbacks in the conventional wisdom for RDMA-based index caching (see §3.2). First, instead of using a *homogeneous* structure to cache a *partial* index, the ML model can cache the *whole* index at the cost of *accuracy*. Therefore, using the learned cache can completely avoid capacity misses, and each lookup only needs one RDMA READ. Further, the ML model is also famously memory-efficient (e.g., two parameters per LR model). Thus, the learned cache can match the *optimal* throughput of conventional design (a whole-index cache) but with practical memory consumption.

Second, instead of finding the *actual* position by traversing a tree-based index with $O(\log N)$ random memory accesses, the ML model can approximately predict a *range* of positions for a lookup key by performing a single multiplication and addition (e.g., linear regression). It implies that the learned cache might also reduce the end-to-end latency,

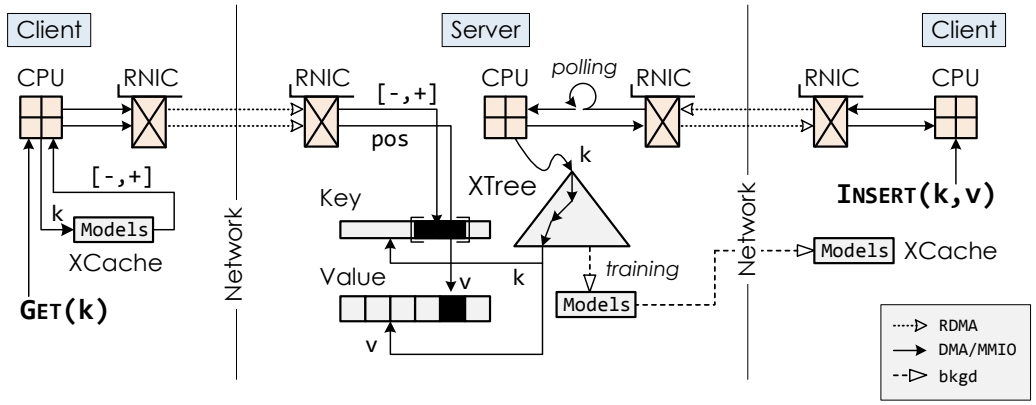


Figure 3–5 The hybrid architecture behind `XStore`: client-direct operations (left) and server-centric operations (right).

even compared to a whole-index cache, due to fewer CPU cache and TLB misses at the clients.

Finally, instead of *fine-grained* and *recursive* invalidation in the tree-based cache for accurate predictions, the ML model can reduce and delay cache invalidations since it only needs to provide approximate predictions. Updates to the index might only decrease the accuracy of the (partial) ML model. Thus, the learned cache can significantly save invalidation cost in terms of network round trips and bandwidth usage, especially compared to a whole-index cache.

Challenge: Dynamic Workloads. Dynamic workloads (e.g., inserts and deletes) would violate an (unrealistic) assumption of ML-based approach that all key-value pairs are stored in sorted order by key [98]. However, retraining ML models and keeping data in order are slow and costly, which is hard to match the high performance of in-memory key-value stores (tens of millions of requests per second). An intuitive solution is to maintain a delta index (e.g., B^+tree) for (in-place or buffer-based) inserts and then periodically compact it with the learned index (data merging and model retraining) [111, 114]. Unfortunately, it cannot work well with RDMA-based index caching. First, additional RDMA-based lookups on the delta index would incur more network round trips and severely increase the latency. Second, it is also hard to cache a fast-changing (tree-based) delta index at the clients. Finally, the data and model compaction definitely interrupts (RDMA-based) remote accesses and completely invalidates the learned cache. Hence, *how to make learned cache keep pace with dynamic workloads at low cost* becomes a key challenge.

Overview of XStore. XStore is an in-memory ordered key-value store using a client-server model, where the server and the clients are connected with a high-speed, low-latency network with RDMA.① Using ML models as the index (aka *learned index*) is famously efficient and cheap for static workloads (e.g., gets and scans), while it is notoriously slow and costly for dynamic workloads (e.g., inserts and deletes). It is because the inserts would amplify the prediction error and incur model retraining frequently. Prior work [98, 114-115] relies more on the profit from efficiently handling static workloads to amortize the negative influence on dynamic workloads. We argue that the *learned cache* opens the opportunity to solve this dilemma. Unlike prior work [98, 114-116], which replaces or augments the tree-based index with the learned index, we propose a *hybrid* architecture that *retains the tree-based index at the server to handle dynamic workloads and uses the learned cache at the clients to handle static workloads*.

Figure 3–5 shows the architecture of XStore. The server hosts a B^+tree index (XTree) in the main memory and stores key-value pairs at the leaf level physically, like the common practice. Each client interacts with the server through a library, which hosts a local learned cache (XCache). XStore uses the client-direct design for read-only requests (i.e., Get(k) and Scan(k,n)) and the server-centric design for the rest (i.e., Update(k,v), Insert(k,v), and Delete(k)). For client-direct operations, like Get(k) in Figure 3–5, the client first predicts a range of positions for the key k using XCache and then fetches them using one RDMA READ. Finally, the client uses a local search to find the actual position and fetches the value using another RDMA READ. For server-centric operations, like Insert(k,v) in Figure 3–5, the client uses RPC over RDMA to ship the request to the server. The server searches the lookup key k by first traversing the B^+tree index and then inserts the new KV pair (k,v). XStore will partially retrain ML models for updated tree nodes in the background, and each client will individually fetch the models for XCache on demand.

3.4 Design and implementation of XStore

We describe the design and implementation of XStore in this section. We start by presenting two main data structures at the server and the clients, respectively (§3.4.1). Then we elaborate on how XStore implements client-direct operations (§3.4.2) and

① The client may not be the end user but the computation node or the front-end of RDMA-based datacenter applications [2, 13, 26, 28, 30, 95, 97].

server-centric operations (§3.4.3). Finally, we discusse how to support durability (§3.4.4) and scale-out (§3.4.5) in XStore.

3.4.1 Data structures

XTree. At the server, XStore retains a B^+tree index (XTree) and stores key-value pairs at the leaf level physically, like common practice, as illustrated in the left part of Figure 3–6. XTree follows the basic design of a concurrent B^+tree [3, 104], except that the leaf node (LN) is optimized for remote reads. Specifically, the leaf node consists of a 24-bit incarnation (INCA), an 8-bit counter (CNT), a 32-bit right-link pointer to next sibling (NXT), keys with N slots ($K_0..K_{N-1}$) and values with N slots ($V_0..V_{N-1}$), as described in the following table.

INCA	The incarnation of the leaf node
CNT	The number of key-value pairs stored
NXT	The offset to the next leaf node if exists
$K_0..K_{N-1}$	The keys of key-value pairs (at most N)
$V_0..V_{N-1}$	The values of key-value pairs (at most N)

Every leaf node is allocated from an RDMA-registered memory region managed by a slab allocator. The node can store at most N key-value pairs in sorted order. For brevity, we assume fixed-length key-value pairs here.① To save the size of RDMA READ for lookup, XStore stores keys and values separately but continuously. This setup can avoid storing the address of the value. For instance, the client can fetch N keys from the leaf node and calculate the (remote) address of the expected value locally (a fixed offset from its key). Moreover, XStore uses incarnation checks [2, 17] to guarantee the consistency of remote accesses. The incarnation in the leaf node is initially zero and is monotonically increased when the leaf node is reused (e.g., split or free). The number of slots (N) can be tuned for RDMA performance (e.g., 16).

XCache. Each client hosts a local learned cache (XCache), which consists of a 2-level recursive ML model (XModel) and a translation table (TT). As illustrated in the right part of Figure 3–6, given a lookup key, XModel is used to predict a range of positions

① Similar to prior RDMA-based key-value stores [2, 17, 28], XStore currently supports fixed-length key and fixed/variable-length value. For variable-length value, the leaf node should store a 64-bit fat pointer [2, 37] (the size and the position of the value) instead of the value. We discuss how to support variable-length key in §3.6 and leave it to future work.

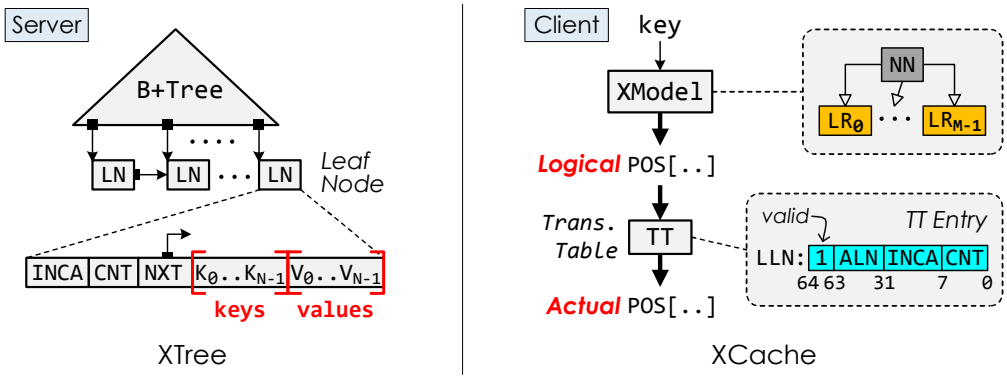


Figure 3–6 The main structures in `XStore`: XTree and XCache.

(`POS[..]`) within a sorted array (logically stitching together all leaf nodes of XTree). `XStore` assumes machine learning (ML) models can effectively learn various data distributions. We describe how `XStore` configures models in §3.5 in details.

The ML model demands the positions (virtual address) of leaf nodes are always sorted by the keys. However, it is almost impossible for dynamic workloads, since the insertion of key-value pairs may insert a new node at the leaf level and break the sorted order of leaf nodes. To this end, the `XStore` server maintains an additional translation table (TT) for leaf nodes that translates logical positions to actual positions. Each client caches a part of the table on demand. The entry of TT is located by the logical leaf-node number (`LLN`) and consists of a valid bit, a 31-bit actual leaf-node number (`ALN`), a 24-bit expected incarnation (`INCA`), and an 8-bit counter, as shown in Figure 3–6. The client can calculate the (host) virtual address of the target leaf node using `ALN` and the base address of an RDMA-registered memory region. Further, the match of incarnation between TT's entry and target leaf node guarantees that the leaf node has not been reused.

Training models and TT. The server (re-)trains a 2-level ML model (XModel) with a translation table (TT) over XTree's leaf nodes in the background, and each client (re-)fills the learned cache (XCache) on demand. Figure 3–7 shows the pseudo-code of training a XModel and TT from scratch. Starting from a sorted array of keys with logical positions (line 4), we first train the top model. Based on the prediction of the top model, we then evenly partition keys into `M` sub-models (line 9). Finally, we train each sub-model on a sorted array of its keys with a private logical position at a leaf node granularity (line 12-21) and calculate min- and max-error for every sub-model (line 22). Note that the keys in the leaf node across sub-models will be trained by both of sub-models. Moreover, each

```
▸ M: Max. number of sub-models
▸ N: Max. number of keys in each leaf node
XModel {
  Model     top                                ▸ LR: k → [0,1)
  Model[M] subs              ▸ LR: k → [0,pos) w/ min/max_err
}

TRAIN_XMODEL(xmodel)
▸ train top-model
1   cdf = []                                   ▸ training set
2   pos = 0
3   foreach k in xtree                       ▸ in sorted order
4   | cdf.add(k, pos++)
5   xmodel.top = new LR trained on cdf
▸ assign keys to sub-models
6   kset = [][]                 ▸ key set for each sub-model
7   foreach k in xtree
8   | mid = xmodel.top.predict(k) × M
9   | kset[mid].add(k)
▸ train sub-models
10  for i in [0:M)
11  | TRAIN_SUBMODEL(xmodel.subs[i],
    |                MIN(kset[i]), MAX(kset[i]))

TRAIN_SUBMODEL(model, min, max)
12  cdf = []                                   ▸ training set
13  LLN = 0                       ▸ logical leaf-node number
14  start = xtree.find_lnode(min)
15  end = xtree.find_lnode(max)
16  for lnode in [start:end]
17  | pos = LLN × N
18  | foreach k in lnode.keys              ▸ key-sorted order
19  | | cdf.add(k, pos++)
20  | model.tt[LLN++] = {1, ALN(lnode),
    |                     lnode.inca, lnode.cnt}
21  model = new LR trained on cdf
22  model.calc_err(cdf)              ▸ calculate min/max_err
```

Figure 3–7 Pseudo-code of training XMODEL and TT over XTREE.

sub-model has independent logical positions and an own translation table, making it easy to retrain a sub-model individually when necessary.

In practice, training XMODEL is fast and low-cost, since (1) all of the models in XMODEL are simple linear/multi-variate regression models, can be efficiently trained; (2) XMODEL can be partially retrained at a sub-model granularity; and (3) the top model can be trained over a sampled data. As an example, for 100M keys, XMODEL with 500K sub-models takes about 4 seconds to train the top-model and 8 microseconds for each

```
LOOKUP(key, &addr)
1   mid = xmodel.top.predict(key) x M
2   model = xmodel.subs[mid]
3   pos = model.predict(key)                    ▸ prediction
4   start = (pos - model.min_err)/N               ▸ lnode ID
5   end = (pos + model.max_err)/N                 ▸ lnode ID
6   rdma_doorbell = []
7   for n in [start:end]                  ▸ from LLN to ALN
8   | entry = model.tt[n]                         ▸ TT entry
9   | if entry.valid == 0 then
10  | | return invalid                            ▸ fallback
11  | ra = RA(entry.ALN)                   ▸ remote address
12  | rdma_doorbell.add(ra)
    ▸ one RDMA to read disjoint memory regions
13  lnodes = RDMA_READ(rdma_doorbell)
14  for n in [start:end]
15  | lnode = lnodes[n-start]
16  | entry = model.tt[n]
17  | if entry.inca != lnode.inca then
18  | | entry.valid = 0                        ▸ invalidation
19  | | return invalid                             ▸ fallback
20  | for i in [0:lnode.cnt)                  ▸ local search
21  | | if key == lnode.keys[i] then
22  | | | addr = calc remote addr of ith value
23  | | | return found
24  return not_found                       ▸ non-existent key
```

Figure 3–8 Pseudo-code of LOOKUP operation based on XCACHE.

sub-model using a single thread. Further, the client can fill a 500K sub-models XCACHE from scratch in less than one second.

A memory-performance trade-off. The ML model is famously memory-efficient [98]. In XMODEL, the basic sub-models are 14B large and consist of two 32-bit floating-point model parameters ①, two 8-bit min- and max-error, and a 32-bit TT size. Thus, XMODEL with 500K sub-models only needs less than 6.7MB. In contrast, TT might dominate the memory usage of XCACHE. For 100M keys, suppose each leaf node has 16 slots (N) and is half-full, TT requires nearly 100MB (15% of the tree-based index). In practice, each client could cache sub-models and TT entries on demand, and even just cache XMODEL to save 99% memory at the cost of 20% performance (using one RDMA READ to fetch a few TT entries).

① LR may use more floating-points for prediction.

3.4.2 Client-direct operations

XStore adopts the client-direct design (§3.1) for read requests, namely GET(K) and SCAN(K,N), as shown in the left part of Figure 3–5.

3.4.2.1 GET

Given a key, the client uses XCACHE to lookup the remote position of the value using one RDMA READ in the common case, replacing RDMA-based traversal in a tree-based index. As shown in Figure 3–8, the client first uses XMODEL to predict leaf nodes that cover the lookup key (from `start` to `end`) and then calculates the actual (remote) address of these leaf nodes with TT (line 11). The client can use one RDMA READ with doorbell batching to fetch disjoint memory regions if necessary (line 13).① Note that the unit of remote read is a leaf node (`N` keys with a 64-bit header); it is the most likely to read just one leaf node due to the low prediction error of XMODEL. Next, the client uses a local search (e.g., scanning) to find the key from leaf nodes retrieved (line 20-23) and calculates the remote address of the value if it is found (line 22). Finally, the client uses another RDMA READ to fetch the value. Note that any invalid TT entry (line 9 and 17) would result in a fallback path, which ships the GET operation to the server and fetches updated models and TT entries using a single request (i.e., server-centric design).

3.4.2.2 SCAN

SCAN(K,N) implements a form of range query that returns first (up to) N key-value pairs (in order by key), starting with the next key at or after K. The client first uses the lookup operation with K to determine the remote address of the first key-value pair (larger than or equal to K) and then predicts the leaf nodes that contain the next N key-value pairs, with the help of TT. The translation table provides the number of key-value pairs (`CNT`) and the actual remote address (`ALN`) of adjacent leaf nodes (`LLN`) in sorted order by key. Thus, the client can use one RDMA READ with doorbell batching to fetch these leaf nodes, including keys and values. In general, `XStore` only requires two RDMA READs for each range query. In the rare case, the unexpected result, such as an invalid leaf node (incarnation mismatch) due to dynamic workloads, would cause a fallback path, similar to GET. Note that the range query in `XStore` is also not atomic with respect to updates and

① One RDMA READ can only read a continuous memory region. Yet, we can use an RDMA-aware optimization called doorbell batching [66] to read multiple disjoint memory regions in one network roundtrip.

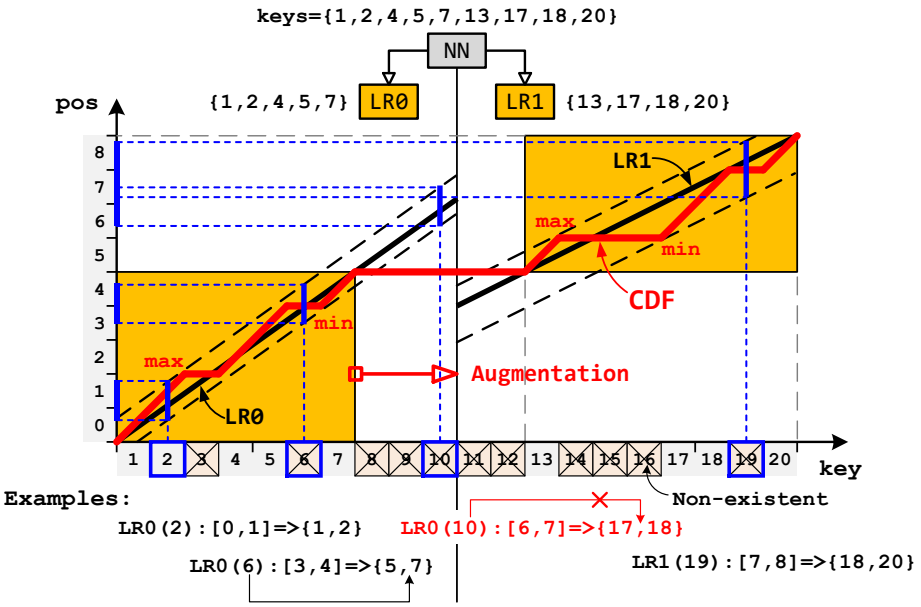


Figure 3–9 An example of the prediction for non-existent keys.

inserts as usual [28, 104]; it could be implemented by applications (e.g., transaction [13, 108]).

3.4.2.3 Non-existent keys

Intuitively, the ML model guarantees to find all keys have been trained since it stores the worst over- and under-prediction for a CDF (i.e., min- and max-error). However, for non-existent keys, the model should be *monotonic* to guarantee the correct upper and lower bound of a prediction [117-118], so that a local search could make sure the lookup key does not exist (see line 24 in Figure 3–8). Hence, XMODEL adopts monotonic models (e.g., linear regression). As shown in Figure 3–9, for a non-existent key (KEY=6), the sub-model LR0 can provide a proper prediction (LR0(6)=[3,4]) that covers the non-existent key (KEYS={5,7}).

However, a hierarchy of models might leave a gap of non-existent keys between neighboring models. Consequently, it still might provide a wrong prediction for these non-existent keys, even if every model is monotonic. For example, the top model selects LR0 for KEY=10 (non-existent), and then LR0 will return a wrong prediction (LR0(10)=[6,7]) that cannot determine whether the key does not exist or the model is out of date from the results (KEYS={17,18}). Worse yet, the non-existent key is common in the range query (e.g., SCAN(K,N)), which demands to retrieve first (up to) N keys larger

than or equal to K. As illustrated in Figure 3–9, the lookup (LR0(10)) for a range query SCAN(10,3) will miss a key (KEY=13), so the result (KEYS={17,18,20}) is also wrong.

Data augmentation. To remedy this, we augment the training set of sub-models to cover the gap of non-existent keys between neighboring models. However, data augmentation would increase the prediction error. We thus carefully add a boundary key to both sub-models, which can fill the gaps with minimal overlap between models. For example, in Figure 3–9, we add a non-existent key in the gap (KEY=10) with the position of a previous KEY=4 into both sub-models (LR0 and LR1). After that, the lookup of non-existent keys would always return a correct prediction. Further, since the keys in the leaf node across sub-models have been trained by both, there is no need for data augmentation in most cases.

3.4.3 Server-centric operations

`XStore` clients use a server-centric design and communicate with the server to perform UPDATE(K,V), INSERT(K,V), and DELETE(K) operations, as shown in the right part of Figure 3–5. The server updates XTREE concurrently and retrains XMODEL in the background.

Correctness. The correctness condition in `XStore` is *no lost keys* [104]: the reader must return a correct value for a given key, regardless of concurrent writers. Specifically, when a reader and a writer run concurrently, the reader should return either the old or the new value in an atomic way.

Concurrency. The hybrid architecture behind `XStore` not only provides separate and appropriate execution paths for static and dynamic workloads (see Figure 3–5), but also simplifies the mechanism to guarantee the correctness of concurrent operations. It is critical to the performance of RDMA-based systems due to the lack of richness of RDMA primitives [58]. In Figure 3–10, by using the learned cache (XCACHE), `XStore` restricts (client-direct) remote accesses to the leaf nodes (the dotted red arrow). Thus, we can avoid using sophisticated mechanisms to retrofit a concurrent tree-based index [28].

XTREE reuses an HTM-based concurrent B^+tree [3]① to coordinate concurrent index updates (e.g., node splits) and lookups on internal nodes, without the concern of

① The implementation is based on Intel's restricted transactional memory (RTM), a mature feature available on modern Intel CPUs (e.g., Skylake).

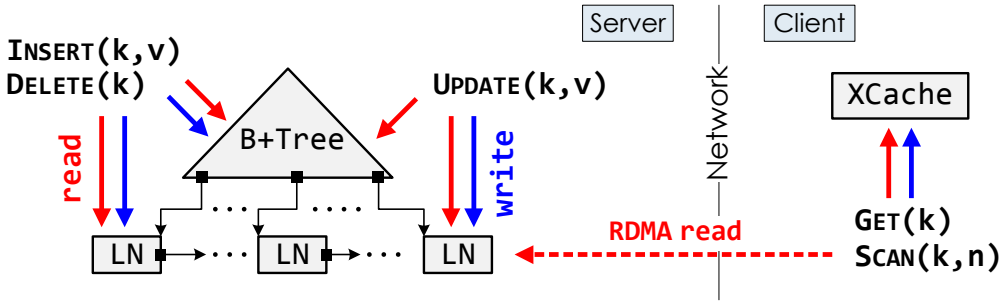


Figure 3–10 The access types of different operations for the main components in XStore. Red and blue arrows denote read and write accesses.

RDMA-based remote accesses. For leaf nodes, XStore follows the technique proposed in DrTM+R [18]. Each tree operation at the server is enclosed within an HTM region, that provides strong atomicity in a single machine [119]. In addition, the strong consistency feature of RDMA (where an RDMA operation will abort an HTM transaction that accesses the same memory location [17]) further extends the atomicity when encountering remote accesses. Moreover, as the RDMA operation is only cache-coherent within a cache line, XStore adopts versioning [2] for consistent remote reads across multiple cache lines. For the data stored in the leaf node across multiple cache lines, a 16-bit version number is stored both in the header of data and at the start of each cache line. The remote reader matches these versions to detect inconsistent read and must retry if the versions differ. Note that XStore hides these versions to applications by automatically converting the data on reads and writes. Finally, the key is also stored in the header of its value, which guarantees consistent remote reads to the key and the value separately.

3.4.3.1 UPDATE

For UPDATE(K,V), the server first traverses XTREE to reach the leaf node and updates the value with V if the key (K) exists. Note that since the update to the value will not change the index, it will not influence the learned cache and belongs to static workloads.

Optimization: position hint. Server-side UPDATE(K,V) can also benefit from the learned cache, especially when the server CPU becomes a bottleneck. For instance, the client can use XCACHE to predict a position (the remote address of leaf nodes) for the key (see line 1-12 in Figure 3–8) and then ship the update request together with the position hint to the server. The server first checks the leaf nodes (by matching incarnation) according to the hint and updates the value if successful. Then it might skip index traversal and

relax the burden on server CPUs. This optimization would increase the performance of update-heavy workloads, like YCSB A (50% update and 50% read).

3.4.3.2 INSERT and DELETE

INSERT(K,V) and DELETE(K) are shipped to the server and performed on XTREE, as is usual on B^+tree. The *in-place* inserts and deletes require moving many key-value pairs within a leaf node to preserve the order of keys. Thus, XTREE chooses not to keep key-value pairs sorted within a leaf node, which can avoid moving key-value pairs and reduces working set in the HTM region. Note that the lookup based on the learned cache will not be affected since it fetches all keys (N) of a leaf node. For DELETE(K), we always overwrite the key and value slot for K with the last key-value pair in the leaf node and update the counter (CNT). Further, the empty leaf node will not be reclaimed to avoid thrashing and model retraining. For INSERT(K,V), we directly append K and V to the key and value slots in the leaf node if K does not exist (see K_A in Figure 3–11). Inserting a key-value pair into a full leaf node will result in a node *split* (see K_B in Figure 3–11). A new leaf node is allocated, and all key-value pairs (plus the new one) are evenly assigned to two leaf nodes in sorted order by key. The original leaf node should increment its incarnation, which makes the clients realize the split. The rest of the split process will execute on the tree index as well as usual.

Retraining and invalidation. The insert of a new leaf node (aka a split) will break the sorted (logical) order of leaf nodes and cause model retraining.

An interesting observation behind our solution is that TT decouples model retraining from index updating and allows a *stale* combination of XMODEL and TT to provide a *correct* prediction for the lookup key as long as it is not overlapped with a split. This is because any insert will not cause data movement across leaf nodes, except the split node. For example, LR_8 initially maps K_A to logical node number LN_2, which stores the leaf's physical address 102. After leaf node LN_1 splits due to inserts (a new leaf node with physical address 327), the latest logical node number for K_A is LN_3 after retraining. Yet, the stale TT still maps K_A to physical address 102, the correct position of K_A. Thus, the client can still use a combination of stale models and TTs to find the keys as long as they are not overlapped with split leaf nodes.

Based on this, after a split, the server will *individually* retrain the sub-model and its translation table in the background (see TRAIN_SUBMODEL in Figure 3–7) and perform

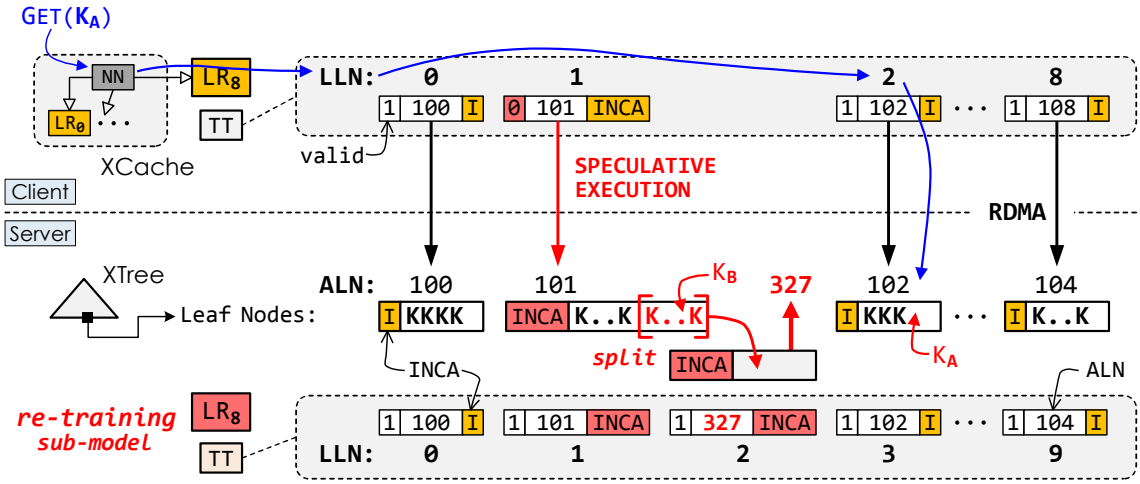


Figure 3–11 An example of model retraining for LR_8 due to a split of LN_{101}. The leaf node is named by its actual leaf-node number (ALN).

all kinds of operations as usual based on XTree. Meanwhile, the clients can still directly perform read-only operations based on XCache. The incorrect prediction can be detected by incarnation mismatch between the leaf node and cached TT entry (line 17 in Figure 3–8) and results in a fallback, which ships the operation to the server. The client will update XCache with a retrained model and its translation table fetched by the fallback. Noted that concurrent splits will not affect model retraining in progress and just make it stale. The new incarnation of the split leaf node ensures the client with this new (stale) model to realize the change of concurrent splits. Each split will issue a retraining task. The training thread currently does not merge or optimize the pending tasks to the same sub-model since it happens very rarely.

Optimization: speculative execution. A split of leaf node just moves the second half of its key-value pairs (sorted by key) to its new sibling leaf node. Therefore, the prediction to the split node must still be mapped to this node or its new sibling, like LN_{101} and LN_{327} in Figure 3–11. Based on this observation, *speculative execution* is enabled to handle the lookup operation on a *stale* TT entry (i.e., failed incarnation check). The client will still find the lookup key in the keys fetched from the split leaf node. If not found, the client will use its right-link pointer to fetch (the second half) keys from its sibling (one more RDMA READ). It means there is roughly half of the chance to avoid incurring a performance penalty. Currently, we only consider one sibling before using a fallback since a cascading split happens rarely. This optimization is important for insert-dominate

workloads (e.g., YCSB D) since insert operations and retraining tasks might keep server CPUs busy; the fallbacks will also take server CPU time.

Model expansion. The growing size of key-value pairs in the ML model will likely increase the prediction error, resulting in performance degradation. Prior work [114] uses a sophisticated model split to adapt its learned structure for dynamic workloads, which demands physical data moving and atomic top-model replacement. Differently, `XStore` supports *model expansion* that increases the number of sub-models in XMODEL at once (e.g., doubling) when necessary (e.g., exceeding a threshold of min- and max-error). The model expansion requires a complete training (see Figure 3–7) on XTREE to build a new version of XMODEL and TT. Note that model expansion will not affect any requests performed by both the server and the client for several reasons. First, training models will not change or move data. Second, the top model can be trained over incomplete data. Third, the conflicting sub-model retraining could be made up later. Finally, the client can use the originally learned cache during model expansion. Moreover, after deleting a large number of key-value pairs, `XStore` can also resize XMODEL to shrink the number of sub-models using a similar process.

3.4.4 Durability

`XStore` should log writes (updates, inserts, and deletes) to log files stored in reliable storage for persistence and failure recovery (e.g., server's local disk). As RDMA-based remote accesses are restricted to reads (lookups, gets, and scans), they will not involve in logging and recovery. In addition, XMODEL and TT are tightly associated with XTREE (e.g., virtual address). Thus, they should be rebuilt after recovery.

