Scythe: A Low-latency RDMA-enabled Distributed Transaction System for Disaggregated Memory

KAI LU, Huazhong University of Science and Technology, Wuhan, China
SIQI ZHAO, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
HAIKANG SHAN, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
QIANG WEI, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
GUOKUAN LI[*], Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
JIGUANG WAN[*], Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
TING YAO, Huawei Cloud Computing Technologies Co Ltd, Shenzhen, China
HUATAO WU, Huawei Cloud Computing Technologies Co Ltd, Shenzhen, China
DAOHUI WANG, Huawei Cloud Computing Technologies Co Ltd, Shenzhen, China

Disaggregated memory separates compute and memory resources into independent pools connected by RDMA (Remote Direct Memory Access) networks, which can improve memory utilization, reduce cost, and enable elastic scaling of compute and memory resources. However, existing RDMA-based distributed transactions on disaggregated memory suffer from severe long-tail latency under high-contention workloads.

In this paper, we propose Scythe, a novel low-latency RDMA-enabled distributed transaction system for disaggregated memory. Scythe optimizes the latency of high-contention transactions in three approaches: 1) Scythe proposes a hot-aware concurrency control policy that uses optimistic concurrency control (OCC) to improve transaction processing efficiency in low-conflict scenarios. Under high conflicts, Scythe designs a timestamp-ordered OCC (TOCC) strategy based on fair locking

New paper, not an extension of a conference paper.
[*]Corresponding authors.

Authors' Contact Information: Kai Lu, Huazhong University of Science and Technology, Wuhan, Hu Bei, China; e-mail: kailu@hust.edu.cn; Siqi Zhao, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: m202373801@hust.edu.cn; Haikang Shan, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: shanhaikang.shk@oceanbase.com; Qiang Wei, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: m202273609@hust.edu.cn; Guokuan Li, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: liguokuan@hust.edu.cn; Jiguang Wan, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: jgwan@hust.edu.cn; Ting Yao, Huawei Cloud Computing Technologies Co Ltd, Shenzhen, Guangdong, China; e-mail: yaoting17@huawei.com; Huatao Wu, Huawei Cloud Computing Technologies Co Ltd, Shenzhen, Guangdong, China; e-mail: wuhuatao@huawei.com; Daohui Wang, Huawei Cloud Computing Technologies Co Ltd, Shenzhen, Guangdong, China; e-mail: wangdaohui@huawei.com.

ACM 1544-3973/2024/5-ART
https://doi.org/10.1145/3666004

to reduce the number of retries and cross-node communication overhead. 2) Scythe presents an RDMA-friendly timestamp service for improved timestamp management. 3) Scythe designs an RDMA-optimized RPC framework to improve RDMA bandwidth utilization. The evaluation results show that, compared to state-of-the-art distributed transaction systems, Scythe achieves more than 2.5× lower latency with 1.8× higher throughput under high-contention workloads.

CCS Concepts: • **Software and its engineering → Distributed memory**; • **Information systems → Distributed database transactions**.

Additional Key Words and Phrases: Disaggregated memory, RDMA, Distributed transactions

1 INTRODUCTION

Datacenters (e.g., RSA [47], WSC [2], dReDBox [30]), cloud databases (e.g., LegoBase [65], Amazon Aurora [52], PolarDB [8]), HPC systems [42, 53], etc. increasingly favor disaggregated memory (DM for short) architectures for higher resource utilization, flexible hardware scalability and lower costs. As shown in Fig. 1, memory disaggregation decouples compute (CPU) and memory resources from traditional monolithic servers to form independent resource pools. With a single-digit-microsecond level latency, RDMA (Remote Direct Memory Access) networks are generally adopted to connect the compute and memory pools. Recently RDMA-based disaggregated memory systems have received significant attention in both academia and industry, involving distributed databases [8, 33, 65], shared memory systems [19, 44, 49], software stacks [7, 38], distributed transactions [11, 29, 58, 63], data organization and indexes [35, 55, 66].

Distributed transactions (dtxns) are important for disaggregated memory systems to ensure that remote data in the memory pool is atomically and consistently accessed by the compute pool [63]. However, the current real-world database and datacenter workloads have two characteristics that bring challenges for dtxns: 1) **Highly skewed**. A small fraction of data (hotpots) is accessed frequently by a large number of concurrent transactions. For example, online transaction processing (OLTP) workloads feature severe skew in data access frequency [12, 17, 24, 26, 37, 62]. Similar workload examples exist at Facebook [9] and Twitter [60], making transaction contention and conflicts common. 2) **Latency sensitive**. Many datacenter and database applications require data services to deliver fast responses and microsecond-scale tail latency [12, 15, 21, 39, 40, 51]. Application examples include real-time gaming, video conferencing, cloud services.

However, existing RDMA-based distributed transactions [11, 19, 44, 58] suffer from long-tail latency under high-contention workloads [12, 18, 26, 63]. What's worse, we observe that ***the long-tail latency problem is exacerbated on DM*** due to the characterizes of DM: 1) The memory resources are physically distant from CPUs, resulting in increased access latency as data needs to be fetched remotely during transaction execution; 2) The memory pool lacks CPUs to handle extensive computational tasks frequently during dtxn processing. The typical practice is to leverage one-sided RDMA to bypass the CPU in memory pools [19, 44, 55, 58], but this leads to significant round trips and access contentions [26, 63]. As the state-of-the-art distributed transaction for disaggregated memory, FORD [63] improves the transaction throughput by efficient locking scheme, backup-enabled read and selective remote flushing, but still suffers from long-tail latency due to high abort overhead. In summary, it is imperative to redesign low-latency distributed transactions for disaggregated memory, yet it is confronted with following problems:

(1) Heavy retry overhead of optimistic concurrency control. RDMA-based dtxns [11, 19, 29, 44, 58, 63] prefer optimistic concurrency control (OCC) to achieve higher throughput and lower latency in low-conflict scenarios. However, OCC-based transactions encounter frequent aborts when dealing with high-contention situations. Additionally, the validation phase is performed after remote reads, so frequent retries after aborting lead to additional round-trip times (RTTs), causing long-tail latency. FORD detects the conflicts in advance by adding read locks during the execution phase, thus reducing the retry overhead. However, FORD fails to address the excessive abort rates of OCC, which is the root of the long-tail latency problem.

(2) High-latency timestamp management. Timestamp is a critical component of dtxn systems to support strict serializability [44]. Timestamp management based on the physical clocks is commonly implemented in current systems [14, 44]. However, this approach introduces a uncertain waiting time, which increases the overall latency of transaction processing. Furthermore, in high-contention scenarios, frequent retries occur, and each retry requires fetching the timestamp again, which further increases the latency. Many studies have proposed hybrid logic clock methods [20, 57], but they introduce complex maintenance algorithms and communication protocols, which are not suitable for disaggregated memory architectures due to near-zero computation power in memory pool. Moreover, logic clock algorithms typically do not provide strong consistency guarantees.

(3) Low RDMA bandwidth utilization. RDMA communication is the main bottleneck of distributed transactions on DM [18, 63]. Typically, distributed transaction systems utilize one-sided RDMA primitives for RDMA communication [11, 58, 63]. However, this approach has significant drawbacks: 1) The CPU of the memory server remains idle most of the time except for establishing connections and allocating memory, resulting in inefficient CPU utilization; 2) A single data query (e.g., data index search) may involve multiple read and write operations, leading to increased one-sided RTTs; 3) Pure one-sided RDMA operations cannot access non-contiguous data on the memory server through scatter-gather, resulting in wasted RDMA bandwidth. Therefore, many studies propose high-performance RPC frameworks based on RDMA (e.g., FaSST [29]) or adopt general RPC libraries without RDMA primitives [27]. However, we find that the existing RDMA communication methods are suboptimal and do not take full advantage of RDMA bandwidth (§3.4).

In this paper, we address these challenges with ***Scythe***, a low-latency RDMA-enabled distributed transaction system on DM. To mitigate the aforementioned challenges, we have introduced a set of innovative techniques to achieve low-latency and high-throughput transaction performance. Specifically, this paper makes the following contributions:

- **Hot-aware concurrency control.** To reduce abort/retry overhead due to conflicts, Scythe proposes a fair locking mechanism to achieve heat awareness during transaction execution and fair queuing between conflicting transactions. Scythe retains OCC in low-heat scenarios with fewer hotpots. In high-heat scenarios with high conflicts, Scythe proposes the TOCC (Timestamp Ordered OCC) strategy based on the idea of timestamp ordering [26] to reduce the cross-node communication overhead caused by frequent retries and alleviate the long-tail latency problem.
- **RDMA-friendly TSO service.** To avoid uncertain waiting, Scythe provides a low-latency centralized timing service based on the Timestamp Oracle (TSO) [25, 41]. In addition, Scythe utilizes RDMA NIC (network interface card) memory and proxy strategies to reduce the additional communication overhead caused by TSO.
- **RDMA-optimized RPC framework.** To enhance the utilization of RDMA bandwidth utilization, Scythe introduces a high-performance RDMA-optimized RPC framework (RRPC). RRPC combines the RPC approaches with one-sided RDMA primitives and leverages batch processing to reduce the transaction abort rates and cross-node communication overhead.

We implement a Scythe prototype and conduct performance evaluations using OLTP benchmarks, including Smallbank and TPC-C workloads [5, 17, 50], and YCSB benchmarks. The evaluation results demonstrate that Scythe reduces latency by approximately 60% and improves throughput by approximately 50% compared to state-of-the-art distributed transaction systems. The open-source code of Scythe is available at https://github.com/PDS-Lab/scythe.

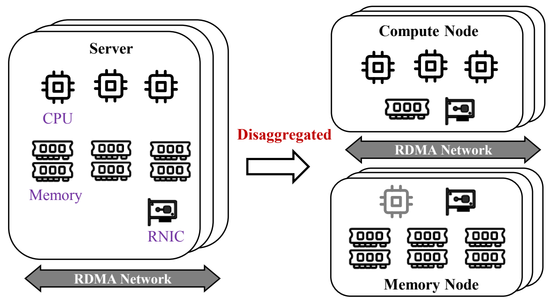


Fig. 1. The disaggregated memory architecture.

2 BACKGROUND

2.1 Memory Disaggregation

Datacenters today mostly use monolithic server architectures in which CPUs and memory are tightly coupled. However, as memory requirements increase, such an architecture suffers from low resource utilization, lack of elasticity, and high costs. For example, in monolithic servers, since the memory resource occupied by a single instance cannot be allocated across server boundaries, it is difficult to fully utilize memory resources. In Microsoft Azure's [34] and Google's clusters[43], the memory utilization is generally below 50%.

