Localized Validation Accelerates Distributed Transactions on Disaggregated Persistent Memory

MING ZHANG, YU HUA, PENGFEI ZUO, and LURONG LIU, Huazhong University of Science and Technology

Persistent memory (PM) disaggregation significantly improves the resource utilization and failure isolation to build a scalable and cost-effective remote memory pool in modern data centers. However, due to offering limited computing power and overlooking the bandwidth and persistence properties of real PMs, existing distributed transaction schemes, which are designed for legacy DRAM-based monolithic servers, fail to efficiently work on the disaggregated PM. In this article, we propose FORD, a *F*ast *O*ne-sided *R*DMA-based *D*istributed transaction system for the new disaggregated PM architecture. FORD thoroughly leverages one-sided remote direct memory access to handle transactions for bypassing the remote CPU in the PM pool. To reduce the round trips, FORD batches the read and lock operations into one request to eliminate extra locking and validations for the read-write data. To accelerate the transaction commit, FORD updates all remote replicas in a single round trip with parallel undo logging and data visibility control. Moreover, considering the limited PM bandwidth, FORD enables the backup replicas to be read to alleviate the load on the primary replicas, thus improving the throughput. To efficiently guarantee the remote data persistency in the PM pool, FORD selectively flushes data to the backup replicas to mitigate the network overheads. Nevertheless, the original FORD wastes some validation round trips if the read-only data are not modified by other transactions. Hence, we further propose a localized validation scheme to transfer the validation operations for the read-only data from remote to local as much as possible to reduce the round trips. Experimental results demonstrate that FORD significantly improves the transaction throughput by up to 3× and decreases the latency by up to 87.4% compared with state-of-the-art systems.

CCS Concepts: • **Information systems** → **Distributed storage**; **Distributed database transactions**; • **Hardware** → **Non-volatile memory;**

Additional Key Words and Phrases: Disaggregated data center, persistent memory, distributed transaction processing, one-sided remote direct memory access

ACM Reference format:
Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2023. Localized Validation Accelerates Distributed Transactions on Disaggregated Persistent Memory. *ACM Trans. Storage* 19, 3, Article 21 (June 2023), 35 pages.
https://doi.org/10.1145/3582012

The preliminary version of this work appeared in *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)* as "FORD: Fast One-Sided RDMA-Based Distributed Transactions for Disaggregated Persistent Memory" [94].
This work was supported in part by the National Natural Science Foundation of China (NSFC) under grants 62125202 and U22B2022, and Key Laboratory of Information Storage System, Ministry of Education of China.
Authors' address: M. Zhang, Y. Hua (corresponding author), P. Zuo, and L. Liu, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, Hubei, China, 430074; emails: {csmzhang, csyhua, pfzuo, lrliu}@hust.edu.cn.

1553-3077/2023/06-ART21 $15.00
https://doi.org/10.1145/3582012

1 INTRODUCTION

Memory disaggregation decouples the compute and memory resources from the traditional monolithic servers into independent compute and memory pools, and has received extensive interests from both industry [18, 22, 62] and academia [1, 32, 68, 78, 94, 96]. With efficient resource pooling in datacenters, the resource utilization, elasticity, failure isolation, and heterogeneity are largely improved [66]. Specifically, the compute pool runs programs with a small DRAM buffer, which is leveraged to store metadata and intermediate results. Moreover, the memory pool stores application data with weak compute units, which are used only for memory allocations and interconnections [96]. High-speed networks (e.g., **Remote Direct Memory Access (RDMA)**) are adopted to connect the compute and memory pools [78]. Recently, **Persistent Memory (PM)** has become available on the market [20], which exhibits non-volatility and low latency with high density and low costs [89]. Therefore, the efficient use of PM becomes important to build a persistent, large, and cost-effective disaggregated PM pool [75].

To ensure that the data are atomically and consistently accessed in the PM pool, the compute pool needs **Distributed Transactions (dtxns)** to access remote data. However, existing RDMA-based dtxn systems are designed for traditional monolithic servers, in which each server hosts the CPU and DRAM resources. These systems fail to work on the disaggregated PM, since the PM pool does not contain CPUs to frequently handle extensive computation tasks during dtxn processing, such as concurrency control in HTM [14, 82], data retrieving [81], locking [31, 46, 65], and busy polling [30]. Moreover, legacy systems do not consider the bandwidth and persistence properties of real PM, leading to low throughputs and inconsistent remote writes. To run dtxns on the disaggregated PM, an intuitive solution is to leverage one-sided RDMA to bypass the CPU in the PM pool. However, we observe that using one-sided RDMA in existing dtxn systems incurs substantial round trips and access contentions, which significantly decrease the performance. It is non-trivial to design a high-performance dtxn system for disaggregated PM due to the challenges as follows.

(1) Long-Latency Processing. Legacy systems adopt **Optimistic Concurrency Control (OCC)** [49] to serialize dtxns, and the **Primary-Backup Replication (PBR)** for high availability. OCC is efficient for **Read-Only (RO)** dtxns due to no locks on RO data. However, for the read-write dtxns, the data in the read-write set consume three round trips to be read, locked, and validated before writing remote replicas, thus heavily increasing the latency. Furthermore, to ensure that the dtxn can roll forward once the primary fails, prior designs consume two round trips to write remote replicas—that is, writing redo logs to backups and then updating primaries, which however delays the dtxn commit.

(2) Limited PM Bandwidth on the Primary. When using the PBR, legacy systems only allow the primary to be read, since the newest data in backups are still stored in redo logs after the dtxn commits. Hence, all the RDMA read/write requests are issued to the primary to be handled. However, the PM DIMM suffers from lower write bandwidth (e.g., 12.9 GB/s of six interleaved 256-GB PM DIMMs [89]) than recent **RDMA-capable NICs (RNICs)** (e.g., 25GB/s for a dual-port ConnectX-5 RNIC [24]). The substantial RDMA reads saturate PM bandwidth and further block write requests. As a result, the primary's PM becomes a performance bottleneck, which decreases the throughput.

(3) Lack of Remote Persistency Guarantee. Existing DRAM-based systems overlook the persistence property of PM. When issuing RDMA writes to the PM pool, the data are cached in RNIC but not immediately persisted to PM. Hence, the remote persistency [29, 35] is not guaranteed, which possibly causes the remote data to be lost or partially updated once a crash occurs in the PM pool, leading to data inconsistency. Therefore, it is important to ensure the remote persistency in dtxn processing with low network overheads.

Existing studies do not efficiently address these challenges on disaggregated PM. FaSST [46] uses the **Remote Procedure Call (RPC)** to reduce round trips, but RPC requires the CPU in the

PM pool to frequently query, lock, and update data. DrTM+H [81] employs hybrid RDMA verbs to improve performance, but the two-sided RDMA fails to work in the PM pool due to consuming the remote CPU. NAM-DB [91] decouples the compute and storage servers to run dtxns. It adopts snapshot isolation and operation logs without checkpointing to disks. The data are not replicated, thus hurting the availability. After commit, the inputs, descriptions, and timestamps of dtxns are recorded in the operation logs. Once the operation logs fill up the memory, the system cannot serve write requests. Moreover, NAM-DB works on DRAM and disks, which is not designed for PM.

To tackle the preceding challenges, we propose FORD, a *F*ast *O*ne-sided *R*DMA-based *D*istributed transaction system. Unlike prior systems, FORD fully leverages one-sided RDMA to process dtxns for the new disaggregated PM architecture with efficient round trip reductions and PM-conscious designs. Specifically, this article makes the following contributions:

- *Hitchhiked locking* and *coalescent commit to reduce latency*. FORD efficiently attaches the locks with read requests in a hitchhiker manner, to read remote data that belong to the *read-write set* in a single round trip during the dtxn execution phase. Hence, it is unnecessary to consume extra round trips for locking and validating the read-write set after the execution phase (Section 4.2). Furthermore, FORD leverages a coalescent commit scheme to *in-place update all the primaries and backups* in a single round trip to accelerate commit. To ensure that the dtxn can roll back once the replica crashes, FORD writes undo logs in parallel with the dtxn execution. To prevent the updated data from being partially read, FORD temporarily marks the data to be invisible in the commit round trip. After commit, the data are made visible in the background, which consumes at most 0.5 **Round Trip Time (RTT)** (Section 4.3).
- *Backup-enabled read to release the PM bandwidth on the primary replicas*. FORD allows the backups to serve the read requests, thus freeing up the PM bandwidth in the primary to serve other requests. Since the backups are in-place updated by using our coalescent commit scheme, the compute pool can easily read the newest data from the backups after the dtxn commits. By balancing the load on the primaries and backups, FORD eliminates the performance bottleneck on the primary to improve the throughput (Section 4.4).
- *Selective remote flush to guarantee remote persistency with low overheads*. FORD leverages one-sided RDMA flush schemes to persist the written data from a remote RNIC cache to PM for remote persistency. However, flushing each RDMA `WRITE` to each remote replica incurs substantial round trips. To avoid this, FORD selectively issues the flushes only after the final write and to the backups. Since the $(f+1)$-way PBR tolerates at most f replica failures, once the updates are persistently stored in the f backups, the remote persistency is guaranteed. Hence, FORD significantly reduces the remote flush operations (Section 4.5).
- *Localized validation to mitigate RDMA round trips*. FORD utilizes the DRAM caches in the compute pool to maintain the versions of hotspot remote objects. In this way, coordinators validate the RO set in a local manner. By transferring the validation from the remote PM to local DRAM, FORD efficiently reduces the RDMA round trips to improve the transaction performance (Section 4.7).
- *Real implementation and extensive experiments*. We implement FORD[1] using C++ (about 18.6k lines of codes) and compare it with two state-of-the-art RDMA-based dtxn systems: FaRM [31] and DrTM+H [81]. The experimental results demonstrate that FORD significantly improves the transaction throughput by up to 3× and reduces the latency up to 87.4% under three **Online Transaction Processing (OLTP)** benchmarks.