To ensure correct recovery from a machine failure, `XStore` reuses the existing durability mechanism in the concurrent tree-based index extended by XTREE, like version numbers [3, 104]. Each worker thread at the server appends the log (key, value, and version) to its in-memory log buffer. A corresponding logging thread, sharing the same core with the worker thread, writes out the log buffer to its log file in the background. The logger batches the log entries to avoid the storage backend becoming the bottleneck. During recovery, `XStore` scans log files to sort logs of the same key by its version number and applies the latest log of keys in parallel. Finally, `XStore` rebuilds XMODEL and TT by training over recovered XTREE.

3.4.5 Scaling out `XStore`

`XStore` follows a coarse-grained scheme [97], the dominant solution, to distribute an ordered key-value store span multiple servers (scale-out). `XStore` first assigns key-value pairs to the servers based on a range-based partitioning function for the keys. Then each server constructs XTREE individually for its assigned key-value pairs and further trains a corresponding XMODEL and TT. Note that the boundary keys should be added to the training set to cover the gap of non-existent keys between neighboring servers.

The client maintains a separate learned cache for each server and uses the same partitioning function to decide which server should perform a given request. Based on it, the client can perform requests as mentioned in §3.4.2 and §3.4.3, with one exception—SCAN(K,N) reads a range of key-value pairs, which can span multiple servers. For instance, after the lookup of K on a specified server, the client might find that the expected number (N) exceeds the remaining key-value pairs in this server. Starting from the first logical leaf node on the next server, the client can predict the leaf nodes that contain the rest of key-value pairs. Finally, the client uses one RDMA READ for each server involved to fetch these leaf nodes.

3.5 Implementation of XModel

`XStore` assumes XMODEL can effectively learn various data distributions (e.g., log-normal [98, 114-115]). `XStore` has also pre-built with various machine learning (ML) models for XMODEL, including linear regression (LR), multivariate linear regression (MLR), and fully-connected neural nets (NN). The NN can be configured with the number of layers, the number of neurons and which activation function to use. We implement XMODEL with static polymorphism such that it can use different models without the cost of virtual function calls. The user can also deploy customized models by implementing the model trait defined by the XMODEL.

In this section, we describe how we implement various ML models (§3.5.1), and how `XStore` chooses ML models for various data distributions (§3.5.2).

3.5.1 Implementation of ML models

Implementing different ML models efficiently from scratch is non-trivial. We use Intel Math Kernel Library (MKL) [120], an efficient math library and PyTorch [121], a powerful ML framework to implement different ML models.

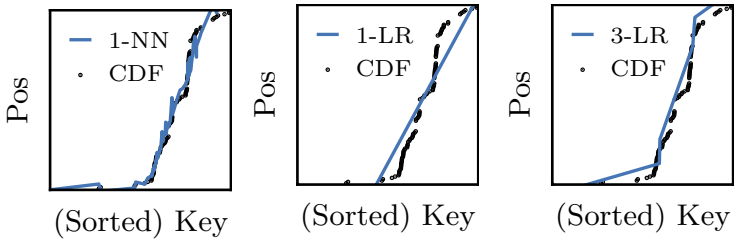
For linear models (LR and MLR), we use `LAPACKE_dgels` in MKL for training. `LAPACKE_dgels` directly calculates the optimal parameters of linear models, which has two benefits compared to the common iterative training procedure (e.g., Stochastic Gradient Descent [122]). First, training models with `LAPACKE_dgels` is fast. For example, it can train a 1000-KVs dataset in less than $50\mu s$. In comparison, iterative training may take several seconds to converge. Second, training linear models with `LAPACKE_dgels` is automatic; i.e., the designer does not need to tune the hyperparameter to train the optimal parameters. Automatic training is particularly important in `XStore` because the sub-model is retrained by the server automatically (§3.4.3) under dynamic workloads. For predictions of linear models, we use floating-point arithmetic to implement them since they are simple (i.e., simple multiplications and additions).

Since we cannot directly calculate the optimal parameters of NN, we train it using stochastic gradient descent with Pytorch, which is efficient and powerful on training. However, we found Pytorch is not suitable for prediction in `XStore` due to its high invocation cost. For example, it takes 31μs to predict with a two-layer NN with one hidden layer of 16 neurons. This invocation latency is an order of magnitude higher than the latency of RDMA (2μs). Kraska et al. [98] has also made a similar observation on TensorFlow [123], another popular ML framework. Since existing ML frameworks are not optimized for ultra-low latency prediction, we manually implement the NN prediction by first extracting the learned parameters from Pytorch, and then implement the forward pass using lightweight MKL. For small NN models (e.g., a two-layer NN with a hidden layer of 64 neurons), `XStore` can finish prediction within 1μs.

3.5.2 ML Model selection

To best utilize XMODEL, the designer should balance predict accuracy, memory consumption and retraining cost when selecting ML models. Using linear models that are efficient on training (§3.5.1) is a common setup for prior learned index [98, 114-115]. However, linear models cannot learn sophisticated data distribution well. Using complex models like NN can improve the accuracy. However, they require more memory to store the parameters and are slower on (re-)training.

We now describe our experiences in choosing ML models for XMODEL from sub-model (level 1) to top-model (level 0). Our focuses are on linear models (i.e., LR and MLR)) and Neural Nets (NN).



Setup	Train	Mem	Err
1-NN	3s①	228B	27
1-LR	20μs	8B	46
3-LR	20μs	24B	15

Figure 3–12 A comparison of the learned cumulative distribution function (CDF), the retrain speed (**Train**), the memory usage (**Mem**), and the prediction error (**Err**) using different sub-model setups on a 100-key OpenStreetMap [100] dataset. The keys are float vectors of dimension 2 (total 8B). **Setup**: **1-NN** uses one three-layer fully-connected NN where each hidden layer has 4 neurons as the sub-model, and the NN uses ReLU as the activation function; **1-LR** uses one LR as the sub-model; **3-LR** uses three sub-models where each sub-model uses LR.

Sub-model. `XStore` chooses the LR for the sub-model due to the following two observations. First, the sub-model has tight memory constraints: XMODEL typically uses one sub-model to predict tens or hundreds of keys. Complex models like NN is not memory efficient for such small datasets since it still needs a non-trivial amount of memory to achieve high accuracy. As shown in Figure 3–12, NN achieves 41% smaller prediction error than LR at the cost of 29X more memory ③.

Second, a sub-model is frequently retrained under the dynamic workload. Thus, XCACHE requires a high retrain speed of sub-model. However, NN is much slower on training compared to linear models. As shown in Figure 3–12, NN uses 3s to converge to the optimal parameters. Consequently, it is not practical to dynamically retrain NN when there are insertions.

One might wonder whether using LR alone can learn complex dataset well. We found LR does have high prediction error (46) for the 100-key irregular OpenStreetMap dataset (Figure 3–12). To remedy this, we suggest using more models to reduce the overall prediction error, which is typically more memory-efficient than NN. This is based on the observation that reducing the number of keys partitioned to a sub-model can significantly reduce the sub-model prediction error in a recursive-model index [98]. As shown in Figure 3–12, using three LR sub-models achieves a smaller prediction error (15) compared to NN, while it only uses 11% of the NN memory.

② We use CPU to train the NN in this experiment. We tried with GPU but found that it is slower. Since the sub-model model and training-set are small, invoking GPU and copying data to GPU will dominate the training performance.

③ We tune the NN architecture to achieve the best accuracy within a limited memory budget (256B).

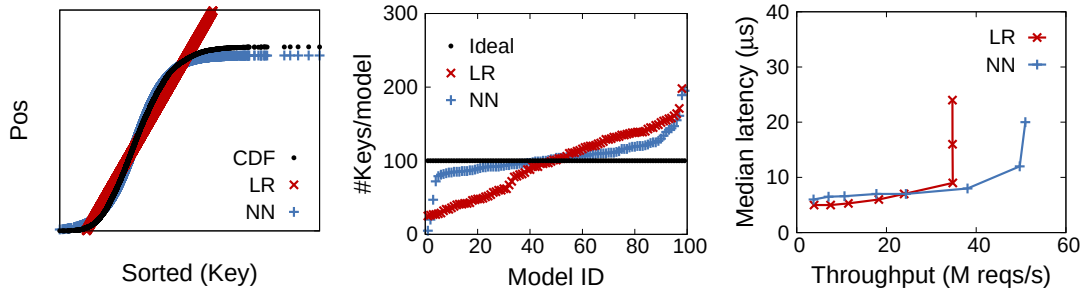


Figure 3–13 A comparison of Neural Nets (NN) and Linear Regression (LR) for the top-model of XCache on a lognormal dataset with 10,000 keys. The keys are 8B integers. The NN has one hidden layer with 8 neurons using *sigmoid* activation, while the XModel has 100 sub-models. We skip MLR in this experiment because it has the same results as LR. (a) The CDF of the dataset and the learned CDF of different models. (b) The number of keys partitioned to each sub-model. (c) A comparison of throughput-latency using different top-models on a 100% read workload.

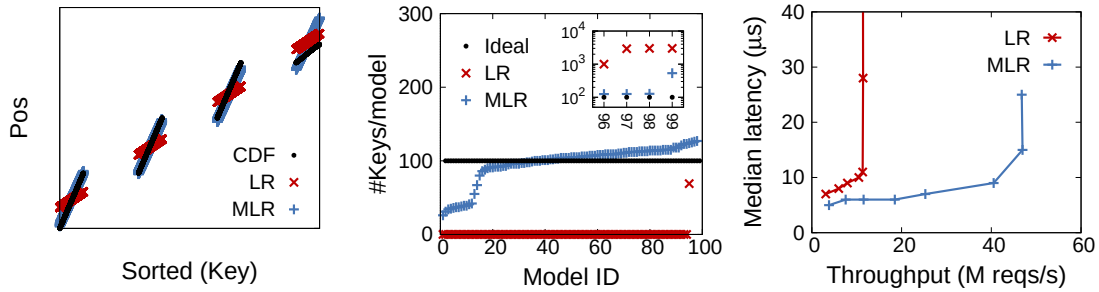


Figure 3–14 A comparison of Multi-variate Linear Regression (MLR) and Linear Regression (LR) for the top-model of XCache on the OrderLine table in TPC-C with 10,000 keys. The keys are 8B integers. The MLR model uses 4 predictors, while the XModel has 100 sub-models. We skip NN in this experiment because MLR is sufficient to learn the distribution. (a) The CDF of the dataset and the learned CDF of different models. (b) The number of keys partitioned to each sub-model. (c) A comparison of throughput-latency using different top-models on a 100% read workload.

Top-model. Different from sub-model, we suggest using complex model for the top-model to achieve high accuracy, so as to evenly partition keys into sub-models (§3.4.1). Top-model can use a complex model because it has less memory constraint and is rarely retrained in `XStore`. More specifically, `XStore` only uses one top-model for all the keys. Thus, the increase of memory usage in top-model is negligible to the XCache. Further, the top-model is only retrained upon model expansion (see §3.4.3), which is rare.

To see how the top-model accuracy affects the overall performance of `XStore`, Figure 3–13 and Figure 3–14 compare the performance of `XStore` with different top-model setups using lognormal dataset and TPC-C dataset [124], respectively. The lognormal

dataset is widely evaluated by prior learned index [98, 114, 116]. As shown in Figure 3–13c, XStore has 1.46X higher peak throughput when using NN as the top-model compared to LR. LR does not evenly partition keys to sub-models since it cannot perfectly learn the non-linear lognormal dataset (Figure 3–13a). As a result, it has a greater variance of the number of keys partitioned to each sub-model compared to NN (3751 vs. 640 in Figure 3–13b). This greater variance causes a larger prediction error compared to NN (11.5 vs. 8.1). Note that using NN as the top-model consumes 12% more XMODEL memory than LR.

Figure 3–14 further presents the results on TPC-C, the de facto standard for OLTP workloads. We generate the dataset by sampling keys from the OrderLine table in TPC-C. The table uses four fields for the primary key: Warehouse ID, District ID, Order ID, and OrderLine ID. Although each field in the primary key follows a linear distribution, the overall key as a binary is non-linear (see Figure 3–14a). Thus, it cannot be effectively learned by LR. As shown in Figure 3–14b, LR partitions most keys to a few sub-models and leaves others empty, resulting in a large variance of the number of keys partitioned to each sub-model. Meanwhile, we can use MLR with four predictors to model the TPC-C OrderLine key distribution. As a result, XStore with MLR achieves 4.1X higher peak throughput than XStore with LR at the cost of 1% more XMODEL memory.

3.6 Discussion

Support variable-length keys. XStore currently supports fixed-length key and variable/fixed-length value. This is not a significant constraint for distributed transactions, since their data typically has a schema.

To support variable-length key, XStore should store a fat pointer in the leaf node of XTREE (instead of the actual key), which encodes the size and position of the key. Though this scheme can traverse variable-length key locally by CPUs (i.e., server-centric design), it would be hard to do it efficiently by using one-sided RDMA READs (i.e., client-direct design). This is because XStore has to retrieve the actual keys using an additional RDMA READ for each (line 21 in Figure 3–8). Therefore, XStore further stores a fixed hash code of the key within the fat pointer. Consequently, the client could directly compare the hash codes instead of keys, after fetching the leaf node for a given key. Note that the actual (variable-length) key should be checked to avoid a hash collision. For example, the client can fetch the value associated with the key.

Table 3–1 YCSB workload description. **R**, **U**, **I**, **M**, and **S** denote read, update, insert, read-modify-update, and scan, respectively. Scan accesses `N` values, where `N` is uniformly distributed in [1,100].

YCSB	**A**	**B**	**C**	**D**	**E**	**F**
Type	R : U	R : U	R	R : I	S : I	R : M
Ratio (%)	50 : 50	90 : 10	100	95 : 5	95 : 5	50 : 50

Table 3–2 Data distribution description for evaluating datasets.

Name	**Description**	**Workloads**
L	Linear	YCSB[99], Nutanix[101]
NL	Noised linear	YCSB[99]
OSM	Longitude location	Open Street Map[100]

3.7 Evaluation

3.7.1 Experimental Setup

Without explicit mention, we evaluate `XStore` on the **VAL** cluster (see Table 2–1), and use one server machine and (up to) 15 client machines for `XStore`.

Workloads. We use YCSB [99] and two production workloads from Nutanix [101]. Our main focus is on YCSB as it contains various types of workloads [125]: update heavy (A), read mostly (B), read only (C), read latest (D), short ranges (E), and read-modify-write (F). Table 3–1 shows a summary of YCSB workloads (A-F). Since small requests dominate in real-life workloads [126], we evaluate KV stores with 100 million KV pairs initially (a 7-level tree-based index and a leaf level), where 8-byte key and 8-byte value are used, similar to prior work [28, 65, 104, 114]. Both Uniform and Zipfian key distributions are evaluated for all YCSB workloads. Note that YCSB D only has Uniform and Latest key distributions; the client is likely to query its recently inserted keys in Latest distribution. In addition, each client generates their insert key uniformly and randomly in YCSB D and E. The two production workloads both have a profile of 57:41:2 write:read:scan ratio, while the access patterns of them are relatively uniform (Prod1) and skewed (Prod2), respectively. Both of them have 500 million KV pairs with 8-byte key and 64-byte value. Finally, besides the default data distribution of the above workloads, we also use two synthetic and one real-life datasets (see Table 3–2) to study the behavior of learned cache in depth.

Comparing targets. We compare `XStore` to three state-of-the-art RDMA-based ordered KV stores: DrTM-Tree [18] and eRPC+Masstree [65] (*server-centric* design), as well as Cell [28] (*client-direct* design). eRPC+Masstree (EMT) adopts eRPC [65] (RDMA-based RPC library) to extend Masstree [104] (in-memory ordered KV store). We implement DrTM-Tree and Cell in the same framework to provide an apple-to-apple comparison with two typical designs, but also because DrTM-Tree uses similar B^+tree [3] and RDMA library [58] with `XStore`, and Cell is not open-source.① We further consider RDMA-Memcached v0.9.6 [127] (RMC) in our experiments, which is an RDMA version of memcached [128], a widely used network-attached KV in industry.

All systems fully utilize all of the 24 CPU cores (with hyperthreading disabled) and two RNICs. As EMT and RMC cannot use multiple NICs simultaneously, we deploy two instances at the server on different sockets, and each instance uses the RNIC attached to that socket. This actually makes them faster during experiments since it avoids cross-socket synchronizations. `XStore` uses (up to) two auxiliary threads to train ML models in the background for dynamic workloads. XTree is configured with a fanout of 16. XModel uses 500K sub-models for static workloads and 2M models for dynamic workloads to avoid model expansion during evaluation (because `XStore` can insert more than 150M KV pairs in 60s). In addition, without explicit mention, we disable logging in all systems and evaluate `XStore` with all the optimizations (e.g., speculative execution) enabled.

3.7.2 YCSB performance

Figure 3–15 compares the peak throughput of various RDMA-based key-value stores for YCSB with Uniform and Zipfian distributions. Note that RMC performs poorly in all experiments as it is bottlenecked by CPU synchronizations [64, 96]. So we omit the discussion of it.

Read-only workload (YCSB C). For Uniform distribution, `XStore` can achieve 82 million requests per second, even a little higher than the *optimal* throughput (a whole-index

① For DrTM-Tree, our experimental results were confirmed by the authors. For Cell, we follow the same caching strategy—the client caches nodes at least four levels above the leaf node at the clients with LRU policy to minimize churn and maximize hits. Based on a comparison against published numbers, we believe that the large performance difference between `XStore` and other systems (e.g., 27M reqs/s from our implementation vs. 0.95M reqs/s from Cell [28] for YCSB A with Zipfian distribution) offsets performance variations due to system and implementation details.

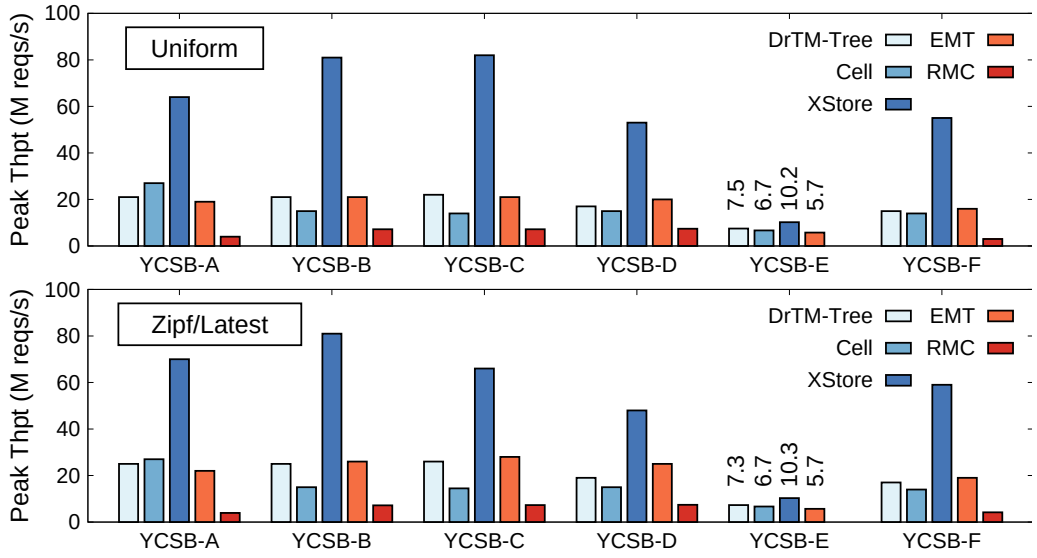


Figure 3–15 Comparison of throughput on various RDMA-based KVs using YCSB. Note that RMC does not support range queries.

cache), since it only uses one RDMA READ to fetch one leaf node per lookup; the payload is 16B smaller by avoiding a sophisticated mechanism for consistency (i.e., min-max fence keys [28]). The prediction error of XCache is just 0.74. This number outperforms EMT, DrTM-Tree, and Cell by 3.9×, 3.7×, and 5.9×, respectively. Both DrTM-Tree and EMT are bottlenecked by server CPUs, while Cell is bottlenecked by RDMA amplifications; it still needs four RDMA READs to traverse tree nodes even index caching is enabled.

For Zipfian distribution, `XStore` can still outperform EMT, DrTM-Tree, and Cell by 2.4×, 2.5×, and 4.6×, respectively. The systems with server-centric design perform better due to better CPU cache locality. However, the peak throughput of `XStore` drops by 18% since RDMA has relatively poor performance when massive clients read a small range of memory simultaneously. We suspect that our current RNIC (ConnectX-4) checks conflicts between one-sided RDMA operations based on request's address [66], so that these operations may compete for NIC's internal processing resources, even if there is no conflict.

Static read-write workloads (YCSB A, B, and F). For update-heavy workloads (YCSB A), `XStore` is still bottlenecked by server CPUs for handling updates. However, compared to server-centric KVs (e.g., DrTM-Tree and EMT), the clients in `XStore` can directly perform read requests with the help of learned cache, which completely bypasses

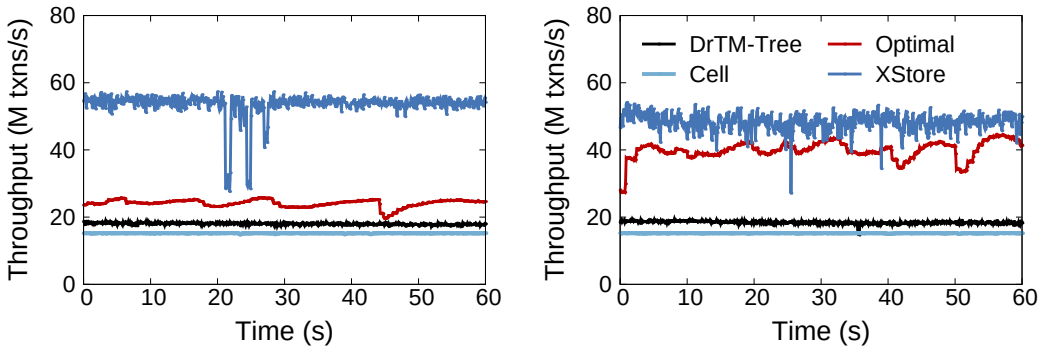


Figure 3–16 The performance timeline of YCSB D with (a) Uniform and (b) Latest workloads.

server CPUs. Therefore, `XStore` can still provide up to 3.1× and 3.2× (from 2.4× and 2.6×) throughput improvements for Uniform and Zipfian distributions, respectively, compared to other KVs. For read-mostly workloads (YCSB B), the speedup of throughput in `XStore` further reaches up to 5.4× (from 3.4×). There are two reasons: (1) the read requests are less skewed interleaved with (10%) updates, compared to read-only workloads (YCSB C); (2) the server of `XStore` has not been saturated (less than 40% of CPU utilizations); thus it is still sufficient to perform updates, compared to update-heavy workloads (YCSB A). The performance of `XStore` on YCSB F is somewhere in between since it has about 75% reads.

Dynamic workloads (YCSB D and E). The throughput of every system is impacted by dynamic workloads due to the contention between reads and inserts. For DrTM-Tree and EMT, the contention happens on the tree-based index. For `XStore` and Cell, the performance slowdown is mainly due to cache invalidations. However, Cell only caches the top four levels, where node split is rare. The overhead in `XStore` mainly comes from two parts: (1) cache invalidations would increase RDMA operations due to fallbacks (RDMA-based RPC) and speculative execution (50% one more RDMA READ); (2) a dynamic dataset is always harder to learn than a static dataset due to the randomly inserted new keys; the prediction error would stably increase to 8.3 for YCSB D.① Fortunately, the clients can still use stale learned cache for most read requests, and model retraining is also very fast. Thus, for YCSB D, `XStore` can provide up to 3.5× and 3.2× (from 2.7× and 1.9×) speedup and achieve 53M and 48M reqs/s throughput for Uniform and Latest distributions, respectively. For YCSB E, the performance is dominated by scanning a

① The data distribution of dynamic workloads (i.e., YCSB D and E) is close to noised linear (NL). Hence, `XStore` can only achieve 61M reqs/s for YCSB D with 2M models even no inserts (see Figure 3–20b and Figure 3–23a).

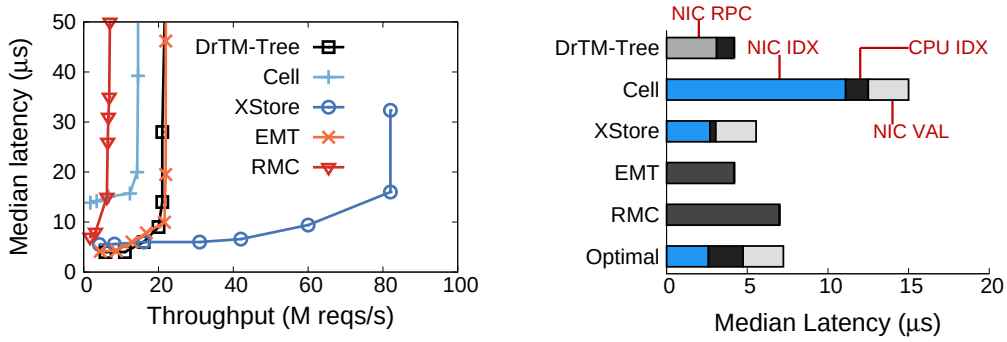


Figure 3–17 Comparison of (a) throughput-latency and (b) end-to-end median latency at low load for YCSB-C with a uniform distribution.

large range of KV pairs. Thereby the performance difference is relatively small, and `XStore` outperforms other systems by up to 1.8× (from 1.4×).

Figure 3–16 further shows the timelines for YCSB D with Uniform and Latest workloads. The optimal throughput of tree-based index cache can only achieve about 25M reqs/s, more than 3× lower than its read-only throughput (78M reqs/s), and suffers from severe performance fluctuations due to frequent cache invalidations, especially for Uniform distribution. For Latest distribution, each client will focus on a small range of KV pairs (latest inserted by itself), which significantly reduces cache misses and invalidations due to accessing internal nodes split by other clients. `XStore` preserves relatively high throughput and has steady cache invalidation rates, 5% for Uniform, and 21% for Latest. It is mainly because stale learned cache can still provide a correct prediction for most read requests. The speculative execution also helps to halve the rate (from 10% to 5%). In addition, in Latest distribution, each client will frequently access KV pairs just inserted. If the insert incurs a node split, `XStore` might not fetch a new model immediately (wait for model retraining) and would increase cache misses.

CPU utilizations of `XStore`. Note that `XStore` uses two auxiliary threads to retrain XMODEL for dynamic workloads, causing increased server CPU usage. Yet, `XStore` still saves server CPUs compared to server-centric KVs (e.g., DrTM-Tree) due to handling read requests in the clients. For example, DrTM-Tree saturates all CPUs (24 × 100%) for YCSB D, while `XStore` just consumes under half for serving insert requests and retraining sub-models.

End-to-end latency. Figure 3–17a shows the throughput-latency curves for YCSB C with a uniform distribution. Due to space limitations, we omit other workloads that are

similar. When using few clients (low load), server-centric KVs have lower latency, as one RPC round trip is faster than two one-sided RDMA operations, namely DrTM-Tree (NIC_RPC) vs. `XStore` (NIC_IDX and NIC_VAL) in Figure 3–17b. However, the throughput of them (e.g., DrTM-Tree) is saturated by CPUs much earlier (about 20M reqs/s), and the latency would rapidly collapse. On the other hand, the latency of Cell is limited by multiple RDMA READs for each lookup (NIC_IDX) even at low load. In contrast, `XStore` only needs one RDMA READ, thanks to the learned cache. As a reference, we provide the latency of using whole-index cache (Optimal) that also takes just one RDMA READ. However, traversing tree-based index locally still takes more time (2.14µs in CPU_IDX) due to many random memory accesses, compared to `XStore` (0.35µs). Moreover, `XStore` can keep low latency at much high load (82M reqs/s with median latency of 16µs) by eliminating CPU bottleneck at the server.

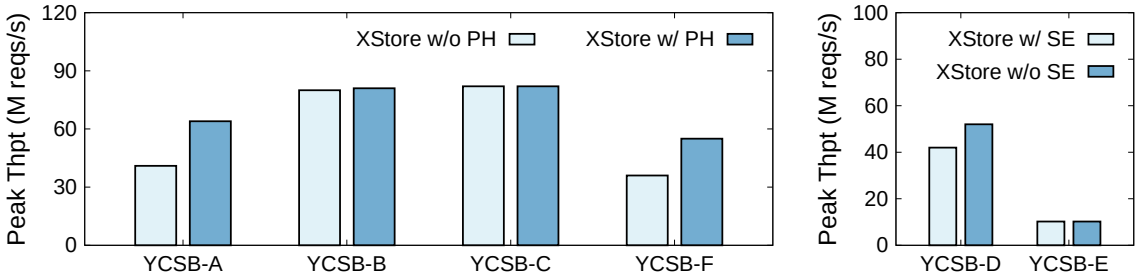


Figure 3–18 *Effects of optimizations (a) position hint (**PH**) to* UPDATE *and (b) speculative execution (**SE**) to* INSERT *on YCSB uniform workload.*

3.7.3 Effects of optimizations

Position hint. Reducing server-side tree traversal with client-side XCACHE is effective for the server-centric UPDATE. As shown in Figure 3–18a, position hint improves the performance of YCSB A, B and F by 1.56X, 1.01X and 1.53X, respectively. It is mostly effective in YCSB A, which is a write-heavy workload with 50% updates. On the other hand, it does not affect YCSB C since YCSB C is a read-only workload.

Speculative execution. Figure 3–18b exams the impact of speculative execution. It improves YCSB D by 1.23X thanks to the reduced invalidation rate (§3.7.2) under dynamic workloads. On the other hand, speculative execution has little effect to YCSB E because YCSB E has nearly no invalidations: the performance of YCSB E is dominated by scanning a range of KV pairs.

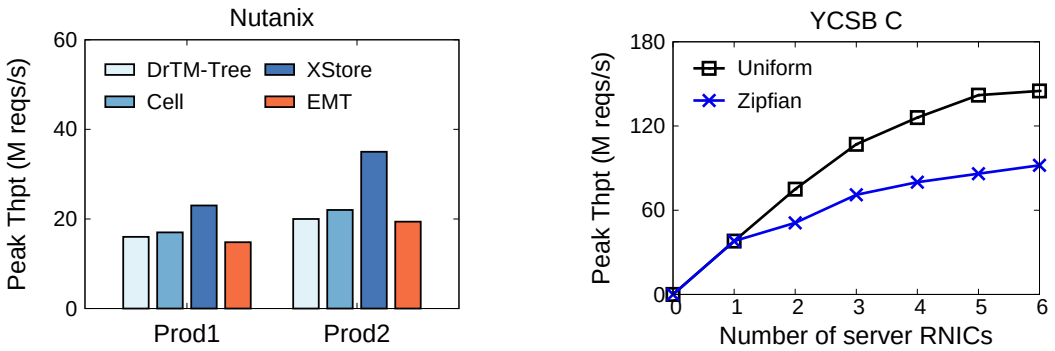


Figure 3–19 (a) Performance comparison with production workloads. (b) Scalability of `XStore` on YCSB C with the increase of RNICs.

3.7.4 Production workload performance

Figure 3–19a shows the peak throughput of `XStore` and other systems on two write-intensive production workloads, similar to YCSB A. The performance is also mainly bottlenecked by server CPUs due to 57% of writes. In the first workload (Prod1), `XStore` outperforms DrTM-Tree, EMT, and Cell by 1.44×, 1.55×, and 1.35×, respectively. The speedup in the second workload (Prod2) increases to 1.75×, 1.80×, and 1.60× since this workload is more skewed.