In response, *memory disaggregation* is proposed to solve these problems and has received significant attention in both academia and industry [2, 22, 30, 34, 47, 65]. Memory disaggregation separates the memory resources from the compute resources in a datacenter, forming independent resource pools connected with fast networks. This allows different resources to be managed and expanded independently, enabling higher memory utilization, elastic scaling, and lower costs. As shown in Fig. 1, in DM, the compute nodes (CNs) in the compute pool contain numerous CPU cores and small local DRAM; the memory nodes (MNs) in the memory pool host high-volume memory with near-zero computation power. The microsecond-latency networks (e.g. RDMA) are generally the physical transmission approach from CNs to MNs.

2.2 RDMA Technology

RDMA is a series of protocols that enable direct data access from one machine to remote ones over the network. RDMA protocols are typically realized directly on RDMA NICs (RNIC) and offer high bandwidth (>10 GB/s) and low latency at the microsecond level (~2μs). These protocols are widely supported by technologies such as InfiniBand, RoCE, OmniPath. [23, 46, 61]. RDMA provides data transfer services based on two types of operational primitives: *one-sided verbs* including RDMA `READ`, `WRITE`, `ATOMIC` (e.g. `FAA`, `CAS`) and *two-sided verbs* including RDMA `SEND`, `RECV`. RDMA communication is implemented through a message queue model called the Queue Pair (QP) and the Completion Queue (CQ). QP consists of Send Queue (SQ) and Receive Queue (RQ). A sender posts the request to SQ (one-sided or two-sided verbs); RQ is used for queuing RDMA `RECV` requests in two-sided verbs. CQ is associated with the specified QP. Requests in the same SQ are executed sequentially. Using doorbell batching [46, 55], multiple RDMA operations can be combined into a single request. These requests are then read by the RNIC, which asynchronously writes or reads data from a remote memory. Once the sender's request completes, the RNIC writes the completion entry to CQ, so that the sender can know it by polling CQ. Waiting for network transmissions accounts for most of RDMA's latency overhead [10], making it suitable for optimization using coroutine techniques.

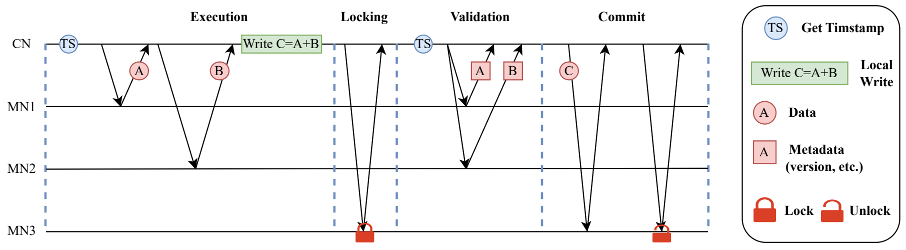


Fig. 2. The workflow of MVOCC.

Coroutines are lightweight threads that are not managed by the operating system but by the program. Coroutines allow a program to pause execution at any location and resume execution later without interrupting the entire program. This makes coroutines well suited for asynchronous tasks like RDMA where I/O wait and compute can be executed under the same thread.

2.3 Distributed Transactions on Disaggregated Memory

For disaggregated memory, distributed transactions (dtxns) are still important to ensure that remote data in the memory pool is atomically and consistently accessed by the compute pool [63]. Optimistic concurrency control (OCC) and its variants are commonly employed in RDMA-enabled distributed transactions. Among them, multi-version OCC (*MVOCC*) [16, 58, 59] is a popular scheme that modifies the original OCC protocol to support multi-versioning. Fig. 2 illustrates how existing RDMA systems [58, 63] process dtxns over MVOCC. The MVOCC process is divided into five phases: 1) Execution: The transaction first obtains a timestamp and reads the required data (i.e., read set = A, B) from MNs and executes a dtxn locally. The updated data (write set = A, B, C) are buffered in a local cache without being immediately synchronized to the remote end. 2) Locking: Lock the write set and abort the dtxn if the locking fails. 3) Validation: If locking succeeds, verify that the records in the read set remain unchanged after getting the write timestamp. If the validation fails, abort the dtxn. 4) Commit: If the validation succeeds, a two-phase commit (2PC) is adopted. This involves synchronizing the write set in the local buffer to the remote end and unlocking.

3 MOTIVATION AND RELATED WORK

In this section, we will analyze the long-tail latency problems faced by RDMA-based distributed transactions on disaggregated memory and its causes, and present related work.

3.1 Long-tail Latency Problems on Disaggregated Memory

To demonstrate the long-tail latency problem, we implement the transaction mechanism of FaRM [19], a typical RDMA-based transaction system, in a disaggregated memory architecture (called FaRM-DM). we test the performance of FaRM and FaRM-DM using TPC-C workloads [5] with varying skewness (tuned by the parameter θ), which determines the contention level. $\theta = 0.5$ indicates a skewed distribution and is typically used to model low or medium contention scenarios; $\theta = 0.99$ means a highly skewed distribution (high-contention scenarios) [12, 26, 64]. Experimental details can be found in §5. The results are shown in Fig. 3 and Fig. 4, where the 99th percentile (P99) latency and throughput are collected as we increase the number of threads from 1 to 32. It can be found that the tail latency of FaRM and FaRM-DM increases dramatically under high-contention workloads (32 threads and $\theta = 0.99$, high concurrency and highly skewed). The long-tail latency problem is exacerbated on DM.

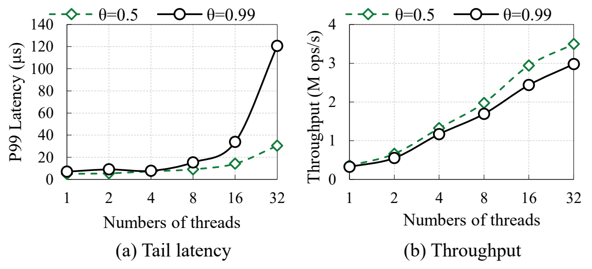


Fig. 3. The tail latency and throughput of FaRM.

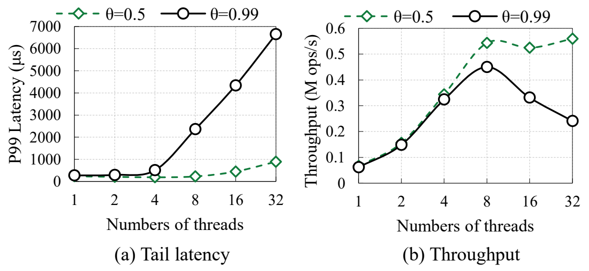


Fig. 4. The tail latency and throughput of FaRM-DM.

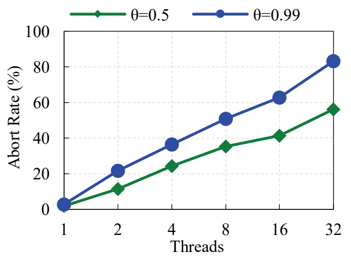


Fig. 5. Abort rate test.

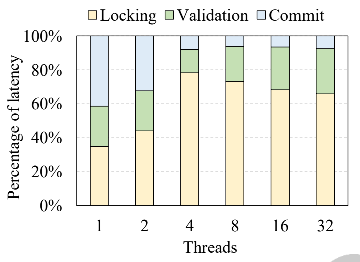


Fig. 6. P99 latency decomposition.

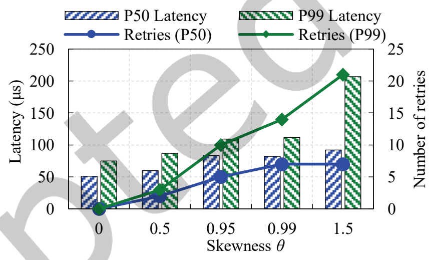


Fig. 7. RDMA CAS Locking Latency.

As shown in Fig. 4, the tail latency of FaRM-DM under high contention reaches milliseconds, and the throughput initially increases and then decreases with the number of threads.

Next, we will explore the causes of high latency from three aspects: concurrency control algorithms, timestamp management, and communication frameworks. We will also explore the most appropriate strategies for disaggregated memory.

3.2 Concurrency Control for Disaggregated Memory

To investigate the underlying reasons of long-tail latency problem, three experiments are conducted:

(1) Abort rate. First, we measure the abort rate of transactions in FaRM-DM. As shown in Fig. 5, with the increase of concurrency, the transaction abort rate reaches 56% when there are fewer conflicts ($\theta = 0.5$) and exceeds 83% when the conflicts are severe ($\theta = 0.99$). The rising abort rate directly leads to frequent retries of OCC transactions during the execution phase. Consequently, the number of cross-node communications increases, resulting in long-tail latency.

(2) Latency decomposition. As shown in Fig. 6, we measure the decomposition of P99 latency, excluding the execution phase. The results indicate notable changes in the locking and validation phases as the proportion of write transactions increases. The locking operation accounts for 80% of the latency in the three phases, excluding the execution phase. Theoretically, the number of locking operations should be similar to that of validation operations, but the locking latency is significantly higher due to two reasons: 1) Serial locking process. The locking operation is executed serially in OCC to avoid deadlocks, while the validation phase is performed by sending RDMA unilateral read requests in parallel; 2) Lacking fairness. Existing locks in RDMA-based transactions use a single bit as the data structure for locks and employ the RDMA CAS operation to implement distributed locking algorithms. However, they do not consider fairness between conflicting lock acquisitions, thus starving certain client requests and consequently inducing high tail latency [55].

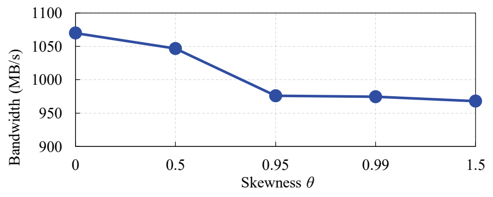


Fig. 8. Data transfer bandwidth affected by RDMA CAS locking with different skewness.