[1]Open source code: https://github.com/minghust/ford.

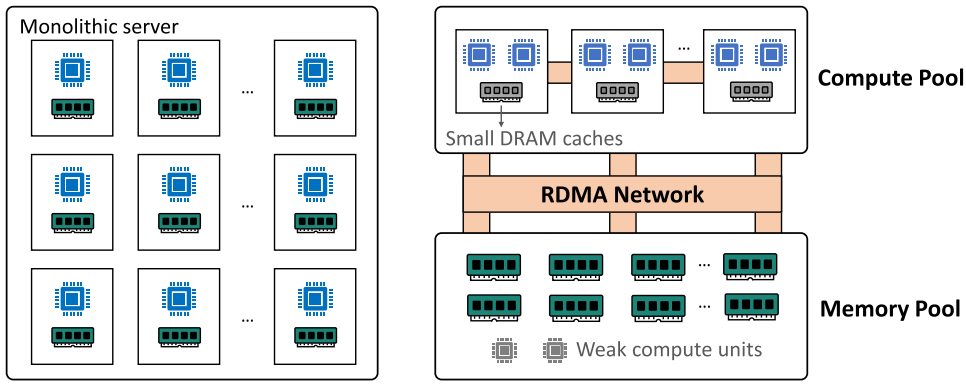

Fig. 1. The comparisons between monolithic architecture, which contains many monolithic servers (a), and disaggregated architecture, which decouples the compute and memory resources into independent and distributed resource pools connected by RDMA (b).

2 BACKGROUND

2.1 Remote Direct Memory Access

RDMA delivers high bandwidth and low latency [30, 45]. RDMA provides two types of *verbs* for accessing remote data. The first type is one-sided verbs, including `READ`, `WRITE`, `CAS` (compare-and-swap), and `FAA` (fetch-and-add), which bypass the remote CPU to directly access the memory. RDMA is well known for its one-sided verbs that do not involve the remote CPU and kernel [81]. The second type is two-sided verbs, including `SEND` and `RECV`, which require the remote CPU to process messages but bypass the traditional network stack [30, 46]. Given these high-performance verbs, RDMA has been widely applied in modern datacenters [31, 65, 67, 76, 81].

To use the two types of verbs, a client uses **Queue Pairs (QPs)** to communicate with a server via the RNIC. A QP contains a **Send Queue (SQ)** and a **Receive Queue (RQ)**:

(1) When using one-sided verbs, the client posts a request to the SQ, and polls the completion event of the request from a completion queue, which is bound with the QP. The server is not aware of the request since one-sided RDMA does not require the involvement of the server-side CPU.
(2) When using two-sided RDMA, the client posts a request to the SQ, and the server uses the CPU to poll the request from the RQ. According to the request types, the server-side CPU runs the pre-defined RPC functions to process the requests, and posts corresponding responses to the SQ. These responses are sent back to the client's RQ.

To transfer data through QPs, three transport modes are available to programmers: **Reliable Connection (RC)**, unreliable connection (UC), and unreliable datagram (UD). Among them, only RC supports all one-sided verbs [72], since RC guarantees that the packets are reliably delivered in order without any loss. We hence use RC for one-sided RDMA in this article.

2.2 Disaggregated PM

Traditional datacenters are built on the monolithic architecture, which contains a set of monolithic servers, as shown in Figure 1(a). Each server hosts a small amount of compute units and memory modules. However, such an architecture suffers from low resource utilization, poor elasticity, and coarse failure domain [78].

To address the preceding drawbacks, modern datacenters leverage *memory disaggregation* to decouple the compute and memory resources from monolithic servers to independent and RDMA-connected resource pools, as shown in Figure 1(b). The compute pool contains substantial compute units (e.g., CPU cores) to execute computation tasks with small DRAM caches. The memory pool consists of many memory units (e.g., DRAM DIMMs) to store the application data, and contain a small number of weak compute units only for memory allocations and network interconnections [78, 96]. The compute units in the memory pool are too weak to be involved when executing computation tasks. Hence, the compute pool generally leverages one-sided RDMA to access data in the memory pool to bypass the remote CPUs. Since each compute and memory pool is flexibly deployed and scaled, the disaggregated memory architecture becomes the important and new trend in modern datacenters due to significantly improving the resource utilization and elasticity, and narrowing the failure domain [94, 96], as analyzed in the following:

- *Improving the resource utilization.* As reported by Google [74], the servers in datacenters have only about 60% memory utilization on average. For example, if needing more compute power, we have to add more servers in which the memory modules are wasted. To address this issue, memory disaggregation decouples the memory resources from the monolithic servers to build a large memory pool, which enables the applications to use the memory capacity from the memory pool in an on-demand manner to avoid wastes, thus improving the resource utilization.
- *Improving the elasticity.* A monolithic server hosts a fixed number of CPUs and memory modules, which makes it hard to dynamically meet the compute and memory requirements of different types of applications (e.g., memory- or compute-intensive workloads), leading to poor elasticity when offering cloud services to multiple tenants. To address this issue, the disaggregated memory architecture separates the compute and memory into scalable resource pools. Therefore, the datacenter is able to dynamically provide proper amounts of CPUs and memory modules to satisfy the requirements for different types of applications, thus improving the elasticity.
- *Narrowing the failure domain.* In a monolithic server, if the CPU is broken, the whole server is unusable, which enlarges the failure domain. To address this issue, the disaggregated memory architecture manages the CPU and memory in independent pools. Therefore, the damaged CPUs in the compute pool do not affect the normal uses of the decoupled memory modules in the memory pool. In this way, the failure domain is significantly reduced to improve the hardware availability.

Nevertheless, the memory pool does not guarantee data persistence when using DRAM as the memory unit. Plugging UPS [31, 82] adds "non-volatility" on DRAM, which however increases the costs and energy consumptions. If a power failure occurs, the data in DRAM are flushed to disks using UPS, which incurs I/O overheads. Moreover, it is difficult to increase the capacity of one DRAM DIMM due to its limited scalability [73], causing high costs to build a large memory pool.

PM addresses the preceding issues by providing persistence, high density (e.g., 512 GB/DIMM [21]), and low costs (e.g., 39.2% $/GB of DRAM [4]), while exhibiting DRAM-like latency [89]. As memory disaggregation meets the needs of datacenters, disaggregating PM also enjoys the same benefits [75]. Hence, we leverage PM as memory units to build the disaggregated PM, which forms a persistent and cost-effective memory pool.

2.3 RDMA-Based Dtxns

Due to the benefits of bypassing the remote CPU and traditional TCP/IP stack, recent studies leverage RDMA to run dtxns [31, 46, 65, 81, 82, 94]. Specifically, a coordinator is leveraged to read remote data, run dtxn logic, and commit the updated data back to remote machines. The concurrency

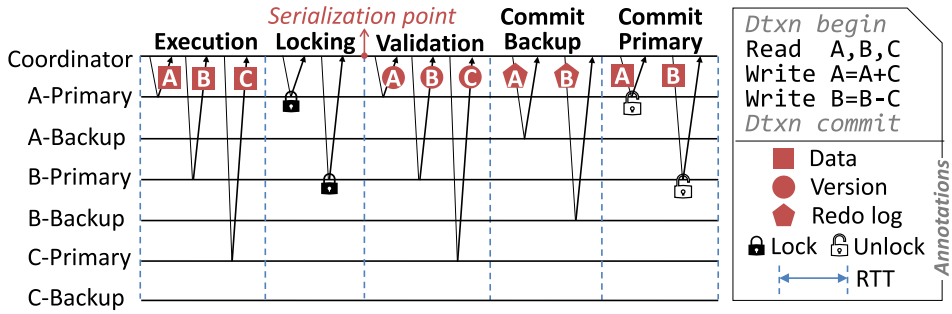

Fig. 2. Using OCC and PBR to process dtxns.

control schemes, such as 2PL (two-phase locking) [6] and OCC [49], are used to serialize dtxns. 2PL acquires locks for all data before execution and releases all locks after commit. OCC does not lock data during execution but acquires (or releases) locks for all the written data before (or after) commit. Many systems adopt OCC due to not locking the RO data, which benefits RO dtxns. Moreover, the PBR [50] is incorporated in dtxn processing for high availability [31, 81, 93]. The $(f + 1)$-way PBR contains one primary and f backups for each data shard, and tolerates at most f replica failures. We assume that the fail-stop failures [40] can occur in arbitrary replicas at any time. The failed replica can be quickly detected and recovered by using RDMA [31]. Like FaRM [30, 31, 65], DrTM [14, 81, 82], and FaSST [46], we currently do not consider the Byzantine failures [41].