3.7.5 Scale-out performance

Figure 3–19b shows the scalability of `XStore` with up to 6 server RNICs (3 server machines). We scale `XStore` by range-based partitioning a YCSB dataset with 600M keys into different numbers of RNICs. The performance is measured using up to 13 client machines (26 RNICs) with a read-only workload. For a uniform request distribution, `XStore` achieve a peak throughput of 145M reqs/s, which is limited by the number of client machines. Note that, on our testbed, `XStore` needs about eight client RNICs to saturate one server RNIC. `XStore` scales to 1.97× and 2.81× by using 2 and 3 server RNICs, respectively. For a skewed request distribution (Zipfian), `XStore` just reaches 92M reqs/s by using 6 server RNICs since most requests (more than 35%) are sent to one RNIC. It throttles the entire system.

3.7.6 Model (re-)training and expansion

Figure 3–20a shows the throughput of training models using one or two threads and model invalidation speed for dynamic workloads (YCSB D and E). Empirically, using

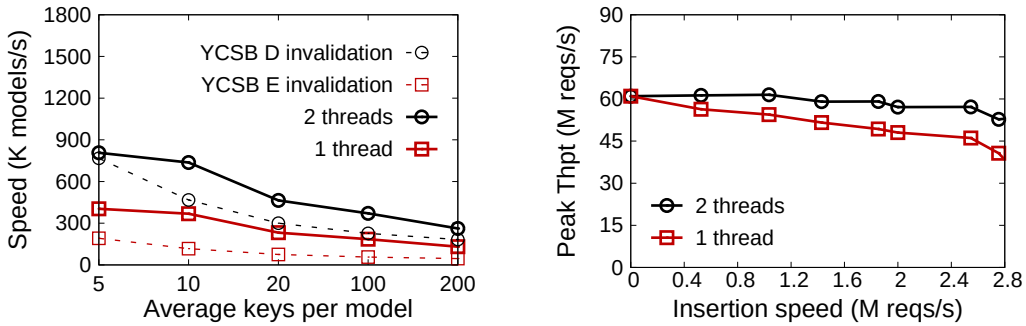


Figure 3–20 (a) Comparison between sub-model retraining and invalidation speed. (c) Performance of `XStore` with the increase of insertion speed.

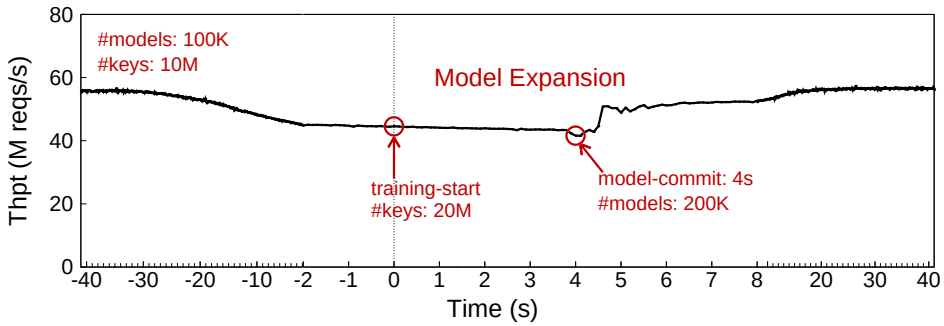


Figure 3–21 Performance timeline with model expansion.

two threads for model retraining is sufficient for `XStore` to reach a throughput of 53M reqs/s (YCSB D). `XStore` can retrain sub-models individually and takes 8µs on average to retrain a model with 200 keys. Note that the insertion speed reaches about 2.65M reqs/s for YCSB D (5% inserts). For dynamic workloads, the throughput of `XStore` would decrease when stale sub-models can not retrained in time. To quantify the performance overhead, we evaluate `XStore` with the increase of insertion speed, similar to YCSB D (except that one client is dedicated to insert key-value pairs with a given speed, and the rest of clients still issue reads). As shown in Figure 3–20b, the throughout drops below 40% (61M vs. 37M reqs/s) under the peak insertion speed (2.8M reqs/s, limited by server CPUs) when using a single retraining thread. Further, when using two threads, the performance degradation is limited to 13%.

Finally, the growing size of KV pairs in the ML model will likely increase the prediction error, resulting in performance degradation. `XStore` supports model expansion to increase models in the background if needed. As shown in Figure 3–21, starting from 10M keys and 100K models, several clients continuously insert KV pairs, and the perfor-

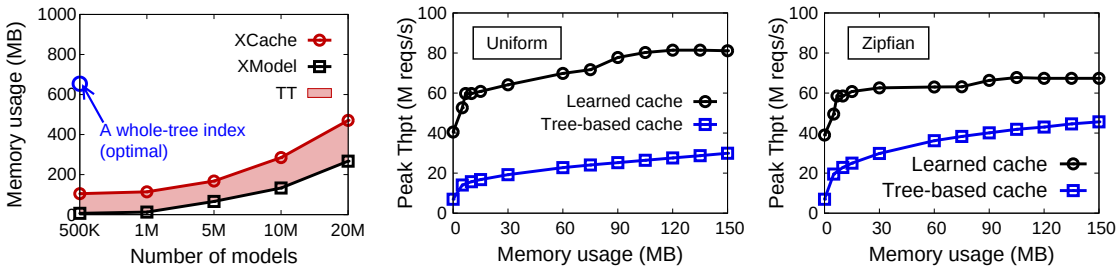


Figure 3–22 (a) Memory usage of learned cache (XCache). Comparison of peak throughput between learned cache and tree-based cache with different memory footprint at the client for YCSB C using (b) Uniform and (c) Zipfian distributions.

mance of `XStore` slowly degrades for read requests. When the average number of keys per model exceeds 200 (a user-defined threshold), the server starts to train a new XModel with double sub-models (200K) in the background from 0s to 4s, with negligible overhead. After that, the server will commit the new model, and clients could individually fetch new sub-models on demand. The performance resumes rapidly in 2s.

3.7.7 Memory footprint of XCache

Figure 3–22a presents the memory usage of XCache with the increase of sub-models for 100M KV pairs. Note that the entire XTree has 654MB internal nodes. The size of TT depends on the number of leaf nodes. Since each leaf node has 16 slots for KV pairs, TT occupies around 98MB as the tree-based index is half-full. Thus, TT would dominate the memory usage for a small XModel since each sub-model is 14B large. To achieve peak throughput, XModel with 500K sub-models is enough for read-only workloads (YCSB C) with 100M KV pairs, while it needs 2M sub-models for dynamic workloads (YCSB D) with 250M KV pairs.

As shown in Figure 3–22b and Figure 3–22c, compared to conventional tree-based index cache, `XStore` can provide competitive performance with much lower memory footprint at the clients, even (almost) no memory footprint. XCache prefers to store XModel, which may only occupy 1% memory (6.8M vs. 654MB). It means that, for YCSB C with Uniform and Zipfian distributions, `XStore` can achieve 74% and 87% of optimal throughput (a whole-index cache), where the client uses one additional RDMA READ to fetch several 8-byte TT entries for each lookup. Even if the client only stores a 16-byte top model, `XStore` can still achieve about 40M reqs/s by using one RDMA READ to fetch a 14-byte sub-model first.

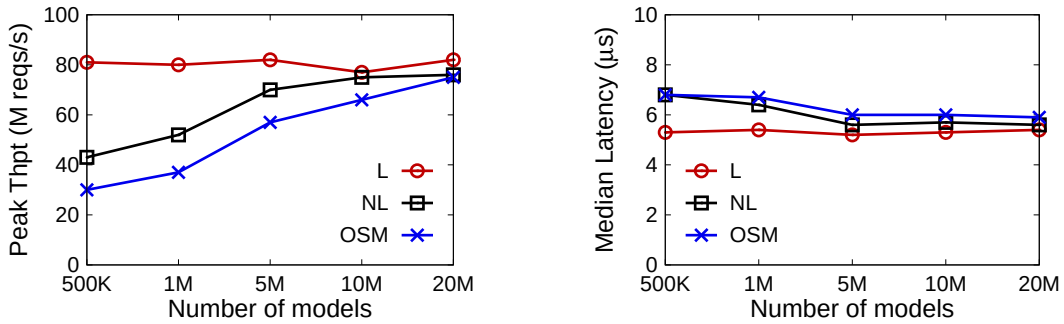


Figure 3–23 Comparison of (a) peak throughput and (b) median latency on `XStore` with the increase of models for various data distributions (see Table 3–2).

Table 3–3 The impact of durability on the throughput (M reqs/s) of `XStore`.

YCSB *(Uniform)*	**A**	**B**	**C**	**D**	**E**	**F**
w/o logging	41	80	82	53	10.2	36
w/ logging	31	78	82	51	9.9	33

3.7.8 Data distribution

We further evaluate `XStore` on a 100M-key dataset with different data distributions (Table 3–2) using a read-only workload (YCSB C). Since the throughput of `XStore` is sensitive to the prediction error due to bandwidth amplification, `XStore` requires more simple sub-models (e.g., LR) to learn complex data distributions (e.g., OSM) for the same prediction error. For example, as shown in Figure 3–23a, `XStore` requires about 20M sub-models for OSM to achieve a peak throughput of 80M reqs/s. On the other hand, the median latency at a low load is relatively stable (see Figure 3–23b) for various data distributions, as the latency of RDMA is insensitive to payload sizes [35].

3.7.9 Durability

To study the overhead of logging for durability, we evaluate the peak throughput of `XStore` for various YCSB workloads with logging enabled. As shown in Table 3–3, the performance drops by up to 24% for update-heavy workloads (e.g., YCSB A) due to additional writes to SSD. On the other hand, it does not degrade the performance of read-heavy workloads much (e.g., YCSB C), because first first, `XStore` executes read operations (e.g., GET) using one-sided RDMA primitives bypassing the logging threads thoroughly. Second, `XStore` flushes the logs in a batched manner [104] to hide the impact of slow storage (§3.4.4).

3.8 Related work on learned index

The learned cache used in XStore is motivated by earned index. Specifically, Kraska et al. [98] argue that all existing index structures can be replaced with machine learning (ML) models, which are termed "learned index". They further propose several example learned indexes for various index structures, e.g., tree-based range index. As the original learned indexes focus on static workloads, there have been several efforts of adapting learned indexes to handle dynamic workloads [114-115, 129]. XIndex [114] adds a delta index to each sub-model in a learned index and proposes a new concurrent RCU-based compaction scheme to split models. Adapting RCU to XStore for dynamic workloads is inefficient, because updating the index after model retraining requires synchronizing with all the clients in this approach. ALEX [115] uses a gapped array to accommodate new key-value pairs, similar to the leaf node of XTree. However, it is non-trivial to enable the gapped array in a distributed xstoretem since it requires complex coordinations when expanding the array upon full. Bourbon [116] is a log-structured merge (LSM) tree that leverages the learned index to speedup lookups. FITing-Tree [111] is a form of a learned index to balance prediction error and memory cost. It uses extra sorted buffers to store inserts and merges them back when reaching a threshold. SIndex [130] is a concurrent learned index for variable-length string keys.

3.9 Conclusion

XStore is an RDMA-based in-memory ordered key-value store with a new *hybrid* architecture to leverage ML model as RDMA-based index cache. We term this cache as "leaned cache". With the help of the learned cache, XStore avoids the costly index traversal in traditional RDMA-based ordered key-value store. By maintaining a layer of indirection, XStore further decouples the ML model retraining from index updating and allow a stale learned cache to continue predicting a correct position of a lookup key. To minimize the cost of learned cache misses, XStore was built with a set of optimizations like speculative execution. Evaluations with YCSB benchmarks and production workloads confirmed the benefit of designs in XStore compared to the state-of-the-art RDMA-based ordered key-value stores.

Distributed transactions can leverage XStore to accelerate data retrieval over secondary indexes, where ordered accesses are common. We will discuss it in §6.1.

Chapter 4 Phase-by-phase Analysis for Hybrid RDMA-enabled Concurrency Control

This chapter presents DrTM+H, a distributed transaction processing system designed with RDMA. Distributed transaction used to seem slow [131]. Yet,the prevalence of fast networking features such as RDMA (§2.2) has boosted the performance of distributed transactions by orders of magnitudes [13, 17-18, 25].

Though there are many transactional systems designed with RDMA (§4.1), it is often challenging for system designers to choose the *best* design, due to the lack of a systematic study on these designs. People are also debating over how to choose RDMA primitives for transactions recently (§4.2). To this end, DrTM+H conducts the first systematic study on how different choices of RDMA primitives and designs affect the performance of distributed transactions.① Our phase-by-phase analysis reveals that none of them has the optimal performance (§4.3). Based on the analysis results, DrTM+H can provide better performance than the prior designs (§4.4).

4.1 Background on RDMA-enabled distributed transactions

There is an active line of research in using RDMA for serializable distributed transactions [13, 18, 25]. Most of such systems use variants of optimistic concurrency control (OCC) for consistency [22] and variants of primary-backup replication (PBR) for availability [132]. PBR uses fewer round trips and messages to commit one transaction than Paxos [13], which fits distributed transactions in a well-connected cluster.

Although these systems have different design choices and leverage different RDMA primitives, they use a similar transaction protocol (OCC)② to execute and commit serializable transactions. The operations performed in the protocol can be briefly summarized as four consecutive phases, as shown in Figure 4–1. A transaction first executes by reading the records in its read set (**Execution**). Then it executes a commit protocol, which locks the records in the write set and validates the records in the read set

① Note that optimizing distributed transaction protocol is not the focus of this chapter.

② While DrTM [17] implements a two phase locking (2PL) scheme using HTM and RDMA, it provides no high availability support and a later version [18] uses a variant of OCC to provide high availability. We are not aware of other RDMA-enabled distributed transaction systems using 2PL. Hence, we focus on OCC in this chapter.

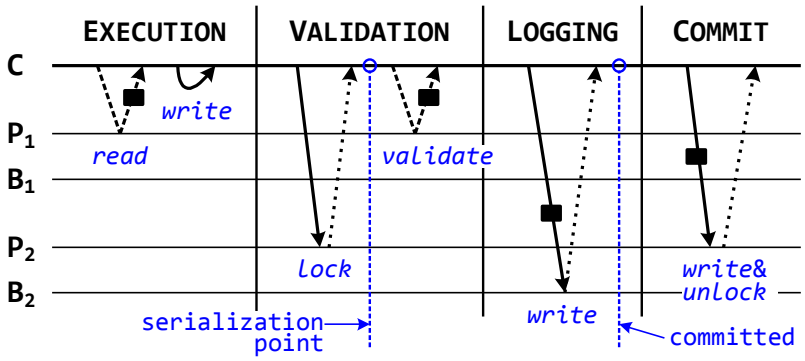


Figure 4–1 A phase-by-phase overview of transaction processing with OCC. **C**, **P**, and **B** stand for the coordinator, the primary and the backup of replicas, respectively. $\mathbf{P}_1$ is read and $\mathbf{P}_2$ is written. The dashed, solid, and dotted lines stand for read, write, and hardware ack operations, and rectangles stand for record data.

is unchanged (**Validation**). If there is no conflicting transaction, the coordinator sends transaction's updates to each backup and waits for the accomplishment (**Logging**). Upon successful, the transaction will be committed by writing and unlocking the records at the primary node (**Commit**). Note that the execution order of the protocol is very important. For example, the transaction is considered to be committed if and only if the log replies have been received [13, 25]. Thus the commit phase must be executed after the completion of logging.

Read-only transaction is an important building block for modern applications [59]. OCC use a two-phase protocol to execute it: the first phase reads all records (**Read**), and then the second phase validates all of them have not been changed (**Validation**).

4.2 One-sided vs. Two-sided: an on-going debate

Recently, there is an active debate over which RDMA primitive, namely one-sided or two-sided, is better suited for distributed transactions. One-sided primitive (e.g., `READ`, `WRITE`, and `ATOMIC`) provides higher performance and lower CPU utilization [2, 13, 17-18]. On the other hand, two-sided primitive simplifies application programming and is less affected by hardware restrictions such as the limitation of RNIC's cache capacity [25, 30].

It is often challenging for system designers to choose the right primitive for transactions based on previous studies. Most work on RDMA-enabled transactions presents

a new system built from scratch and compares its performance with previous ones using other codebases. Some only compare the performance of different primitives or designs using micro-benchmarks. This makes their results hard to interpret: differences in hardware configurations and software stacks affect the observable performance. Further, different RDMA primitives may significantly affect the overall performance [30, 66].

There have been several valuable studies in the database community in comparing different transactional systems [133-134]. Harding et al. [134] conduct a comprehensive study on how different transaction protocols behave under different workloads in a distributed setting using a single framework. However, for a particular protocol, there may be many different implementations which have very different performance, especially when embracing new hardware features like RDMA.

In this dissertation we present the first systematic study on how different choices of RDMA primitives and designs affect the performance of distributed transactions. Unlike most previous research efforts which compare different overall systems, we compare different designs within a single, well-tuned execution framework designed for RDMA (§2.2). Our goal is to provide a guideline on optimizing distributed transactions with RDMA, and potentially, for other RDMA-enabled systems (e.g., distributed file systems [32, 51] and graph processing systems [34-36]).

4.3 A phase-by-phase performance analysis

Our systematic study adopts a phase-by-phase analysis with different RDMA primitives. The goal is to answer how to choose RDMA primitives for RDMA-enabled transactions. A phase-by-phase analysis can effectively find the best choice of OCC-based RDMA-enabled transactions because OCC executes all phases in a strictly serial order (§4.1).

To provide an apple-to-apple comparison on different primitives and transactions, we conduct our analysis on `R2` (§2.2.2). Table 4–1 summaries whether we apply the optimization discussed in §2.2.3 at different phases of transactional execution. Below is some highlights of the analysis results:

- One-sided primitive is faster when the number of round trips is the same and the completion acknowledgement of requests are required (§4.3.1,4.3.2,4.3.4,4.3.5).
- It is always worth checking and filling the lookup cache for one-sided primitive, even using two-sided primitive (§4.3.1).

Table 4–1 A summary of optimizations on RDMA primitives at different phases (§2.2.3). **OR**, **DB**, **CO** and **PA** stand for outstanding request, doorbell batching, coroutine, and passive ACK. **RW** and **RO** stand for read-write and read-only transactions. I and II stand for one-sided and two-sided primitives.

		OR		DB		CO		PA	
		I	II	I	II	I	II	I	II
RW	E	✗	✗	✗	✓	✓	✓	✗	✗
	V	✓	✓	✓	✓	✓	✓	✗	✗
	L	✓	✓	✗	✓	✓	✓	✗	✗
	C	✓	✓	✓	✓	✓	✓	✓	✓
RO	R	✗	✗	✗	✓	✓	✓	✗	✗
	V	✓	✓	✗	✓	✓	✓	✗	✗

- One-sided primitive is faster, even using more network round trips, for CPU-intensive workloads (§4.3.1).
- Two-sided primitive with passive ACK has comparable or better performance than one-sided (§4.3.3).

In this chapter, we focus on in-memory distributed transactional processing that does not provide durability. In §6.3, we will present how to use RDPMA (§2.3) to support durability for DrTM+H using Optane PM.

Benchmarks. We use two popular OLTP benchmarks, TPC-C [124] and SmallBank [135], to measure the performance① of every phase with different primitives, since they represent *CPU-intensive* and *network-intensive* workloads respectively. We use a partitioned data store where data is sharded by rows and then distributed to all machines. We enable 3-way logging and replication to achieve high availability, namely each primary partition has two backup replicas.

TPC-C simulates an order processing application. We scale the database by deploying 384 warehouses to 16 machines. We use this benchmark as a CPU-intensive workload. TPC-C is known for good locality: only around 10% of transactions access remote records. To avoid the impact of local transactions, which our work does not focus on, we only run new-order transaction of TPC-C and make transactions always distributed,

① We scale up the concurrent requests handled by the server to achieve the peak throughput.

which is a major type of transaction (45%) and representative in TPC-C.①

SmallBank simulates a simple banking application. Each transaction performs simple reads and writes operations on account data, such as transferring money between different users. We use this benchmark as a network-intensive workload because transaction only contains simple arithmetic operations on few records. We do not assume locality as previous work [25], which means that all transactions use network operations to execute and commit transactions. To scale the benchmark, we deploy 100,000 accounts per thread, while 4% of records are accessed by 90% of transactions.

Symmetric model. We use a *symmetric model* in our experiments as prior RDMA-enabled distributed transactions [13, 17-18, 25]. In a symmetric model, each machine acts both a client and a server.

4.3.1 Execution (E)

Overview. In the *execution* phase, the transaction coordinator fetches the records a transaction reads. This requires traversing the index structure and fetching the record. We can simply send an RPC to remote server to fetch the record, which only requires one round-trip communication. On the other hand, we can also leverage one-sided RDMA READs to traverse the data structure and read the record. This typically requires multiple round trips but saves remote CPUs. Prior work has proposed two types of optimizations to reduce the number of round trips required by one-sided primitives [2, 17, 28-29].

RDMA-friendly key-value store. Many hash-based data structures can be optimized to reduce the number of RDMA operations for traversing the remote server to find the given key, these include cuckoo hashing [29], hopscotch hashing [2], and cluster hashing [17]. We adopt DrTM-KV [17], a state-of-the-art RDMA-friendly key-value store in all experiments.

RDMA-friendly index cache. The ideal case for one-sided primitive is to use one one-sided READ to get the record back. DrTM [17] introduces a location-based cache to eliminate the lookup cost (one RDMA READ) in the common case. FaRM [13] and Cell [28] use a similar design for caching the internal nodes of B-tree. In our experiment, we maintain a 300MB index cache on each machine, which will be used and filled in the execution

① For brevity, we refer to our simplified TPC-C benchmark as TPC-C/no.

phase. Note that the index cache is quite effective since a relatively small cache is usually enough for skewed OLTP workloads [126, 136-139], such as SmallBank [135], TATP [140], and YCSB [99].

Evaluation. Figure 4–2 compares the performance of using one-sided and two-sided primitives for the execution phase on TPC-C/no and SmallBank, respectively. Two-sided uses one RPC to fetch the record. One-sided fetches records with at least two one-sided READs (one for index and one for payload). One-sided/Cache always fetches the indexes from the local index cache and then get the record from a remote server using a single one-sided READ. This presents an ideal case for the performance of the execution phase using one-sided primitives.

TPC-C/no: One-sided/Cache outperforms Two-sided by up to 1.45X in throughput (from 1.26X), and the median latency is only around 69% of Two-sided (from 89%). The benefits mainly come from the better performance of one-sided READs. Two-sided outperforms One-sided (no cache) by up to 1.28X in throughput. Without caching, the coordinator requires an average of double round trips (one for lookup and another for read) to fetch one record.

Interestingly, when increasing the number of coroutines, the peak throughput of One-sided (no cache) outperforms that of Two-sided (about 13%). The median latency is also slightly better when using more than 10 coroutines. The performance gain comes from lower CPU utilization on each machine. This confirms the benefits of using one-sided primitives when remote servers are busy [28-29]. The adaptive caching scheme in prior work [28-29] can be used to get better performance by balancing CPU and network.

SmallBank: Not surprisingly, One-sided/Cache still outperforms Two-sided by up to 1.36X in throughput due to the better performance and CPU utilization of one-sided READ. However, compared to One-sided (no cache), the speedup of peak throughput for Two-sided reaches up to 2.01X (from 1.13X). This is due to two reasons. First, without location-based cache, one-sided uses more round trips to finish the execution phase. Further, the performance of Smallbank is bottlenecked by network bandwidth since it is a network-intensive workload.

Summary. If one round-trip RDMA READ can retrieve one record using the index cache, one-sided primitive is always a better choice than two-sided one. Otherwise, two-

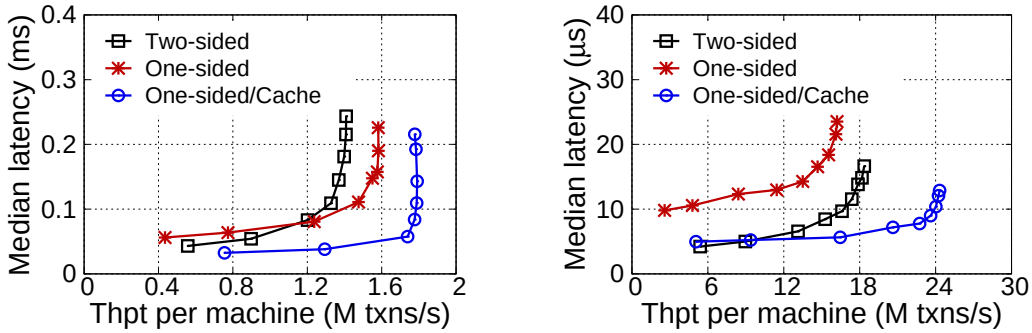


Figure 4–2 The performance of (a) `TPC-C/no` and (b) `SmallBank` with different implementations of Execution phase.

sided primitive should be used when servers are not overloaded. Hence, a *hybrid* scheme should be used in the execution phase. Specifically, we should *always enable the index cache and look from it* before choosing either one-sided primitive (on cache hit) or two-sided primitive (on cache miss). We should also *always refill the cache even if two-sided primitive is chosen upon a miss*.

4.3.2 Validation (V)

Overview. To ensure serializability, OCC atomically checks the read/write sets of the transaction in the validation phase. The coordinator first *locks* all records in the transaction's write set and then *validates* all records in the read/write set to ensure that they have not been changed after the execution phase.

Lock. RDMA provides one-sided *atomic compare and swap* operations (`ATOMIC`), which can be used to implement distributed spinlock [17-18]. Although `ATOMIC` is slower than other two-sided primitives, on recent generation of RNIC (e.g., ConnectX-4), `ATOMIC` can achieve 48M reqs/s, which is sufficient for many OLTP workloads (e.g., `TPC-C`). More importantly, the throughput of two-sided primitive (76M) was evaluated with an empty RPC workload. When locking the record in the RPC routine, the impact of CPU efficiency may change the relative performance of one-sided and two-side primitives. This is especially the case for the symmetric model adopted by transaction systems [13, 17-18, 25-26], when the servers are busy processing transactions.

Validate. Different from the execution phase, a single RDMA `READ` is enough to retrieve the current version of the record for validation, thanks to caching the index in the execu-

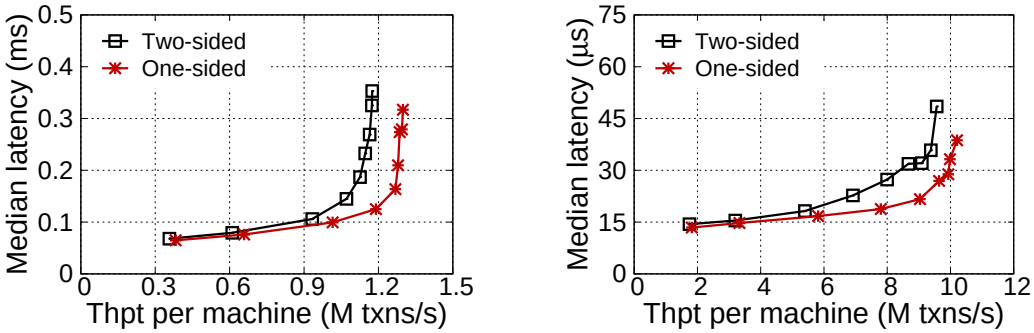


Figure 4–3 The performance of (a) `TPC-C/no` and (b) `SmallBank` with different implementations of locking in Validation phase.

tion phase of the transaction. Therefore, one-sided primitive is always a better choice for read-only records compared to two-sided one, according to the results in §2.2.

Optimization. OCC demands the validation should start exactly after locking all records [7, 13]. This takes two round trips for every read-write record in the validation phase. Fortunately, the locked record can be validated immediately since it can not be changed again. Therefore, each read-write record can be handled by both one-sided and two-sided primitives in one round trip. For one-sided, the RDMA `READ` request will be posted just after the RDMA `CAS` request in a doorbelled way to the same send queue of target QP, since they are processed in a FIFO manner. Further, with passive ACK, the `CAS` request can be made unsignaled (§2.2.3). For two-sided, the RPC routine will first lock the record and then read its version. On commodity x86 processors, compiler fences are sufficient to ensure the required ordering.

Restriction of RDMA atomicity. Currently, the key challenge for using one-sided primitive (RDMA `ATOMIC`) for distributed locking is that `ATOMIC` cannot correctly work with CPU's atomic operations (e.g., CAS). To remedy this, local atomic operations must also use RNIC's atomic operations [17], which will slow down the validation phase of local transactions. Leveraging advanced hardware features, like hardware transactional memory (HTM), can overcome this issue [17].

Evaluation. Figure 4–3 compares the performance of using one-sided and two-sided primitives for the validation phase on `TPC-C/no` and `SmallBank`, respectively. Since the read/write sets are the same in `TPC-C/no` and `SmallBank`, one-sided will send one `ATOMIC` and one `READ` sequentially to lock the record and retrieve the current version in

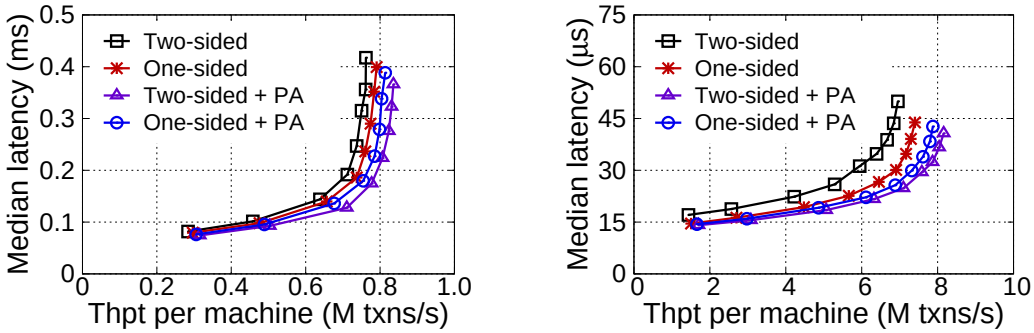


Figure 4–4 The performance of (a) `TPC-C/no` and (b) `SmallBank` with different implementations of Commit phase.

one round trip. We can see in Figure 4–3 that for both workloads, one-sided primitive (`ATOMIC`) is faster, even it has lower peak throughput.

Summary. Although RDMA `ATOMIC` is slower than other RDMA network primitives, it may not be the bottleneck for many applications and can further improve the performance of many workloads. If the atomicity between RNIC and CPU will not cause a performance issue, One-sided RDMA `ATOMIC` is a better choice to implement distributed locking due to high CPU efficiency. Otherwise, two-sided primitive is preferred in this phase since local CASs are much faster than RNIC's CASs.

4.3.3 Commit (C)

Overview. In the commit phase, the coordinator first writes the updates of the transaction back and then releases the locks. One-sided `WRITE` can be used to implement the commit operation with two requests, one to write updates back and one to release the locks (i.e., zeroing the lock state of the record).

Similar to the validation phase, two one-sided `WRITE`s (one to write the update back and one to release the lock) will be posted sequentially to the same QP in a doorbelled way, which preserves the required ordering (release after write-back). Therefore, the commit phase can be handled by both one-sided and two-sided primitives in one round trip.