	Uniform	Zipfian
P50 Latency (μs)	8.4	11.7
P90 Latency (μs)	16.3	72.9
Throughput (Mops)	1.76	0.42

Table 1. Latency and throughput of RDMA CAS operations under different access distributions

(3) Locking mechanism based on RDMA CAS. To investigate issues related to a single-bit lock with RDMA CAS, we enable 64 threads to request 1000 locks (with varying levels of Zipfian skewness) in a remote MN. Each thread holds the locks for approximately 25μs after acquiring them, and requests a lock again after releasing it. The experiment is repeated 10000 times and the latency between requesting a lock and obtaining a lock is recorded for each iteration. The latency under different conflict rates (skewness) is presented in Fig.7. As the data become more skewed, the latency of the locking operation increases significantly due to the increased number of retries. For example, in high-contention workloads with a θ value of 0.99 (indicating severe conflicts), the P99 latency of the lock operation reaches 112μs. This latency represents a significant overhead compared to RDMA communication, which typically has a much lower latency of only 2–4μs. Additionally, we perform the 2048B data transfer during the locking test. As shown in Fig. 8, it is observed that frequent RDMA CAS locking operations under high contention congest the bandwidth and result in a degradation of data transfer throughput by approximately 10%.

In summary, there are two main reasons for the performance collapse of OCC-based transaction systems under high-contention workloads: **high abort rate** and **unfair locking mechanism**.

3.3 Distributed Timestamp Management

Distributed timestamp management can be broadly classified into three approaches: true time with physical clocks (e.g. Spanner [14]), hybrid logic clocks (e.g. CockroachDB [48]) and globally ordered timestamp service, namely *Timestamp Oracle* (e.g. Percolator [4], TiDB [25]). Additionally, many RDMA-enabled timestamp management techniques (e.g. FaRMv2 [44]), DST [57]) based on these approaches have been proposed. However, these strategies exhibit notable limitations when it comes to disaggregated memory.

(1) True Time. FaRMv2 adopts a TrueTime API based on specific hardware (GPS and atomic clocks) to provide scalable timestamps. Furthermore, FaRMv2 utilizes RDMA networks and the Marzullo [36] algorithm for clock synchronization to achieve clock uncertainty wait time of about 22~24μs. However, this approach has two drawbacks: 1) Prohibitive costs of hardware equipment; 2) The uncertainty wait (11~12 RTTs) is required for each transaction to ensure the match of timestamp and transaction ordering. Although increasing the clock synchronization frequency can reduce the uncertainty wait time, frequent RDMA communication consumes the bandwidth for data transfer and degrades the system performance.

(2) Hybrid Logic Clock (HLC). The main idea is to integrate physical and logical time to generate timestamps using well-designed algorithms. However, these solutions [20, 48, 57] suffer from two drawbacks: 1) they introduce complex communication protocols and maintenance overhead, and 2) they are unable to provide linearizability guarantees.

(3) Timestamp Oracle (TSO). As a centralized timestamp server, TSO is considered as the simplest and most effective method for timestamp management. However, it faces scalability limitations. TiDB [25] implements high-throughput and highly scalable TSO services through strategies such as follower proxy and batching timestamp requests. The remaining problem is that a significant influx of requests can lead to high CPU utilization of the

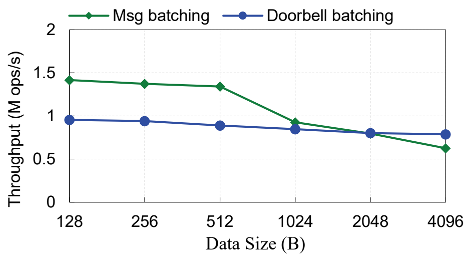


Fig. 9. Different batch processing.

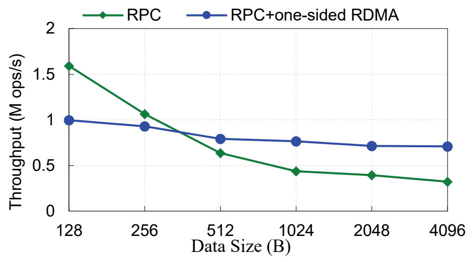


Fig. 10. Different communication strategies.

TSO node, consequently impacting the transaction latency. RDMA networks offer the feature of bypassing the remote CPU, providing an opportunity to solve the problem. However, there have been limited studies exploring the utilization of TSO in RDMA-based transactions.

In summary, building the TSO service using RDMA primitives is a promising solution, but also challenging. Table 1 displays the throughput and latency of RDMA `CAS` operation (8 threads) under uniform/zipfian workloads. It is observed that RDMA `CAS` suffers from high latency under skewed workloads. This problem will be more pronounced for centralized TSO services. The underlying cause is that a single RDMA `CAS` operation necessitates two PCIe transactions on the remote RNIC: i) reading data from CPU memory into the RNIC and ii) writing back after modification. As a result, the RNIC experiences queuing of requests to access the same virtual address, leading to a significant increase in latency [26, 55]. Scythe aims to efficiently utilize the RNIC and achieve an RDMA-friendly timestamp management.

3.4 RDMA Communication Framework

The existing RDMA communication methods have been investigated to be suboptimal and do not fully exploit the potential of RDMA bandwidth. Specifically, several previous systems propose several optimization guidelines for RDMA communication [11, 19, 26, 29, 55, 57, 63], which can be summarized as follows:

(1) Using *Hugepages* and *QP sharing* can reduce the NIC cache miss rate, thereby reducing communication latency;

(2) Employing *Doorbell batching* [28] can reduce the network latency in the transaction process;

(3) Designing high-performance *RPC* frameworks based on RDMA [19, 29] or adopting general RPC libraries without RDMA primitives [27] improves communication performance.

Our experiments demonstrate the efficacy of guideline (1). Regarding guideline (2), we evaluate the throughput of RDMA `WRITE` operations under varying data sizes using different batching modes. Surprisingly, as shown in Fig. 9, we observe that when transferring small data, the throughput of the Doorbell batching is significantly lower than message batching (Msg batching). Msg batching batches data in a continuous buffer (2K size by default). Doorbell batching allows the RNIC to process requests from the same QP in batches, reducing PCIe transactions initiated by the CPU. Hence, Doorbell batching offers the advantage of decreasing CPU usage and optimizing the utilization of PCIe bandwidth. However, it does not directly reduce RTTs because RNIC does not consolidate the data into a single RDMA request packet for transmission in Doorbell batching mode. As shown in Fig.9, Doorbell batching performs better when the data is larger. Based on this observation, Scythe attempts to combine these two batching methods to fully leverage of RDMA bandwidth.

Regarding guideline (3), we test the throughput using various data sizes with two popular frameworks including 1) only-RPC (using eRPC [27]); 2) One-sided RDMA and RPC hybrid mode [19, 29], which utilizes RPC to obtain remote data address first and accesses data via one-sided RDMA verbs. As shown in Fig. 10, the result indicates

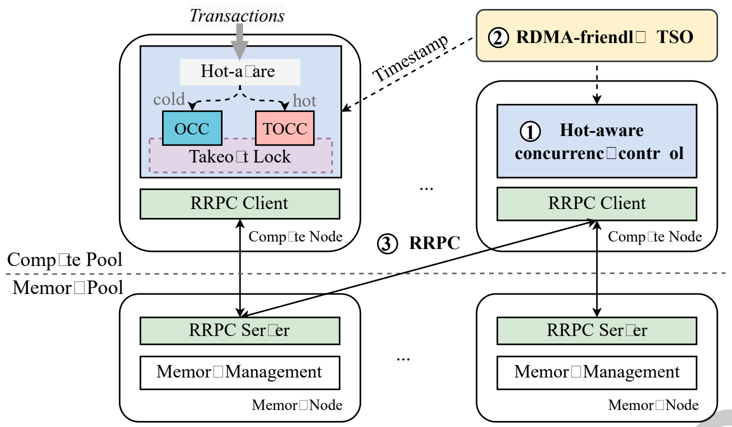


Fig. 11. The architecture of Scythe.

that RPC communication is suitable for small data transmission, while the hybrid mode exhibits higher throughput for large data scenarios. The 512 bytes is a cut-off point, which inspires Scythe to design a dynamic strategy.

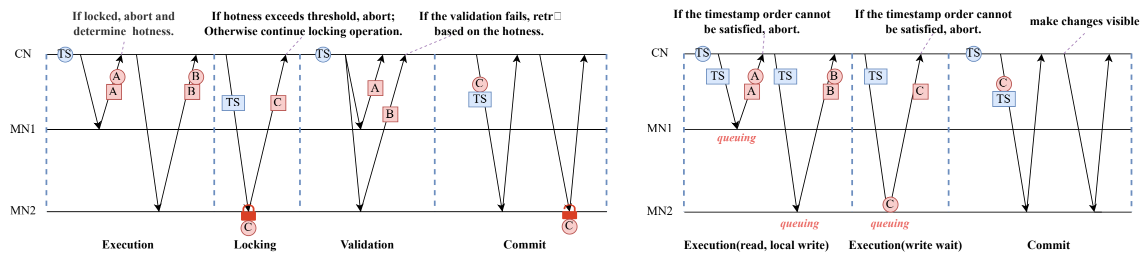
4 SCYTHE DESIGN

4.1 System Overview

This paper proposes Scythe, a novel distributed transaction system with low latency for disaggregated memory. Fig. 11 shows its key design components:

- ***Hot-aware concurrency control*** (**§4.2**). Scythe proposes a hot-aware concurrency control algorithm with a fair locking mechanism to mitigate the long-tail latency problem caused by frequent transaction retries and unfair locking under high contention. Specifically, OCC is used for transactions in low-heat datasets (low-contention scenarios). For transactions under high contention, Scythe proposes a low-latency concurrency control, *TOCC* (Timestamp Ordered OCC, **§4.4**), which improves upon the OCC algorithm by incorporating the concept of timestamp ordering (TO [3, 26]). In addition, Scythe designs a fair locking scheme, *Takeout Lock* (**§4.3**), which is based on the single-machine fair locking *Ticket Lock* algorithm [31]. Takeout Lock enables fair queuing of conflicting transactions and provides heat awareness.
- ***RDMA-friendly TSO service*** (**§4.5**). Scythe introduces a centralized timestamp management solution based on TSO to mitigate the latency issues resulting from uncertainty waiting of physical clocks. In addition, Scythe leverages RDMA-friendly TSO with the RNIC memory to eliminate the high overhead of RDMA `CAS` operations caused by PCIe transactions.
- ***RDMA-optimized RPC framework (RRPC)*** (**§4.6**). Scythe proposes a high-performance RRPC framework with a hybrid transmission mode, batch processing and other optimizations to take full advantage of the high bandwidth of RDMA networks.

As shown in Fig. 11, Scythe consists of a set of memory nodes (MNs) and compute nodes (CNs). The CNs run dtxns and access application data stored in MNs. A TSO service is running on MN's RNIC memory to provide timestamp service. The communication between the compute and memory pools is facilitated by the RRPC framework, which manages the RDMA queue pair connections.