Our paper focuses on the efficient use of OCC and PBR. Figure 2 presents how existing RDMA systems [31, 81] process dtxns over OCC and PBR. Without loss of generality, we use a two-way replication as an example. In general, there are five phases. The first phase is execution. A coordinator reads the required data (i.e., read set = {A, B, C}) from primaries and locally executes a dtxn. The updated data (i.e., write set = {A, B}) are buffered in a local cache. The second phase is locking. After execution, the coordinator locks the write set in primaries to serialize dtxns. If the locking fails, the coordinator aborts the dtxn. The third phase is validation. If the locking succeeds, the coordinator reads the data versions from primaries to validate that the versions of read and write sets are unchanged. If the validation fails, the coordinator aborts the dtxn. The fourth phase is commit backup. If the validation succeeds, the coordinator sends the redo logs to remote backups. The fifth phase is commit primary. After receiving all the **Acknowledgments (ACKs)** from backups, the coordinator updates and unlocks primaries to commit the dtxn.

3 DTXNS ON DISAGGREGATED PM

3.1 System Model

In the disaggregated PM architecture, PM is used as remote memory with persistence to durably store the application data, which is replicated into one primary and f backup replicas. In this configuration, the $(f + 1)$-way replication tolerates at most f replica failures (e.g., hardware faults) to provide availability, and the persistence of PM supports to restore the application data from a power outage or system crash in any replica. The PM pool contains a small number of weak compute units only for memory allocations and RDMA connections during the *initialization* [78, 96]. Afterward, these compute units are not used during the *execution* since they are too weak to frequently and efficiently handle substantial tasks. The compute pool uses RDMA to access the data stored in remote PMs at the byte granularity (no page swap), while there is no PM in the compute pool. To ensure the atomicity and strong consistency for data accesses, the compute pool uses coordinators to run transactions that read/write data across remote PMs. All transactions

are hence distributed, and the replication is accessed by multiple coordinators, which use RDMA to commit each dtxn. Although two-sided RDMA-based RPC reduces the network round trips by consuming remote CPUs to handle multiple operations in one round trip [46], the PM pool does not contain CPUs to process requests during execution, resulting in RPC-based schemes not efficiently working. Hence, the coordinators need to use one-sided RDMA to bypass remote CPUs.

3.2 Deficiencies of Existing Systems

Legacy RDMA-based dtxn systems become inefficient on disaggregated PM since they are not designed for memory disaggregation and real PM. Directly using one-sided RDMA will incur extensive round trips that decrease the performance. The reasons are presented next.

First, as shown in Figure 2, due to not using locks in the execution phase, the intersected data between read and write sets (i.e., read-write set = {A, B}) are operated in the execution, locking, and validation phases, which consume three RTTs before updating the replicas. In general, the read-write set is equal to the write set, since the data need to be read before being written back [46] to confirm the existence of the old data or empty slots in the remote memory pool. This can be interpreted as all three write scenarios. The first is *update*, in which the coordinator reads the old data from remote replicas and writes the new data back.The second is *insert*, in which the coordinator reads the remote slots to select an empty one, then writes the newly inserted data back to the remote empty slot. The third is *delete*, in which the coordinator reads the old data from remote replicas, marks them to be invalid, and finally writes the invalid data back. In consequence, the write set is always a subset of the read set in the context of dtxn processing on the disaggregated memory. As a result, the round trips for reading, locking, and validating the read-write data widely exist, causing extra latency. Moreover, if the locking (or validation) fails, the dtxn aborts, which wastes the execution (or execution+locking) phases. As a result, the coordinator consumes useless round trips before processing the next dtxn, thus decreasing the throughput. DrTM+H [81] merges the locking and validation phases but still consumes an RTT to validate the read-write set.

Second, Figure 2 shows that existing systems [31, 46, 65, 81] consume two RTTs to first write backups (i.e., redo logs) and then write primaries (i.e., in-place updates) for high availability. By doing so, the dtxn is ensured to commit after receiving all ACKs from backups, since even if the primary fails, the new data can be recovered from redo logs in the backup. In the monolithic architecture, the coordinator can colocate with a primary or backup in the same server. Hence, the coordinator commits the new data to the local storage, which saves an RTT. However, in the disaggregated architecture, the compute pool does not store any replica. As a result, the coordinator inevitably spends two RTTs to commit each read-write dtxn, which incurs high latency.

Moreover, prior systems work on the DRAM+SSD environment. FaRM [31] and DrTM [82] regard the battery-backed DRAM as PM, but the bandwidth and persistence properties of real PM are different from DRAM, causing inefficiency on the disaggregated PM, for two reasons.

First, prior systems [14, 31, 65, 81] do not allow backups to serve read requests, since in backups the redo logs are asynchronously migrated to the in-place locations after updating the primary. Hence, only the primary can serve the latest data after commit [65]. As a result, all requests from coordinators are sent to the primary, causing a high load on the primary's PM. However, PM shows lower write bandwidth than the new generations of RNICs, as mentioned before. We use 128-GB Optane PM DIMMs and ConnectX-5 RNIC with 100-Gbps InfiniBand to evaluate the throughput of RDMA writes when mixing different frequencies of RDMA reads. As shown in Figure 3, when using 32 threads to concurrently issue read requests, the write throughput decreases by up to 87.5%. Hence, only using the primary to serve all requests makes the PM bandwidth become a performance bottleneck.

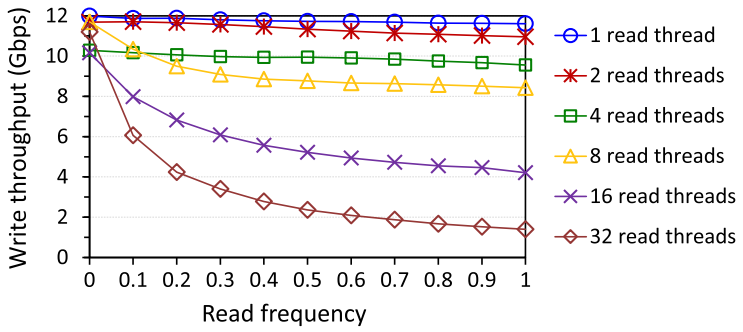

Fig. 3. The throughput of RDMA writes to the remote PM when mixing different frequencies of RDMA reads in RDMA writes. For example, 0.5 means that five reads are mixed with every 10 writes.

Second, there is a lack of remote persistency guarantee. Current RDMA verbs have no persistency semantic [29]. For RDMA writes, the data are first buffered in a volatile cache in the remote RNIC, which ACKs the writes once validated [80]. Hence, even if the client receives all ACKs, some data may not be persisted to the remote PM in case of a crash. This misleads the client into considering that the data are durably stored in the remote PM. Hence, it is important to guarantee the remote persistency for RDMA writes, which is however not considered in prior dtxn systems that use DRAM.

In summary, state-of-the-art dtxn systems become inefficient on the disaggregated PM due to causing substantial round trips and overlooking the PM properties. This motivates our article to propose FORD, an efficient one-sided RDMA-based dtxn processing system for the new disaggregated PM architecture.

4 THE FORD DESIGN

4.1 Overview

Figure 4 shows the system overview of FORD. The compute pool runs coordinators that process dtxns and access application data in the PM pool. The compute and PM pools communicate using **Connection Managers (CMs)**, which maintain the RDMA QP connections.

FORD's workflow contains two stages. The first stage is the `Init` stage. ❶ The clients use the weak compute units in the PM pool (by RPCs) to allocate and register memory for subsequent RDMA operations [78, 96], then load **Database (DB)** tables. The DB tables are organized by indexes (Section 5.1). ❷ The compute and PM pools build RDMA connections using CMs. To calculate the remote addresses for one-sided RDMA in the compute pool, the CM in the PM pool sends the metadata of all the indexes to the compute pool. These metadata only consume several megabytes and are buffered in the compute pool (Section 5.1). Moreover, the PM pool notifies the compute pool about the location and role (i.e., primary or backup) of each DB table so that the coordinators can correctly access remote data during processing. The second stage is the Run stage. ❸ The clients issue substantial dtxns to the compute pool, which spawns threads as coordinators to leverage our runtime library for fast dtxn processing. This library contains our novel designs in Sections 4.2 through 4.7 and exposes easy-to-use interfaces (Section 5.2). ❹ Each coordinator uses one-sided RDMA to process dtxns, which are serialized by locking and version validations. Hence, there is no consistency requirement among compute units. ❺ After processing, the coordinators report "Tx_commited" or "Tx_aborted" to clients.