Optimization with passive ACK. Since the transaction is considered to be committed after the completion of logging, the completion of the commit message can be acknowledged passively by piggybacking with other messages. Thus we enable passive ACK optimization to both one-sided and two-sided primitives in the commit phase.

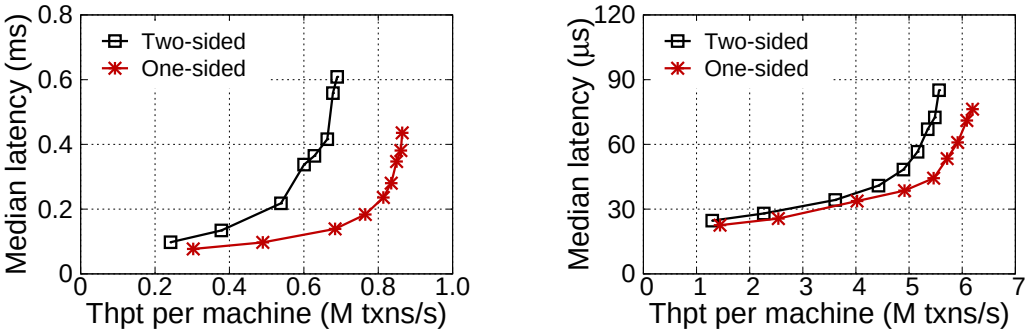


Figure 4–5 The performance of (a) `TPC-C/no` and (b) `SmallBank` with different implementations of Logging phase.

Evaluation. Figure 4–4 presents the performance of `TPC-C/no` and `SmallBank` using different commit approaches. Note that we use two-sided as the validation implementation in this experiment. This is because one-sided ATOMICs cannot work correctly with the commit phase with two-sided primitive due to the atomicity issue with our current RNIC.

For both workloads, without passive ACK, one-sided WRITEs are faster due to better CPU utilization at the receiver's side. With passive ACK, two-sided is faster. This is because, although two-sided primitive costs more CPU at the receiver side, it can save CPU at sender side due to doorbell batching [66] (see Table 4–1). One-sided primitive requires multiple MMIOs to commit multiple records, while two-sided primitive can chain these requests by using one doorbell. Passive ACK can further save the cost of two-sided primitives when sending the replies back. These results match up with the results observed in our primitive-level performance analysis (§2.2.3).

Summary. To commit transactions, two-sided primitive with passive ACK is a better choice.

4.3.4 Logging (L)

Overview. In the logging phase, the coordinator writes transaction logs with all updates to all backups. After receiving the completion acknowledgements from all backups, the transaction commits. The coordinator will notify backups to reclaim the space of logs lazily by updating records in-place.

One-sided primitive. To enable logging with one-sided RDMA `WRITE`, each machine

Table 4–2 A summary of execution time (cycles) and payload size (bytes) in different phases for TPC-C and SmallBank.

	TPC-C		SmallBank	
	Time	Payload	Time	Payload
Execution	342	68	678	71
Validation	454	157	185	105
Logging	363	1006	134	149
Commit	108	34	87	20

maintains a set of ring-buffers for remote servers to log. The integrity of the log is enforced by setting the payload size at the begin and end of the message, inspired by previous work [2]. Note that since we use RC (Reliable Connection) QP to post one-sided WRITEs, the logging is considered success after polling the ACK from the RNIC. We use two-sided primitive to reclaim the log since it must involve remote CPUs [13]. Since log reclaiming is not on the critical path of transactional execution, this request can be marked as unsignaled and the claiming can be done in the background.

Two-sided primitive. Logging with two-sided primitive is relatively simple. The RPC routine copies the log content to a local buffer after receiving the log request, and then it sends a reply to the sender. The log reclaiming can also be executed in the background.

Evaluation. Figure 4–5 presents the performance of TPC-C/no and SmallBank using different logging approaches. For both of them, one-sided logging always has higher throughput and lower latency than its two-sided counterpart, thanks to offloading write operations to one-sided primitives. Using one-sided logging increases the throughput of TPC-C/no and SmallBank by up to 1.29X (from 1.24X) and 1.12X (from 1.10X), respectively. One-sided logging has more improvements in peak throughput in TPC-C/no since the payload size of logs in TPC-C is much larger than that of SmallBank (1,006B vs. 149B), as shown in Table 4–2.

Summary. Since the logging phase can be offloaded using one-sided RDMA WRITEs with one round trip, one-sided primitive is always preferred to write logs.

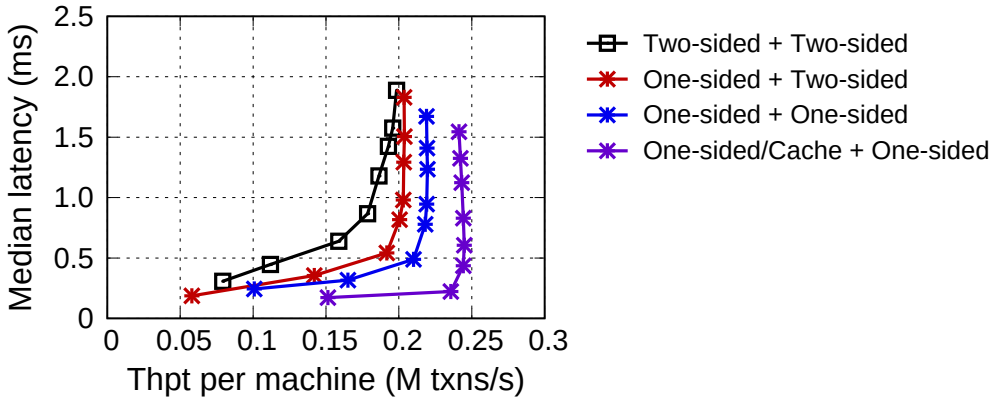


Figure 4–6 The performance of `customer-position` in `TPC-E` with different implementations of the read-only transaction (Read and Validation phases).

4.3.5 Read-only transaction (R+V)

Overview. We use a simplified two-phase protocol to run read-only transactions as prior work [59]. The first phase reads all records like the execution phase, and the second phase validates that the versions of all records have not been changed, which is similar to the operations in the validation phase for the records in read set. For single-key read-only transactions, the validation phase can be ignored. These transactions are popular in many OLTP workloads (e.g., `TATP` [140]), as reported by prior work [13, 25].

Evaluation. With a proper sharding, there is no distributed read-only transaction in `TPC-C`, which needs remote data accesses. Further, there is only one single-key read-only transaction in `Smallbank` (i.e., `Balance`), which does not require the second phase (validation) [13, 25]. Therefore, we use the `customer-position` transaction in `TPC-E` [141] to evaluate the performance of distributed read-only transactions.

TPC-E. is designed to be a more realistic OLTP benchmark, which simulates the workload of a brokerage firm. One of well-known characteristics is the high proportion of read-only transactions, reaching more than 79%. The `customer-position` transaction is read-only and has the highest execution ratio. It simulates the process of retrieving the customer's profile and summarizing their overall standing based on current market values for all assets. The assets prices are fetched in a distributed way.

Figure 4–6 compares different choices of primitives for distributed read-only transactions. As expected, by offloading read operations to RNICs and bypassing remote CPUs, using one-sided primitives for both the read and validation phases can gain the

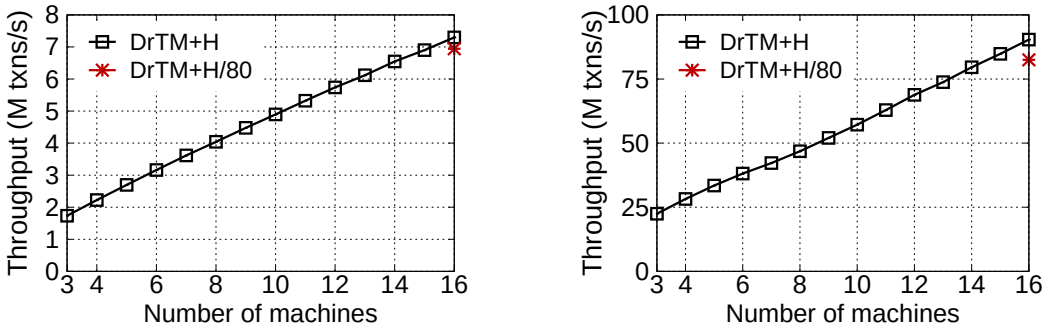


Figure 4–7 The performance of DrTM+H with the increase of machines for (a) TPC-C/no and (b) SmallBank.

best performance in both throughput and latency. One-sided outperforms Two-sided by about 10% in peak throughput (0.19 vs. 0.21), and the median latency is around 80% of Two-sided. Enabling the index cache (One-sided/Cache) in the read phase will further improve the peak throughput by close to 20% (0.25 vs. 0.21) and reduce the median latency more than 20%.

Summary. The hybrid scheme used in the execution phase (see §4.3.1) is also suitable to the first phase, and one-sided READ is always a better choice for the second phase (see §4.3.2). For single-key read-only transactions, a single one-sided READ is usually efficient.

4.4 DrTM+H: Fast transactions using hybrid schemes

In this section, we conclude our studies of using RDMA for transactions by showing how to improve the performance of prior designs by choosing appropriate primitives and techniques at different phases of transactional execution. This leads to DrTM+H, an efficient distributed transaction system using hybrid schemes.

4.4.1 Design of DrTM+H

DrTM+H optimizes different phases of the transaction by choosing the right primitives guided by our previous studies (§4.3). It supports serializable transaction with log replication for high availability. Currently, we have not implemented the reconfiguration and recovery, which is necessary to achieve high availability. Yet, since our replication protocol is exactly the same as the one used in FaRM [13], DrTM+H can use its method

to recover from failure.

Execution. DrTM+H uses a hybrid design of one-sided READs with caching and two-sided RPC. If the record's address has been cached locally, one RDMA READ is sufficient to fetch the record. Otherwise, DrTM+H uses RPC to fetch the record and its address.

Validation. DrTM+H uses one-sided ATOMIC for validation if there is no atomic issue (e.g., Network accesses do not conflict with local ones). Otherwise two-sided is preferred since using RDMA atomic operations will slow down local operations [17].

Logging. DrTM+H always uses one-sided WRITEs to replicate transaction logs to all backups and uses two-sided primitive to lazily reclaim logs on backups.

Commit. DrTM+H uses one-sided WRITEs to commit if one-sided ATOMIC is used in the validation phase. Otherwise DrTM+H uses two-sided RPC. DrTM+H always uses passive ACK optimization since the completion of commit message is not on the critical path of transactional execution.

Using outstanding request with speculative execution. In §4.3.1, we disable the outstanding request optimization at the execution phase to avoid requiring advance knowledge of read/write set. However, this usually means that transaction must fetch records one-by-one, which increases the latency of a single transaction.① We found that even the record has not been fetched to local, the transaction can still speculatively execute until the involved value is really used. This can greatly reduce the lifespan of a transaction. For example, the remote records required by new-order transaction in TPC-C are independent. Thus DrTM+H uses speculative execution to fetch these records in parallel.

4.4.2 Performance evaluation

Figure 4–7 presents the throughput and scalability of DrTM+H using TPC-C/no and SmallBank. To show that DrTM+H's usage of one-sided primitive has good scalability on a larger-scale cluster, we use the QP setting which is enough to run on an 80-node cluster (DrTM+H-80). Each thread uses 80 QPs (16x5) to connect to 16 nodes and chooses the usage of QP in a round-robin way.

① We still send multiple requests in parallel for different transactions using coroutines.

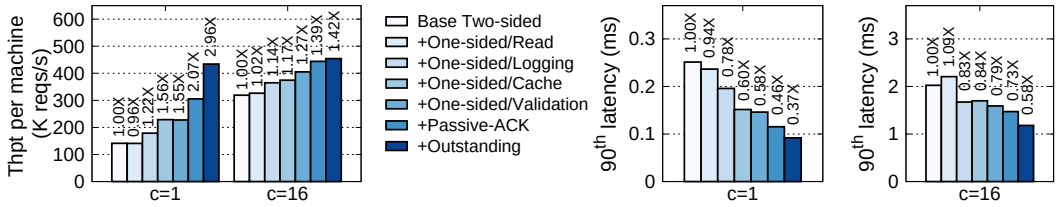


Figure 4–8 The contribution of optimizations to (a) throughput and (b,c) latency to TPC-C/no for DrTM+H using 1 and 16 coroutines, respectively. Optimizations are cumulative from left to right.

Performance and scalability. DrTM+H scales linearly with the increasing of machines. The throughput of TPC-C/no and SmallBank decrease 5% and 9% on the emulated 80-node connection setting, respectively. SmallBank is more sensitive to the number of QPs since its payload size is much smaller than that of TPC-C/no. However, SmallBank is still 1.3X higher than a pure two-sided solution in throughput, with a significant decrease in the tail latency. The 50^{th} (median), 90^{th}, and 99^{th} latency are reduced by 22%, 39%, and 49%, respectively.

Factor analysis. To investigate the contribution of the primitive choices in DrTM+H, we conduct a factor analysis in Figure 4–8. Due to space limits, we only report the experimental results of TPC-C/no; SmallBank is similar. First, we observe that using one-sided primitives can significantly improve the throughput and latency when servers are underloaded (1 coroutine). This is because one-sided primitive has lower CPU utilization and lower latency compared to two-sided one. Second, by increasing coroutines, the two-sided implementation has close throughput with one-sided one. However, a hybrid scheme in DrTM+H improves both median and tail latency. Finally, when leveraging RDMA, the number of round trips has more impacts on latency but not throughput, especially for CPU-intensive workloads (e.g., TPC-C). When using 16 coroutines, the throughput increases even using more network round trips (adding one-sided READs). This is because coroutines hide most of waiting for request's completion while one-sided primitive has lower CPU utilization.

4.4.3 Comparison against prior designs

There have been several designs to optimize transactional execution using RDMA. To understand the effects of RDMA primitive decisions, we implemented and evaluated

Table 4–3 A review of the existing RDMA-enabled transaction systems. I and II stand for one-sided and two-sided primitives.

	RW-TX				RO-TX	
	E	V	L	C	R	V
FaRM	I	II+I	I	II	I	I
DrTM+R	I	I+I	I	I+I	I	I
FaSST	II	II	II	II	II	II
DrTM+H	I/II	I/II	I	I/II	I/II	I

emulated versions of FaRM [13], DrTM+R [18] and FaSST [25].① We adopted the same codebase and transaction protocol (OCC) of DrTM+H, but choosing the RDMA primitives and techniques at different phases of transactional execution as the originals. Table 4–3 summarizes the primitives used in the three systems and compares the performance of emulated versions of them with DrTM+H. Note that all existing optimizations on RDMA primitives are enabled, including coroutine, outstanding requests, and doorbell batching.

Emulating FaRM. FaRM [13] is designed to run transactions atop of a global memory space over RDMA networking. FaRM uses one-sided READ at the execution/logging phase and one-sided WRITE at the logging phase, as well as a hybrid choice at the validation phase. Moreover, FaRM adopts an RDMA-friendly memory store (FaRM-KV) proposed in their prior work [2]. Our emulated store (DrTM-KV) has been shown to have a comparable performance even without the location cache [17]. Further, our two-sided RPC implementation has also better performance than the implementation in FaRM [25]. Hence, we believe our emulated version has similar or even better performance compared to the vanilla FaRM.

Emulating DrTM+R. DrTM+R [18] offloads all network operations to one-sided RDMA primitives for CPU efficiency, including using one-sided ATOMIC for locking remote records in the validation phase. Further, DrTM+R exploits hardware transactional memory (HTM) [142] to handle local transactions, but does not leverage coroutines to

① FaRM is not open-sourced, DrTM+R depends on hardware transactional memory, and FaSST uses a simplified OCC and protocol.

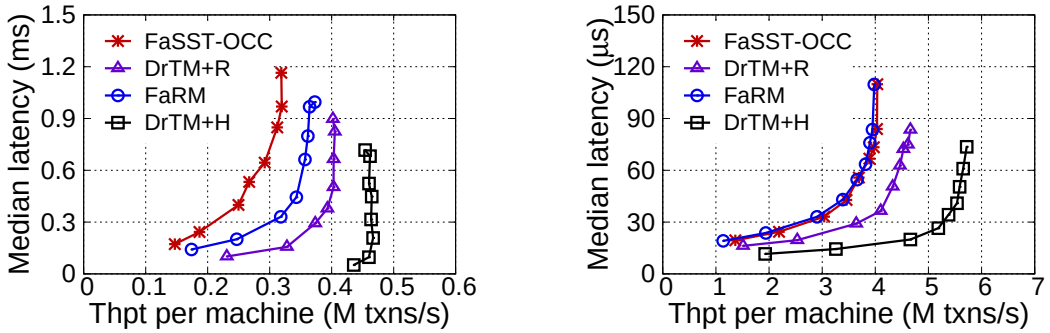


Figure 4–9 An end-to-end comparison of different designs on (a) `TPC-C/no` and (b) `SmallBank`.

obtain higher throughput. To focus on comparing different choices of RDMA primitives, our emulated version disables HTM (similar to the implementation of DrTM-OCC [143]) but enables coroutine optimization.

Emulating FaSST. FaSST [25] proposes a well-optimized RPC implementation based on two-sided primitives for running transactions. Since our framework provides a similar UD-based RPC implementation, it is straightforward to emulate FaSST by using two-sided primitives at all phases of transactional execution. Note that FaSST uses a simplified OCC protocol [25] by moving lock operations from the validation phase to the execution phase. To avoid confusion, we use FaSST-OCC to name the pure two-sided implementation on our platform with OCC protocol.

Evaluation. Compared to other prior designs, `DrTM+H` always embraces the best performance in terms of latency and throughput. Figure 4–9 presents our results. `DrTM+H` has the best throughput than previous designs with the right choice of RDMA primitives and a set of optimizations to better leverage the chosen primitive. On `TPC-C/no`, `DrTM+H`'s throughput is up to 2.96X of FaSST (from 1.41X), up to 1.89X of DrTM+R (from 1.12X) and up to 2.50X of FaRM (from 1.21X). When using 16 coroutines, the median latency is reduced by 33%, 23% and 34%, respectively. We broke down the performance improvements in §4.4.2. FaRM optimizes baseline two-sided (FaSST) by using one-sided operation for logging and execution. DrTM+R further adds location cache and use one-sided for validation and commit. In `TPC-C/no`, FaRM and DrTM+R outperforms FaSST due to better leveraging one-sided primitives for CPU-intensive workloads. DrTM+R outperforms FaRM due to the usage of location cache at the execution

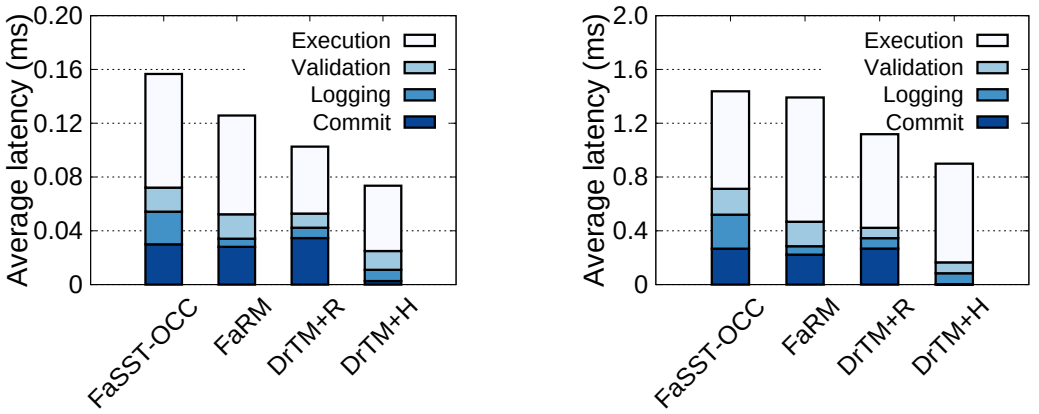


Figure 4–10 The latency breakdown of `TPC-C/no` using (a) 1 coroutine and (b) 16 coroutines.

phase and the usage of atomics at the validation phase. FaSST has a comparable performance to FaRM for `SmallBank` since two-sided primitive is faster at the execution phase.

Latency breakdown. To study the performance influence of choosing RDMA primitives, we further show a latency breakdown in each phase for different designs in Figure 4–10. By leveraging one-sided `READ`s, the latency of the execution phase is reduced by 13% and 41% in FaRM and DrTM+R respectively—for one coroutine. Increasing the number of coroutines can narrow the performance gap by hiding the latency of network operations. Further, FaSST can outperform FaRM by 22% when using 16 coroutines, since FaRM requires more network round trips to read remote data. To remedy this, DrTM+R enables the location-based cache [17] for one-sided operations and achieves the lowest latency (less than 0.7ms). In the validation phase, DrTM+R has the lower latency by offloading lock operations to RDMA NICs. Using one-sided `WRITE`s, the latency of the logging phase in DrTM+R and FaRM is reduced by about 69% and 75% respectively, compared to using two-sided primitives (FaSST). Finally, `DrTM+H` can always choose appropriate RDMA primitives to embrace the latency reduction at each phase. Note that `DrTM+H` has the lowest latency at the commit phase due to enabling Passive ACK optimization (§2.2.3), such that receiving the acknowledgement of commit messages is done off the critical path.

4.5 Discussion

Trends, features, and extensions. Our studies focus on Mellanox ConnectX-4 RNIC. Previous generations of RNICs like ConnectX-3 yields slower performance of one-sided `READ`s. However, we have seen a trend that one-sided primitives become faster and more scalable in recent RNICs, from Connect-IB to ConnectX-4 to ConnectX-5. Further, new generation RNIC may introduce more features for one-sided primitives. For example, ConnectX-5 has integrated with NVMe over Fabrics [144], suggesting an optimistic opinion about providing offloading features in modern data centers.

On the other hand, one-sided primitive still has many limitations due to the lack of expressiveness [17]. For example, it is not competent for complicated operations, like searching in a sorted store. Furthermore, one-sided primitive is unlikely to have orders of magnitude higher performance than messaging, because we have also seen a trend on providing fast messaging rate in later generation RNICs [145]. Hence, how to properly choose the right primitive is very important given a specific workload. This chapter gives an example of how to optimize transactional processing with a combination of different primitives in a phase-by-phase way. The resulting system and insights may be reused for further studies.

Some proposed RDMA extensions, including the coherence of atomic operations, atomic object reads [146], and multi-address atomics [147], may provide further exploration spaces once being commercialized. We believe that there will be a continued line of research in this field with more new features, implementations and application domains.

Emulating a large-scale RDMA cluster. Currently, we mainly focus on emulating massive RDMA connections in a rack-scale cluster, because QP cache misses will dominate the impact on the performance of various primitives. Consequently, we do not consider other scalability issues in a real large-scale RDMA cluster. For example, a large cluster has to use multiple layers of RDMA networking such as multiple switches or congestion control mechanism [148].

4.5.1 Related work on RDMA-enabled systems

Besides distributed transactions [13, 17-18, 25-27], a large number of systems have used RDMA features to improve performance. These include but not limited to key-value stores [2, 28-31], distributed file systems [32, 51], consensus algorithms [33] and graph

processing systems [34-37]. Such systems also have different RDMA primitive choices according to their own demands. We hope our study of DrTM+H can further inspire an optimal use of RDMA primitives for these systems.

4.6 Conclusion

DrTM+H is the first systematic study on how different choices of RDMA primitive affect the performance of transactional execution. Unlike previous studies, it compares different primitives and techniques using one well-optimized RDMA framework. This makes the comparison of techniques and primitives comparable and comprehensive. The main observation of DrTM+H is that no single primitive is the winner all the time, even at different phases of transactional execution. It then adopts a hybrid solution which uses the most appropriate primitive at each phase of transactions. This not only improves the throughput but also reduces the latency of transactions. Finally, our study gives hints about whether it is cost-effective to offload RDMA one-sided, or just use two-sided for easy porting. We hope it can stimulate and provide a guideline for future system co-design with RDMA.

Chapter 5 Using DST for Scalable Multi-version Concurrency Control

As we have presented in the last chapter, by co-designing OCC with RDMA, DrTM+H can fully utilize RDMA for RDMA-enabled distributed transactions. However, in workloads dominated by read-only transactions, the OCC protocol used in DrTM+H can perform poorly due to excessively aborts. As shown in Figure 5–1, there is a notable performance gap between using OCC and read committed (RC) protocol for TPC-E.

Read-only transactions are common and requires strong transactional isolation. For instance, our examination of TPC-E [141], a sophisticated online transaction processing benchmark that models stock exchange, uncovers that 79% of transactions are read-only ones at run time. Facebook has also reported that 99.8% of accesses to its distributed data store is reads [105], and these reads require strong transactional isolation with writes [149]. Enforcing transactional isolation for read-only transactions is challenging, because one such transaction may fork into thousands of sub-queries [149].

Multi-version concurrency control (MVCC) is a common approach [150-151] to unleash the parallelism between concurrent readers and writes. By maintaining multiple database snapshots, readers can read tuples from a stale snapshot while writers can concurrently write the tuples. Therefore, nearly all commercial database has adopted MVCC, including but not limited to PostgreSQL [152], Oracle [153], MySQL/InnoDB [154], Hekaton [5], and SAP HANA [155].

While MVCC extracts more concurrency from transactions (especially for read-only transactions), it does not necessarily attain optimal performance and/or scalability improvement (see Figure 5–1). MVCC relies on a timestamp to delimitate different snapshots, while maintaining timestamp ordering at scale (§5.1.3) is costly. More specifically, a centralized sequencer (timestamp oracle) is usually used to provide snapshot timestamp to transactions, which reflects a total order among transactions (i.e., global timestamp (GTS)). However, such a mechanism not only adds more communications but also causes overly-constrained concurrency control for read-write transactions, leading to performance degradation and scalability bottlenecks [156]. Vector timestamp (VTS), which leverages a clock per worker or machine, only mitigates the scalability bottleneck of centralized timestamp schemes but causes more network traffic, which grows linearly

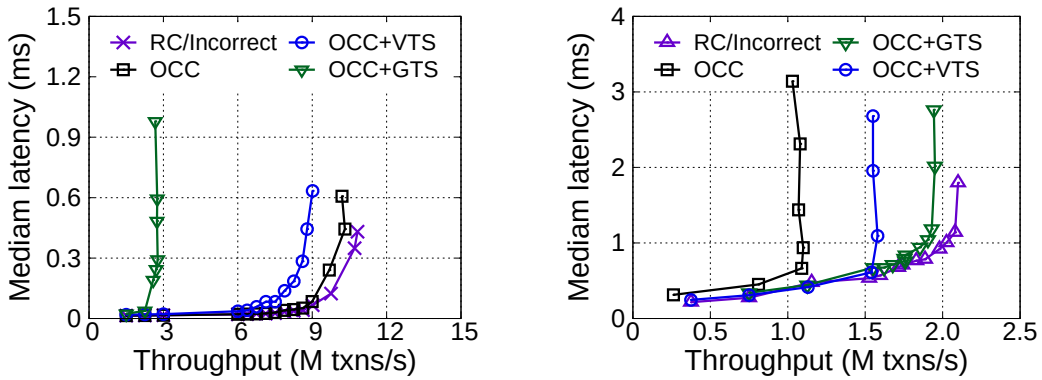


Figure 5–1 Performance of (a) TPC-E and (b) TPC-C on DrTM+H using different CC protocols and TS schemes (see §5.5 for details). GTS and VTS stand for using OCC protocol, while the read-only transactions read the snapshots delimitated by GTS and VTS. RC/Incorrect stands for using RC protocol, which can provide optimal performance, but at the expense of correctness.

with the increase of workers or machines in the system.

This chapter presents DST (*decentralized scalar timestamp*), a scalable timestamp mechanism for MVCC without a centralized sequencer or vector timestamps. DST is motivated by a key observation: *transaction ordering provided by existing CC protocols already implies serializable ordering among transactions, which can be reused to maintain timestamp ordering in a lightweight and scalable way.* This is because any pair of conflicting transactions must have conflicting accesses to a particular tuple. Thus, the *later* transaction should see the timestamp of the *former* transaction from the conflicting tuple and have a *larger* timestamp. By piggybacking on CC protocols to derive a scalable timestamp, DST avoids the performance overhead and scalability bottleneck in existing centralized approaches.

Besides high performance and scalability, DST is also a general timestamp mechanism: we have ported it to three representative transactional systems with different CC protocols, namely DrTM+H (§4, OCC), MySQL cluster [60] (2PL), and Rococo [19]. Further, DST does not depend on specific hardware features (e.g., RDMA). The experimental results on three clusters show that DST can achieve more than 95% of optimal performance (using RC protocol) without compromising correctness. With DST, DrTM+H achieves up to 1.8X and 6.1X performance improvements for TPC-E and TPC-C. DST is also up to 1.7X and 6.3X faster than the state-of-the-art solutions of centralized timestamp schemes.

5.1 Background and motivation

5.1.1 Target systems

We design DST for general distributed transactions over database data partitioned to multiple storage nodes. The client's transaction request is handled by a coordinator, which interacts with storage nodes for executing the transaction. During the transaction's execution, the coordinator may send read/write requests to read/write data from the storage nodes; or send transactional requests (e.g., lock or unlock) according to the database's concurrency control protocol. It may batch requests (e.g., write and unlock) to avoid extra network roundtrip.

Our goal is to support serializable read-only transaction that never aborts, and does not interfere with read-write transaction. Further, it is desirable to execute reads in the read-only transaction in one-roundtrip, i.e., the coordinator can retrieve a consistent view of the data from the storage nodes in one request.

5.1.2 MVCC and timestamps

A common approach to support serializable read-only transaction without interfering with read-write transaction is through multi-version concurrency control (MVCC). When designing MVCC systems, designers have two major considerations compared with single-version mechanisms [151]. The first is *how to cheaply allocate a globally-ordered version for updating tuples transactionally*. Deciding the version installed with tuples should have minimal impacts on read-write transactions. The second is *how to efficiently allocate a freshly-stable version for reading tuples consistently*. Read-only transactions should have access to consistent snapshots with low latency and high freshness. MVCC schemes typically adopt the concept of *timestamps* for tuple versions. However, it is non-trivial to design a general timestamp scheme that supports *efficient* snapshot reads while incurring *minimal overhead* for broad CC protocols.

To motivate the design of DST, we start by briefly reviewing how existing timestamp schemes are applied to two-phase locking (2PL) for MVCC and snapshot reads [151, 157].

Global timestamp (GTS). This approach leverages a timestamp service, namely *timestamp oracle*, to manage globally ordered timestamps [5, 158-159]. It provides two functions for MVCC systems, as shown in Figure 5–2. First, the read-write transaction contacts the oracle for a commit timestamp (GlobalTS) at the commit phase. Upon a successful

2PL with global timestamp (GTS)

```
At Oracle:                                  ▸ timestamp server
+ GlobalTS                          ▸ monotonic global timestamp
+ StableGTS                          ▸ snapshot global timestamp
+ Queue                               ▸ pending global timestamp

  INSTALL(gts):
+1  add gts to Queue

  STABILIZE( ):                             ▸ run asynchronously
+1  for each gts in Queue do
+2    if gts is ready then
+3      dequeue gts and gts → StableGTS

At Worker_i:                       ▸ i denotes the worker number
  WRITE(tx, id, data)
 1  acquire lock
 2  add ⟨id, data⟩ to tx.wset

  READ(tx, id)
 1  acquire lock and get latest ⟨data⟩
 2  add ⟨id, data⟩ to tx.rset
 3  return ⟨data⟩

  COMMIT(tx)
+1  tx.TS ← Oracle.GlobalTS          ▸ network round trip
 2  for each w in tx.wset do
:3    update ⟨w.data, tx.TS⟩ and release lock
 3  for each r in tx.rset do
 4    release lock
+5  Oracle.INSTALL(tx.TS)            ▸ network round trip

  ROTX(tx)                                  ▸ snapshot read
+1  tx.TS ← Oracle.StableGTS
+2  for each r in tx.rset do
+3    get ⟨r.data⟩ up to tx.TS
```

Figure 5–2 Using GTS (i.e., blue code lines) to enable consistent snapshots for read-only transactions with 2PL. `+N` and `:N` denote new and modified lines of code respectively.

commit, this transaction creates a new version denoted by the commit timestamp for each tuple in the write set (*line:3* of COMMIT) and sends back the committed timestamp to the oracle (*line:5*). Second, the read-only transaction contacts the oracle for a read timestamp (StableGTS) and retrieves tuples in the read set with versions no larger than the read timestamp (*line:2-3* of ROTX).