If locked, abort and determine hotness.
If hotness exceeds threshold, abort; Otherwise continue locking operation.
If the validation fails, retr based on the hotness.
CN
MN1
MN2
TS
A
B
C
Execution
Locking
Validation
Commit
If the timestamp order cannot be satisfied, abort.
If the timestamp order cannot be satisfied, abort.
make changes visible
queuing
queuing
queuing
Execution(read, local write)
Execution(write wait)
Commit

(a) OCC for low-heat mode (b) TOCC for high-heat mode

Fig. 12. Hot-aware concurrency control in Scythe.

4.2 Hot-aware Concurrency Control

Fig. 12 illustrates the workflow for Scythe's concurrency control algorithm that comprises of two modes, namely low-heat and high-heat modes. These modes employ distinct concurrency control strategies, specifically OCC and TOCC. The current transaction dynamically selects either low or high heat mode, depending on the object's contention for locking.

4.2.1 OCC for low-heat mode. In low-heat mode, Scythe maintains the main flow of OCC with the following modifications:

(1) During the execution phase, locking phase and validation phase, the heat information is obtained (provided directly by Takeout Lock). It determines whether the transaction will continue in low-heat mode or switch to high-heat mode when retrying after aborting.

(2) During the locking phase, transactions are assigned timestamps and locked in parallel at the remote side. This approach serves two purposes: 1) Ensuring compatibility between the OCC and TOCC algorithms; 2) Fully Leveraging the high concurrency advantage of OCC and while reducing transaction processing latency through parallel locking. Additionally, this approach can further decrease the time spent in the critical zone, thereby reducing the conflict rate.

(3) During the execution phase, if a transaction is observed to be locking or queuing, it will be aborted early. This is due to the fact that when an object is found to be write-occupied during reading, only two possibilities during the validation step: 1) The object is still write-occupied; 2) Another transaction has completed updating the object and committed. In either case, the transaction will be aborted during the validation phase. In cases where the abort rate is low in low-heat mode, this approach can minimise retry overhead and lower latency.

4.2.2 TOCC for high-heat mode. In high-heat mode, Scythe proposes the TOCC concurrency control algorithm, aiming to guarantee that the execution order of transactions aligns as closely as possible with their obtained timestamps. TOCC essentially employs the concept of timestamp ordering (TO) transaction systems [3, 26] to alleviate the issue of tail latency problem of OCC under high contention. In comparison to TO, TOCC employs the following optimizations: 1) TO needs to maintain the read/write timestamps of objects, whereas TOCC only utilizes start timestamps to sequence transactions; 2) Instead of queuing or caching transaction contexts on memory nodes to ensure transaction execution order, TOCC transfers waiting to compute nodes, making it more friendly to disaggregated memory; 3) TOCC employs Takeout Lock to ensure fair transaction processing and alleviate the starvation issue present in TO; 4) TOCC caches write operations locally instead of executing them directly remotely. Specifically, Fig. 12 (b) illustrates TOCC flow:

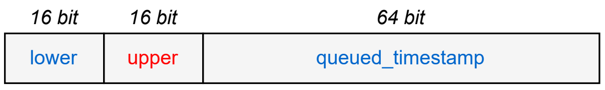


Fig. 13. Takeout Lock structure.

(1) During the execution phase, read operations include timestamp information when accessing remote memory objects, and the remote Takeout Lock ensures transactions are queued fairly in timestamp order. Moreover, RRPC batches read operations on the same node to minimize the number of network communications.

(2) During the execution phase, write operations are initially executed in the local cache, similar to OCC. Before committing the transaction, TOCC initiates queuing requests for the write sets in parallel through RRPC, thereby reducing retries caused by out-of-order transaction timestamps.

(3) TOCC does not need the validation phase since the read set of transactions executed in timestamp order is not preempted or modified by other conflicting transactions during execution. Additionally, if the transaction's timestamp order is inconsistent with the execution order, the transaction will be aborted. Based on a heuristic waiting method [26], TOCC uses RRPC's message merging and scheduling strategy between CNs and MNs to reduce transaction abort rate and minimize cross-node communication (§4.4).

(4) The two-phase commit step remains the same as OCC.

4.3 Takeout Lock

To alleviate the issues of unfairness and starvation in the existing locking mechanism, we propose the Takeout Lock based on *Ticket Lock* algorithm [31] and *Hangout Timeout* mechanism. The basic concept of the ticket lock is similar to the ticket queue management system. There are two integer values initialized to 0. The first value represents the queue ticket, indicating the thread's position in the queue, while the second value is the dequeue ticket, representing the ticket or queue position currently holding the lock.

The Takeout Lock is designed for disaggregated memory and optimized for RDMA networks. Each object is assigned a Takeout Lock by the MN. The structure of the Takeout Lock is shown in Fig. 13 and the core algorithm is shown in Fig. 14. The *lower* and *upper* fields are similar to the queue and dequeue ticket in the ticket lock. The queued_timestamp (*queued_ts* for short) records the maximum timestamp of the currently queued transaction on the Takeout Lock. With the Takeout Lock, Scythe can mitigate the long-tail latency of cross-node communication caused by unfair locking and frequent retries. Besides, Takeout Lock can provide a hot-aware mechanism to identify hot and cold objects to be locked. In summary, Takeout Lock possesses the following features:

(1) Takeout Lock enables fair queuing of conflicting transactions. At MN-side, Takeout Lock assigns a queue number to the transaction requesting the lock, based on the order in which the requests arrive (line 10 of server code, i.e., *lower* adds 1). The transaction awaits the matching of the current dequeue number with its queue number (line 10 of client code, i.e., reads *upper* the remote lock), indicating that the transaction is holding the lock. When a transaction executes an unlock operation, the *upper* field is updated (line 7 of client code, i.e., adds 1 to *upper* of the remote lock). Since only the transaction that has acquired the lock can perform the unlock operation, the requests are responded in the same order as they arrived, which ensures fairness in locking.

(2) Takeout Lock enables locking wait at CN-side. Takeout Lock prevents unnecessary waiting at MN with near-zero computation power without interrupting other operations (e.g., indexing, memory allocation). In addition, Takeout Lock enables the Hangout timeout mechanism to avoid multiple invalid locking operations, resulting in reduced RDMA bandwidth consumption. Specifically, Takeout Lock divides the locking step into two stages:

Step 1: Locking process using RPC, i.e., line 3 in client code and line 3 in server code, respectively representing the RPC request initiated by CN and the RPC processing by MN.

```
struct Takeout_Lock {
    u16 lower;
    u16 upper;
    u64 queued_ts;
};
struct Lock_Reply {
    bool is_queued;
    Takeout_Lock lock_info;
    u64 remote_lock_addr;
};
enum Mode { COLD, HOT };
```

```
1  // Client (CN-side)
2  Takeout_Lock* local_buf;
3  Lock_Reply Lock(u64 ts, Mode mode) {
4      return RPC_CALL("Lock", ts, mode, ctx);
5  }
6  void Unlock(u64 remote_addr) {
7      RDMA_FAA(local_buf, remote_addr, 1<<16);
8  }
9  void Poll(u64 remote_addr) {
10     RDMA_READ(local_buf, remote_addr, sizeof(u32));
11 }
```

```
1  // Server (MN-side)
2  Takeout_Lock lock;
3  Lock_Reply Lock(u64 ts, Mode mode) {
4      CurrentQueuedTxnNum = lock.lower - lock.upper;
5      if (ts < queued_ts) or (mode == COLD and CurrentQueuedTxnNum > ColdWatermark) or (
           CurrentQueuedTxnNum > HotWatermark){
6          return Lock_Reply (false, lock, &lock);
7      }
8      tmp = lock;
9      lock.queued_ts = ts;
10     lock.lower = lock.lower + 1;
11     return Lock_Reply (true, tmp, &lock);
12 }
```

Fig. 14. Pseudo code of Takeout Lock.

Step 2: CN verifies if the transaction holds a lock by using the RDMA READ primitive, i.e., *Poll* function (line 9) in client code. To decrease invalid cross-node communication, Takeout Lock incorporates an active waiting mechanism at CN-side, i.e., **Hangout timeout**. It is evident that Takeout Lock possesses the following property:

$$lower - upper = CurrentQueuedTxnNum \tag{1}$$

The difference between the *lower* and *upper* of the current transaction equals the number of transactions queued in front of it. To account for this, a timeout value $T_{rtt,i}$ is applied when polling for the lock, which represents an estimated one-way RTT from the CN where the current transaction resides to MN i. $T_{rtt,i}$ is maintained by Equation 2, similar to Smoothed RTT method in TCP [1].

$$T_{rtt,i} = \alpha T_{rtt,i} + (1-\alpha)Current\_T_{rtt,i} \tag{2}$$

Where $Current\_T_{rtt,i}$ is the precise time (RTT) for the current communication, and $\alpha = 0.9$) by default. Based on this timeout value, Takeout Lock can obtain the $T_{Hangout,i}$:

$$T_{Hangout,i} = CurrentQueuedTxnNum * T_{rtt,i} \tag{3}$$

If the Takeout Lock is not obtained directly at step 1, it will wait for $T_{Hangout,i}$ before performing a check. In general, an RDMA one-sided RTT takes less time than the execution of a transaction held by the lock. This means if the current transaction fails to acquire the lock and then immediately sends another request for the lock, it will most likely fail again. As a result, the timeout mechanism minimizes the number of invalid retries of the lock and avoids over-waiting.

(3) Takeout Lock enables heat awareness. When a transaction attempts to lock a specific object, the heat information of the object can be expressed by *CurrentQueuedTxnNum*, calculated according to Equation 1. We

Zipf factor	Longest queue	Second longest queue	Other queues	Highest access rate	Second highest access rate
0.9	6	5	≤2	4%	4%
0.95	10	6	≤4	9%	4%
0.99	13	4	≤3	10%	3%
1.2	31	9	≤5	25%	7%

Table 2. Queuing states under different Zipf factor

test the queuing behavior of 128 clients across 100,000 objects, following a Zipfian distribution with varying factors that are indicative of transaction conflicts. The results, as presented in Table 2, indicate a positive correlation between the access rate and the number of queued transactions. This means that a high-access memory object correspondingly had more transactions queued. Furthermore, it can be observed that most objects have fewer than 5 queued transactions, with the top two accessed objects having the longest queues. This suggests that $CurrentQueuedTxnNum$ can serve as an indicator for identifying access hotspots in objects.