It is worth noting that the `Init` stage performs once before the Run stage, and the weak compute units in the PM pool are *not* involved in the Run stage.

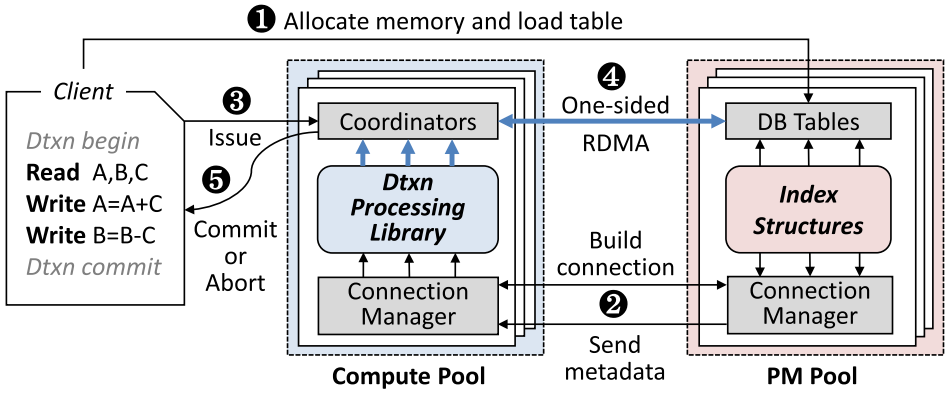

Fig. 4. The system overview of FORD.

4.2 Hitchhiked Locking

As analyzed in Section 3.2 and shown in Figure 5(a), prior works consume three RTTs to separately read, lock, and validate data to process a general read-write dtxn shown in Figure 4.

To reduce the heavy round trips, FORD proposes a *hitchhiked locking* scheme to lock the data that belong to the *read-write set* when reading them in the execution phase. The read and write sets are known according to the transaction logic. FORD sends the lock request together with the read request in a hitchhiker manner. In this way, the read-write data do not need to be locked and validated after execution, since other transactions cannot modify the locked data. Therefore, the total round trips of processing a dtxn are efficiently reduced.

Due to not involving the CPUs in the PM pool, it is hard to lock and read data using one-sided RDMA in one round trip. To address this issue, FORD adopts the doorbell mechanism [45] to batch the RDMA `CAS` followed by an RDMA `READ` in one request, which is delivered and ACKed in one round trip, instead of being separately issued in two round trips, as depicted in Figure 5(b). The RDMA `CAS` first tries to lock the remote data, and the RDMA `READ` further fetches the data. By using the RC transport mode for RDMA QPs, the two RDMA operations are reliably delivered to the remote RNIC in order [72]. The batched operations are then executed by RNIC as the delivering order to ensure the correctness. After receiving the ACK of the batched request, the coordinator checks whether the locking is successful by comparing the return value of RDMA `CAS` with the previously sent lock value (i.e., only equality means a successful locking). If the locking fails, the coordinator aborts the dtxn and unlocks the previously locked data to avoid deadlocks. Figure 5(c) shows our hitchhiked locking scheme, which locks and reads the read-write data (e.g., {`A`, `B`}) using one-sided RDMA in one RTT, thus reducing the latency.

Our hitchhiked locking is different from 2PL, which locks all the data before execution. FORD still maintains the optimistic feature of OCC to avoid contentions for the data that are only read. Specifically, the RO data (e.g., data `C` in Figure 5(c)) are not locked in the execution phase, and the locked read-write data can still be read by other coordinators (but cannot be locked). Moreover, if a read-write dtxn contains any RO data, there is a validation phase to guarantee that the versions of these RO data are not changed. The validation is generally performed by reading the data versions from the remote replicas and checking whether the versions are modified. In Section 4.7, we present that FORD further accelerates this procedure by storing the data versions in the compute pool. In this way, the validation is completed by reading a local version cache, which efficiently mitigates the network round trips.

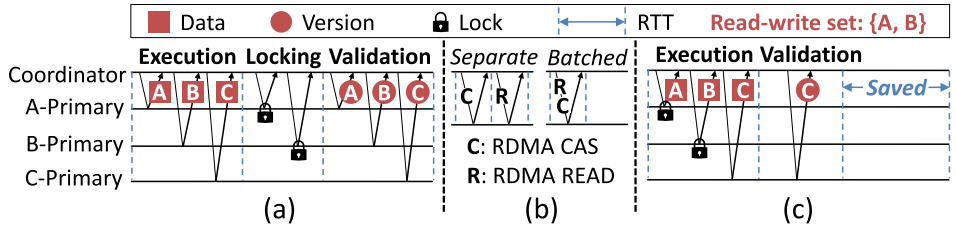

Fig. 5. (a) Legacy schemes serially read, lock, and validate the read-write set. (b) Adopting doorbell batching reduces the number of RDMA round trips. (c) Our hitchhiked locking eliminates extra RDMA round trips for locking and validating the read-write set.

Enabling hitchhiked locking requires remote data addresses for one-sided RDMA. FORD leverages a hash indexing scheme for the coordinator to compute the remote address of a bucket and read it (Section 5.1). Due to hash collisions, it is hard to accurately lock the slot in a remote bucket at the first read. However, directly locking the entire bucket prevents other coordinators from locking different slots in the same bucket, causing unnecessary contentions. Hence, the hitchhiked locking is disabled when the data are read at the first time. After reading, the coordinator obtains the remote data addresses and buffers them in its local address cache. Each time the previous data are read again, the local address cache provides the addresses to enable hitchhiked locking. If some remote data addresses in the PM pool are changed by a coordinator (e.g., some data are deleted and then inserted to different slots), another coordinator can easily discover that its buffered addresses become stale, since the key of the fetched data mismatches the queried key. In this case, the coordinator re-reads the bucket to obtain the correct data and updates its buffered addresses.

Our hitchhiked locking is different from (1) FaRM [30, 31, 65] and DrTM+H [81], which consume a dedicated RTT to lock data; (2) DrTM+R [14], which exclusively locks all the data in the read and write sets; and (3) FaSST [46], which uses RPC to lock data, which fails to work on the disaggregated memory. Unlike FaSST, FORD leverages one-sided RDMA to read and lock data in one round trip. Hitchhiked locking does not lengthen the lock duration due to eliminating the locking and validation phases for the read-write data. In the preceding systems that support OCC and PBR, we summarize the *lock duration* as follows:

- *FaRM* [30, 31, 65]: Four phases = lock + validate + commit backup + commit primary&unlock.
- *FaSST* [46]: Five phases = lock + validate + log + commit backup + commit primary&unlock.
- *DrTM+R* [14]: Four phases = lock + validate + update + unlock.
- *DrTM+H* [81]: Three phases = lock&validate + commit backup + commit primary&unlock.
- *Our FORD*: Case (1): Three phases without RO data = read&lock read-write set + commit all replicas (Section 4.3) + background unlock (Section 4.3); Case (2): Four phases with RO data = read&lock read-write set + validate RO set + commit all replicas + background unlock.

Although our lock duration experiences four phases with RO data, the coordinator can immediately detect lock conflicts in the execution phase and run the next dtxn as early as possible. Hence, FORD avoids the aforementioned wastes of the execution (or execution+locking) phases due to the lock (or validation) failures in prior systems [14, 31, 81]. This trade-off is beneficial for improving the transaction throughput.

4.3 Coalescent Commit

As analyzed in Section 3.2 and shown in Figure 6(a), existing dtxn systems spend two RTTs to separately write redo logs to the backup and then update the primary to finish commit. Hence,

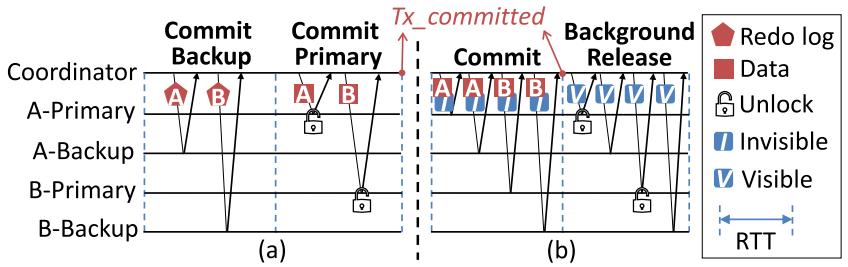

Fig. 6. The comparisons between separate commit (a) and coalescent commit (b).

if the primary crashes, the dtxn can roll forward by using the redo logs in backups. However, such separate commit design incurs high network overheads on the disaggregated PM, since each read-write dtxn needs two RTTs to replicate the updated data.