Given the specification of extensions to 2PL with GTS in Figure 5–2, we analyze the transaction behavior in the case shown in Figure 5–3 to explain the design of GTS.

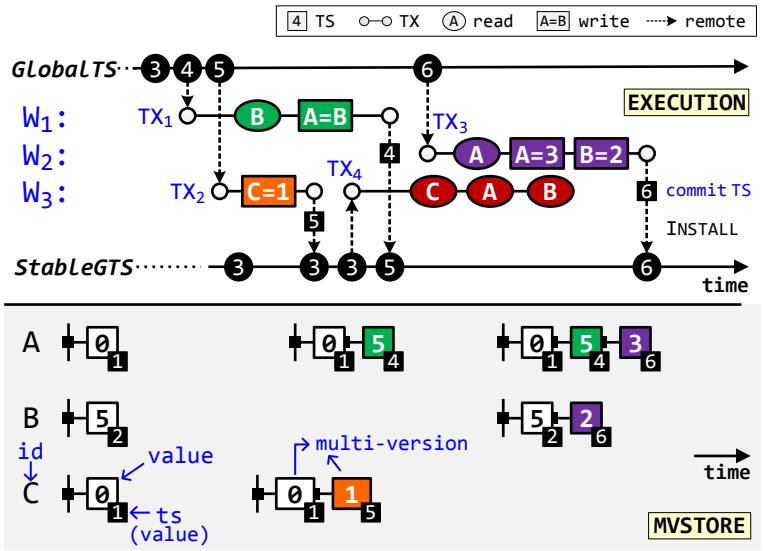


Figure 5–3 A sample case of using GTS, where four transactions (TX_1-TX_4) operate on three tuples (A, B, and C).

There are four transactions (TX_1–TX_4), which operate on three tuples (A, B, and C). Note that non-conflicting transactions TX_1 (green) and TX_2 (orange) are both forced to acquire GlobalTS according to the specification. This operation is necessary to maintain the global timestamp ordering, yet results in overly-constrained concurrency control and an extra network round trip compared to the vanilla 2PL.

The necessity of the oracle to maintain StableGTS can be revealed with the conflict between the timestamp order and the commit order concerning TX_1 and TX_2. In this case, TX_2 acquires a larger GlobalTS but commits before TX_1. When read-only transaction TX_4 (red) starts, it cannot simply use the latest committed timestamp (GlobalTS=5) for snapshot reads. The snapshot would be inconsistent if the read-only transaction observes TX_2 before TX_1 commits. Thus, transactions must install commit timestamps so that the oracle can determine the read timestamp (StableGTS=3) for TX_4.

Vector timestamp (VTS). To reduce the overhead of acquiring GlobalTS in the critical path of read-write transactions, VTS replaces the global timestamp counter with a vector of local timestamps. The vector contains a slot for each worker, which records the per-worker timestamp. In each worker, a local counter (LocalTS) is used to assign the commit timestamp for transactions, hence reducing one network round trip compared to GTS. However, the oracle is retained in VTS to maintain the StableVTS with similar reasons as GTS. Figure 5–4 shows the specification of extensions to 2PL with VTS.

```
2PL with vector timestamp (VTS)
At Oracle:                                  ▸ timestamp server
+ StableVTS                          ▸ snapshot global timestamp
+ Queues                              ▸ pending global timestamp

  INSTALL(⟨i:ts⟩, deps)
+1  add ⟨⟨i:ts⟩, deps⟩ to a Queues[i]

  STABILIZE( )                              ▸ run asynchronously
+1  for each queue in Queues do
+2    for each ⟨⟨i:ts⟩, deps⟩ in queue do
+3      if deps is ready then              ▸ stability protocol
+4        dequeue ⟨⟨i:ts⟩, deps⟩
+5        ⟨i:ts⟩ → StableVTS

At Worker_i:                       ▸ i denotes the worker number
+ LocalTS                           ▸ monotonic local timestamp

  WRITE(tx, id, data)
:1  acquire lock and get latest ⟨i:ts⟩
 2  add ⟨id, data⟩ to tx.wset
+3  add ⟨i:ts⟩ to tx.deps

  READ(tx, id)
:1  acquire lock and get latest ⟨data, ⟨i:ts⟩⟩
 2  add ⟨id, data⟩ to tx.rset
+3  add ⟨i:ts⟩ to tx.deps
 4  return ⟨data⟩

  COMMIT(tx)
+1  tx.TS ← LocalTS
 2  for each w in tx.wset do
:3    update ⟨w.data, ⟨i:tx.TS⟩⟩ and release lock
 4  for each r in tx.rset do
 5    release lock
+6  Oracle.INSTALL(⟨i:tx.TS⟩, tx.deps)   ▸ NT round trip

  ROTX(tx)                                  ▸ snapshot read
+1  tx.TS ← Oracle.StableVTS
+2  for each r in tx.rset do
+3    get ⟨r.data⟩ up to tx.TS
```

Figure 5–4 Using VTS (i.e., blue code lines) to enable consistent snapshots for read-only transactions with 2PL. `+N` and `:N` denote new and modified lines of code respectively.

Figure 5–5 presents a concrete case of using VTS. Each worker maintains its local counter (W_1:3, W_2:2, and W_3:5). The version of a tuple is represented as $\langle i : ts \rangle$, where i is the worker ID, and ts is the commit timestamp of the transaction that writes the tuple. The initial StableVTS is $(3, 2, 5)$, which means that tuples with versions less than $\langle 1 : 3 \rangle$, $\langle 2 : 2 \rangle$, and $\langle 3 : 5 \rangle$ can be consistently read by read-only transactions.

Maintaining the stable timestamp (StableVTS) becomes more complex in VTS because we cannot directly compare the per-worker timestamps [26, 160]. To convey the

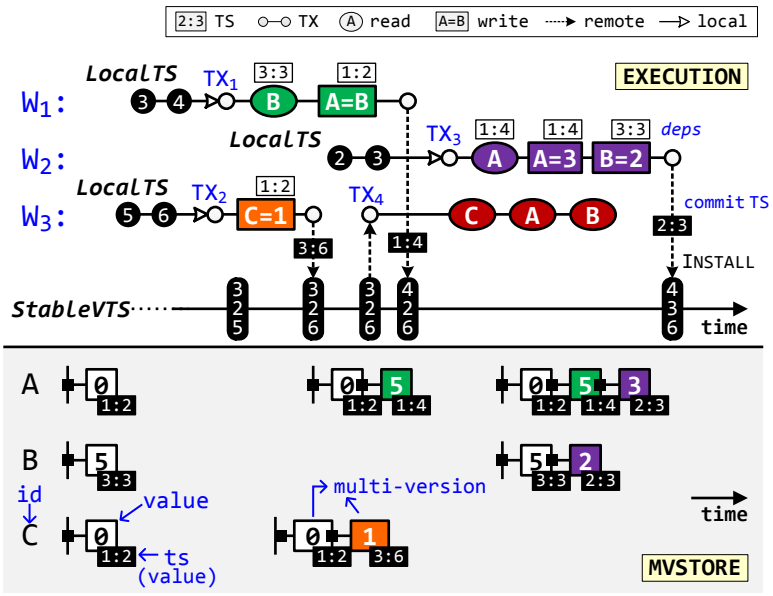


Figure 5–5 A sample case of using VTS, where four transactions (TX_1-TX_4) operate on three tuples (A, B, and C).

ordering of transactions to the oracle for deciding StableVTS, workers should collect observed timestamps of accessed tuples from other workers (e.g., $\langle 1 : 2 \rangle$ of C for TX_2). Note that when read-write transactions (TX_1, TX_2, and TX_3) commit, they must send all observed timestamps (*deps*) to the oracle (INSTALL in Figure 5–4). Moreover, read-only transactions (TX_4) must request the whole vector timestamp (StableVTS) from the oracle to start a snapshot read.

5.1.3 Analysis of network overhead

We present an in-depth analysis of centralized timestamp schemes① and attribute performance overhead and scalability bottleneck to three main aspects:

Non-scalable timestamp oracle. Prior work [7, 133, 158, 161-163] has shown that a centralized timestamp oracle will become the scalability bottleneck of MVCC systems. The throughput of schemes using a shared counter with atomic operations (GTS) [5, 164-165] is limited to less than 10 M ops/s (`GlobalCNT` in Figure 5–6(a)). The throughput will further decrease due to maintaining the stable timestamp for read-only transactions (`+StableTS`). VTS mitigates the scalability issue by using a local counter for read-write

① For brevity, we avoid prior sophisticated optimizations (incl. batching requests [66, 158], timestamp compression and dedicated fetch thread [26]) for timestamps in here, but enable all of them in the evaluation (§5.5).

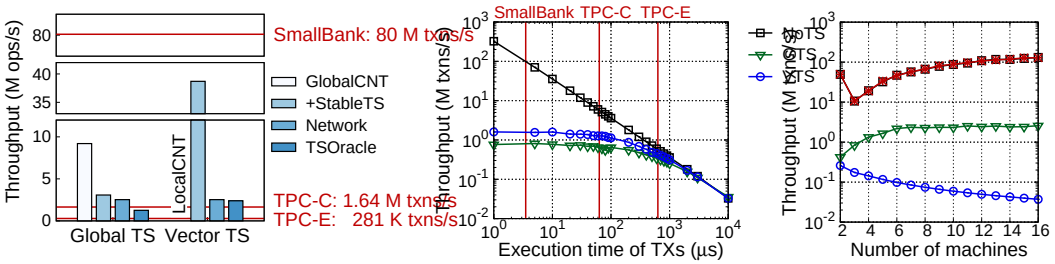


Figure 5–6 (a) Analysis of peak performance and bottleneck of timestamp oracle for GTS and VTS using a 24core machine with 10GbE. (b) The performance of *read-write* transactions with the increase of execution time for different timestamp schemes. The weighted average median latency of read-write transactions in TPC-E, TPC-C and SmallBank are labeled by red lines. (c) The performance of *read-only* transactions with the increase of machines for different timestamp schemes. All experiments are conducted on a local 16-node cluster with 10GbE network (§5.5). One machine is dedicated for timestamp oracle, even NoTS has no need. Each machine spawns 24 server workers.

transactions. Besides, prior work [26] avoids the mechanism for the stable timestamp (reaching close to 40 M ops/s) at the expense of increasing transaction aborts. However, the network will first become the bottleneck for both GTS and VTS (`Network`). Consequently, the throughput of timestamp oracle (`TSOracle`) can only reach 1.26 M and 2.39 M ops/s for GTS and VTS respectively, which may be enough for TPC-E (281 K txns/s) but far not enough for TPC-C (1.64 M txns/s) and SmallBank (80 M txns/s) even only scaling out to 16 machines.

Using fast networks can boost the throughput of timestamp oracle, while the performance of transactional systems will also increase much [13, 17, 25-26], and CPU may first become the bottleneck [7]. Moreover, batching requests [66, 158] or dedicated fetch thread [26] can alleviate the timestamp-related load on the network①, while these techniques also amplify the staleness of the data retrieved by read-only transactions, and increase the abort rate and the end-to-end latency of read-write transactions (see §5.5.1).

Costly timestamp allocation. A centralized timestamp scheme will inevitably cause extra network communication overhead for each read-write transaction. GTS demands two network round trips, one for obtaining the commit timestamp and one for installing it. VTS uses per-worker local counters to assign the commit timestamp, but still demands one network round trip to install the timestamp. Given that most transactions operate on

① We enabled these optimizations for GTS and VTS in our evaluation (§5.5).

tuples in local partitions [124, 135, 140], especially for read-write transactions, additional network round trips will notably lengthen the critical section of transactions and further increase the chance of conflicts, causing extra transaction aborts or blocking time. Thus, it is non-trivial to hide the network round trips without sacrificing the latency of transactions (e.g., batching requests [66, 158]).

The overhead of timestamp allocation highly depends on the execution time of transactions. Hence, we implement a microbenchmark only consisting of read-write transactions, which do not access any tuples and just spin in a loop for a given time. As shown in Figure 5–6(b), the overhead of VTS is moderate (from 10% to 30%) compared to not using timestamp schemes (NoTS), when the execution time is close to that of read-write transactions in TPC-E (from 1,400μs to 470μs). The throughput will significantly drop more than 80% when transactions execute in about 50μs, which is similar to that of read-write transactions in TPC-C. Further, GTS can only achieve half of VTS throughput, since it demands one more round trip to obtain the commit timestamp.

Large traffic size. VTS mitigates the timestamp overhead by using per-worker local counters as the commit timestamp for read-write transactions. However, a critical downside is that a whole vector of per-worker timestamps must be obtained as the read timestamp first, and then be transferred to every tuple for performing consistent snapshot reads. In contrast to the scalar timestamp (e.g., GTS), this overhead grows linearly with the increase of workers or machines in the system. For most transactional workloads [124, 135, 140-141], the size of the vector timestamp can become orders of magnitude larger than the tuple size, even in a moderate-sized cluster. Using the per-machine counter in VTS (i.e., all workers on one machine share one timestamp slot) can reduce traffic size [160]. However, these workers have to share a local counter by using *atomic* operations (e.g., CAS), which will incur additional overhead on read-write transactions [26].

To demonstrate the impact of traffic size, we implement a microbenchmark only consisting of read-only transactions, which read ten 8-byte tuples with 90% of which being local accesses. In Figure 5–6(c), the performance collapse of VTS is due to the increase of timestamp vector obtained from the oracle and transferred to remote tuples. Note that GTS is still one order of magnitude slower due to extra one round-trip to fetch the read timestamp (even scalar), compared to NoTS.

5.2 Decentralized scalar timestamp (DST)

This section presents the design of DST, a *decentralized scalar timestamp* that facilitates the multi-version concurrency control (MVCC) implementation for broad CC protocols with efficient snapshot read support and minimal overhead. DST aims at fundamentally overcoming the above drawbacks of traditional timestamp schemes. The intuition behind it is that *the timestamp scheme can piggyback on concurrency control protocols to maintain the timestamp ordering with low cost and no new scalability bottleneck to read-write transactions*.

Roadmap. We first use two-phase locking (2PL) as an example to explain the basic protocol of DST for read-write and read-only transactions (§5.2.1 and §5.2.2). We then prove the serializability of read-only transactions with DST (§5.2.3). One key challenge is how to derive a *consistent* yet *fresh* snapshot. §5.2.4 introduces a hybrid scalar timestamp to provide snapshot reads with bounded staleness (§5.2.4). Finally, we discuss the impact of DST on the fault-tolerance scheme (§5.2.5).

5.2.1 Timestamps in read-write transaction

DST is a fully decentralized timestamp without a centralized sequencer (timestamp oracle) to provide total order timestamps for read-write transactions and stable timestamps for read-only transactions. Therefore, DST must ensure that the derived timestamps for read-write transactions always match the transaction ordering.

The CC protocol is used to ensure the serializable transaction ordering and provide the following three properties, where Transaction A (TX_A) commits before Transaction B (TX_B), and both of them access a conflicting tuple O.

Property 1: Write-Write. TX_B's write ($W_B(O)$) should overwrite TX_A's write ($W_A(O)$) or generate a newer version.

Property 2: Write-Read. TX_B's read ($R_B(O)$) should retrieve TX_A's write ($W_A(O)$).

Property 3: Read-Write. TX_A's read ($R_A(O)$) should not retrieve TX_B's write ($W_B(O)$).

To match the transaction ordering, DST should ensure TX_B's commit timestamp (TS_B) is larger than TX_A's commit timestamp (TS_A) under the above case. The general idea is to piggyback over the CC protocol to derive a commit timestamp from conflicting tuples. Figure 5–7 presents how DST is integrated with two-phase locking (2PL), and Figure 5–8 illustrates the execution of sample transactions with DST. DST leverages conflicting

```
Read-write Transaction: 2PL with DST
At Worker_i:                    ▸ i denotes the worker number
+  LocalTS                      ▸ monotonic local timestamp

   START(x)
+1    x.TS ← LocalTS

   WRITE(x, id, data)
:1    acquire lock and get ts
 2    add ⟨id, data⟩ to x.wset
+3    x.TS ← max(x.TS, ts+1)

   READ(x, id)
:1    acquire lock and get latest ⟨data, ts⟩
 2    add ⟨id, data⟩ to x.rset
+3    x.TS ← max(x.TS, ts+1)
 4    return data

   COMMIT(x)
 1    for each w in x.wset do
:2      update ⟨w.data, x.TS⟩ and release lock
 3    for each r in x.rset do
:4      update ⟨x.TS⟩ and release lock
+5    LocalTS ← max(LocalTS, x.TS)
```

Figure 5–7 Specification of *read-write* transaction for 2PL with DST. `+N` and `:N` denote new and modified lines of code respectively.

tuples and the above three properties to transmit commit timestamps between dependent transactions. The additional codes for DST in WRITE, READ, and COMMIT (see Figure 5–7) are commented on corresponding operations in the following explanations.

Write-Write property. Transaction TX_A installs value (V_A) with commit timestamp (TS_A) into the tuple O.

$$\langle V_A, TS_A \rangle \rightarrow O$$
$$TS_A \rightarrow O.ts \qquad \triangleright \text{ COMMIT } line{:}2$$

Transaction TX_B reads the timestamp of tuple O (O.ts) and installs new value (V_B) with a *larger* commit timestamp (TS_B) into the tuple O.

$$\begin{aligned} \texttt{O.ts} &\rightarrow \texttt{ts} && \triangleright \textsc{Write}\ \textit{line:1} \\ \textit{max}(\texttt{ts+1}, \texttt{TS}_\texttt{B}) &\rightarrow \texttt{TS}_\texttt{B} && \triangleright \textsc{Write}\ \textit{line:3} \\ \langle \texttt{V}_\texttt{B}, \texttt{TS}_\texttt{B} \rangle &\rightarrow \texttt{O} && \\ \texttt{TS}_\texttt{B} &\rightarrow \texttt{O.ts} && \triangleright \textsc{Commit}\ \textit{line:2} \end{aligned}$$

In Figure 5–8, $\texttt{TX}_1$ (green) commits before $\texttt{TX}_3$ (purple), and both of them write tuple A. Therefore, the commit timestamp of $\texttt{TX}_3$ should be larger than that of $\texttt{TX}_1$. Using DST, $\texttt{TX}_1$ installs its value (5) with its commit timestamp ($\texttt{TS}_1$=4) into tuple A. After that, $\texttt{TX}_3$ should derive a larger timestamp ($\texttt{TS}_3$=5) from the timestamp of tuple A (A.ts=4) and use it to install new value (3) into tuple A. Note that the write operations will update both the tuple's timestamp and the value's timestamp (as a tuple may have multiple values with different versions).

Write-Read property. Transaction $\texttt{TX}_\texttt{A}$ installs value $\texttt{V}_\texttt{A}$ with its commit timestamp $\texttt{TS}_\texttt{A}$ into tuple O.

$$\begin{aligned} \langle \texttt{V}_\texttt{A}, \texttt{TS}_\texttt{A} \rangle &\rightarrow \texttt{O} && \\ \texttt{TS}_\texttt{A} &\rightarrow \texttt{O.ts} && \triangleright \textsc{Commit}\ \textit{line:2} \end{aligned}$$

Transaction $\texttt{TX}_\texttt{B}$ reads value $\texttt{V}_\texttt{A}$ of tuple O with timestamp O.ts and installs a larger commit timestamp $\texttt{TS}_\texttt{B}$.

$$\begin{aligned} \texttt{O} &\rightarrow \langle \texttt{V}_\texttt{A}, \texttt{ts} \rangle && \triangleright \textsc{Read}\ \textit{line:1} \\ \textit{max}(\texttt{ts+1}, \texttt{TS}_\texttt{B}) &\rightarrow \texttt{TS}_\texttt{B} && \triangleright \textsc{Read}\ \textit{line:3} \\ \texttt{TS}_\texttt{B} &\rightarrow \texttt{O.ts} && \triangleright \textsc{Commit}\ \textit{line:4} \end{aligned}$$

In Figure 5–8, $\texttt{TX}_1$ (green) commits before $\texttt{TX}_3$ (purple), and $\texttt{TX}_1$ writes tuple A before $\texttt{TX}_3$ reads it. Therefore, the commit timestamp of $\texttt{TX}_3$ should be larger than that of $\texttt{TX}_1$. Using DST, $\texttt{TX}_1$ installs its value 5 with its commit timestamp ($\texttt{TS}_1$=4) into tuple A. After that, $\texttt{TX}_3$ reads the timestamp of tuple A (A.ts=4) and derives a larger timestamp ($\texttt{TS}_3$=5).

Read-Write property. Transaction TX_A installs commit timestamp TS_A into tuple O since it has read the value of tuple O.

$TS_A \rightarrow O.ts$ ▷ COMMIT *line:4*

TX_B reads timestamp of tuple O (O.ts) and installs new value V_B with a larger timestamp TS_B into tuple O (O.ts).

$O.ts \rightarrow ts$ ▷ WRITE *line:1*

$max(ts+1, TS_B) \rightarrow TS_B$ ▷ WRITE *line:3*

$\langle V_B, TS_B \rangle \rightarrow O$

$TS_B \rightarrow O.ts$ ▷ COMMIT *line:2*

In Figure 5–8, TX_1 (green) commits before TX_3 (purple), and TX_1 reads tuple B before TX_3 writes it. Therefore, the commit timestamp of TX_3 should be larger than that of TX_1. Using DST, TX_1 reads an old value (5) of tuple B and installs its commit timestamp (TS_1=4) into tuple B. After that, TX_3 will derive a larger timestamp (TS_3=5) and use it to install new value (2) into tuple B.

5.2.2 Timestamps in read-only transaction

DST ensures that the order of derived commit timestamps for read-write transactions always matches the transaction ordering. Therefore, read-only transactions can directly pick any timestamp (TS_{RO}) to read a consistent snapshot by comparing its read timestamp with the timestamps of tuples.

Since the (snapshot) read-only transaction does not follow the CC protocol (e.g., lock/unlock tuples before/after reading values), the read-only transaction may read a part of updates of a concurrent read-write transaction. For example, in Figure 5–8, the read-only transaction TX_4 (red) and the read-write transaction TX_3 (purple) are concurrently executed. If TX_3 commits between the read operations to tuple A and tuple B in TX_4, and then TX_4 will read an old version of tuple A (5) and a new version of tuple B (2).

To ensure the serializability of read-only transactions, DST asks the read-only transaction to claim its operations actively before reading the tuple. It first installs its read

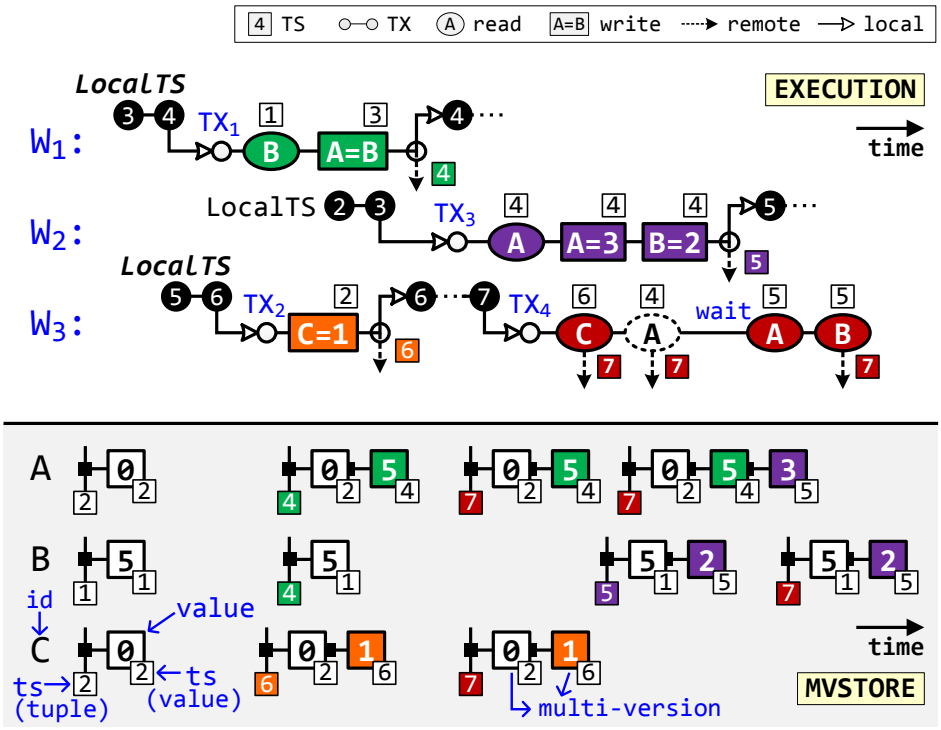


Figure 5–8 A sample case of using DST, where four transactions (TX_1-TX_4) operate on three tuples (A, B, and C).

timestamp (TS_{RO}) into the tuple and waits until the conflicting read-write transaction commits (e.g., the tuple is not locked), if the timestamp of the tuple is not larger than the read timestamp (DEP_READ in Figure 5–9). Note that the read-only transaction will only wait for at most one conflicting read-write transaction because if the concurrent read-write transaction starts after the claim, it will definitely see the read timestamp through accessing the tuple and derive a larger commit timestamp. Consequently, the read-only transaction will skip all of the updates from this transaction. If the concurrent read-write transaction starts before the claim, it will hold the lock of the tuple. The read-only transaction will wait until the read-write transaction commits. No matter the commit timestamp is larger or smaller than the read timestamp, a read-only transaction can always read a consistent snapshot by ignoring or reading all of the updates from conflicting transactions. Note that CC protocols ensure the atomicity of read-write transaction's updates.

As shown in Figure 5–8, the read-only transaction TX_4 will install its read timestamp (TS_4=7) into tuples with smaller tuple timestamps (*line:1* of DEP_READ in Figure 5–9). For unlocked tuple C, TX_4 will directly read the value up to the timestamp (1). For locked tuple A and B, TX_4 will wait until the concurrent read-write transaction TX_3 commits. In

```
Read-only Transaction: 2PL with DST
At Worker_i:                    ▸ i denotes the worker number
+  LocalTS                        ▸ monotonic local timestamp

  ROTX(x)                                    ▸ snapshot read
+1   x.TS ← LocalTS
+2   for each r in x.rset do
+3     DEP_READ(x,r)

  DEP_READ(x,r)
+1   if r.ts <= x.TS then
+2     r.ts ← x.TS                          ▸ atomic (CAS)
+3     wait until r not locked               ▸ if conflict
+4   get ⟨r.data⟩ up to x.TS
```

Figure 5–9 Specification of *read-only* transaction for 2PL with DST.

this example, since TX_3 does not see the read timestamp of TX_4, the commit timestamp of TX_3 is still smaller than the read timestamp of TX_4 (5 vs. 7). Hence, TX_4 can read all updates from TX_3 (A=3 and B=2).

5.2.3 Proof of correctness

Theorem (Serializability). *DST implements serializable read-only transactions, which always read a consistent snapshot generated by serializable read-write transactions.*

Proof sketch. The intuition of the proof is that if a read-only transaction can be serialized with read-write transactions, then it reads a consistent snapshot. We provide a proof sketch by contradiction based on this intuition: i.e., if a read-only transaction cannot be serialized with read-write transactions, then it leads to a contradiction. Before giving the proof, we need to prove following two lemmas first:

Lemma 1. *Given two dependent read-write transactions TX_1 and TX_2, if TX_2 depends on TX_1, then TX_2's timestamp (TS_2) is larger than TX_1's timestamp (TS_1).*

Proof If TX_2 directly depends on TX_1[①], this lemma follows directly from the algorithm (see §5.2.1) that TX_2 always calculates TS_2 based TS_1. If TX_2 transitively depends on TX_1, in a proof by contradiction we assume TS_1 is not smaller than TS_2, then in the partial dependent graph denoted by $TX_1 \rightarrow ... \rightarrow TX_i \rightarrow TX_j ... \rightarrow TX_2$[②], there exists TX_i and TX_j that TX_j directly depends on TX_i, but its timestamp is not larger than TX_i's, which is a contradiction with the first case. □

① TX_2 is conflicting with TX_1, and TX_2 accesses the conflicting tuples immediately after TX_1.

② The symbol → indicates the happen-before relation.

Lemma 2. *Given a read-only transaction TX_{RO} and a read-write transaction TX_{RW}, TX_{RO} observes TX_{RW}'s update on tuple O①, if and only if TX_{RO}'s timestamp (TS_{RO}) is not smaller than TX_{RW}'s timestamp (TS_{RW}).*

Proof First, if TX_{RO} observes TX_{RW}'s update on O, then TS_{RO} is not smaller than TS_{RW}. Because TX_{RW} updates O with TS_{RW} and content atomically (e.g., 2PL), TX_{RO} waits for TX_{RW}'s commit. Second, if TS_{RO} is not smaller than TS_{RW}, then TX_{RO} eventually observes TX_{RW}'s update on O. Assume TX_{RO} does not observe TX_{RW}'s update, then TX_{RO} reads O before TX_{RW} commits its update. One situation is TX_{RO} reads O before TX_{RW}'s request arrives, it leads a contradiction that TX_{RO} update O's timestamp to be TS_{RO} before the read. Another situation is TX_{RO} reads O after TX_{RW} calculates TS_{RW}, but before committing its update. This leads to the contradiction that TX_{RO} always waits for the concurrent TX_{RW} to commit (e.g., 2PL).□

Proof (of the Theorem) TX_1 updates A, TX_2 updates B, and TX_2 depends on TX_1. Assume read-only transaction TX_{RO} only observes TX_2's update on B, but does not observe TX_1's update on A (i.e., inconsistent reads).② From LEMMA 2, we have TS_{RO} is not smaller than TS_2, while TS_1 is larger than TS_{RO}. Therefore, we have TS_1 is larger than TS_2, which is contradictory to LEMMA 1. □

5.2.4 Hybrid timestamp and bounded staleness

Hybrid timestamp. The commit timestamp of a read-write transaction is derived from the timestamps of tuples in its read/write set, and the read timestamp of a read-only transaction can be any timestamp in the past, at present, or even in the future. Therefore, the local timestamp (LocalTS) is not essential for the correctness of DST. However, the read-only transaction may suffer from either staleness or performance issues if using an improper read timestamp. If the read timestamp is too small (past), the read-only transaction may read an excessively stale snapshot. If the read timestamp is too large (future), the read-only transaction will frequently install its read timestamp into tuples and wait until conflicting read-write transactions commit (DEP_READ in Figure 5–9).