Takeout Lock sets a hotspot threshold (*ColdWatermark*, default value is 3), which defines the threshold for cold and hot heat modes. Under the low-heat mode, if it is found that the number of queued transactions surpasses the *ColdWatermark*, the OCC process is aborted and the high-heat mode (TOCC process) is started. In addition, a threshold, *HotWatermark* (default value is 10) is set to limit the number of queued transactions on the hotspot data. The threshold can reduce the number of transaction abortions caused by timestamp disorder in TOCC, and prevent conflicts on hotspot objects from spreading uncontrollably to other objects.

4.4 Timestamp Ordered OCC

In this section, we examine the availability of TOCC based on workflow shown in Fig. 12 (b). We will discuss deadlocks, request reordering, and consistency in detail.

(1) TOCC avoids deadlocks. The fundamental concept behind TOCC is to ensure that the order of transaction execution aligns with the order of their start timestamps. This mitigates extensive cross-node communication caused by frequent retries triggered by *request reordering* (which means requests are not executed in the order of their timestamps [26]). We employ mathematical induction to establish the practicability of TOCC, demonstrating its ability to complete tasks within a finite timeframe without any risk of deadlock. ① When there is only 1 transaction, it is clear that the transaction can be completed without deadlocks; ② Assuming that TOCC can ensure processing is completed without deadlocks when N transactions are queued for processing; ③ When there are N+1 transactions queued for processing, there must be a transaction that holds the smallest timestamp to be at the top of the waiting queue for each object. This prioritises the transaction for execution without blocking, reducing the problem to the ② scenario. Therefore, the usability of TOCC is demonstrated.

(2) TOCC mitigates request reordering. Theoretically, controlling the order of transaction execution to align with timestamps could reduce the incidence of transaction conflicts. In practice, as illustrated in Fig. 15 and described in Aurogon[26], there are two scenarios whereby this order cannot be strictly regulated. 1) In Case 1, for instance, transaction 1 (*TXN1*) acquires the timestamp prior to transaction 2 (*TXN2*), nonetheless, *TXN2* reads the object X first since X and *TXN2*'s execution reside on the same server. As a consequence, *TXN1* is aborted. The issue arises frequently in a large heterogeneous cluster because the latency of accessing data from different servers is usually nonuniform due to different network hop counts and uneven traffic distribution. For this case, TOCC utilizes a heuristic waiting method [26]. When a read/write request arrives at MN-side, the transaction needs to wait for approximately 20μs before retrying if the number of queued transactions is greater than *ColdWatermark* and less than *HotWatermark*. Aurogon's test results demonstrate that a 20μs deferral for reads on hotspots reduces write failures by 50% in most cases [26]. 2) In Case 2, *TXN1* acquires the timestamp before *TXN2*, but receives a larger timestamp. As *TXN2* has a smaller timestamp, it will abort when accessing

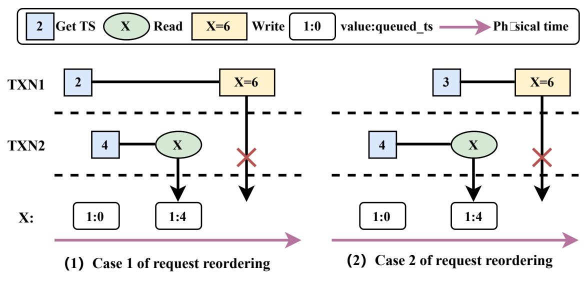


Fig. 15. The cases of request reordering.

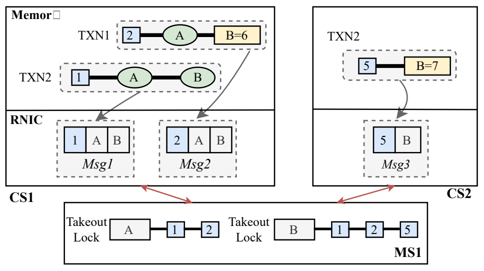


Fig. 16. TOCC mitigates request reordering by merging messages.

X. Inaccurate distributed clocks cause the order of two transaction timestamps to not be consistent with the physical time order in which they get timestamps. Since Scythe uses a centralized timestamp mechanism that avoids imprecise distributed clocks, case (b) does not occur.

In addition, TOCC utilizes RRPC's message merging (§4.6) between CNs and MNs to reduce request reordering, as shown in Fig. 16. At CN-side, we use a coroutine framework[29] to schedule and overlap all RDMA requests from different transaction clients, with each coroutine corresponding to a client. We share a ring buffer for clients that belong to the same thread (e.g., *TXN1* and *TXN2* in Fig. 16). The read/write operations of a transaction are not immediately translated into RDMA communication primitives to communicate with the remote memory, but are first cached in the buffer (e.g. Msg1 and Msg2 in Fig. 16) until a transaction client needs to obtain the corresponding object data or metadata information (including version, lock information, etc.). At MN-side, the requests received from different nodes over a period of time will be grouped and organized. A batch of requests is sorted according to the object key as the first keyword and the request timestamp as the second keyword (e.g., 'A-1-2' and 'B-1-2-5' in Fig. 16), which can be processed with a scan operation. Takeout Lock is acquired in the order of the transaction timestamp, thus reducing aborts caused by request reordering.

(3) Consistency guarantee of TOCC and OCC. The consistency of OCC and TOCC themselves is easily guaranteed, and it is necessary to analyze whether conflicts between OCC and TOCC can be detected. Fig. 17 illustrates three scenarios: 1) In Case 1, a TOCC transaction with a larger timestamp first locks or queues on object *A*. An OCC transaction with a smaller timestamp will be aborted immediately when reading object *A*, instead of discovering the update during the validation phase. Second, if the OCC transaction holds a larger timestamp than the TOCC transaction and starts before the TOCC transaction, since the OCC's read process is lock-less and queue-less, it will not trigger the abort of the TOCC transaction. 2) In Case 2, an OCC transaction reading object *A* with a lower timestamp than TOCC transaction is aborted due to a subsequent locking operation. if the OCC transaction has a higher timestamp than the TOCC transaction, the TOCC transaction is not aborted if object *B* is hot and the number of queues exceeds the *HotWatermark*. 3) In Case 3, the OCC transaction and the TOCC transaction are executed in the order of timestamps. However, to ensure consistency, OCC transaction will be aborted during the validation phase when a transaction is discovered to be locked, queued, or if the object in the read set has been updated with a newer version. In summary, the consistency can be guaranteed in the mixed case of OCC and TOCC. Especially, in high contention scenarios, Scythe prioritizes TOCC transactions to process and ensures that they are not aborted.

4.5 RDMA-friendly TSO

We propose an RDMA-friendly TSO service to provide monotonically incrementing timestamps, as shown in Fig. 18. The following techniques are utilized to reduce the latency:

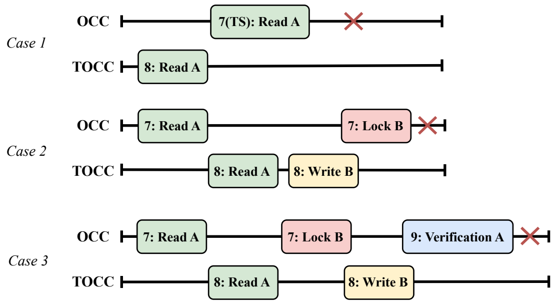


Fig. 17. Conflict cases between TOCC and OCC.

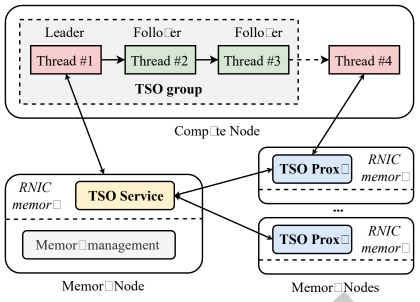


Fig. 18. RDMA-friendly TSO.

(1) The TSO service operates within the MN's RNIC memory to minimize PCIe transactions at MN-side and consequently achieve lower latency. Our tests revealed a throughput of approximately 110 Mops for RDMA CAS operation. However, as the number of CNs increases, each CN has to establish an RC (Reliable Connection) link with the TSO service. Therefore, the TSO service has to maintain a significant number of communication queues, such as RDMA QPs, CQs, etc., that compete for RNIC memory. High contention with limited RNIC memory leads to frequent cache invalidation and greater communication latency [19, 63].

(2) TSO Group. Multiple threads within a CN may request TSO service for timestamps during the same interval, which will then be organized on a linked list, with the current head node referred to as the leader thread (*Leader*), as shown in Fig. 18. The leader thread amalgamates timestamp requests from other concurrent threads (*Followers*) to form a TSO group, which will then be sent to the TSO service. By obtaining TSO in batches, the RDMA communication latency may be amortised within TSO group, enhancing efficiency and reducing overhead.

(3) TSO Proxy. In addition, a TSO proxy layer is proposed to limit the number of RDMA QPs on a TSO service. The proxy service sends a batch of incoming requests from the TSO group to the TSO service, thereby diminishing the interactions between the CN and the TSO service. Moreover, the TSO proxy maintains the currently maximum timestamps assigned, enabling quick recovery of the timestamp service in case of a single point failure in TSO service.

The TSO workflow is as follows: 1) Add the threads that need to get timestamps to the TSO group, with the first thread as the Leader; 2) The Leader counts the number of timestamped requests in the group (e.g. n), and then randomly selects a TSO proxy or TSO service to send a batch of TSO requests. The TSO proxy packages incoming requests and sends them to the TSO service; 3) The leader allocates n timestamps and returns them to the corresponding threads, and then adds n to the current timestamp (using RDMA *FAA* verb).

4.6 RRPC Framework

Based on the guidelines and observations in §3.4, the main features of RRPC are as follows:

(1) Decoupling the RDMA QPs and ring buffers. This can reduce the NIC cache miss rate by QP sharing, enabling efficient disconnection and reconnection of QPs during cluster changes.