To reduce latency, FORD proposes a *coalescent commit* scheme to update the primaries and backups together in only one RTT. The coordinator commits the dtxn if the ACKs from all replicas are received. Otherwise, the dtxn aborts and rolls back. In fact, there is a trade-off between the replica commit latency and recovery state—that is, two RTTs + roll forward vs. one RTT + roll back. In practice, the commit latency is more important for the disaggregated PM, since we need to decrease the number of round trips to accelerate dtxn processing in the common case, in which no ACK is lost. Hence, we choose to commit all replicas together to improve the performance and support to roll back dtxns in case of failures.

In the disaggregated PM architecture, we need to consider how to update replicas when using coalescent commit. For primaries, it is efficient to in-place update data, since the coordinators can directly read and lock the remote data without address redirections. But for backups, it is inefficient to send redo logs like FaRM [31, 65] and DrTM+H [81]. Because the CPUs in the PM pool are not involved in processing dtxns, the new data in redo logs will not be installed after commit. As a result, the backup cannot work after the memory is filled up by logs. Hence, we choose to in-place update the backup. In general, it is challenging to in-place update the backups and primaries in one round trip, because in case of a crash, the old data in remote replicas can be partially overwritten, which prevents the dtxn from being rolled back. To tackle this challenge, FORD sends *undo logs* to all the replicas before in-place updates. Hence, the dtxns can roll back using the old data in undo logs. Unlike redo logs, the undo logs are simply discarded by setting the log status to be "committed" after the dtxn commits, which is completed by coordinators in the background. Hence, undo logging meets the requirement of PM pool—that is, not involving the remote CPU to move data.

4.3.1 Parallel Undo Logging. The next question is how to send undo logs to remote replicas. One solution is to spend a dedicated round trip to send logs, which however causes extra RTTs. We observe that undo logs can be immediately generated once the old data of the read-write set have been read in the execution phase. Based on this observation, we design a *parallel undo logging* scheme to generate and send undo logs in parallel with the transaction logic execution. Therefore, it is unnecessary to consume an extra round trip to send undo logs. To ensure atomicity, the coordinator only needs to check that all the ACKs of log writes (i.e., RDMA `WRITE`) are returned before updating the replicas. Note that the redo logs cannot be sent in the execution phase, because we have to wait for completing the transaction execution to obtain the newest data to generate redo logs, which heavily weakens the parallelism especially in the transaction that goes through a long-time execution logic, such as the `New Order` transaction in the `TPCC` benchmark [27].

In general, sending undo logs to all replicas consumes the network bandwidth, but the consumption is not significant. Specifically, the OLTP workloads typically involve inserting, updating, and deleting *small* amounts of data, as pointed out by Oracle [61]. We observe that the record size in a DB table is basically small in OLTP workloads—for example, up to 16B, 48B, and 672B in the `SmallBank`, `TATP`, and `TPCC` benchmarks, respectively, as introduced in Section 6.1. Moreover, in our experiments, a read-write dtxn averagely updates 3, 2, and 13 records in the `SmallBank`, `TATP`, and `TPCC` benchmarks, respectively. The updated record generally contains a small number of modified fields. Therefore, the size of undo logs in a read-write dtxn is small. However, the network bandwidth of InfiniBand is rapidly increasing at twice the speed—for example, 100 Gbps/port in ConnectX-5 [24], 200 Gbps/port in ConnectX-6 [26], and 400 Gbps/port in ConnectX-7 [25]. Therefore, sending small-sized undo logs will not significantly consume the bandwidth.

4.3.2 Visibility Control. To ensure consistency, our coalescent commit protocol guarantees that the data that being updated in the replicas are not partially read by other dtxns. Since our hitchhiked locking scheme does not block RO requests, a coordinator possibly reads some remote data that are being updated, causing inconsistency. To avoid this, FORD proposes a one-sided RDMA-based *visibility control* to decide whether the data are visible to coordinators, as shown in Figure 6(b). The idea is to batch an invisible request followed by an RDMA `WRITE` into one request to update the remote replicas. First, the invisible request prevents other coordinators from reading data by setting the invisible flag to 1. FORD implements the 1-bit invisible flag and 63-bit lock value in an 8B value, called `VLock`, which is atomically modified via an RDMA `CAS`. Second, after setting the invisible flag, the RDMA `WRITE` in-place updates the remote replica. After receiving *all* ACKs, the coordinator reports "Tx_committed" to clients. Otherwise, for example, if a replica fails, the coordinator rolls back the dtxn by using undo logs. After commit or rollback, the coordinator unlocks data and makes them visible by writing an 8B zero to `VLock` in a background *release* phase.

The release phase does not exist on the critical path of the dtxn commit. It incurs only 0.5 RTT or can be fully overlapped. First, once the remote RNIC receives the RDMA `CAS` request and clears the `VLock`, other coordinators can immediately access the remote data. It is unnecessary to wait for returning the ACK, thus only consuming 0.5 RTT to make data visible. If some data are currently invisible, a coordinator can re-read them until visible. After all the required data become visible, the coordinator continues to process dtxns to guarantee the atomic visibility. Second, if there is no coordinator currently reading the invisible data, the background release phase is completely overlapped with other in-flight dtxns, thus avoiding re-read operations.

4.4 Backup-Enabled Read

As discussed in Section 3.2, only leveraging the primary to handle all the requests decreases the throughput due to the limited write bandwidth of PM. To tackle this challenge, FORD *enables the backups to serve read requests* for the RO data—that is, the coordinators are allowed to read the RO data and their versions from backups, as shown in Figure 7(a). This frees up the PM in the primary to serve other requests (e.g., lock and write), thus balancing the load to improve throughput. Based on our coalescent commit that *in-place* updates all replicas, it is easy for a coordinator to read the RO data from backups due to no address redirection.

FORD guarantees the correctness of reading the RO data from backups under different contention cases, as demonstrated in Figure 7(b). If a dtxn (e.g., `dtxn1`) reads all its RO data before (or after) another dtxn (e.g., `dtxn2`) commits the replicas, `dtxn1` will obtain the old (or new) data, which guarantees the correctness since `dtxn2` is uncommitted (or committed). However, if `dtxn1` reads multiple RO data and goes through `dtxn2`'s execution and commit phases, the data that `dtxn1` has read are possibly stale after `dtxn2` updates the replicas. To address this issue, FORD

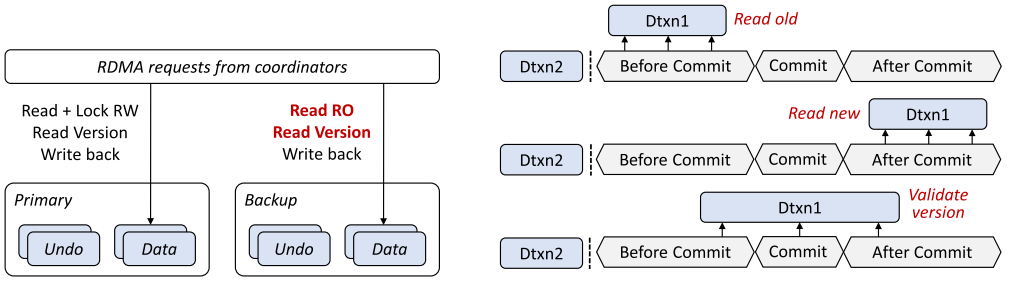

Fig. 7. Enabling the backup replicas to serve read requests.

validates the versions of all dtxn1's RO data before dtxn1 commits, as guaranteed by our hitchhiked locking scheme in Figure 5(c). If the validation fails, the coordinator aborts dtxn1 to guarantee the correctness.

Existing systems unfortunately fail to efficiently read data from backups. First, for legacy DB systems, such as Microsoft Azure [28] and Amazon Aurora [77], the primary (or backup) replica handles the write (or read) requests from clients. After a client writes data to the primary, the backup needs to wait for receiving and installing the new data that are sent from the primary. Hence, after updating the primary, the clients are delayed to read the latest data from backups, thus causing extra latency. However, in the disaggregated PM, the CPUs in the primaries and backups are not involved in dtxn processing. Hence, the data send/receive operations between replicas fail to work in the PM pool. Second, for prior RDMA-based dtxn systems [31, 65, 81], the coordinator writes the updated data to the backups (i.e., redo logs) and primaries (i.e., in-place updates) to commit a dtxn. However, other coordinators cannot read the backups after the dtxn commits, since the latest data in backups have not been transmitted from the redo logs to the in-place locations. Moreover, in the disaggregated PM, the backups fail to extract the updated data in redo logs and transmit these updates due to involving the CPU in the PM pool during dtxn processing. Unlike the preceding systems, our coalescent commit protocol in-place updates the backups and primaries together without involving the CPU in the PM pool. Hence, the coordinators are allowed to read the latest in-place data from backups after the dtxn commits, which alleviates the load on the primary's PM to improve the throughput.

4.5 Selective Remote Flush

It is important to guarantee the remote persistency when committing the transaction updates to the PM pool, which is however not considered in prior dtxn systems that use DRAM as the memory. Recently, the one-sided RDMA FLUSH [35, 69] was proposed to persist data from the remote RNIC to PM. However, flushing each data to each remote replica (i.e., full flush) consumes many round trips. As shown in Figure 8(a), updating two data requires eight round trips when using remote flushes.