DST adopts a combination of *physical* clock and *logic* counter as a hybrid timestamp. The 64-bit timestamp consists of the 48-bit physical part (high-order bits) and the 16-bit logic part (low-order bits). DST uses a loosely synchronized clock as the physical part

① It means TX_{RO}'s read on O happens after TX_{RW}'s update.

② The proof is also correct for TX_1 and TX_2 are the same transaction.

and uses a monotonically increasing counter as the logical part. At the beginning of the transaction, it will acquire a local hybrid timestamp composed of the current physical clock and zero-initialized logic counter (START in Figure 5–7 and *line:1* of ROTX in Figure 5–9). The logical part of the hybrid timestamp is used to avoid possible overflow of the physical part since the timestamp will be incremented when calculating the maximum timestamp (e.g., *line:3* of WRITE in Figure 5–7). On the other hand, the physical part of the hybrid timestamp is used to ensure the read-only transaction can read a fresh snapshot.

Bounded staleness. Based on the hybrid timestamp, `DST` can provide snapshot reads with bounded staleness.

Theorem (Bounded Staleness). *The updates of read-write transactions can be observed in at most Δ, where Δ is the maximal duration any machine needs to make its local clock increased by $2 \times \varepsilon$, and ε is the maximal clock drift between any two machines in the cluster.*

Proof sketch. First, we prove the following two lemmas:

Lemma 1. *Given a read-write transaction TX_{RW}, its commit timestamp (TS_{RW}) is not larger than $t_m + \varepsilon$, where t_m is the local machine time on TX_{RW} commits.*

Proof If $\mathsf{TS}_{\mathsf{RW}}$ is larger than $t_m + \varepsilon$, then there is a TX_{i} which accesses a tuple before $\mathsf{TX}_{\mathsf{RW}}$, and TS_{i} is larger than $t_m + \varepsilon$. As the timestamp is calculated from its local machine time or the tuples it accessed, we can inductively find a transaction TX_{j} whose timestamp is larger than $t_m + \varepsilon$, and it is calculated from its local machine time. It is a contradiction to the maximal clock drift between any two nodes is ε. □

Lemma 2. *For any read-only transaction TX_{RO} starts after TX_{RW} commits, its read timestamp TS_{RO} is larger than $t_m - \varepsilon$.*

Proof This follows that $\mathsf{TX}_{\mathsf{RO}}$ calculates its timestamp based on local machine time and the clock drift between any two nodes cannot be larger than ε. □

Proof (<u>of the Theorem</u>) With LEMMA 1 and 2, we can have a fact that, if $\mathsf{TX}_{\mathsf{RO}}$ starts after $\mathsf{TX}_{\mathsf{RW}}$, then $\mathsf{TS}_{\mathsf{RO}}$ cannot be smaller than $\mathsf{TS}_{\mathsf{RW}} - 2 \times \varepsilon$. Since any machine is able to increase its local machine time by $2 \times \varepsilon$ in Δ, we can conclude that the updates of $\mathsf{TX}_{\mathsf{RW}}$ will be visible in the duration of Δ. □

5.2.5 Failure and recovery

The CC protocol should provide a proper fault-tolerance scheme to recover the transactional system from various failures. For example, the primary-backup replication [132] is widely used to provide high availability in prior work [13, 18, 25]. The fault-tolerance schemes can usually work with various timestamp schemes by replicating tuples together with the commit timestamps of read-write transactions. However, the fully decentralized design of DST has two sides. The advantage of this approach is to avoid handling the failure of centralized timestamp oracle, which may cause a stop-the-world recovery [26]. The disadvantage is the potential cost to maintain the consistency of decentralized timestamps before and after some failure occurs.

An obvious, but costly solution is to replicate the read timestamps of read-only transactions together with tuples, as the commit timestamps of read-write transactions. Because the missing read timestamp may cause a new conflicting read-write transaction to use a smaller commit timestamp to write tuples; the read-only transaction may read some tuples with an old version and other tuples with a new version before and after the failure occurs, respectively.

To avoid replicating or persisting read timestamps, DST provides two alternative solutions that can be selected according to the behavior of workloads or the CC protocol associated. More specifically, after recovery, DST can selectively abort and re-execute either the remaining read-only transactions that read tuples on crashed machines or the remaining read-write transactions that write tuples on crashed machines. Consequently, there is no additional overhead and modification associated with the normal execution of transactions, regardless of which approach is selected.

5.3 Generality of DST

DST is a general timestamp scheme to enable efficient read-only transactions with little impact on read-write transactions. Hence, it is easy to integrate DST with various CC protocols, and DST can also cooperate with many optimizations [19, 163] on CC protocols. In this section, we lay out a general guideline for piggybacking DST on various CC protocols, and demonstrate the efficacy of this guideline by applying it to three representative transactional systems (DrTM+H, MySQL cluster, and Rococo) with different CC protocols (OCC, 2PL, and Rococo).

5.3.1 A guideline for integrating DST

Read-write transaction. DST should allocate a commit timestamp for the read-write transaction that is larger than any dependent transactions' timestamp. Thus, two following tasks (*RW1* and *RW2*) should piggyback on CC protocols.

1. *select a commit timestamp larger than both the current local timestamp and the timestamps of tuples in the read/write set.* (*RW1*)
2. *install the commit timestamp to tuples in the read/write set before the transaction commits.* (*RW2*)

Read-only transaction. DST should guarantee the read-only transaction can read the value of tuples up to the read timestamp. Thus, two following tasks (*RO1* and *RO2*) should piggyback on CC protocols.

1. *select an appropriate read timestamp according to the current local timestamp.* (*RO1*)
2. *ensure the tuple has an equal or larger timestamp before reading its value up to the read timestamp.* (*RO2*)

5.3.2 Case study

We now present how we apply DST to existing systems. Note that the description below focuses on the general comments about integrating DST; we omit a few details and corner cases due to space limitations.

DrTM+H. Optimistic concurrency control (OCC) is widely adopted by modern transactional systems [3, 5, 7, 13, 18, 25, 58, 166-167]. The read-only transaction in OCC will take two or more rounds of reads for consistent results without MVCC and timestamp schemes, due to conflicting read-write transactions. We use DrTM+H (§4) to demonstrate how DST piggybacks on OCC.

For the read-write transaction, we can obtain the timestamp of tuples in the read and write set when validating and locking them respectively and then derive a larger commit timestamp (*RW1*). Before committing, we should install the commit timestamp to the tuples in the read and write set (*RW2*). Note that there is no need to lock tuples in the read set since dummy timestamps from aborted transactions are benign. For the read-only transaction, all CC protocols share (almost) the same implementation (see Figure 5–9). The only difference is how to wait for conflicting transactions (*line:3* of DEP_READ).

For OCC, the conflicting read-write transaction will lock the tuple when installing its timestamp for updates. Therefore, similar to 2PL, the read-only transaction will confirm that the tuple is not locked before reading the value up to its read timestamp.

After applying DST to DrTM+H, DrTM+H can no longer use one-sided RDMA READ to execute queries in the read-only transaction. It is because the semantic of DST reads (see Figure 5–9) is more complicated than that of the READ. Thus, DrTM+H+DST executes DEP_READ with two-sided RDMA. This design choice may seem to contradict the design of DrTM+H, which has shown that reads with one-sided RDMA are faster (see §4.3). Nevertheless, with DST, DrTM+H only requires one roundtrip to commit the read-only transaction, and the read-only transaction will never abort. Hence, we believe it is a reasonable tradeoff. §5.5 will present how DST improves the performance of DrTM+H in workloads with read-only transactions.

MySQL cluster. Two-phase locking (2PL) is another classic CC protocol used by many transactional systems [20, 60, 168]. The read-only transaction in 2PL will be blocked without MVCC and timestamp schemes, due to conflicting read-write transactions. We use MySQL cluster [60] (v7.6.8) to show the integration of 2PL and DST, mainly following the specification in Figure 5–7 and 5–9.① To support the read-write lock in MySQL cluster, the transaction only needs to install timestamp into tuples in the read set atomically (i.e., compare-and-swap) and avoids overwriting a larger timestamp. Further, we leverage the lock queue mechanism in MySQL cluster to wait for conflicting transactions (*line:3* of DEP_READ in Figure 5–9), which avoids spinning on the tuple.

Rococo. Rococo [19] is a research CC protocol that outperforms traditional protocols under high contention workloads. It reorders conflicting read-write transactions instead of aborting them. Its protocol is a two-phase mechanism. The *start* phase explores a dependency graph, and then the *commit* phase executes conflicting transactions with a serializable order according to the dependency graph. The read-only transaction in Rococo is blocked until the completion of conflicting transactions and uses multiple rounds for reading consistent results.

To extend Rococo② with DST, the general idea is to use the dependency graph to collect timestamps of dependent tuples and derive a larger commit timestamp for the

① Although MySQL cluster uses read committed (RC) protocol by default, it also provides serializability by using per-row 2PL.

② Source code: https://github.com/shuaimu/rococo.

read-write transaction in the start phase (*RW1*). Then the commit timestamp can be installed to tuples in the commit phase (*RW2*). For the read-only transaction, DST reuses the blocking mechanism in Rococo to wait for conflicting transactions (*line:3* of DEP_READ in Figure 5–9).

5.4 Discussion

Performance overhead. Compared to traditional centralized timestamp schemes, DST needs to update the timestamps of tuples in the read set for read-write transactions, which may incur additional costs. However, these operations can easily piggyback on original operations in CC protocols (see Figure 5–7), like the locking and the validating in 2PL and OCC, respectively. Moreover, the read-only transaction may also update the timestamps of tuples, while it only happens as the read timestamp is larger (DEP_READ in Figure 5–9). Thus, using a hybrid timestamp can effectively mitigate it. To study the potential performance overhead for DST, we designed two microbenchmarks to model the worst-case scenarios (see §5.5.4), and the experimental results show limited cost.

Range scans and phantom reads. DST relies on the CC protocol to detect conflicts, including range scans and phantom reads, and also needs to assign timestamps to certain "guard" (e.g., index structures) [169-170]. For example, the next-key locking mechanism [171] is widely used by 2PL to support range scans. The CC protocol acquires such locks, and DST assigns timestamps to them. For OCC, DST assigns timestamps to the internal nodes in the index structure as the versions during the validation phase.

The SNOW theorem. The SNOW Theorem [59] describes the fact that strict serializability (*S*), non-blocking read-only transactions (*N*), one-response from each tuple (*O*), and compatible with conflicting write transactions (*W*) cannot be satisfied at the same time. Yet, SNOW-optimal and latency-optimal read-only transactions can achieve three of the above properties (i.e., *N*+*O*+*W*) without strict serializability (*S*). DST also relaxes *S* to serializability for read-only transactions, and satisfies *O* and *W* apparently. DST can simply satisfy *N* by letting reads return a relatively stale data. However, it may be not reasonable; thus, DST chooses to provide bounded staleness with much fewer blocking operations (see §5.5.4).

Session strict serializability. DST only ensures *serializability* to read-only transactions rather than *strict serializability*, while it is equal to or better than most snapshot-based sys-

Table 5–1 Additional measurement clusters used in DST.

Cluster	#Nodes	Descriptions
AWS	32	r4.2xlarge (8x vCPU, 61GB DRAM, up to 10GbE)
VALE	16	2x Intel Xeon E5-2650 v4 (12 cores), 128GB DRAM, 1x Intel I350 10GbE

tems [15, 20, 59-60]. Further, DST can provide session guarantees [160, 172] (i.e., read-my-write [173] and read-after-write [174] consistency), such that read-only transactions can always observe the latest updates of read-write transactions within the same session (e.g., issued from the same client or handled by the same server). DST returns the commit timestamp to the session manager (e.g., client or server) after the transaction commits. The session manager will always use the largest observed commit timestamp as the read timestamp for successive read-only transactions.

5.5 Evaluation

As mentioned in §5.3, we have integrated DST with three representative transactional systems, namely DrTM+H, MySQL cluster, and Rococo, with different CC protocols. We also implemented two centralized timestamp schemes (GTS and VTS) by following the state-of-the-art [26, 158]① with many carefully tuned optimizations (e.g., batching requests [66, 158], cooperative multitasking [25], timestamp compression and dedicated fetch thread [26]). These optimizations have significant performance improvements on GTS and VTS. For example, cooperative multitasking improves the peak per-machine throughput of GTS on DrTM+H by 3.04X, and timestamp compression improves VTS by 2.7X on a 16-node cluster.

Testbed and setup. Since DST is a general timestamp scheme, besides clusters used in Table 2.1, we also evaluate it in traditional clusters without RDMA capability (see Table 5–1). For each system, we dedicate on machine in each cluster as the timestamp oracle, even only GTS and VTS need. Other machines serve as both database nodes and clients. Further, we configure these machines in a *symmetric* setting [13]—namely each machine both executes transactions and store database data—to better saturate each system's peek performance.

① Different than Percolator [158], we use the stabilization process to avoid holding locks when acquiring write timestamp, since it will significantly increase transaction abort rate.

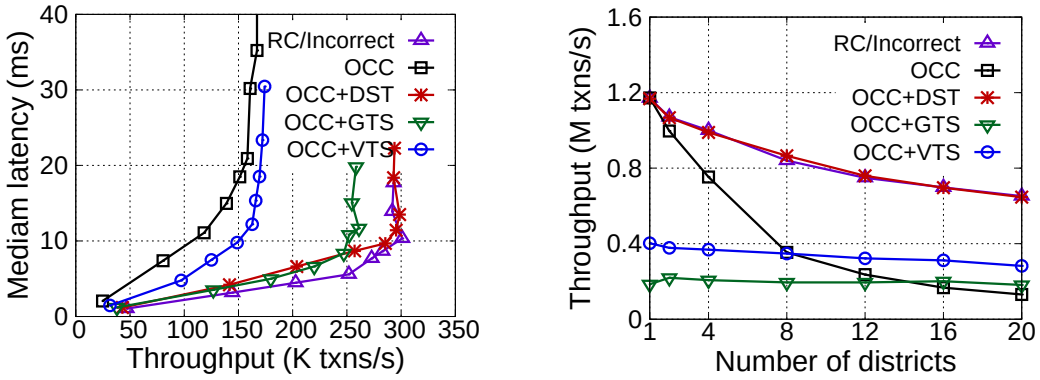


Figure 5–10 Performance of (a) TPC-E and (b) TPC-C on **AWS**.

Benchmarks and performance overview. As the performance benefit of using MVCC and snapshot reads is sensitive to characteristics of read-only transactions in OLTP workloads, we chose three different benchmarks, namely TPC-E, TPC-C, and SmallBank, to show the benefits of DST comprehensively. TPC-E [141] presents the workload of a brokerage firm with a high proportion of read-only transactions (79% of the standard mix) and complicated operations (massive range queries and distributed accesses). DST is expected to improve the performance much compared to the vanilla CC protocols for this target workload, with *a relaxed consistency level from strict serializability to serializability*. TPC-C [124] simulates a warehouse-centric order processing application with a few read-only transactions (8% of the standard mix). DST is expected to show gradual improvement with the increase of execution time in read-only transactions (not affect proportion). We increase the number of districts (one district by default) accessed by the read-only *stock-level* transactions (4%). SmallBank [135] models a simple banking application where transactions perform very simple read and write operations (less than four) on user accounts. DST is expected not to incur perceptible overhead and show order-of-magnitude speedup compared to centralized timestamp schemes (GTS and VTS). In all benchmarks, DST should achieve close to optimal performance using RC (5%) but without compromising correctness, which can be backed by the experimental results of DST on motivating microbenchmarks (see Figure 5–6).

5.5.1 DrTM+H

This section presents the evaluation results of DST on DrTM+H. Due to space limitations, we do not report the experimental results on SmallBank, which are as expected.

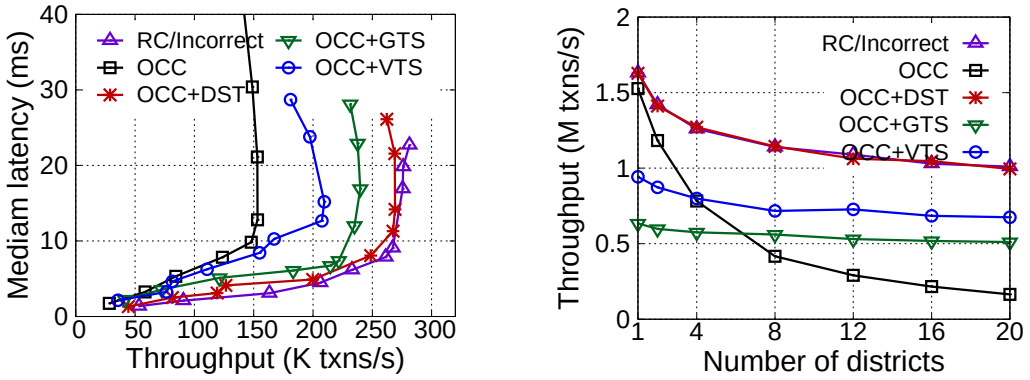


Figure 5–11 Performance of (a) TPC-E and (b) TPC-C on **VALE**.

TPC-E. Figure 5–10(a) shows the results of TPC-E on AWS. TPC-E has a high proportion of read-only transactions, and most of them are distributed. Compared to using snapshot reads (GTS, VTS, and DST), the vanilla OCC protocol provides *strict serializability* and requires an additional round to validate tuples in the read set. Thus, many read-only transactions will abort under heavy workloads. As a reference, RC can outperform OCC by 1.79X (yet with incorrect results), since it simply skips the validation phase. DST achieves almost the same performance as RC, as it also avoids the validation phase and never aborts read-only transactions. Differently, DST ensures the read-only transaction can read a consistent yet fresh snapshot. Moreover, compared to GTS and VTS with the same consistency level (*serializability*), DST can outperform the throughput of them by 1.16X and 1.72X, respectively. Because DST omits the communication to the timestamp oracle and avoids large traffic size due to using a fully decentralized design and scalar timestamps (see §5.1.3).

We further evaluate TPC-E on VAL. As shown in Figure 5–11(a), DST can still achieve similar performance as RC and provides 1.13X and 1.29X speedup compared to GTS and VTS, respectively. VTS performs slightly better on VAL due to using a relatively smaller vector timestamp.

TPC-C. Figure 5–10(b) and Figure 5–11(b) show the peak throughput of TPC-C on AWS and VAL with the increase of districts accessed by the read-only *stock-level* transactions. Note that in default TPC-C accesses one district (the first data point of every line). Besides, we retain all default settings, like the proportion of *stock-level* transactions (4%).

As shown in Figure 5–10(b), when accessing one district, DST has a very close performance compared to RC. These results indicate that DST has little overhead to read-write transactions. In comparison to DST, GTS and VTS are 6.29X and 2.93X slower than

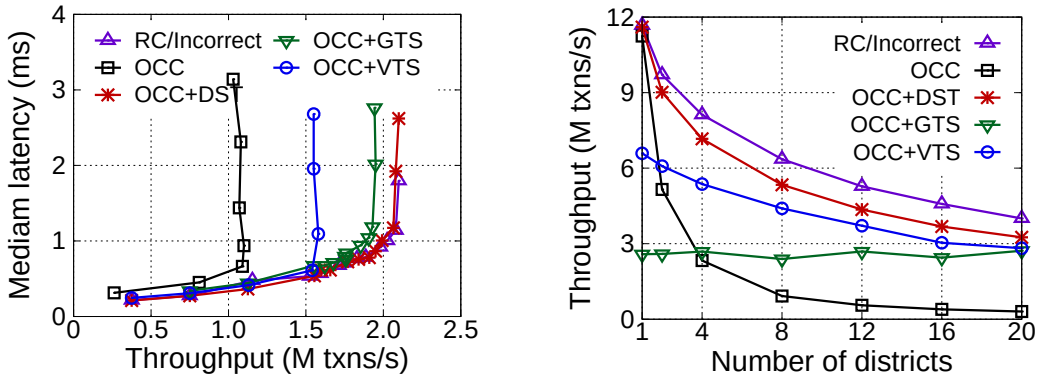


Figure 5–12 Performance of (a) TPC-E and (b) TPC-C on **VAL**.

RC, due to the significant cost for maintaining centralized and/or vectorized timestamps (see §5.1.3).

OCC performs well on the original TPC-C due to the limited read-only transactions in the standard-mix (8%). On the other hand, when increasing the execution time of read-only *stock-level* transactions (by accessing more districts), the performance difference between RC and OCC is more evident because OCC has more overheads for validating the read-set of the *stock-level*. DST still performs close to RC and is 4.94X faster than vanilla OCC (accessing 20 districts) with a relaxed consistency level. Finally, DST still outperforms VTS and GTS by 2.29X and 3.56X when accessing 20 districts, respectively.

In Figure 5–11(b), the performance of DST is also very close to RC for TPC-C on VAL. On the other hand, the overhead of GTS and VTS still incurs up to 2.57X (from 1.95X) and 1.73X (from 1.47X) slowdown, compared with DST. Different than AWS, the lower latency of network round-trip on VAL (90μs) is beneficial for centralized timestamp schemes, but the effect is quite limited.

Using RDMA. By using 100Gbps RDMA, the CPU may become the bottleneck in the timestamp oracle for GTS, about 3.0 M ops/s (see §5.1.3). For VTS, the timestamp oracle will not limit the performance of TPC-E and TPC-C with only 16 machines, while the increase of transaction abort rate (due to optimizations [26]) and large traffic size still incur non-trivial costs, compared to the decentralized scalar timestamp (like DST).

As shown in Figure 5–12, the fast network (RDMA) in VLR has a significant positive impact on all of the settings, as expected. For TPC-E, DST still outperforms GTS and VTS by 1.07X, and 1.32X, respectively. RDMA reduces the overhead of centralized timestamp allocation for GTS, while the impact of traffic size in VTS remains. For TPC-C DST is still 4.49X (from 1.19X) and 1.76X (from 1.15X) faster than GTS and VTS.

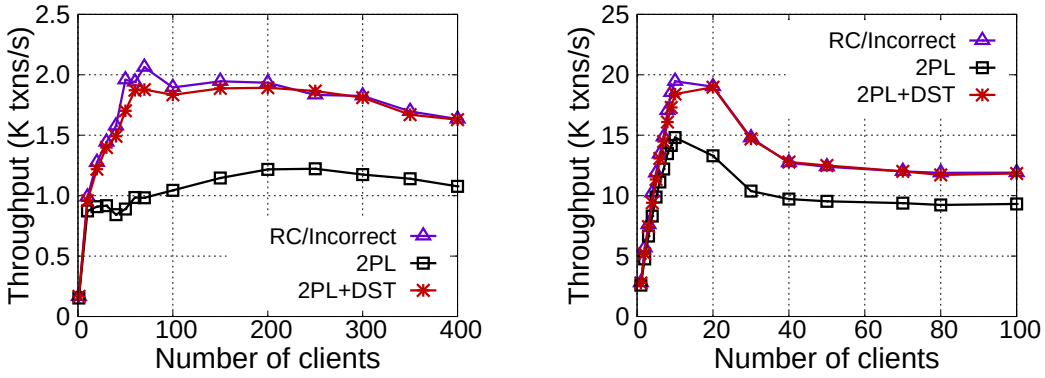


Figure 5–13 Performance of (a) `TPC-C` and (b) `SmallBank` for `MySQL cluster` with different CC protocols on **VALE**.

5.5.2 `MySQL cluster`

We evaluate `MySQL cluster` with `DST` by using `TPC-C` and `SmallBank` on VAL. We increase the number of clients until the throughput is saturated. As shown in Figure 5–13, with `DST`, `MySQL cluster` achieves up to 1.91X (from 1.09X) and 1.28X (from 1.07X) higher throughput for `TPC-C` and `SmallBank`, respectively. The main reason is due to enabling snapshot reads to avoid blocking for the read-only transactions. It also mitigates the contention in the read-write transactions. `DST` is more effective in `TPC-C` since it is more sensitive to blocking time from conflicting transactions due to relatively longer execution time compared to `SmallBank`. On the other hand, `DST` can provide comparable performance to RC but still guarantee serializability for correctness.

5.5.3 `Rococo`

We follow the methodology (benchmarks and settings) in prior work [19, 59] to evaluate `Rococo` on VAL.①

Figure 5–14 shows the performance of `Rococo` by increasing the number of concurrent requests per server. In Figure 5–14(a), using `DST` on `Rococo` can improve the throughput of *new-order* transactions by 2.09X with 100 concurrent requests per server, due to reducing transaction aborts and skipping the validation process in read-only transactions. For example, less than 4% of *stock-level* transactions can be committed when there are more than 50 concurrent requests per server. Thus, the server CPU is wasted

① We try our best to compare with ROCOCO-SNOW [59], which also optimizes the read-only transaction of ROCOCO. Unfortunately, it failed to run on our testbed.

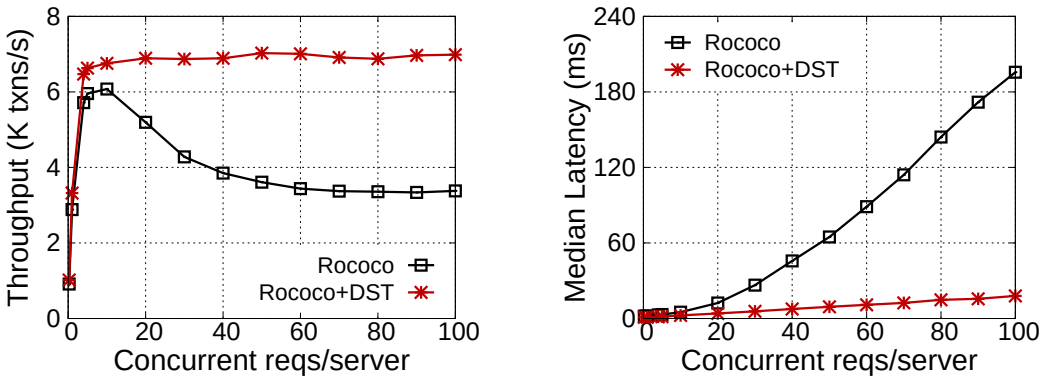


Figure 5–14 (a) Throughput of *new-order* transactions and (b) median latency of *stock-level* transactions in TPC-C mixed workload on **VALE**.

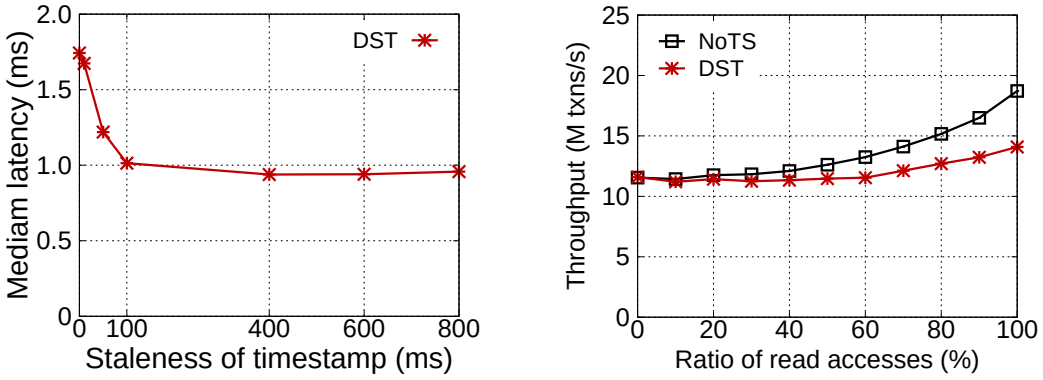


Figure 5–15 (a) The impact of latency for read-only transactions by using stale timestamp. (b) The overhead of DST with the ratio increase of read-only accesses in read-write transactions.

on retrying and validating read-only transactions. Further, as shown in Figure 5–14(b), Rococo+DST has a much lower median latency of (read-only) *stock-level* transactions, thanks to reading a consistent snapshot by one round of execution without validation.

5.5.4 A study of DST cost

To study the overhead from blocking and additional timestamp updates in DST, we use two workloads that share most characteristics with TPC-C. We tuned the workload behavior to better reflect these overheads.

Blocking overhead. One read-only transaction accesses 10 tuples, while another write-only transaction continuously updates these tuples with locking. This is considered as the worst-case scenario for DST, since the read tuples are locked most of the time. Figure 5–15(a) shows the impact on the median latency of read-only transactions when varying the

staleness of read timestamps. When using the current time (staleness=0ms) as the read timestamp, 10% of the reads are blocked by concurrent writes, which incur 83% overhead of the median latency (1.72ms vs. 0.96ms). With the increase of staleness (smaller timestamp), fewer reads are blocked since the tuples have been updated with larger commit timestamps. The blocking overhead becomes trivial when staleness exceeds 100ms. Note that this is an extreme case for blocking: reads always touch the locked tuples. In reality, we only observe about 160 and 200 blocks per second at each machine under peak throughput for TPC-E and TPC-C, respectively.

Timestamp update overhead. Figure 5–15(b) presents the overhead of DST to read-write transactions. Each transaction accesses 10 tuples, while some tuples are made read-only. We can see that when all tuples are updated, there is no overhead for DST, since the timestamp update will piggyback on the unlock operation. With the increase of read(-only) ratio, DST adds up to 25% overhead to the overall performance. Because DST will update the timestamp of tuples even just reading them, which requires additional synchronizations using atomic operations. Fortunately, most of the read and write sets are overlapping in OLTP workloads.

5.6 Related work on timestamps

Using timestamp for snapshot reads. A centralized timestamp is the most straightforward way to support MVCC for snapshot reads, which is widely adopted by centralized systems [5, 164-165, 175-178]. Many distributed systems also use timestamps to provide MVCC [15, 20, 26, 156, 158, 179-180], while most of them only support weaker isolation guarantees (e.g., Snapshot Isolation) [15, 26, 156, 158]. For example, Percolator [158] uses a global timestamp oracle, and NAM-DB [26] uses vectorized centralized timestamps. Spanner [20] is based on a combination of 2PL and MVCC developed in previous decades [157]. Spanner relies on TrueTime API to provide scalable timestamps for strict serializable read-only transactions and snapshot reads, which requires specific hardware (GPS and atomic clocks) to ensure clocks with bounded uncertainty. Further, the read-write transactions still require blocking to ensure the match of timestamp and transaction ordering. DST chooses to support serializable read-only transactions with bounded staleness. It requires no external timestamp service and does not block read-write transactions. RAMP [181] introduces Read Atomic isolation and uses timestamps to identify

and retry inconsistent reads. TxCache [182] provides a distributed transactional cache that always returns a consistent snapshot by lazily selecting the timestamps for transactions. Causalspartan [183] also uses Hybrid Logical Clocks to optimize timestamps in causal consistency systems.

DST naturally piggybacks timestamp allocation to existing CC protocols, which avoids additional communications for maintaining timestamps. Further, DST can work with a border range of CC protocols and is orthogonal to prior optimizations on CC protocols [19, 163].

Using timestamp for concurrency control. Many systems directly leverage a timestamp-based mechanism to commit transactions orderly [14, 160, 184-188]. CLOCC [185] combines optimistic timestamp ordering with loosely synchronized clocks, which avoids a centralized counter for checking serializability in the original OCC protocol [22]. Granola [186] uses the timestamp based on a distributed voting mechanism to order independent transactions deterministically and treats distributed transactions in locking mode. TAPIR [14] uses loosely synchronized clocks at the clients in OCC's validation for read-write transactions. The clock drift in these systems will increase false aborts and impact the execution of read-write transactions. On the contrary, the clock drift in DST only affects the freshness of snapshot reads.