The architecture of RRPC is shown in Fig. 19, which consists of the buffer layer and the connection layer. 1) In the buffer layer, each thread exclusively occupies multiple *Rockets*, each of which represents a message ring buffer for communication with a specific remote node. The small RPC messages in a Rocket form a Msg batch, which is subsequently transmitted to the connection layer. 2) The connection layer consists of *RdmaStraws* shared between multiple threads. *RdmaStraws* represents a collection of QPs that have established a connection with a specific remote node. Each *Rocket* dynamically selects QPs based on load conditions. The number of QPs is also not static, instead, it is dynamically created or released by using the average number of threads served by

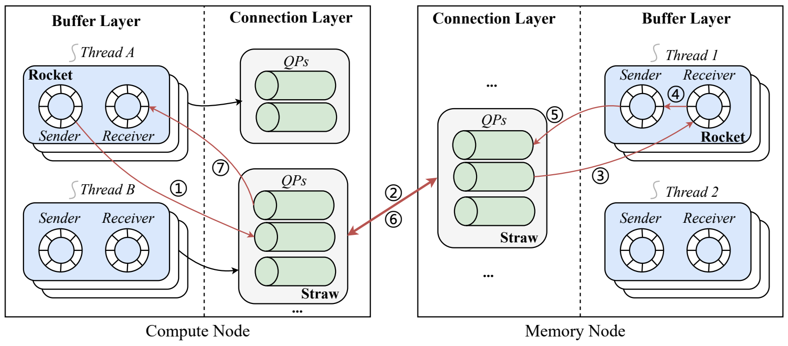


Fig. 19. The architecture of RRPC.

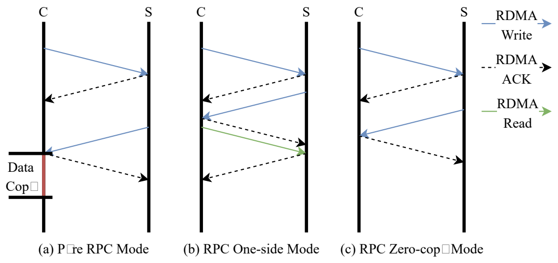


Fig. 20. Different communication modes in RRPC.

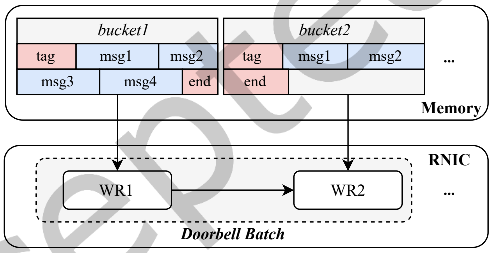


Fig. 21. Message ring buffer in RRPC.

each QPs in *RdmaStraws* as a threshold. In addition, in order to mitigate the effect of RPC communication on one-sided RDMA primitives, each *RdmaStraws* divides QPs into two categories: one for RPC communication, while the other is specialized for one-sided RDMA operations. 3) at the MN-side, requests from different CNs are directed to the receive ring buffer.

Fig. 19 illustrates an RPC procedure. It is important to note that this figure displays solely a pure RPC process, whereas RRPC supports a hybrid communication method, which will be described later. ① RDMA work requests are generated by merging Msg batching and Doorbell batching, as explained subsequently. ②/③ Batch-processed messages are transmitted to the MN's ring buffer by means of RDMA `WRITE` operations. ④ The MN-side CPU proceeds to handle the messages. ⑤/⑥/⑦ represent the symmetric communication processes from MNs to CNs.

(2) Hybrid communication modes. Compared with the traditional RDMA and RPC frameworks, RRPC incorporates a hybrid approach that can flexibly select between RPC and one-sided RDMA communication protocols depending on data patterns. RRPC is inspired by the test results in Fig. 10 and uses 512B as the threshold to dynamically choose the communication modes. Fig. 20 shows the three communication modes included in RRPC. (a) *Pure RPC mode* for communicating data that is less than 512B in size, and is applicable in cases such as transactions that involve locking, data index queries, and memory allocation. (b) *RPC & one-sided mode* is more appropriate for handling unstructured big data whose size exceeds 512B, or in scenarios where the data size is unknown, for instance, in object storage. This is because it can be challenging for clients to determine the size of the object that they intend to access before requesting the server. In this case, it is necessary to obtain the remote address via RPC first, allocate a specified size of space locally, and finally remote fetch via an RDMA one-sided `READ` operation. (c) *RPC zero-copy mode* is designed for structured data that exceeds 512 bytes in size and has a fixed format, such as SQL scenarios. Because data has a fixed size, this communication mode can incorporate the address of local space when initiating an RPC request, and facilitate direct writing of the data through an RDMA one-sided `WRITE` operation.

(3) Combining Msg batching and Doorbell batching. Based on the test results in Fig. 9 and Fig. 10, RRPC design a new ring buffer strategy by combining Msg batching and Doorbell batching. As shown in Fig. 21, RRPC logically divides a contiguous ring buffer into buckets (default 512B size per bucket). Messages within a bucket are combined into a single memory area to form a Msg batch and an RDMA request. When there are a large number of batches, the work requests (e.g., WR1 and WR2 in Fig. 21) generated in each bucket are linked together in a linked list to form a Doorbell batch. Then, the linked list of work requests is dispatched to RNIC via an *ibv_post_send* operation. Such a design takes full advantage of message batching and Doorbell batching. In addition, a label is recorded at the start address and at the end of message batch for each bucket, respectively. Their values indicate the length of the message batch, and the message receiver polls these two labels to see if they are non-zero and consistent, avoiding incomplete message writes.

4.7 Implementation

In Scythe, RDMA RPC framework is implemented on eRPC[27]. We implement a log-structure in-memory KV store (called *LogDB*) for memory management in memory nodes, providing read and write interfaces. LogDB is constructed on DRAM and persistent memory (PM). The data index part (concurrent SkipList) is situated in DRAM. In PM, LogDB preserves distinct version values of each object through a linked list, enabling efficient data management and garbage accumulation. The source code for this system is accessible via https://github.com/PDS-Lab/scythe.

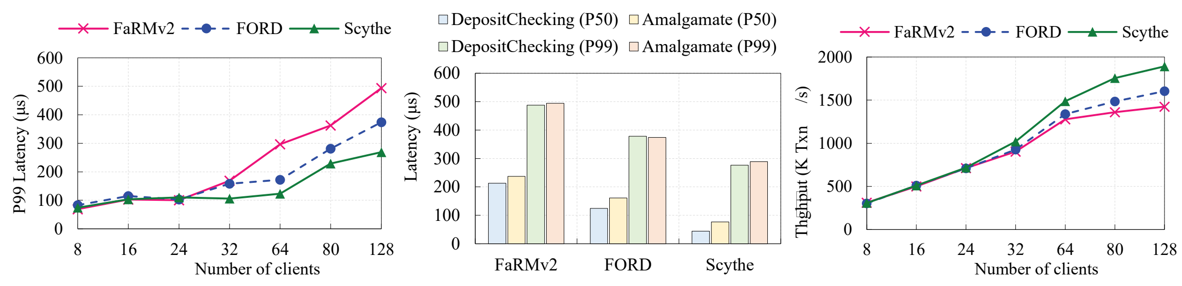
5 EVALUATION

5.1 Experiment Setup

Environment. All experiments are conducted on three machines, each equipped with Intel Xeon Gold 5218R CPUs @2.10Ghz, 46 GB DRAM, one 100Gbps Mellanox ConnectX-5 RNIC and two 128GB Intel(R) Optane Persistent Memory for memory expansion. The operating system running on all servers is Ubuntu 20.04 with Linux 5.4.0-144-generic. One machine is leveraged as the compute pool to run benchmarks. In this machine, a task thread starts 8 coroutines and each coroutine executes a transaction request (considered a transaction client). We run 1, 2, 3, 4, 8, 10, and 16 threads in our tests, corresponding to 8, 16, 24, 32, 64, 80, and 128 transaction clients. Other two machines form the memory pool: one machine operates 2 TSO proxies and 1 memory node, while the other machine runs 1 TSO service and 2 memory nodes.

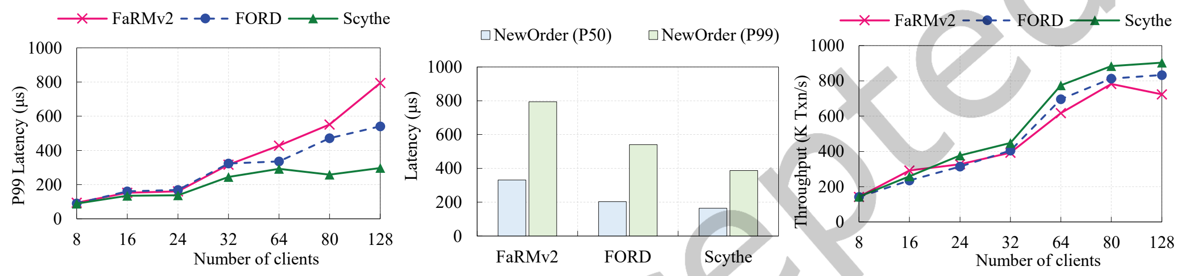
Comparisons. Scythe is compared with two other state-of-the-art RDMA-based distributed transaction systems: 1) **FaRMv2** [44] and 2) **FORD** [63]. We execute FORD using open-source codes. Since FaRMv2's code is not publicly available, we use the code in GAM system[6], which provides an implementation of FaRMv2. To test for fairness, we implement the following modifications to these systems: 1) Note that FaRMv2 is not really "disaggregated" architecture, as its compute nodes have local memory of the same size as the remote memory. We modify some configurations (reducing local memory at CN-side) to port it to a disaggregated architecture. 2) FaRMv2 utilizes disks for storage and FORD uses PM. To be fair, we store data in DRAM and PM (as extended memory mode). 3) FaRMv2 and FORD use different data indexes to manage data. In our experiments, all systems use LogDB, Scythe's log-structured storage engine. 4) We do not consider primary-backup replication, which should be decoupled from the transaction processes. 5) We do not use FaRMv2's recovery feature to improve its performance.

Workloads. 1) We first test the performance of these systems using two OLTP benchmarks, i.e., SmallBank [50], and TPC-C [5], which are widely used in previous studies [11, 19, 26, 58, 63]. **SmallBank** simulates a banking application that includes 2 tables, in which 85% of transactions are read-write, and the record size is



(a) P99 latency (b) P50/P99 latency (128 clients) (c) dtxn throughput

Fig. 22. The dtxn throughput and latency on SmallBank workloads.