To guarantee remote persistency with low network overhead, we propose a *selective remote flush* scheme, as shown in Figure 8(b). The idea is to issue an RDMA FLUSH after the final RDMA WRITE and only to backups, for two reasons. First, RDMA FLUSH supports to flush all the previous written data. Hence, it is sufficient to use one RDMA FLUSH after the final write to one replica. Second, in the $(f+1)$-way PBR, once the data are persisted in all backups, even if the primary crashes, we can recover the primary by using backups. Hence, it is sufficient to issue RDMA FLUSH to only backups. Note that if all the $(f+1)$ replicas fail, the data cannot be recovered [31]. FORD guarantees the

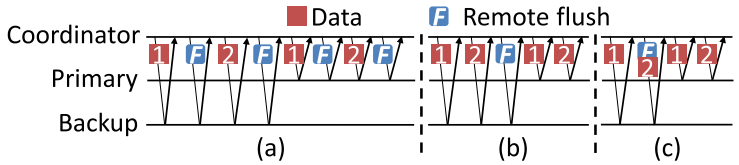

Fig. 8. Ensuring remote persistency using full flush (a), selective flush (b), and selective flush with request batching (c).

remote persistency under at most f replica failures. Thus, by issuing necessary flush operations, FORD significantly reduces the round trips.

As RDMA FLUSH is currently unavailable in programming due to the needs of modifying RNIC and PCIe [69], we leverage one-sided RDMA READ-after-WRITE to flush the data in RNIC to memory like existing studies [44, 48]. Specifically, the RDMA READ fetches any size (e.g., 1B) of the data that are written by RDMA WRITE. Then, the remote RNIC will issue all PCIe writes before issuing PCIe reads to satisfy the RDMA READ. In this way, the data in RNIC are written to PM. We further optimize this procedure by batching the write and read into one request to eliminate the extra read round trip. This implementation is compatible with the future one-sided RDMA FLUSH—that is, replacing RDMA READ with RDMA FLUSH, as shown in Figure 8(c). In essence, our selective remote flush scheme aims to reduce the round trips when ensuring remote data persistency. Hence, this scheme is not affected by the specific implementation of remote data flushing—for example, using the future RDMA FLUSH primitive or current READ-after-WRITE method.

In datacenters, UPS [31] offers a standby battery that supplies power to the machine to provide enough time to properly power down the machine in case of an interruption in the utility power. However, it is worth noting that merely relying on UPS to enable the weak CPU in the memory pool to flush the data in RNIC to PM is not sufficient to guarantee the remote persistency. The reason is that UPS can only handle power interruptions but fails to support the machine to keep running upon kernel panics and CPU failures. In case of kernel panics and CPU failures, the data in the RNIC cache are lost even if the machine is backed by UPS, since there is no way for the machine to flush the cached data from RNIC to PM. As a result, the client cannot know whether the RDMA-written data are actually persisted into the remote PM. Our selective remote flush scheme efficiently addresses this issue by explicitly flushing the data from the remote RNIC to PM, and returning the ACK to the coordinator. Such remote flush operation fully bypasses the remote kernel and CPU by using one-sided RDMA verbs. Therefore, even though the kernel panics and CPU failures occur, the coordinator can accurately distinguish whether the RDMA-written data are persisted into the remote PM, thus guaranteeing the remote persistency.

4.6 FORD Transactions

Figure 9 illustrates how our designs (Sections 4.2–4.5) work together to process dtxns by using one-sided RDMA verbs. The requests in one RTT are issued and ACKed in parallel.

(1) Execution. A coordinator reads and locks the required read-write data from primaries using batched RDMA CAS+READ in one round trip. The RO data can be fetched from backups or primaries using RDMA READ. The undo logs are immediately generated and written to remote replicas by RDMA WRITE in parallel with the execution. The concurrent dtxns that have conflicting accesses to the same remote data are serialized by locks. If any lock operation fails, the coordinator aborts the dtxn and unlocks the remote data.

(2) Validation. After execution, the coordinator reads the versions of the RO data (if any) using RDMA READ and verifies that the data versions are not modified by other coordinators. If a version

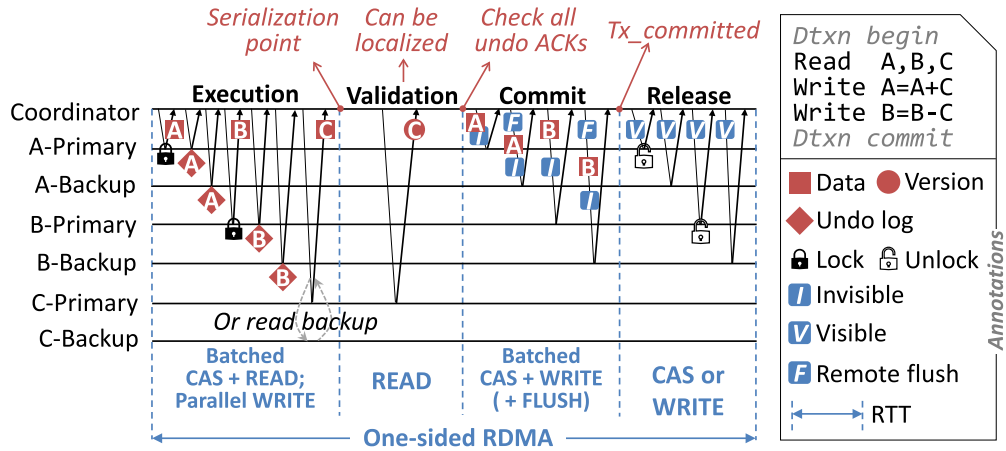

Fig. 9. Dtxn processing in FORD.

changes, the coordinator aborts the dtxn and unlocks the remote data. As discussed in Section 4.7, FORD further localizes the validation operations by enabling the coordinators to check the data versions using a local DRAM, which efficiently eliminates the network round trips of validations.

(3) Commit. After validation, the coordinator checks that all the ACKs of undo logs are received, then commits the updated data to all the replicas in one round trip. The data in primaries are marked to be invisible and updated using batched RDMA `CAS+WRITE`. The data in backups are updated and further flushed from RNIC to PM using remote data flushing operations. Therefore, the coordinator uses batched RDMA `CAS+WRITE+FLUSH` to update backups. After receiving all the ACKs from replicas, the coordinator reports "Tx_committed" to the client. Afterward, the coordinator starts processing the next dtxn.

(4) Release. After the dtxn commits, the coordinator releases the remote data by setting them visible and unlocking them in the background—that is, writing 0 to the 8B `VLock` field by using RDMA `CAS` or RDMA `WRITE` [81].

FORD efficiently handles different types of dtxns. First, for RO dtxns, FORD reads remote data and validates the versions before commit. Prior systems [31, 81] adopt similar operations. However, the difference is that FORD supports coordinators to read the latest data from backups after the transaction commits to improve the throughput, whereas prior systems do not support this. Second, for read-write dtxns, only two RTTs (i.e., Execution+Commit, without RO data) or three RTTs (i.e., Execution+Validation+Commit, with RO data) are on the critical path. Compared with existing designs that require five RTTs [31, 46] or four RTTs [81] to process a read-write dtxn (as analyzed in Section 3.2), FORD significantly improves the performance for the disaggregated PM architecture.

4.7 Localized Validation

Due to not locking the RO data, FORD consumes an RTT to validate whether the RO set is modified by other transactions to guarantee serializability. However, we observe that if the read set of a transaction is not modified by any other transactions, validating the data versions by using RDMA `READ` becomes wasteful. The root cause of this phenomenon is that all the versions are stored together with their data in remote replicas. Hence, the coordinators have to always perform validations over the network, which incurs high latency in the disaggregated architecture.

To reduce the round trips for validations, FORD leverages a *localized validation* scheme to decouple the storing locations of data and their versions. Specifically, this scheme utilizes the local

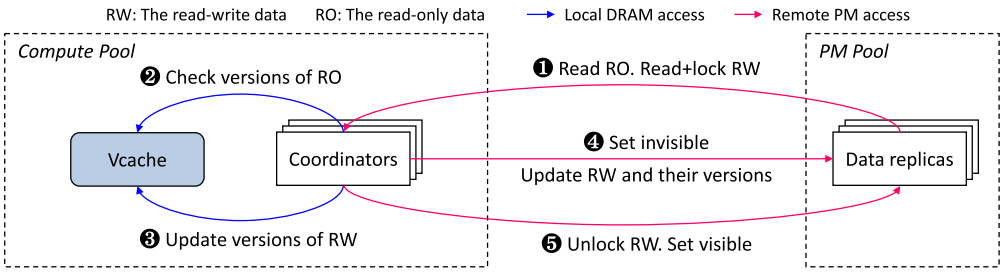

Fig. 10. Localizing the validation operations to reduce RDMA round trips.