Several variant timestamp schemes have been proposed to *mitigate the cost from frequent aborts due to the violation between timestamp and transaction ordering*. Lomet et al. [189] introduce timestamp ranges to reduce transaction conflicts, while the timestamp management is centralized. MaaT [190] uses dynamic timestamp ranges to avoid distributed locking for the atomic commitment in OCC. Further, some prior systems also use decentralized timestamp schemes, but most of them focus on *optimizing one particular CC protocol*. TicToc [191] introduces a data-driven timestamp scheme for multicore platforms, which allows each read-write transaction to compute a valid commit timestamp from tuples before it commits. However, the read-only transaction still needs additional validations and incurs more aborts due to conflicts. Clock-SI [192] also uses loosely synchronized clocks to create consistent snapshots with fewer network round trips, while snapshot reads must be delayed due to concurrent transactions and clock drift. Sundial [193] uses logical timestamps as leases to reduce aborts in distributed read-write transactions. Pelieus [173] derives a commit timestamp for the read-write transaction from all involved servers (not tuples), which is used in the validation phase with different

rules to support different concurrency levels (e.g., SI and Serializability).

Differently, DST is a decentralized timestamp scheme for various CC protocols and can piggyback on them efficiently. Thus, DST will not interfere with the execution of read-write transactions and has no need of extra validations and aborts.

5.7 Conclusion

This chapter presents DST, a decentralized scalar timestamp that can unify timestamp management with existing CC protocols. We have integrated DST with two classic protocols, namely 2PL and OCC, and a recent research proposal, Rococo. Our evaluation with three transactional systems (MySQL cluster,DrTM+H and Rococo) and three benchmarks (TPC-E, TPC-C and SmallBank) confirmed the benefit of DST.

DST is a general timestamp scheme, which applies to settings with or without RDMA. Our experience with DST reveals that besides offloading transaction protocols to new hardware features (e.g., RDMA), people should also improve transaction protocol for different workloads. DST enables efficient MVCC for distributed transactions and can improve the performance of read-only transactions.

Chapter 6 Put It All Together: A Fast Distributed Transaction System

In this chapter, we put all our prior designs together (§2—§5) and present DrTM+X, a fast distributed transaction system using RDMA and NVM. Since we have conducted extensive end-to-end comparisons of its core components (e.g., DrTM+H (§4)) with their counterparts, the evaluations in this chapter focus on the uncovered parts, namely, how we boost the secondary index lookup with XStore and the efficiency of durability support with RDPMA in DrTM+X.

Overview. Figure 1–2 presents an overview of DrTM+X, which has two system layers atop of R2 and RDPMA:

- **Storage layer (§6.1).** DrTM+X adopts an RDMA-friendly layer to store all the data required for distributed transactions. We follow priori designs to use a key-value interface for transactions [13, 17, 25]. The key-value store adopts a hybrid deployment of DrTM-KV [17] for unordered accesses and XStore (§3) for ordered key-value accesses.
- **Transaction execution layer (§6.2).** The coordinator at the transaction layer receives the transaction requests from clients and executes them with DrTM+X' s protocol. DrTM+X mainly leverages DrTM+H (§4) to execute read-write transactions and uses DST (§5) to accelerate read-only transactions. The read-write transactions use logging to ensure high availability (§4.3.4). The transaction layer can further deploy Optane PM to support fast durable transactions (§6.3) with the help of RDPMA (§2.3).

6.1 RDMA-friendly storage layer

As we have mentioned before, DrTM+X adopts an RDMA-friendly key-value store to store transaction' s data. Specifically, all the static key-value operations (e.g., GET) are offloaded to one-sided RDMA, while complex dynamic key-value operations (INSERT) are handled by two-sided RDMA. DrTM+X further differentiates the mechanisms to support ordered key-value and unordered key-value operations, since unordered key-value opera-

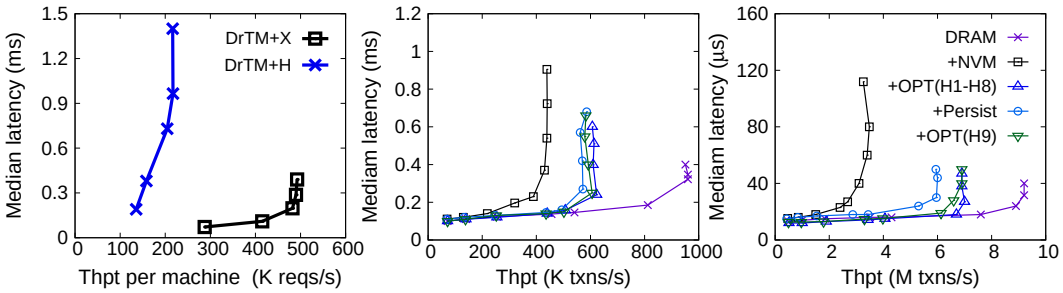


Figure 6–1 (a) Comparison of DrTM+H and DrTM+X on TPC-C. DrTM+X further adopts XStore. The performance of DrTM+X on (b) TPC-C/no and (c) SmallBank.

tions backed by hash index can better utilize one-sided RDMA than the ordered key-value operations backed by tree-based index (§3.1). In default, DrTM+X adopts an in-memory setup. Nevertheless, DrTM+X can deploy Optane PM to support a larger memory capacity. Since Optane PM has the same interface as DRAM, DrTM+X can seamlessly switch between these two types of memory.

Unordered access. For data that only has unordered accesses, DrTM+X use DrTM-KV [17], a state-of-the-art unordered key-value store to store them. DrTM-KV uses an RDMA-friendly hash structure—cluster–chaining to facilitate hash lookup over one-sided RDMA and concurrent updates at the server. In most cases, it only requires one one-sided RDMA READ to retrieve the transaction ' s data. We have used it as the default storage engine in DrTM+H and evaluate its effectiveness in common transaction workloads (§4).

Ordered access. Ordered key-value access is an important workload in distributed transactions, e.g., searching the secondary index. DrTM+X uses XStore (§3) to support such accesses. XStore adopts a learned approach to accelerate ordered key-value access over one-sided RDMA. It can retrieve the value of a given key only using two network roundtrips.

6.1.1 Evaluations

As we have extensively evaluated the effectiveness of XStore as a stand-alone key-value store (§3) and the performance of unordered accesses in DrTM+H (§4) , we focus on the performance of unordered key-value access in distributed transactions in this section.

Experimental setup. We use TPC-C [124] to compare the performance of DrTM+X and DrTM+H (§4), since they adopt the same concurrency control protocol, allowing an apple-to-apple comparison. In default, DrTM+H uses two-sided RDMA to support ordered key-value accesses while DrTM+X adopts XStore.

We run both systems in an asymmetric setting, which is widely adopted in cloud databases [26, 94, 194]. More specifically, we deploy 96 warehouses on four data servers and use the rest of the machines in our testbed as clients in the **VAL** cluster (see Table 2–1). Both DrTM+H and DrTM+X rely on the data server to update tuples, while DrTM+X uses one-sided RDMA READs to retrieve tuples from the data server with the help of XStore. Therefore, we use a read-heavy TPC-C workload in the experiment, which consists of NEW-ORDER transactions (10%) and ORDER-STATUS transactions (90%). NEWORDER transaction inserts a new order with five to fifteen order lines; ORDERSTATUS transaction retrieves the recently inserted orders first and then scans related order lines.

Performance. As shown in Figure 6–1 (a), DrTM+X improves the peak throughput of DrTM+H by 2.27×, reaching 490K reqs/s. DrTM+H is bottlenecked by server CPUs since the data server traverses the index and performs the read request locally. Consequently, the read requests of ORDERSTATUS transactions would compete CPUs with the write requests of NEWORDER transactions at the servers. Differently, DrTM+X uses RNICs at the clients to lookup and retrieve tuples for ORDERSTATUS transactions. Hence, it relaxes the burden on server CPUs and improves performance significantly.

6.2 RDMA-friendly transaction execution layer

The transaction layer of DrTM+X mainly adopts DrTM+H (§4) to execute transactions. DrTM+H uses an RDMA-friendly optimistic concurrency control to provide strict serializability and primary-backup replication with vertical paxos to provide high availability. It chooses the optimal RDMA primitive for each of the transaction' s execution phase. Based on DrTM+H, DrTM+X further supports MVCC with the help of DST (§5). Table 6–1 summarizes DrTM+X' s primitive choice with other RDMA-enabled distributed transaction systems. DrTM+X only makes a different choice than DrTM+H when the user uses DST to accelerate read-only transactions.

Table 6–1 A comparison of the primitive choice of DrTM+X with existing RDMA-enabled transaction systems. DrTM+X only makes a different choice than DrTM+H when supports MVCC with DST. I and II stand for one-sided and two-sided primitives. '-' means not required. **E** states for the execution phase, **V** states for the validation phase, **L** represents the logging phase, **C** represents the commit phase and **R** states for the read phase. §4 described these phases.

	Read-write TX				Read-only TX	
	E	V	L	C	R	V
FaRM [13]	I	II+I	I	II	I	I
DrTM+R [18]	I	I+I	I	I+I	I	I
FaSST [25]	II	II	II	II	II	II
DrTM+H (§4)	I/II	I/II	I	I/II	I/II	I
DrTM+X	I/II	I/II	I	I/II	II/I	-/I

Execution. DrTM+X uses a hybrid design of one-sided READs with caching and two-sided RPC, as DrTM+H. If MVCC is enabled, the execution will further read the timestamp of the record according to DST (§5.3.2).

Validation. If MVCC is disabled, DrTM+X follows DrTM+H, which selects RDMA primitive based on whether the NIC has an atomic issue. Otherwise, DrTM+X use two-sided RDMA since integrating with DST requires more functions executed in the validation phase.

Logging. As DrTM+H, DrTM+X always uses one-sided WRITEs to replicate transaction logs to all backups and uses two-sided primitive to lazily reclaim logs on backups.

Commit. If MVCC is disabled, DrTM+X uses one-sided WRITEs to commit if one-sided ATOMIC is used in the validation phase. Otherwise DrTM+X uses two-sided RDMA. Besides, DrTM+X always enables the passive ACK optimization (§2.2.3) in the commit phase. If MVCC is enabled, DrTM+X only uses two-sided RDMA in this phase.

Read. If MVCC is disabled, DrTM+X follows DrTM+H, which uses the same primitive choice as the execution phase. Otherwise, DrTM+X uses two-sided RDMA since traversing the multiple versions with one-sided RDMA is inefficient.

Validation (Read-only). If MVCC is disabled, DrTM+X adopts one-sided RDMA for the validation in read-only transactions. Otherwise, DrTM+X does not need to execute this phase because it uses DST to guarantee that results read in the **Read** phase are always consistent.

Since we have extensively evaluated each design choice of the transaction execution layer in §4.4 and §5.5, we omit the evaluation in this section.

6.3 Supporting durability with RDPMA

Finally, we describe how we use RDPMA (§2.3) to support durable transactions in DrTM+X. As we have mentioned in the introduction, DrTM+X leverages RDMA and NVM to reduce the durability cost in distributed transactions. To add durability support with Optane PM and RDMA, we need to put all the storage (including transaction' s data and log) in Optane PM, and ensures that when the logging phase finishes, the log should be persistently written to the Optane PM. Therefore, we first use RDPMA for all the data allocation in the storage layer. Second, we use the interface in RDPMA to write logs and records.

Using RDPMA alone is not sufficient to fully leverage Optane PM, because several RDMA-NVM optimization hints summarized in §2.3.4 depend on application semantics (e.g., **H1**). Thus, DrTM+X further applies several optimizations according to the study. These optimizations (some have been already incorporated in RDPMA) are summarized as follows:

- Separate the memory pool from different sockets to avoid cross-socket NVM access (**H1**).
- Configure DrTM+X with DDIO disabled (**H3**). Note that RDPMA has incorporated this optimization.
- Use `ntstore` to optimize the *commit phase* (**H4**). RDPMA has incorporated this optimization.
- Align and pad logs/records larger than 256B to XPLine granularity (**H5**). RDPMA has incorporated this optimization.
- Align and pad logs/records smaller than 256B to 64B granularity (**H6** + **H7**). RDPMA has incorporated this optimization.
- Implement a DRAM-based lock service for the *validation phase* (**H8**). Note that it is safe not persisting the locks in NVM because DrTM+X does not require the

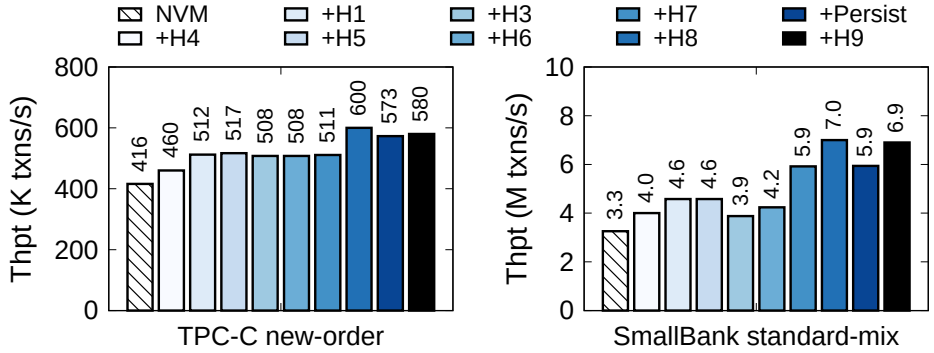


Figure 6–2 The contribution of optimizations from RDPMA to the throughput of DrTM+X for (a) TPC-C and (b) Smallbank.

locks to be persistent even when durability is enabled.

- Implement remote persistent log with **H9** in one roundtrip. RDPMA has incorporated this optimization.

6.3.1 Evaluations

Experimental setup. We evaluate the performance of DrTM+X by comparing the vanilla DrTM+H without optimizations mentioned in the previous chapter. We use the **R74V** cluster (see Table 2–1) where the machine capable of NVM serves as the server, while other machines are configured as clients. Like DrTM+H, we use two representative workloads, namely TPC-C/no and Smallbank for the evaluations. A detailed descriptions of the workloads can be found at §4.3.

Comparing targets. In Figure 6–1 (b) and (c), **DRAM** is the DrTM+X without durability support. **+NVM** runs it on Optane PM, and **+OPT(H1-H8)** further applies optimizations (§6.3). **+Persist** adopts an existing approach[75] to support durability atop of **+OPT(H1-H8)**. Finally, **+OPT(H9)** optimizes **+Persist** with **H9**. It is the DrTM+X that supports durable transactions.

Performance without durability. Athough this section focuses on durable transactions in DrTM+X. DrTM+X can also leverage Optane PM to support a large DRAM capacity, More importantly, most optimizations from this section also apply to this scenario (**H1–H8**). Therefore, we first present the performance of DrTM+X using Optane PM without guaranteeing durability.

Figure 6–1 (b) and (c) present the throughput-latency results of both workloads. We plot the graph by increasing the number of clients until the throughput is saturated. **+OPT(H1-H8)** improves the DrTM+X's performance under TPC-C/no and SmallBank by 1.45X and 2.20X, respectively.

To analyze the contributions of each optimization, Figure 6–2 further presents a factor analyses of evaluation results on TPC-C/no and SmallBank, respectively. First, we can see that several hints are beneficial to both workloads. For example, **H8** (use atomic operations less on NVM) speedups TPC-C/no and SmallBank by 1.17X and 1.19X, respectively. On the other hand, some hints have negative effects for certain workloads: SmallBank drops 15% throughput when adding **H3**, this is because **H3** is only beneficial when the application is bottlenecked by NVM's bandwidth. Finally, some hints have more contributions to SmallBank than TPC-C/no. For example, **H7** has a 1.4X speedup on SmallBank but does not affect TPC-C/no. **H7** only improves transaction utilizations of NVM write throughput, while SmallBank is more sensitive to the NVM write throughput utilization due to its simpler workloads.

Performance with durability. As shown in Figure 6–1 (b) and (c), supporting durable transaction (**+Persist**) adds 5% and 15% performance overhead to **+OPT(H1-H8)** on TPC-C/no and SmallBank, respectively. The overhead is from the additional one-sided RDMA READ at the logging phase. Hence, reducing this network roundtrip with **H9** in RDPMA improves TPC-C/no and SmallBank's performance by 1.01X and 1.17X, respectively.

6.4 Conclusion

This chapter presents DrTM+X, a fast distributed transaction processing system that incorporates all the techniques proposed in this dissertation. DrTM+X can fully utilize RDMA and NVM for executing distributed transactions. Its storage layer adopts XStore for data storage, and its transaction execution layer uses DrTM+H and DST to provide serializability and high availability. Finally, we discuss how to use RDPMA to support durable transactions. Extensive evaluations (including the ones presented in the previous chapters) have shown the benefits of each design choice in DrTM+X.

Conclusion

This dissertation presents a study of how to fully leverage RDMA and NVM to reduce the cost of executing distributed transactions. We show that a bottom-up approach is effective for finding the proper way of using RDMA and NVM in this scenario. Starting from a systematic study of hardware features, we built frameworks to hide specific optimizations to efficiently coordinate RDMA and NVM together. Second, we shown that a learned approach can efficiently bridge the semantic gap between ordered data access—a key function in transactions—with RDMA. Third, we demonstrated how to best select RDMA primitives for RDMA-enabled transactions with a phase-by-phase analysis. Fourth, we presented how to use a new decentralized timestamp scheme to avoid performance and scalability bottleneck—that can happen in traditional centralized timestamp in distributed transactions—under fast-interconnects like RDMA. Finally, we brought all the above techniques together and present how to build a fast distributed transaction processing system with RDMA and NVM.

We believe techniques presented in this dissertation can also benefit other RDMA/RDMA-NVM-enabled systems. Some of them can also apply to traditional distributed transaction processing systems without RDMA or NVM. Finally, we hope the study in this dissertation can stimulate and provide a guideline for future co-design with RDMA and NVM.

Acknowledgements. I would like to thank professor Rong Chen, who co-advisors me throughout the p.h.d. It is him who brings me to the world of system research. He always generously supports and helps me during the research. I would also like to thank my advisor, professor Binyu Zang, for his generous support in my research. I would like to thank professor Haibo Chen, he always gives us valuable insights. There are so many others to thank. Professor Yubin and Professor Zhaoguo, it is always a wonderful experience discussing with you. I would like to thank my girlfriend Xiating, who helps me a lot both in research and life. Finally, I would like to thank my parents who brought me there.

Bibliography

[1] Michael Stonebraker. The Traditional RDBMS Wisdom is All Wrong[EB/OL]. 2013. https://downloads.voltdb.com/datasheets_collateral/voltdb_STONEBRAKERSAYS_webinar2.pdf.

[2] Dragojević A, Narayanan D, Hodson O, et al. FaRM: Fast Remote Memory[C/OL]. in: NSDI'14: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. Seattle, WA: USENIX Association, 2014: 401-414. http://dl.acm.org/citation.cfm?id=2616448.2616486.

[3] Wang Z, Qian H, Li J, et al. Using Restricted Transactional Memory to Build a Scalable In-memory Database[C/OL]. in: EuroSys'14: Proceedings of the Ninth European Conference on Computer Systems. Amsterdam, The Netherlands: ACM, 2014: 26:1-26:15. http://doi.acm.org/10.1145/2592798.2592815. DOI: 10.1145/2592798.2592815.

[4] Cowling J, Liskov B. Granola: low-overhead distributed transaction coordination[C]. in: USENIX ATC'12: Proceedings of the 2012 USENIX conference on Annual Technical Conference. 2012.

[5] Diaconu C, Freedman C, Ismert E, et al. Hekaton: SQL server's memory-optimized OLTP engine[C]. in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. 2013: 1243-1254.

[6] Thomson A, Diamond T, Weng S C, et al. Calvin: Fast Distributed Transactions for Partitioned Database Systems[C/OL]. in: SIGMOD'12: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. Scottsdale, Arizona, USA: ACM, 2012: 1-12. http://doi.acm.org/10.1145/2213836.2213838. DOI: 10.1145/2213836.2213838.

[7] Tu S, Zheng W, Kohler E, et al. Speedy Transactions in Multicore In-memory Databases[C/OL]. in: SOSP'13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. Farminton, Pennsylvania: ACM, 2013: 18-32. http://doi.acm.org/10.1145/2517349.2522713. DOI: 10.1145/2517349.2522713.

[8] Zhang Y, Power R, Zhou S, et al. Transaction chains: achieving serializability with low latency in geo-distributed storage systems[C]. in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. 2013: 276-291.

[9] Narula N, Cutler C, Kohler E, et al. Phase Reconciliation for Contended In-memory Transactions[C/OL]. in: OSDI'14: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Broomfield, CO: USENIX Association, 2014: 511-524. http://dl.acm.org/citation.cfm?id=2685048.2685088.

[10] Zheng W, Tu S, Kohler E, et al. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism[C/OL]. in: OSDI'14: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Broomfield, CO: USENIX Association, 2014: 465-477. http://dl.acm.org/citation.cfm?id=2685048.2685085.

[11] Xie C, Su C, Kapritsos M, et al. Salt: Combining ACID and BASE in a Distributed Database[C/OL]. in: OSDI'14: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Broomfield, CO: USENIX Association, 2014: 495-509. http://dl.acm.org/citation.cfm?id=2685048.2685087.

[12] Lee C, Park S J, Kejriwal A, et al. Implementing linearizability at large scale and low latency[C]. in: Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP' 15). 2015.

[13] Dragojević A, Narayanan D, Nightingale E B, et al. No Compromises: Distributed Transactions with Consistency, Availability, and Performance[C/OL]. in: SOSP'15: Proceedings of the 25th Symposium on Operating Systems Principles. Monterey, California: ACM, 2015: 54-70. http://doi.acm.org/10.1145/2815400.2815425. DOI: 10.1145/2815400.2815425.

[14] Zhang I, Sharma N K, Szekeres A, et al. Building Consistent Transactions with Inconsistent Replication[C/OL]. in: SOSP'15: Proceedings of the 25th Symposium on Operating Systems Principles. Monterey, California: ACM, 2015: 263-278. http://doi.acm.org/10.1145/2815400.2815404. DOI: 10.1145/2815400.2815404.

[15] Aguilera M K, Leners J B, Kotla R, et al. Yesquel: scalable SQL storage for Web applications[C]. in: SOSP. 2015.

[16] Xie C, Su C, Littley C, et al. High-performance ACID via modular concurrency control[C]. in: Proceedings of the 25th Symposium on Operating Systems Principles. 2015: 279-294.

[17] Wei X, Shi J, Chen Y, et al. Fast In-memory Transaction Processing Using RDMA and HTM[C/OL]. in: SOSP '15: Proceedings of the 25th Symposium on Operating Systems Principles. Monterey, California: ACM, 2015: 87-104. http://doi.acm.org/10.1145/2815400.2815419. DOI: 10.1145/2815400.2815419.

[18] Chen Y, Wei X, Shi J, et al. Fast and general distributed transactions using rdma and htm[C]. in: Proceedings of the Eleventh European Conference on Computer Systems. 2016: 26.

[19] Mu S, Cui Y, Zhang Y, et al. Extracting More Concurrency from Distributed Transactions[C/OL]. in: OSDI'14: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Broomfield, CO: USENIX Association, 2014: 479-494. http://dl.acm.org/citation.cfm?id=2685048.2685086.

[20] Corbett J C, Dean J, Epstein M, et al. Spanner: Google's globally distributed database[J]. ACM Transactions on Computer Systems (TOCS), 2013, 31(3): 8.

[21] Shamis A, Renzelmann M, Novakovic S, et al. Fast General Distributed Transactions with Opacity[C/OL]. in: SIGMOD '19: Proceedings of the 2019 International Conference on Management of Data. Amsterdam, Netherlands: Association for Computing Machinery, 2019: 433-448. https://doi.org/10.1145/3299869.3300069. DOI: 10.1145/3299869.3300069.

[22] Kung H T, Robinson J T. On Optimistic Methods for Concurrency Control[J/OL]. ACM Trans. Database Syst., 1981, 6(2): 213-226. http://doi.acm.org/10.1145/319566.319567. DOI: 10.1145/319566.319567.

[23] Hooker S. The hardware lottery[J]. ArXiv preprint arXiv:2009.06489, 2020.

[24] Thomas S, Voelker G M, Porter G. CacheCloud: Towards Speed-of-light Datacenter Communication[C/OL]. in: 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18). Boston, MA: USENIX Association, 2018. https://www.usenix.org/conference/hotcloud18/presentation/thomas.

[25] Kalia A, Kaminsky M, Andersen D G. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs[C/OL]. in: OSDI'16: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. Savannah, GA, USA: USENIX Association, 2016: 185-201. http://dl.acm.org/citation.cfm?id=3026877.3026892.

[26] Zamanian E, Binnig C, Harris T, et al. The End of a Myth: Distributed Transactions Can Scale[J]. Proc. VLDB Endow., 2017, 10(6): 685-696.

[27] Wei X, Shen S, Chen R, et al. Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing[C/OL]. in: 2017 USENIX Annual Technical Conference (USENIX ATC 17). Santa Clara, CA: USENIX Association, 2017: 335-347. https://www.usenix.org/conference/atc17/technical-sessions/presentation/wei.

[28] Mitchell C, Montgomery K, Nelson L, et al. Balancing CPU and Network in the Cell Distributed B-Tree Store[C]. in: 2016 USENIX Annual Technical Conference (USENIX ATC 16). 2016.

[29] Mitchell C, Geng Y, Li J. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store[C/OL]. in: USENIX ATC'13: Proceedings of the 2013 USENIX Conference on Annual Technical Conference. San Jose, CA: USENIX Association, 2013: 103-114. http://dl.acm.org/citation.cfm?id=2535461.2535475.

[30] Kalia A, Kaminsky M, Andersen D G. Using RDMA Efficiently for Key-value Services[C/OL]. in: SIGCOMM'14: Proceedings of the 2014 ACM Conference on SIGCOMM. Chicago, Illinois, USA: ACM, 2014: 295-306. http://doi.acm.org/10.1145/2619239.2626299. DOI: 10.1145/2619239.2626299.

[31] Szepesi T, Wong B, Cassell B, et al. Designing a low-latency cuckoo hash table for write-intensive workloads using RDMA[C]. in: First International Workshop on Rack-scale Computing. 2014.

[32] Stuedi P, Trivedi A, Pfefferle J, et al. Crail: A High-Performance I/O Architecture for Distributed Data Processing.[J]. IEEE Data Eng. Bull., 2017, 40(1): 38-49.

[33] Poke M, Hoefler T. DARE: High-Performance State Machine Replication on RDMA Networks[C/OL]. in: HPDC '15: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. Portland, Oregon, USA: ACM, 2015: 107-118. http://doi.acm.org/10.1145/2749246.2749267. DOI: 10.1145/2749246.2749267.

[34] Wu M, Yang F, Xue J, et al. GraM: Scaling Graph Computation to the Trillions[C/OL]. in: SoCC '15: Proceedings of the Sixth ACM Symposium on Cloud Computing. Kohala Coast, Hawaii: ACM, 2015: 408-421. http://doi.acm.org/10.1145/2806777.2806849. DOI: 10.1145/2806777.2806849.

[35] Shi J, Yao Y, Chen R, et al. Fast and Concurrent RDF Queries with RDMA-based Distributed Graph Exploration[C/OL]. in: OSDI'16: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. Savannah, GA, USA: USENIX Association, 2016: 317-332. http://dl.acm.org/citation.cfm?id=3026877.3026902.

[36] Zhang Y, Chen R, Chen H. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data[C/OL]. in: SOSP'17: Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai, China: ACM, 2017: 614-630. http://doi.acm.org/10.1145/3132747.3132777. DOI: 10.1145/3132747.3132777.

[37] Xie X, Wei X, Chen R, et al. Pragh: Locality-preserving Graph Traversal with Split Live Migration[C/OL]. in: 2019 USENIX Annual Technical Conference (USENIX ATC 19). Renton, WA: USENIX Association, 2019: 723-738. https://www.usenix.org/conference/atc19/presentation/xie.

[38] Dulloor S R, Kumar S, Keshavamurthy A, et al. System Software for Persistent Memory[C/OL]. in: EuroSys '14: Proceedings of the Ninth European Conference on Computer Systems. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. https://doi.org/10.1145/2592798.2592814. DOI: 10.1145/2592798.2592814.

[39] Xu J, Swanson S. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories[C/OL]. in: 14th USENIX Conference on File and Storage Technologies (FAST 16). Santa Clara, CA: USENIX Association, 2016: 323-338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

[40] Dong M, Chen H. Soft Updates Made Simple and Fast on Non-volatile Memory[C/OL]. in: 2017 USENIX Annual Technical Conference (USENIX ATC 17). Santa Clara, CA: USENIX Association, 2017: 719-731. https://www.usenix.org/conference/atc17/technical-sessions/presentation/dong.

[41] Zuo P, Hua Y, Wu J. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory[C/OL]. in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, 2018: 461-476. https://www.usenix.org/conference/osdi18/presentation/zuo.

[42] Venkataraman S, Tolia N, Ranganathan P, et al. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory[C]. in: FAST'11: Proceedings of the 9th USENIX Conference on File and Stroage Technologies. San Jose, California: USENIX Association, 2011: 5.

[43] Kannan S, Bhat N, Gavrilovska A, et al. Redesigning LSMs for Nonvolatile Memory with NoveLSM[C/OL]. in: 2018 USENIX Annual Technical Conference (USENIX ATC 18). Boston, MA: USENIX Association, 2018: 993-1005. https://www.usenix.org/conference/atc18/presentation/kannan.

[44] Wu M, Zhao Z, Li H, et al. Espresso: Brewing Java For More Non-Volatility with Non-Volatile Memory[C/OL]. in: ASPLOS '18: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. Williamsburg, VA, USA: Association for Computing Machinery, 2018: 70-83. https://doi.org/10.1145/3173162.3173201. DOI: 10.1145/3173162.3173201.

[45] Shull T, Huang J, Torrellas J. AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability[C/OL]. in: PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. Phoenix, AZ, USA: Association for Computing Machinery, 2019: 316-332. https://doi.org/10.1145/3314221.3314608. DOI: 10.1145/3314221.3314608.

[46] Volos H, Tack A J, Swift M M. Mnemosyne: Lightweight Persistent Memory[C/OL]. in: ASPLOS XVI: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. Newport Beach, California, USA: Association for Computing Machinery,

2011: 91-104. https://doi.org/10.1145/1950365.1950379. DOI: 10.1145/1950365.1950379.

[47] Hsu T C H, Brügner H, Roy I, et al. NVthreads: Practical Persistence for Multi-Threaded Applications[C/OL]. in: EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems. Belgrade, Serbia: Association for Computing Machinery, 2017: 468-482. https://doi.org/10.1145/3064176.3064204. DOI: 10.1145/3064176.3064204.

[48] Liu M, Zhang M, Chen K, et al. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory[C/OL]. in: ASPLOS '17: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. Xi'an, China: Association for Computing Machinery, 2017: 329-343. https://doi.org/10.1145/3037697.3037714. DOI: 10.1145/3037697.3037714.

[49] Memaripour A, Badam A, Phanishayee A, et al. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx[C/OL]. in: EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems. Belgrade, Serbia: Association for Computing Machinery, 2017: 499-512. https://doi.org/10.1145/3064176.3064215. DOI: 10.1145/3064176.3064215.