(a) P99 latency (b) P50/P99 latency (128 clients) (c) dtxn throughput

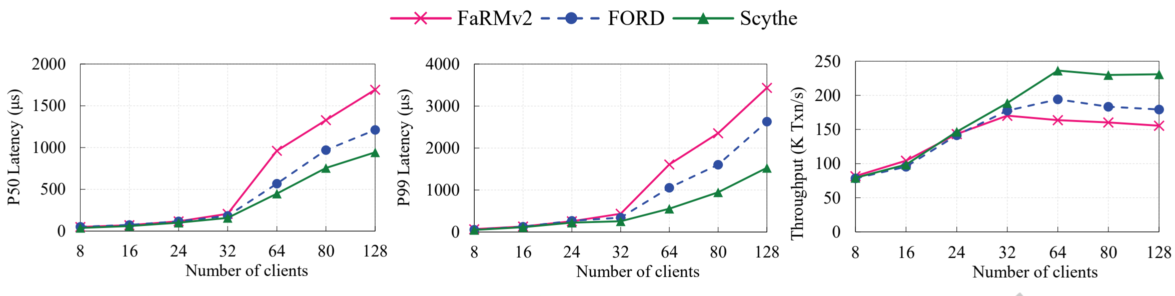
Fig. 23. The dtxn throughput and latency on TPC-C workloads.

16B. We import 100,000 randomly generated account information and select four transaction types (including Balance, DepositChecking, TransactSaving, and Amalgamate) **TPC-C** models a complex ordering system that consists of 9 tables, in which 92% of transactions are read-write, and the record size is up to 672B. We generate 10 warehouses, each serving 10 districts, 3,000 customers, with 100,000 items in inventory. We run 1 million dtxns in each benchmark. The operating accounts in SmallBank and order items in TPC-C for each transaction are generated through a Zipfian distribution (skewness 0.99). 2) We also run a key-value store (*SkipList* as the index) and **YCSB** benchmark [13] on these systems. The kv store stores 1 million key-value pairs in one table, in which the key is 8B and value is 40B. The keys are generated through a Zipfian distribution. **Performance indicators**: we report the throughput by counting the number of committed dtxns per second and report the processing time of the committed dtxn as the latency, including the 50th (P50) and 99th (P99) percentile latency.

5.2 OLTP benchmarks

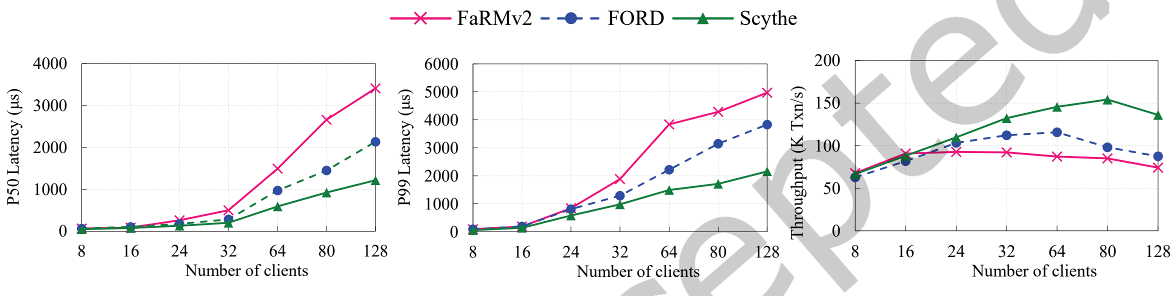
Fig. 22 and Fig. 23 shows the dtxn throughput and latency of different systems under the SmallBank and TPC-C benchmarks. It can be seen that compared to FORD and FaRMv2, Scythe has a lower latency and a higher transaction throughput on OLTP workloads. Specifically, based on the results, the following conclusions can be drawn:

(1) Scythe can significantly reduce tail latency, especially under high-contention and high-concurrent workloads. As shown in Fig. 22 (a), Scythe reduces P99 latency by up to 58% and 30% in comparison with FaRMv2 and FORD on SmallBank workloads. With an increase in concurrent clients, OCC-based concurrency control makes the tail latency of FaRMv2/FORD rise significantly due to high retry overhead. Scythe minimises tail latency increase through the application of a fair TOCC algorithm, which decreases conflicts among concurrent



(a) P50 latency (b) P99 latency (c) dtxn throughput

Fig. 24. The dtxn throughput and latency on YCSB workloads (read:write ratio=3:7).



(a) P50 latency (b) P99 latency (c) dtxn throughput

Fig. 25. The dtxn throughput and latency on YCSB workloads (write-only).

clients and reduces the number of retries and cross-node accesses. In addition, Fig. 23 (a) shows that Scythe's improvement in tail latency is more pronounced for TPC-C workloads with higher conflict probability. Scythe's tail latency is up to 63% and 45% lower than those of FaRMv2 and FORD, respectively.

(2) Scythe can reduce latency across different transaction types. As shown in Fig. 22 (b), For the high-conflict scenario with 128 clients, P50 latency of DepositChecking operations is reduced by 79.3% and 64.5% compared to FaRMv2 and FORD, respectively. P99 latency is reduced by 43.4% and 27%. For Amalgamate operations, P50 latency is reduced by 67.5% and 52.2% than FaRMv2 and FORD, and P99 latency is reduced by 41.5% and 22.7%. Fig. 23 (b) illustrates that Scythe achieves approximately 50% and 30% lower P50 and P99 latency of NewOrder operations. In addition to the low-latency concurrency control strategy, the RDMA-friendly timestamp management and communication framework also helps Scythe reduce transaction latency.

(3) Scythe has the potential to enhance transaction throughput. As shown in Fig. 22 (c) and Fig. 23 (c), Scythe improves transaction throughput by 15%–30% than FaRMv2, and 10%–20% than FORD on SmallBank and TPC-C workloads. Scythe's main advantage is its capability to perform well under high-concurrency scenarios, which pose a serious challenge to FaRMv2 as it suffers from starvation and retry overhead. Scythe overcomes the starvation issue by adopting a fair locking mechanism and leveraging the RRPC framework to increase transaction throughput.

5.3 YCSB Benchmarks

Fig. 24 and Fig. 25 display the dtxn latency and throughput of different systems under YCSB benchmarks. Compared to OLTP workloads, the SkipList scenario has a higher conflict rate. Scythe still performs better on both metrics.

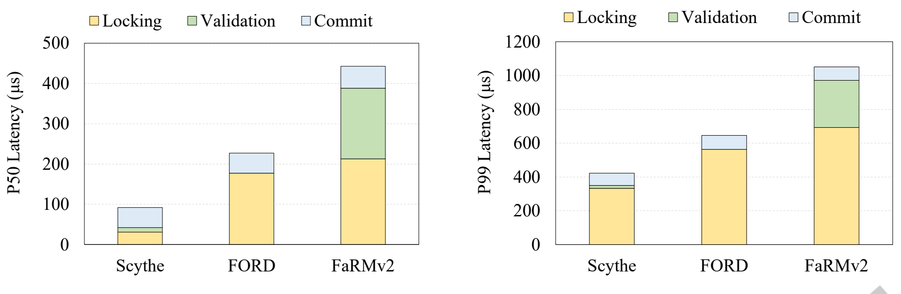


Fig. 26. The latency decomposition on YCSB workloads (write-only).

1) Latency: With a read/write ratio of 3:7, the latency of these transaction systems increases as the number of clients increases. The peak tail latency of FORD and FaRMv2 under 128 clients increases by 49 and 45 times, respectively. In contrast, Scythe mitigates the increase in tail latency by approximately 50% and 30% compared to FaRMv2 and FORD, respectively. Scythe is better on write-only transactions. As shown in Fig. 25 (a)(b), Scythe achieves up to 62% and 46% lower P50/P99 latency than FaRMv2 and FORD. As the proportion of write transactions in YCSB workloads increases, it leads to a corresponding increase in the conflict rate between transactions. Therefore, the test findings underline Scythe's potential to enhance performance in high-contention workloads.

2) Throughput: With a read/write ratio of 3:7, Scythe improves the throughput by up to 20%–40% than other systems. Under write-only workloads with 128 clients, the throughput of all systems degrades due to extremely high conflicts. Still, scythe maintains a 50%–80% throughput improvement due to efficient concurrency control and RDMA-optimized communication mechanisms.

5.4 In-Depth Analysis

In this section, we analyze the performance benefits of Scythe's key techniques in depth. First, we demonstrate the impact of concurrency control strategies on latency through a latency decomposition. Second, we focus on the impact of three strategies on Scythe's performance, including RRPC, Takeout Lock and TSO services.

Latency decomposition. Fig. 26 shows the P50/P99 latency decomposition (excluding execution phase) for different systems under YCSB workloads (write-only). The following conclusions can be drawn based on the test results: 1) The primary variation between Scythe and FORD is their locking overhead. FORD adopts RDMA CAS locking, while Scythe adopts the Takeout Lock, which is a fair lock designed for high-contention cases. Compared to FORD, Scythe reduces about 146μs in P50 latency and more than 230μs in P99 latency. 2) The main differences between Scythe and FaRMv2 are in the validation and locking phases. The reason for locking overhead is consistent with the FORD's. As for validation phase, Scythe reduces the latency by about 90% compared to FaRMv2. The heavy validation overhead in FaRMv2 is caused by multiple retries of OCC transactions under high conflict situations. Scythe employs a TOCC strategy with no validation phase, significantly reducing retry overhead and lowering latency.

Takeout Lock. We compare the latency of locking operations between Takeout Lock and RDMA CAS locking strategy (CAS Lock for short) using 64 threads under different conflict rates (skewness, tuned by θ). The holding time for the lock of each thread is set to 25μs. As shown in Fig. 27, Takeout Lock has a lower latency than CAS Lock under all workloads. As the conflict rate increases, Takeout Lock reduces P50 latency by 17.1%–31.7% and P99 latency by 16%–31.4%. Results show that Takeout Lock is more suitable for highly-skewed, latency-sensitive workloads compared to CAS Lock. Takeout Lock has a fair queuing mechanism, no matter how high the access

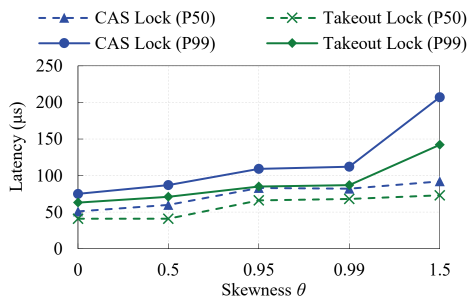


Fig. 27. The latency when using different locking methods.