DRAM in the compute pool to maintain a Vcache to store the versions of hot records. In this way, FORD transfers the validation operations *from remote to local*. The coordinator easily checks whether the RO set is changed by simply looking up the versions in its local Vcache, which is faster than the original RDMA READ design that accesses the remote memory pool through the network. As a version is only 8B, the Vcache does not incur a high space overhead.

Figure 10 shows how FORD processes a read-write dtxn after localizing the validation operations. Before commit, the coordinator checks whether the versions of the RO data are modified in Vcache (❷). After the validation succeeds, the coordinator updates the versions of the read-write data in Vcache (❸), then writes the new data to remote replicas via our coalescent commit protocol described in Section 4.3. Therefore, if a dtxn (e.g., dtxn1) updates the versions of some data, which are RO in another dtxn (e.g., dtxn2), dtxn2 can easily know that its data versions become stale without going through the network. By doing this, FORD avoids the RDMA round trips for checking the versions, thus improving the performance, which is quantitatively demonstrated in Section 6.3.2. However, if a lookup miss occurs in Vcache, the coordinator follows the original design in FORD to validate the data versions via RDMA. After fetching the versions back, the coordinator inserts the versions into Vcache to accelerate subsequent dtxns.

It is worth noting that FORD does not localize the locking operations. The following factors drive this design choice. First, our hitchhiked locking scheme batches the read and lock requests into one round trip. Hence, FORD does not waste additional round trips to lock the remote data. Second, if caching the lock status in compute pool, the lock cache needs to handle concurrent locking operations (i.e., local CPU CAS) and lookups in an efficient way. However, the lock cache is small due to limited DRAM in the compute pool. As a result, the collision rate in a small hash table is high, and substantial CPU CAS operations need to frequently retry to occupy the proper empty slots to insert the <key, lock_status> pairs. It causes the latency to be an order of magnitude higher than the µs latency of remote locking. To verify this, we implement a concurrent hash table for local locking and observe a throughput reduction of up to 13.6× on TPCC. Therefore, FORD does not allow the locking operations to be performed in the compute pool.

In the compute pool, if a coordinator needs to access the Vcaches at other compute nodes to obtain the data versions, it causes in-pool communications and additional overheads of synchronizing the Vcaches in different compute nodes. There are three solutions to tackle this issue, as shown next:

- *Request partition*: This solution partitions the dtxn requests to specific compute nodes according to the DB tables that the transactions access. In this way, the Vcaches in different compute nodes are not overlapped to eliminate the in-pool communications. Nevertheless, when the accessed tables of two different dtxns have intersections, storing the versions in one compute node would still incur the synchronization overheads. To handle these dtxn

requests, FORD allows the coordinators to adopt the original design based on RDMA READ to validate the data versions in remote memory pool.

- *In-compute-node global Vcache*: Apart from partitioning requests, FORD can leverage a dedicated compute node in the compute pool to maintain a global Vcache. In a datacenter, since the compute and memory pools are physically separated, the in-pool communication between nodes within the same pool is generally faster than the cross-pool communication between two nodes of different pools. For example, the nodes in the same pool locate within the same rack, but different pools are distributed in different racks. Therefore, even if the coordinators have to access the Vcache from another compute node, it is still faster than accessing the remote memory pool due to avoiding the cross-rack network transmissions. Moreover, using a global Vcache efficiently avoids the overheads of synchronizing the Vcaches in different compute nodes.
- *In-switch global Vcache*: Apart from storing the Vcache in a compute node, FORD can also store the global Vcache in a switch (e.g., a programmable datacenter switch [52]) near the compute pool. In this way, FORD avoids the additional network hops to the remote memory pool, thus decreasing the latency. A commodity programmable datacenter switch contains 8 GB of RAM [17], which is sufficient to store the Vcache (i.e., 1 GB as evaluated in Section 6.3.2).

It is worth noting that a global Vcache will not become a single-point performance bottleneck, since Vcache is non-blocking for both reads and writes. Specifically, only one coordinator is allowed to modify a particular key's version at one time in step ❸, since the read-write data are already locked by using RDMA CAS. Therefore, multiple coordinators are able to atomically read the newest versions from the Vcache without any wait, since the version size is equal to the atomic unit of CPU write [53]. In this way, looking up the Vcache becomes very fast. FORD leverages a hash table to implement the Vcache, in which the *key* is the record's key and the *value* is the record's version. Moreover, a global Vcache will not lead to the single-point failure risk, since the coordinators update the versions in remote replicas before transaction commit (❹). Hence, the remote replicas also store the newest versions. Thereby, even if the Vcache is lost after a crash, the coordinators are able to continue to access the versions in remote replicas without being stalled. FORD recovers the Vcache by re-collecting the data versions.

In essence, the key idea behind our localized validation scheme is to reduce the number of network accesses to remote memory pool by offloading the validation task to the compute pool as much as possible. The experimental results presented later in Figure 23 demonstrate the performance improvements when allowing the coordinators to locally access the versions in an in-compute-node global Vcache.

4.8 Failure Tolerance

The Replica Fails in the PM Pool. Due to supporting replication in dtxn processing, FORD recovers the data in the failed replicas from other replicas that are alive. If any primary or backup fails, (1) before commit, the coordinator aborts the transaction and unlocks the data; (2), during commit, the coordinator aborts the transaction, reads the remote undo logs to revoke data updates, and unlocks data; and (3) in the release phase, the transaction has already committed. The coordinator clears the VLock in the replicas. If some replicas cannot be recovered, we add new replicas to maintain the $(f + 1)$-way replication and migrate data to the new replicas.

The Coordinator Fails in the Compute Pool. Due to writing undo logs to remote replica, FORD handles coordinator failures by rolling back dtxns. Like FaRM [31] and DrTM [82], FORD supports to use leases [34] to detect failures. After the leases expire (e.g., 5 ms [31]), a failure possibly occurs. However, if a coordinator fails before it reports "Tx_committed," it is unknown whether

the remote replicas have been updated. To address this issue, FORD reads the undo logs in replicas to revoke all the updates and reports "Tx_aborted" to clients. Moreover, even if the cached data of coordinators (e.g., address cache and `Vcache`) are lost due to the volatility of DRAM, the correctness is not affected, since the coordinators can easily rebuild the local caches by gathering the related information (e.g., remote addresses and versions) again from the initial state.

The Network Communication Fails. In practice, it is difficult to distinguish the network failure from node failure. In line with state-of-the-art one-sided RDMA-based membership management system uKharon [37], we assume that network partitions (caused by network failures) are discovered and eventually resolved by the datacenter administrators. If a network partition occurs and is discovered by administrators, either availability or consistency becomes weak based on the CAP theorem [9, 33]. In this case, FORD only allows the replicas in the majority partition [8] to serve for the new requests before the membership is updated to guarantee the strong consistency to provide the ACID properties for OLTP applications. The membership can only be updated in the majority partition [37]. A systematic study of network partitions and membership management is outside the scope of this article and is our future work.

4.9 Correctness and Overhead Analysis

Serializability. FORD leverages locks and validations to guarantee serializability. The committed read-write dtxns are serializable at the point where all the written data are successfully locked. The committed RO dtxns are serializable at the point of their last read. FORD guarantees these serializability points by ensuring that *the data versions at the serialization point are equal to the versions during execution*—that is, locking ensures this for the written data since other coordinators cannot modify the versions of locked data, and validation ensures this for the read data since a version change will abort the dtxn. Moreover, to guarantee serializability across failures, the coordinator waits for all ACKs from all replicas before commit. Once a replica fails during the coalescent commit, FORD aborts the dtxn since an ACK is not received.

ACID. FORD ensures the ACID properties for dtxnsThe first property is *A*tomicity. FORD records undo logs, which are used to revoke the partial updates if a failure occurs before commit. The second property is *C*onsistency. All the data versions are consistent before the dtxn starts and after it commits. The third property is *I*solation. FORD uses locks and version validations to guarantee the serializability among the read-write and RO dtxns. The fourth property is *D*urability. The updated data are persistently stored in PM after commit by using our selective remote flush scheme.

The Number of RDMA Operations. Due to fully using one-sided RDMA to bypass the CPUs in the PM pool, FORD inevitably increases the number of RDMA operations to commit a dtxn. It is worth noting that the new RNICs (e.g., ConnectX-5 [24]) are efficient to handle one-sided RDMA operations including `CAS` [81]. Hence, slightly increasing the number of RDMA operations has negligible impacts on performance. In fact, FORD focuses on reducing the number of RDMA RTTs, which is more important to improve the performance since the RDMA RTT (e.g., 3–8 µs [4]) is still higher than the latency of local PM access (e.g., 62–305 ns [89]).