[50] Hu Q, Ren J, Badam A, et al. Log-Structured Non-Volatile Main Memory[C/OL]. in: 2017 USENIX Annual Technical Conference (USENIX ATC 17). Santa Clara, CA: USENIX Association, 2017: 703-717. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hu.

[51] Lu Y, Shu J, Chen Y, et al. Octopus: an RDMA-enabled Distributed Persistent Memory File System[C/OL]. in: USENIX ATC'17: Proceedings of the 2017 USENIX Annual Technical Conference. Santa Clara, CA: USENIX Association, 2017: 773-785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu.

[52] Yang J, Izraelevitz J, Swanson S. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks[C/OL]. in: 17th USENIX Conference on File and Storage Technologies (FAST 19). Boston, MA: USENIX Association, 2019: 221-234. https://www.usenix.org/conference/fast19/presentation/yang.

[53] Zhang Y, Yang J, Memaripour A, et al. Mojim: A Reliable and Highly-Available Non-Volatile Memory System[C/OL]. in: ASPLOS '15: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. Istanbul, Turkey: Association for Computing Machinery, 2015: 3-18. https://doi.org/10.1145/2694344.2694370. DOI: 10.1145/2694344.2694370.

[54] Shan Y, Tsai S Y, Zhang Y. Distributed Shared Persistent Memory[C/OL]. in: SoCC '17: Proceedings of the 2017 Symposium on Cloud Computing. Santa Clara, California: Association for Computing Machinery, 2017: 323-337. https://doi.org/10.1145/3127479.3128610. DOI: 10.1145/3127479.3128610.

[55] Taleb Y, Stutsman R, Antoniu G, et al. Tailwind: Fast and Atomic RDMA-based Replication[C/OL]. in: 2018 USENIX Annual Technical Conference (USENIX ATC 18). Boston, MA: USENIX Association, 2018: 851-863. https://www.usenix.org/conference/atc18/presentation/taleb.

[56] Ma T, Zhang M, Chen K, et al. AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture[C/OL]. in: ASPLOS '20: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. Lausanne, Switzerland: Association for Computing Machinery, 2020: 757-773. https://doi.org/10.1145/3373376.3378511. DOI: 10.1145/3373376.3378511.

[57] Shen S, Chen R, Chen H, et al. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing[C]. in: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). USENIX Association, 2021.

[58] Wei X, Dong Z, Chen R, et al. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better![C]. in: OSDI '18: 13th USENIX Symposium on Operating Systems Design and Implementation. 2018: 233-251.

[59] Lu H, Hodsdon C, Ngo K, et al. The SNOW theorem and latency-optimal read-only transactions[C]. in: OSDI'16: Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation. 2016: 135.

[60] MySQL Cluster[EB].

[61] Novakovic S, Shan Y, Kolli A, et al. Storm: a fast transactional dataplane for remote data structures[C]. in: Proceedings of the 12th ACM International Conference on Systems and Storage. 2019: 97-108.

[62] Technologies M. Libmlx4 driver[EB/OL]. 2017. http://www.mellanox.com/downloads/ofed/MLNX_OFED-4.0-1.0.1.0/MLNX_OFED_LINUX-4.0-1.0.1.0-ubuntu16.04-x86_64.tgz.

[63] Tsai S Y, Zhang Y. LITE Kernel RDMA Support for Datacenter Applications[C/OL]. in: SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai, China: ACM, 2017: 306-324. http://doi.acm.org/10.1145/3132747.3132762. DOI: 10.1145/3132747.3132762.

[64] Su M, Zhang M, Chen K, et al. RFP: When RPC is Faster Than Server-Bypass with RDMA[C/OL]. in: EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems. Belgrade, Serbia: ACM, 2017: 1-15. http://doi.acm.org/10.1145/3064176.3064189. DOI: 10.1145/3064176.3064189.

[65] Kalia A, Kaminsky M, Andersen D. Datacenter RPCs can be general and fast[C]. in: 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19). 2019: 1-16.

[66] Kalia A, Kaminsky M, Andersen D G. Design Guidelines for High Performance RDMA Systems[C/OL]. in: USENIX ATC '15: 2016 USENIX Annual Technical Conference. Denver, CO: USENIX Association, 2016: 437-450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[67] Stonebraker M, Madden S, Abadi D J, et al. The end of an Architectural Era: (It's Time for a Complete Rewrite)[C/OL]. in: VLDB '07: Proceedings of the 33rd international conference on Very large data bases. Vienna, Austria: VLDB Endowment, 2007: 1150-1160. http://hstore.cs.brown.edu/papers/hstore-endofera.pdf.

[68] Association. I T. InfiniBand Architecture Specification[EB/OL]. 2015. https://cw.infinibandta.org/document/dl/7859.

[69] Liu X, Hua Y, Li X, et al. Write-Optimized and Consistent RDMA-based NVM Systems[J]. ArXiv preprint arXiv:1906.08173, 2019.

[70] The Volatile Benefit of Persistent Memory.[EB/OL]. 2020. https://memcached.org/blog/persistent-memory/.

[71] Ryan Smith. Intel Announces Optane Storage Brand For 3D XPoint Products[EB]. 2015.

[72] Kalia A, Andersen D, Kaminsky M. Challenges and solutions for fast remote persistent memory access[C]. in: Proceedings of the 11th ACM Symposium on Cloud Computing. 2020: 105-119.

[73] Volos H, Magalhaes G, Cherkasova L, et al. Quartz: A Lightweight Performance Emulator for Persistent Memory Software[C/OL]. in: Middleware '15: Proceedings of the 16th Annual Middleware Conference. Vancouver, BC, Canada: Association for Computing Machinery, 2015: 37-49. https://doi.org/10.1145/2814576.2814806. DOI: 10.1145/2814576.2814806.

[74] Yang J, Kim J, Hoseinzadeh M, et al. An empirical guide to the behavior and use of scalable persistent memory[C]. in: 18th {USENIX} Conference on File and Storage Technologies ({FAST} 20). 2020: 169-182.

[75] Intel. The librpmem library[EB]. 2020.

[76] Build Persistent Memory Applications with Reliability Availability and Serviceability.[EB]. 2021.

[77] Ipmctl.[EB/OL]. 2021. https://github.com/intel/ipmctl.

[78] Intel. Intel® Data Direct I/O Technology[EB]. 2019.

[79] Intel. Persistent Memory Replication Over Traditional RDMA[EB]. 2020.

[80] Shi J, Yao Y, Chen R, et al. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration[C/OL]. in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Savannah, GA: USENIX Association, 2016: 317-332. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/shi.

[81] Intel. Intel® Memory Latency Checker v3.7[EB]. 2019.

[82] Neugebauer R, Antichi G, Zazo J F, et al. Understanding PCIe performance for end host networking[C]. in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. 2018: 327-341.

[83] Lim H, Han D, Andersen D G, et al. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage[C/OL]. in: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Seattle, WA: USENIX Association, 2014: 429-444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[84] Wang S, Lou C, Chen R, et al. Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration[C/OL]. in: 2018 USENIX Annual Technical Conference (USENIX ATC 18). Boston, MA: USENIX Association, 2018: 651-664. https://www.usenix.org/conference/atc18/presentation/wang-siyuan.

[85] Ziegler T, Tumkur Vani S, Binnig C, et al. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks[C/OL]. in: SIGMOD '19: Proceedings of the 2019 International Conference on Management of Data. Amsterdam, Netherlands: Association for Computing Machinery, 2019: 741-758. https://doi.org/10.1145/3299869.3300081. DOI: 10.1145/3299869.3300081.

[86] Smolyar I, Markuze A, Pismenny B, et al. IOctopus: Outsmarting Nonuniform DMA[C/OL]. in: ASPLOS '20: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. Lausanne, Switzerland: Association for Computing Machinery, 2020: 101-115. https://doi.org/10.1145/3373376.3378509. DOI: 10.1145/3373376.3378509.

[87] NetCAT[EB/OL]. https://www.vusec.net/projects/netcat/.

[88] Intel. Intel® Xeon® Processor E5 v4 Product Family[EB]. 2019.

[89] Farshin A, Roozbeh A, Jr. G Q M, et al. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks[C/OL]. in: 2020 USENIX Annual Technical Conference (USENIX ATC 20). USENIX Association, 2020: 673-689. https://www.usenix.org/conference/atc20/presentation/farshin.

[90] Kashyap S, Qin D, Byan S, et al. Correct, fast remote persistence[J]. ArXiv preprint arXiv:1909.02092, 2019.

[91] Intel. Intel® Optane persistent memory 200 series[EB]. 2020.

[92] 200Gb/s ConnectX-6 Ethernet Single/Dual-Port Adapter IC[EB]. 2021.

[93] Talpey T, Kamer G. High Performance File Serving With SMB3 and RDMA via SMB Direct[C]. in: Storage Developers Conference. 2012.

[94] Verbitski A, Gupta A, Saha D, et al. Amazon aurora: Design considerations for high throughput cloud-native relational databases[C]. in: Proceedings of the 2017 ACM International Conference on Management of Data. 2017: 1041-1052.

[95] Mitchell C, Geng Y, Li J. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store[C/OL]. in: USENIX ATC'13: Proceedings of the 2013 USENIX Conference on Annual Technical Conference. San Jose, CA: USENIX Association, 2013: 103-114. http://dl.acm.org/citation.cfm?id=2535461.2535475.

[96] Li B, Ruan Z, Xiao W, et al. Kv-direct: High-performance in-memory key-value store with programmable nic[C]. in: Proceedings of the 26th Symposium on Operating Systems Principles. 2017: 137-152.

[97] Ziegler T, Tumkur Vani S, Binnig C, et al. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks[C/OL]. in: SIGMOD '19: Proceedings of the 2019 International Conference on Management of Data. Amsterdam, Netherlands: ACM, 2019: 741-758. http://doi.acm.org/10.1145/3299869.3300081. DOI: 10.1145/3299869.3300081.

[98] Kraska T, Beutel A, Chi E H, et al. The case for learned index structures[C]. in: Proceedings of the 2018 International Conference on Management of Data. 2018: 489-504.

[99] Cooper B F, Silberstein A, Tam E, et al. Benchmarking Cloud Serving Systems with YCSB[C/OL]. in: SoCC'10: Proceedings of the 1st ACM Symposium on Cloud Computing. Indianapolis, Indiana, USA: ACM, 2010: 143-154. http://doi.acm.org/10.1145/1807128.1807152. DOI: 10.1145/1807128.1807152.

[100] OpenStreetMap (OSM) on AWS[EB/OL]. 2021. https://aws.amazon.com/public-datasets/osm.

[101] Lepers B, Balmau O, Gupta K, et al. KVell: the design and implementation of a fast persistent key-value store[C]. in: Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 447-461.

[102] Lim H, Han D, Andersen D G, et al. {MICA}: A holistic approach to fast in-memory key-value storage[C]. in: 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14). 2014: 429-444.

[103] Cassell B, Szepesi T, Wong B, et al. Nessie: A decoupled, client-driven key-value store using RDMA[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(12): 3537-3552.

[104] Mao Y, Kohler E, Morris R T. Cache Craftiness for Fast Multicore Key-value Storage[C/OL]. in: EuroSys'12: Proceedings of the 7th ACM European Conference on Computer Systems. Bern, Switzerland: ACM, 2012: 183-196. http://doi.acm.org/10.1145/2168836.2168855. DOI: 10.1145/2168836.2168855.

[105] Bronson N, Amsden Z, Cabrera G, et al. TAO: Facebook's Distributed Data Store for the Social Graph[C/OL]. in: 2013 USENIX Annual Technical Conference (USENIX ATC 13). San Jose, CA: USENIX Association, 2013: 49-60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.

[106] Aguilera M K, Keeton K, Novakovic S, et al. Designing Far Memory Data Structures: Think Outside the Box[C/OL]. in: HotOS '19: Proceedings of the Workshop on Hot Topics in Operating Systems. Bertinoro, Italy: Association for Computing Machinery, 2019: 120-126. https://doi.org/10.1145/3317550.3321433. DOI: 10.1145/3317550.3321433.

[107] Wang Y, Meng X, Zhang L, et al. C-Hint: An Effective and Reliable Cache Management for RDMA-Accelerated Key-Value Stores[C/OL]. in: SoCC'14: Proceedings of the ACM Symposium on Cloud Computing. Seattle, WA, USA: ACM, 2014: 23:1-23:13. http://doi.acm.org/10.1145/2670979.2671002. DOI: 10.1145/2670979.2671002.

[108] Shamis A, Renzelmann M, Novakovic S, et al. Fast General Distributed Transactions with Opacity[C/OL]. in: SIGMOD '19: Proceedings of the 2019 International Conference on Management of Data. Amsterdam, Netherlands: Association for Computing Machinery, 2019: 433-448. https://doi.org/10.1145/3299869.3300069. DOI: 10.1145/3299869.3300069.

[109] Guo C, Wu H, Deng Z, et al. RDMA over Commodity Ethernet at Scale[C/OL]. in: SIGCOMM'16: Proceedings of the 2016 ACM SIGCOMM Conference. Flo-

rianopolis, Brazil: ACM, 2016: 202-215. http://doi.acm.org/10.1145/2934872.2934908. DOI: 10.1145/2934872.2934908.

[110] Zhang H, Andersen D G, Pavlo A, et al. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes[C]. in: Proceedings of the 2016 International Conference on Management of Data. 2016: 1567-1581.

[111] Galakatos A, Markovitch M, Binnig C, et al. FITing-Tree: A Data-Aware Index Structure[C/OL]. in: SIGMOD ' 19: Proceedings of the 2019 International Conference on Management of Data. Amsterdam, Netherlands: Association for Computing Machinery, 2019: 1189-1206. https://doi.org/10.1145/3299869.3319860. DOI: 10.1145/3299869.3319860.

[112] Graefe G. Write-Optimized B-Trees[C]. in: VLDB ' 04: Proceedings of the Thirtieth International Conference on Very Large Data Bases. Toronto, Canada: VLDB Endowment, 2004: 672-683.

[113] Sowell B, Golab W, Shah M A. Minuet: A Scalable Distributed Multiversion B-Tree[J/OL]. Proc. VLDB Endow., 2012, 5(9): 884-895. https://doi.org/10.14778/2311906.2311915. DOI: 10.14778/2311906.2311915.

[114] Tang C, Wang Y, Dong Z, et al. XIndex: A Scalable Learned Index for Multicore Data Storage[C/OL]. in: PPoPP ' 20: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. San Diego, California: Association for Computing Machinery, 2020: 308-320. https://doi.org/10.1145/3332466.3374547. DOI: 10.1145/3332466.3374547.

[115] Ding J, Minhas U F, Yu J, et al. ALEX: An Updatable Adaptive Learned Index[C/OL]. in: SIGMOD '20: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Portland, OR, USA: Association for Computing Machinery, 2020: 969-984. https://doi.org/10.1145/3318464.3389711. DOI: 10.1145/3318464.3389711.

[116] Dai Y, Xu Y, Ganesan A, et al. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees[C/OL]. in: OSDI '20: 14th USENIX Symposium on Operating Systems Design and Implementation. USENIX Association, 2020. https://www.usenix.org/conference/osdi20/presentation/dai.

[117] Gupta M, Cotter A, Pfeifer J, et al. Monotonic Calibrated Interpolated Look-up Tables[J]. J. Mach. Learn. Res., 2016, 17(1): 3790-3836.

[118] You S, Ding D, Canini K, et al. Deep lattice networks and partial monotonic functions[C]. in: Advances in neural information processing systems. 2017: 2981-2989.

[119] Blundell C, Lewis E C, Martin M M. Subtleties of Transactional Memory Atomicity Semantics[J]. IEEE Computer Architecture Letters, 2006, 5(2).

[120] Intel's Math kernel library[EB/OL]. 2021. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html.

[121] Paszke A, Gross S, Massa F, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library[G/OL]. in: Wallach H, Larochelle H, Beygelzimer A, et al. Advances in Neural Information Processing Systems 32. Curran Associates, Inc., 2019: 8024-8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[122] Robbins H, Monro S. A stochastic approximation method[J]. The annals of mathematical statistics, 1951: 400-407.

[123] Martín Abadi, Ashish Agarwal, Paul Barham, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems[EB/OL]. 2015. http://tensorflow.org/.

[124] The Transaction Processing Council. TPC-C Benchmark V5.11[EB/OL]. 2021. http://www.tpc.org/tpcc/.

[125] Cooper B F. YCSB Core Workloads[EB]. 2021.

[126] Atikoglu B, Xu Y, Frachtenberg E, et al. Workload Analysis of a Large-scale Key-value Store[C/OL]. in: SIGMETRICS '12: Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems. London, England, UK: ACM, 2012: 53-64. http://doi.acm.org/10.1145/2254756.2254766. DOI: 10.1145/2254756.2254766.

[127] High-Performance Big Data (HiBD). RDMA-based Memcached (RDMA-Memcached)[EB].

[128] Memcached[EB].

[129] Nathan V, Ding J, Alizadeh M, et al. Learning Multi-Dimensional Indexes[C/OL]. in: SIGMOD '20: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Portland, OR, USA: Association for Computing Machinery, 2020: 985-1000. https://doi.org/10.1145/3318464.3380579. DOI: 10.1145/3318464.3380579.

[130] Wang Y, Tang C, Wang Z, et al. SIndex: A Scalable Learned Index for String Keys[C/OL]. in: APSys '20: Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems. Tsukuba, Japan: Association for Computing Machinery, 2020: 17-24. https://doi.org/10.1145/3409963.3410496.

[131] Kallman R, Kimura H, Natkins J, et al. H-Store: A High-performance, Distributed Main Memory Transaction Processing System[J/OL]. Proc. VLDB Endow., 2008, 1(2): 1496-1499. http://dx.doi.org/10.14778/1454159.1454211. DOI: 10.14778/1454159.1454211.

[132] Lamport L, Malkhi D, Zhou L. Vertical Paxos and Primary-backup Replication[C/OL]. in: PODC'09: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing. Calgary, AB, Canada: ACM, 2009: 312-313. http://doi.acm.org/10.1145/1582716.1582783. DOI: 10.1145/1582716.1582783.

[133] Yu X, Bezerra G, Pavlo A, et al. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores[J/OL]. Proc. VLDB Endow., 2014, 8(3): 209-220. http://dx.doi.org/10.14778/2735508.2735511. DOI: 10.14778/2735508.2735511.

[134] Harding R, Van Aken D, Pavlo A, et al. An Evaluation of Distributed Concurrency Control[J/OL]. Proc. VLDB Endow., 2017, 10(5): 553-564. https://doi.org/10.14778/3055540.3055548. DOI: 10.14778/3055540.3055548.

[135] The H-Store Team. SmallBank Benchmark[EB/OL]. 2018. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[136] Jin X, Li X, Zhang H, et al. NetCache: Balancing Key-Value Stores with Fast In-Network Caching[C/OL]. in: SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai, China: ACM, 2017: 121-136. http://doi.acm.org/10.1145/3132747.3132764. DOI: 10.1145/3132747.3132764.

[137] Curino C, Jones E, Zhang Y, et al. Schism: A Workload-driven Approach to Database Replication and Partitioning[J/OL]. Proc. VLDB Endow., 2010, 3(1-2): 48-57. http://dx.doi.org/10.14778/1920841.1920853. DOI: 10.14778/1920841.1920853.

[138] Pavlo A, Curino C, Zdonik S. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems[C/OL]. in: SIGMOD'12: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. Scottsdale, Arizona, USA: ACM, 2012: 61-72. http://doi.acm.org/10.1145/2213836.2213844. DOI: 10.1145/2213836.2213844.

[139] Khandelwal A, Agarwal R, Stoica I. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores[C/OL]. in: NSDI'16: 13th USENIX Symposium on Networked Systems Design and Implementation. Santa Clara, CA: USENIX Association, 2016: 485-500. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/khandelwal.

[140] Neuvonen S, Wolski A, Manner M, et al. Telecom Application Transaction Processing (TATP) Benchmark[EB]. 2011.

[141] The Transaction Processing Council. TPC-E Benchmark V1.14[EB/OL]. 2021. http://www.tpc.org/tpce/.

[142] Herlihy M, Moss J E B. Transactional memory: Architectural support for lock-free data structures[C]. in: Proceedings of the 20th annual international symposium on Computer architecture. 1993: 289-300.

[143] Chen H, Chen R, Wei X, et al. Fast In-Memory Transaction Processing Using RDMA and HTM[J/OL]. ACM Trans. Comput. Syst., 2017, 35(1): 3:1-3:37. http://doi.acm.org/10.1145/3092701. DOI: 10.1145/3092701.

[144] Mellanox Technologies. NVMe Over Fabrics Standard is Released[EB/OL]. 2018. http://www.mellanox.com/blog/2016/06/nvme-over-fabrics-standard-is-released/.

[145] Mellanox Technologies. Products Overview[EB/OL]. 2018. http://www.mellanox.com/page/products_overview.

[146] Daglis A, Ustiugov D, Novaković S, et al. SABRes: Atomic Object Reads for In-memory Rack-scale Computing[C/OL]. in: MICRO-49: The 49th Annual IEEE/ACM International Symposium on Microarchitecture. Taipei, Taiwan: IEEE Press, 2016: 6:1-6:13. http://dl.acm.org/citation.cfm?id=3195638.3195646.

[147] Raikin S, Liss L, Shachar A, et al. Remote transactional memory[EB]. 2015.

[148] Zhu Y, Eran H, Firestone D, et al. Congestion Control for Large-Scale RDMA Deployments[C/OL]. in: SIGCOMM'15: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication. London, United Kingdom: ACM, 2015: 523-536. http://doi.acm.org/10.1145/2785956.2787484. DOI: 10.1145/2785956.2787484.

[149] Ajoux P, Bronson N, Kumar S, et al. Challenges to adopting stronger consistency at scale[C]. in: 15th Workshop on Hot Topics in Operating Systems (HotOS XV). 2015.

[150] Bernstein P A, Goodman N. Multiversion Concurrency Control Theory and Algorithms[J/OL]. ACM Trans. Database Syst., 1983, 8(4): 465-483. http://doi.acm.org/10.1145/319996.319998. DOI: 10.1145/319996.319998.

[151] Wu Y, Arulraj J, Lin J, et al. An Empirical Evaluation of In-memory Multi-version Concurrency Control[J/OL]. Proc. VLDB Endow., 2017, 10(7): 781-792. https://doi.org/10.14778/3067421.3067427. DOI: 10.14778/3067421.3067427.

[152] PostgreSQL[EB].

[153] Oracle Database Concepts: Data Concurrency and Consistency[EB]. 2017.

[154] MySQL/InnoDB[EB].

[155] Sikka V, Färber F, Lehner W, et al. Efficient transaction processing in SAP HANA database: the end of a column store myth[C]. in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. 2012: 731-742.

[156] Binnig C, Crotty A, Galakatos A, et al. The end of slow networks: It's time for a redesign[J]. Proceedings of the VLDB Endowment, 2016, 9(7): 528-539.

[157] Pavlo A, Aslett M. What's Really New with NewSQL?[J/OL]. SIGMOD Rec., 2016, 45(2): 45-55. http://doi.acm.org/10.1145/3003665.3003674. DOI: 10.1145/3003665.3003674.

[158] Peng D, Dabek F. Large-scale Incremental Processing Using Distributed Transactions and Notifications[C/OL]. in: 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10). Vancouver, BC: USENIX Association, 2010. https://www.usenix.org/conference/osdi10/large-scale-incremental-processing-using-distributed-transactions-and.

[159] Binnig C, Hildenbrand S, Färber F, et al. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally[J]. The VLDB Journal, 2014, 23(6): 987-1011.

[160] Akkoorath D D, Tomsic A Z, Bravo M, et al. Cure: Strong semantics meets high availability and low latency[C]. in: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). 2016: 405-414.

[161] Vasudevan V, Kaminsky M, Andersen D G. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server[C]. in: Proceedings of the Third ACM Symposium on Cloud Computing. 2012: 8.

[162] Balakrishnan M, Malkhi D, Wobber T, et al. Tango: Distributed data structures over a shared log[C]. in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. 2013: 325-340.

[163] Lim H, Kaminsky M, Andersen D G. Cicada: Dependably fast multi-core in-memory transactions[C]. in: Proceedings of the 2017 ACM International Conference on Management of Data. 2017: 21-35.

[164] Larson P Å, Blanas S, Diaconu C, et al. High-performance concurrency control mechanisms for main-memory databases[J]. Proceedings of the VLDB Endowment, 2011, 5(4): 298-309.

[165] Faleiro J M, Abadi D J. Rethinking serializable multiversion concurrency control[J]. Proceedings of the VLDB Endowment, 2015, 8(11): 1190-1201.

[166] Baker J, Bond C, Corbett J C, et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services.[C]. in: CIDR'11: Proceedings of the 5th biennial Conference on Innovative Data Systems Research. 2011: 223-234.

[167] Ding B, Kot L, Demers A, et al. Centiman: Elastic, High Performance Optimistic Concurrency Control by Watermarking[C/OL]. in: SoCC '15: Proceedings of the Sixth ACM Symposium on Cloud Computing. Kohala Coast, Hawaii: ACM, 2015: 262-275. http://doi.acm.org/10.1145/2806777.2806837. DOI: 10.1145/2806777.2806837.

[168] Li J, Michael E, Ports D R. Eris: Coordination-free consistent transactions using in-network concurrency control[C]. in: Proceedings of the 26th Symposium on Operating Systems Principles. 2017: 104-120.

[169] Larson P, Blanas S, Diaconu C, et al. High-performance Concurrency Control Mechanisms for Main-memory Databases[J/OL]. Proc. VLDB Endow., 2011, 5(4): 298-309. http://dx.doi.org/10.14778/2095686.2095689. DOI: 10.14778/2095686.2095689.

[170] Reimer M. Solving the Phantom Problem by Predicative Optimistic Concurrency Control[C/OL]. in: VLDB '83: Proceedings of the 9th International Conference on Very Large Data Bases. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1983: 81-88. http://dl.acm.org/citation.cfm?id=645911.671118.

[171] Mohan C. ARIES/KVL: A Key-value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes[C/OL]. in: Proceedings of the Sixteenth International Conference on Very Large Databases. Brisbane, Australia: Morgan Kaufmann Publishers Inc., 1990: 392-405. http://dl.acm.org/citation.cfm?id=94362.94465.

[172] Terry D B, Demers A J, Petersen K, et al. Session Guarantees for Weakly Consistent Replicated Data[C/OL]. in: PDIS '94: Proceedings of the Third International Conference on on Parallel and Distributed Information Systems. Autin, Texas, USA: IEEE Computer Society Press, 1994: 140-150. http://dl.acm.org/citation.cfm?id=381992.383631.

[173] Terry D, Prabhakaran V, Kotla R, et al. Transactions with Consistency Choices on Geo-Replicated Cloud Storage[J/OL]., 2013. https://www.microsoft.com/en-us/research/publication/transactions-with-consistency-choices-on-geo-replicated-cloud-storage/.

[174] Lu H, Veeraraghavan K, Ajoux P, et al. Existential Consistency: Measuring and Understanding Consistency at Facebook[C/OL]. in: SOSP '15: Proceedings of the 25th Symposium on Operating Systems Principles. Monterey, California: ACM, 2015: 295-310. http://doi.acm.org/10.1145/2815400.2815426. DOI: 10.1145/2815400.2815426.

[175] Kim K, Wang T, Johnson R, et al. Ermia: Fast memory-optimized database system for heterogeneous workloads[C]. in: Proceedings of the 2016 International Conference on Management of Data. 2016: 1675-1687.

[176] Neumann T, Mühlbauer T, Kemper A. Fast serializable multi-version concurrency control for main-memory database systems[C]. in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. 2015: 677-689.

[177] Wu Y, Arulraj J, Lin J, et al. An empirical evaluation of in-memory multi-version concurrency control[J]. Proceedings of the VLDB Endowment, 2017, 10(7): 781-792.

[178] Leis V, Kemper A, Neumann T. Exploiting hardware transactional memory in main-memory databases[C]. in: ICDE'14: IEEE 30th International Conference on Data Engineering. 2014: 580-591.

[179] Cahill M J, Röhm U, Fekete A D. Serializable Isolation for Snapshot Databases[J/OL]. ACM Trans. Database Syst., 2009, 34(4): 20:1-20:42. http://doi.acm.org/10.1145/1620585.1620587. DOI: 10.1145/1620585.1620587.

[180] Sovran Y, Power R, Aguilera M K, et al. Transactional storage for geo-replicated systems[C]. in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. 2011: 385-400.

[181] Bailis P, Fekete A, Ghodsi A, et al. Scalable atomic visibility with RAMP transactions[J]. ACM Transactions on Database Systems (TODS), 2016, 41(3): 15.

[182] Ports D R, Clements A T, Zhang I, et al. Transactional Consistency and Automatic Management in an Application Data Cache.[C]. in: OSDI: vol. 10. 2010: 1-15.

[183] Roohitavaf M, Demirbas M, Kulkarni S. Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks[C]. in: 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS). 2017: 184-193.

[184] Bernstein P A, Goodman N. Timestamp-based algorithms for concurrency control in distributed database systems[C]. in: Proceedings of the sixth international conference on Very Large Data Bases-Volume 6. 1980: 285-300.

[185] Adya A, Gruber R, Liskov B, et al. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks[C/OL]. in: SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. San Jose, California, USA: ACM, 1995: 23-34. http://doi.acm.org/10.1145/223784.223787. DOI: 10.1145/223784.223787.

[186] Cowling J, Liskov B. Granola: Low-Overhead Distributed Transaction Coordination[C/OL]. in: Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12). Boston, MA: USENIX, 2012: 223-235. https://www.usenix.org/conference/atc12/technical-sessions/presentation/cowling.

[187] Levandoski J, Lomet D, Sengupta S, et al. High performance transactions in deuteronomy[J]., 2015.

[188] Levandoski J, Lomet D, Sengupta S, et al. Multi-version range concurrency control in Deuteronomy[J]. Proceedings of the VLDB Endowment, 2015, 8(13): 2146-2157.

[189] Lomet D, Fekete A, Wang R, et al. Multi-version Concurrency via Timestamp Range Conflict Management[C]. in: ICDE: IEEE 28th International Conference on Data Engineering. 2012: 714-725.

[190] Mahmoud H A, Arora V, Nawab F, et al. MaaT: Effective and Scalable Coordination of Distributed Transactions in the Cloud[J/OL]. Proc. VLDB Endow., 2014, 7(5): 329-340. http://dx.doi.org/10.14778/2732269.2732270. DOI: 10.14778/2732269.2732270.

[191] Yu X, Pavlo A, Sanchez D, et al. TicToc: Time Traveling Optimistic Concurrency Control[C/OL]. in: SIGMOD '16: Proceedings of the 2016 International Conference on Management of Data. San Francisco, California, USA: ACM, 2016: 1629-1642. http://doi.acm.org/10.1145/2882903.2882935. DOI: 10.1145/2882903.2882935.

[192] Du J, Elnikety S, Zwaenepoel W. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks[C]. in: IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS). 2013: 173-184.

[193] Yu X, Xia Y, Pavlo A, et al. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System[J/OL]. Proc. VLDB Endow., 2018, 11(10): 1289-1302. https://doi.org/10.14778/3231751.3231763. DOI: 10.14778/3231751.3231763.

[194] Cao W, Liu Z, Wang P, et al. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database[J/OL]. Proc. VLDB Endow., 2018, 11(12): 1849-1862. https://doi.org/10.14778/3229863.3229872. DOI: 10.14778/3229863.3229872.