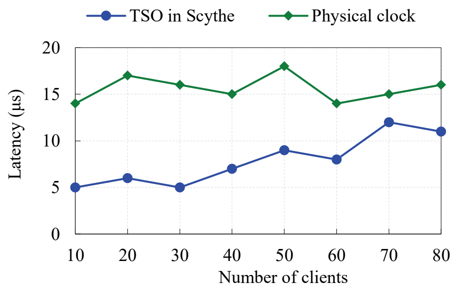
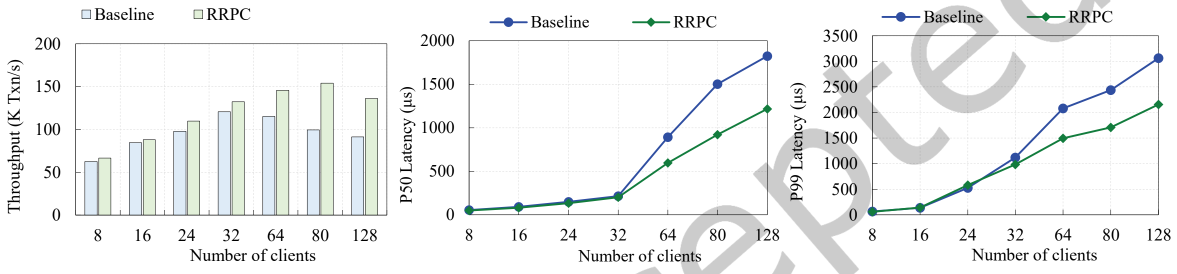


Fig. 28. The latency when using different timestamp management methods.



(a) dtxn throughput (b) P50 latency (c) P99 latency

Fig. 29. The dtxn throughput and latency tests of RRPC.

conflict rate is, the queue length of the queue is at most 64 (64 threads), and the lock can be acquired after a maximum of 63 threads. CAS Lock does not take fairness into consideration, thus starving some requesting lock threads and further inducing high tail latency. In addition, Takeout Lock limits the number of one-sided RDMA reads that ask if the lock was successful, reducing the use of the effective RDMA bandwidth.

RDMA-friendly TSO. We test the latency of obtaining timestamps using Scythe's TSO service and physical timestamp method (FaRMv2) with different number of clients. For simplicity, we use a single thread to run a client (without coroutines), and since there are only 80 cores in a machine, we can run up to 80 clients. As shown in Fig. 28, the timestamp latency is stable for both methods. Compared to physical clock, Scythe's TSO service reduces approximately 50% latency. In addition, Scythe's TSO service achieves a throughput of close to 70 MOPS at 64 clients, which greatly surpasses the transaction throughput and does not become a bottleneck. However, as shown in Fig. 28, the local synchronization operation introduced to improve the throughput rate leads to a slight increase in latency for the RDMA-friendly TSO as the number of clients increases.

RRPC framework. We compare the RRPC with FaRMv2's RPC [19] framework using different RPC clients under YCSB workloads (write-only). Fig. 29 (a) shows that RRPC achieves 1.12–1.49× higher throughput than FaRMv2's RPC, especially in high conflict situations. As shown in Fig. 29 (b)/(c), RRPC exhibits lower latency, with P50 latency reduced by 20–40% and P99 latency reduced by about 30%. RRPC's batching policy can mitigate tail latency and optimize transaction throughput.

6 DISCUSSION

Optimizing current system. Scythe aims to solve the problems of OCC-based transaction systems on disaggregated memory and proposes a hot-aware transaction mechanism to alleviate the long-tail latency under high-contention.

In the future work, we will design automated heat threshold adjustment strategies to improve adaptability and flexibility. In addition, we will focus on the optimization of multi-replica mechanisms and other concurrency control algorithms.

Extending future platform. Currently, Scythe is designed for RDMA-based disaggregated memory, as RDMA technology is more mature and widely used than other network solutions. Nonetheless, RDMA is not the only design options for memory disaggregation and existing RDMA-based memory disaggregation solutions suffer from high latency and additional overheads including page faults and code refactoring[22, 54, 56]. The Compute Express Link (CXL) [45] is an open industry standard protocol to allow interconnections between diverse heterogeneous devices via PCIe while ensuring cache coherence. CXL emerges as a promising technology to realize low-cost, high-performance memory disaggregation [22, 32]. However, existing CXL-based approaches have physical distance limitation and cannot be deployed across racks [54, 56]. In addition, due to the lack of commercially available mass-produced CXL hardware and supporting infrastructure, current research on CXL memory relies on custom FPGA prototypes [22, 54] or simulation using NUMA nodes [56]. In the future work, we will try Scythe and optimize it in different disaggregated memory platforms.

7 CONCLUSION

In this study, we develop Scythe, a low-latency RDMA-enabled distributed transaction system for disaggregated memory. In order to mitigate long-tail latency under high contention, Scythe proposes a hot-aware concurrency control strategy, which adopts timestamp-ordered OCC based on fair locking to minimize the high overhead due to high abort rate and unfair locking mechanism. In addition, Scythe presents an RDMA-aware TSO timestamp mechanism and an RPC framework to improve the efficiency of timestamp management and RDMA communication. Evaluations indicate that Scythe significantly outperforms other RDMA-based distributed transaction systems in terms of throughput and latency under all workloads.

ACKNOWLEDGMENTS

We appreciate all reviewers and editors for their insightful comments and feedback. This work was sponsored by the National Key Research and Development Program of China under Grant No.2023YFB45027-01, the National Natural Science Foundation of China under Grant No.62072196, the Key Research and Development Program of Guangdong Province under Grant No. 2021B0101400003, the Creative Research Group Project of NSFC No.61821003.

REFERENCES

[1] Mark Allman, Vern Paxson, and Ethan Blanton. 2009. *TCP congestion control.* Technical Report.

[2] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013), 1–154.

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

[4] Pramod Bhatotia, Alexander Wieder, İstemi Ekin Akkuş, Rodrigo Rodrigues, and Umut A Acar. 2011. Large-scale incremental data processing with change propagation. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 11).*

[5] TPC Benchmark C. 2010. The Transaction Processing Council. http://www.tpc.org/tpcc/.

[6] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1604–1617.

[7] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* 79–92.

[8] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on*

Management of Data. 2477–2489.

[9] Zhichao Cao and Siying Dong. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*.

[10] Xinyi Chen, Liangcheng Yu, Vincent Liu, and Qizhen Zhang. 2023. Cowbird: Freeing CPUs to Compute by Offloading the Disaggregation of Memory. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 1060–1073.

[11] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–17.

[12] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jiwu Shu. 2022. Plor: General Transactions with Predictable, Low Tail Latency. In *Proceedings of the 2022 International Conference on Management of Data*. 19–33.

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

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

[15] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.

[17] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.

[18] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1930–1943.

[19] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 401–414.

[20] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE, 173–184.

[21] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 249–264.

[22] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[23] Chuanxiong Guo. 2017. RDMA in data centers: Looking back and looking forward. *Keynote at APNet* (2017).

[24] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *Proceedings of the 2021 International Conference on Management of Data*. 658–670.

[25] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[26] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. 2022. Aurogon: Taming Aborts in All Phases for Distributed {In-Memory} Transactions. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 217–232.

[27] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter {RPCs} can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 1–16.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance {RDMA} systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 437–450.

[29] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs.. In *OSDI*, Vol. 16. 185–201.

[30] Kostas Katrinis, Dimitris Syrivelis, Dionisios Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, Kobi Hasharoni, Daniel Raho, Christian Pinto, F Espina, et al. 2016. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 690–695.

[31] Leslie Lamport. 2019. A new solution of Dijkstra's concurrent programming problem. In *Concurrency: the works of leslie lamport*. 171–178.

[32] Hyokeun Lee, Kwanseok Choi, Hvuk-Jae Lee, and Jaewoong Sim. 2023. SDM: Sharing-Enabled Disaggregated Memory System with Cache Coherent Compute Express Link. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 86–98.

[33] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R Narasayya. 2016. Accelerating relational databases by leveraging remote memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data*. 355–370.

[34] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[35] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. {ROLEX}: A Scalable {RDMA-oriented} Learned {Key-Value} Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 99–114.

[36] Keith Ansel Marzullo. 1984. *Maintaining the time in a distributed system: an example of a loosely-coupled distributed service (synchronization, fault-tolerance, debugging)*. Ph.D. Dissertation. Stanford University.

[37] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase reconciliation for contended in-memory transactions. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 511–524.

[38] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant software distributed shared memory. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 291–305.

[39] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[40] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 184–200.

[41] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. (2010).

[42] Ivy Peng, Roger Pearce, and Maya Gokhale. 2020. On the memory underutilization: Exploring disaggregated memory on hpc systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 183–190.

[43] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*. 1–13.

[44] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*. 433–448.

[45] Debendra Das Sharma and Ishwar Agarwal. 2022. Compute Express Link. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf/.

[46] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 255–270.

[47] API Specification. 2019. Intel® Rack Scale Design (Intel® RSD) Storage Services. (2019).

[48] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[49] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. 2021. CoRM: Compactable remote memory over RDMA. In *Proceedings of the 2021 International Conference on Management of Data*. 1811–1824.

[50] The H-Store Team. 2023. Smallbank benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[51] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for {Large-Scale} advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 363–378.

[52] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[53] Jacob Wahlgren, Maya Gokhale, and Ivy B Peng. 2022. Evaluating Emerging CXL-enabled Memory Pooling for HPC Systems. *arXiv preprint arXiv:2211.02682* (2022).

[54] Chenjiu Wang, Ke He, Ruiqi Fan, Xiaonan Wang, Wei Wang, and Qinfen Hao. 2023. CXL over Ethernet: A Novel FPGA-based Memory Disaggregation Design in Data Centers. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 75–82.

[55] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*. 1033–1048.

[56] Zhonghua Wang, Yixing Guo, Kai Lu, Jiguang Wan, Daohui Wang, Ting Yao, and Huatao Wu. 2024. Rcmp: Reconstructing RDMA-Based Memory Disaggregation via CXL. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), 1–26.

[57] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. 2021. Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp.. In *NSDI*. 357–372.

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

[59] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

[60] Juncheng Yang, Yao Yue, and KV Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 191–208.

[61] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2016. The end of a myth: Distributed transactions can scale. *arXiv preprint arXiv:1607.00655* (2016).

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

[63] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. {FORD}: Fast One-sided {RDMA-based} Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 51–68.

[64] Wangda Zhang and Kenneth A Ross. 2020. Exploiting data skew for improved query performance. *IEEE Transactions on Knowledge and Data Engineering* 34, 5 (2020), 2176–2189.

[65] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, et al. 2021. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1900–1912.

[66] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory.. In *USENIX Annual Technical Conference*. 15–29.

Received 23 November 2023; revised 9 April 2024; accepted 20 May 2024