5 IMPLEMENTATIONS

5.1 Data Store in the Memory Pool

FORD supports different indexes to organize DB tables in the PM pool, such as hash tables and B$^+$-trees. These indexing schemes form the data store of FORD, called *FStore*. Our transaction techniques are independent of the specific index used in FStore, since these techniques aim to reduce network RTTs and balance loads, and regard remote data as general objects. For example, when using B$^+$-trees, our hitchhiked locking scheme reads and locks the leaf nodes, and our coalescent

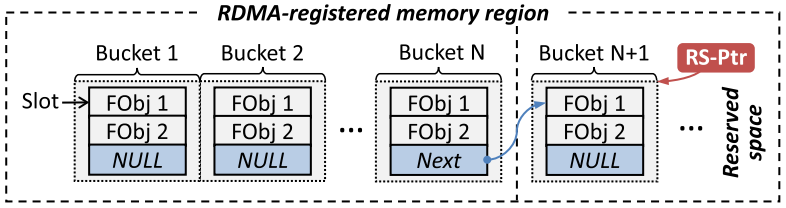

Fig. 11. The hash table structure in FStore.

commit scheme writes the updated tree nodes back to all replicas together. The internal pointer nodes are cached to reduce remote tree traversing. Moreover, since the hash table is widely used in fast RDMA operations [30, 75, 82, 96], we use the hash table as an example to present the implementations of FStore. Each hash table maintains a DB table and supports read/update/insert/delete operations.

The records in DB tables are persistently stored in FStore. Existing hashing schemes that support fixed-size and variable-size records can be used in FORD, such as RACE hashing [96]. When supporting fixed-size records, the records are stored in the hash table for direct access. When supporting variable-size records, the pointers of records are stored in the hash table. In this case, our hitchhiked locking scheme reads and locks the pointer, then fetches the record. Hence, FORD is flexibly to adapt different hash schemes to support fixed or variable record sizes. For simplicity, we show an implementation of storing fixed-size records in this article. Like FaSST [46], the record consists of an 8B key and a maximum-sized value (e.g., 1 KB). Such record meets many OLTP workloads (e.g., `TPCC` [27]). To further support dtxns, FORD packs the record with the following information into an object, called *FObj*:

- `Occupy` (1B): Whether this FObj occupies a slot.
- `TableID` (8B): The DB table that this record belongs to.
- `Version` (8B): The version number of this record.
- `VLock` (8B): 1-bit (in)visibility flag and 63-bit lock.

After allocating and registering a **Memory Region (MR)** in the PM pool, FStore enables clients to load records into hash tables before running dtxns. Figure 11 shows the structure of hash tables. A hash table contains an array of buckets, and each bucket contains several slots and one pointer called `Next`. The numbers of buckets and slots are configured by clients. Each FObj occupies a slot. A client initializes an FObj and hashes its key to obtain the target bucket (e.g., $b1$) to be inserted. After inserting the FObj to an empty slot, its `Occupy` is set to 1. If $b1$ is full, the FObj is inserted to a new bucket (e.g., $b2$) whose address is recorded in the `Next` pointer of $b1$. $b2$ is stored in a **Reserved Space (RS)** of the MR. The size of the RS is set by the client (e.g., 20% of the MR space). The client uses a pointer, called *RS-Ptr*, to trace the current bucket address in the RS. Moving the RS-Ptr forward to a bucket size will generate a new bucket in the RS. If the RS is exhausted but the hash collision still occurs, the client re-allocates memory to load tables.

To enable coordinators to calculate remote addresses for one-sided RDMA in dtxn processing, the CM in the PM pool sends the metadata (as listed in Table 1) of each hash table to the compute pool during network interconnections.

Given the key (e.g., `K0`) of a record, if its remote address is buffered in the local cache, the coordinator directly reads the record using an RDMA `READ`. Otherwise, the coordinator reads a remote record as the following steps:

S1: Calculate the bucket id:
`bucket_id = Hash(K0) mod BucketNum`

Table 1. Metadata of Each Hash Table in FStore

Field	Size (B)	Description
TableID	8	The global unique DB table id
Addr	8	The virtual start address of this hash table
Off	8	The offset between Addr and MR's start address
BucketNum	8	The bucket number of the hash table
BucketSize	4	The size of a bucket (in bytes)
SlotNum	4	The number of slots per bucket

S2: Calculate the bucket offset in the remote MR:
bucket_off = bucket_id × BucketSize + Off
S3: Read the remote bucket (*bkt*) using bucket_off.
S4: Compare K0 with the SlotNum keys in *bkt*. If a key = K0, the record is obtained. Then go to S7. Otherwise, go to S5.
S5: If the next field of *bkt* is NULL, there is no such remote record. Then go to S8. Otherwise, go to S6.
S6: Calculate the next bucket offset as follows and go to S3.
bucket_off = bkt.next − Addr + Off
S7: Exit if the record is visible. Otherwise, re-read the record until it becomes visible.
S8: Exit with a KEY_NOT_EXIST hint.

Since the metadata size of a hash table is only 40B and each remote address is 8B, the local cache in compute pool can buffer all these metadata and addresses, as shown later in Figure 21(a). Caching metadata is scalable, because the compute units do not need to synchronize their metadata with each other: (1) the metadata of index does not change, and (2) if the cached addresses are stale, FORD enables the coordinator to detect this and update its own cached addresses, as discussed in Section 4.2.

5.2 Transaction Interfaces

FORD provides a runtime library, called *FLib*, for applications to process dtxns. FLib exposes the following interfaces:

- TxBegin: Start to execute a dtxn and record its id.
- AddRO: Add an initialized FObj to the RO set.
- AddRW: Add an initialized FObj to the read-write set.
- TxExecute: The dtxn enters the execution phase. In this phase, the coordinator reads the data specified in the RO and read-write sets from remote replicas, and writes the undo logs back to replicas in parallel. After fetching the required data, the coordinator executes the transaction logic to complete the execution phase. Our hitchhiked locking and backup-enabled read schemes are leveraged in the TxExecute interface.
- TxCommit: After the execution phase, the coordinator validates the data versions and commits the updated data to remote primaries and backups. Our localized validation, coalescent commit, and selective remote flush schemes are leveraged in the TxCommit interface.

Our transaction interfaces support general transaction processing. Specifically, the developers are not required to know all the read/write sets at the beginning of each transaction. Instead, developers call AddRO, AddRW, and TxExecute multiple times when reading/writing data occurs during a transaction. Figure 12 illustrates an example of using our interfaces in the Write Check transaction of the SmallBank benchmark [71]. This transaction reads the balances from the Savings and

```
bool WriteCheck(uint64_t dtxn_id, DTXN* dtxn) {
    // The `dtxn` invokes FLib interfaces
    dtxn->TxBegin(dtxn_id);
    // Generate a random account as the key
    uint64_t acct_id = RandomAccount();
    FObj* sav_obj = new FObj(SavingsTableID, acct_id);
    FObj* chk_obj = new FObj(CheckingTableID, acct_id);
    // Add FObjs to the read-only and read-write sets
    // according to the transaction logic
    dtxn->AddRO(sav_obj);
    dtxn->AddRW(chk_obj);
    // Fetch data from remote replicas
    if (!dtxn->TxExecute())
        return false;
    // Get the values of records
    sav_val_t* sav = (sav_val_t*) sav_obj->value;
    chk_val_t* chk = (chk_val_t*) chk_obj->value;
    // Run the transaction logic in coordinator
    if (sav->balance + chk->balance < PredefinedAmount)
        chk->balance -= (PredefinedAmount + 1);
    else
        chk->balance -= PredefinedAmount;
    // Commit the updated data to remote replicas
    bool status = dtxn->TxCommit();
    delete sav_obj;
    delete chk_obj;
    // Report commit (true) or abort (false) to client
    return status;
}
```

Fig. 12. An example of C++ code using FLib interfaces.

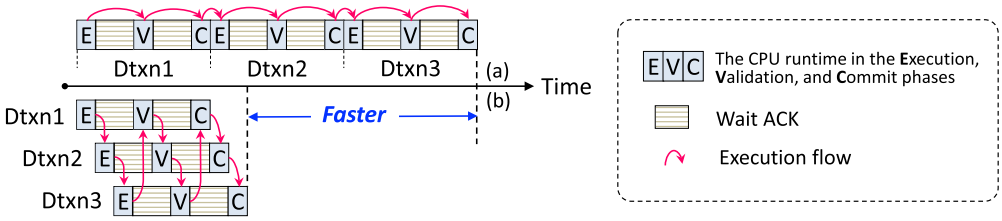

Fig. 13. The comparisons between sequential processing (a) and interleaved processing (b), in one thread.

`Checking` tables, and updates the balance in the `Checking` table. It shows that our interfaces are easy for programmers to use.

5.3 Interleaved Transaction Processing

As shown in Figure 13(a), sequentially processing dtxns in a thread wastes the CPU cycles due to waiting for RDMA ACKs. To avoid CPU idling in the compute pool, FORD leverages an interleaved processing model that enables multiple coordinators in one thread to process different dtxns in the pipeline, as presented in Figure 13(b). In this way, the network RTTs are efficiently hidden and the CPU cores in the compute pool are fully utilized to improve the throughput.

We use coroutines [46, 81] to implement the interleaved processing. Each CPU thread generates several coroutines and each coroutine acts as a coordinator to execute dtxns. After issuing the RDMA requests, a coroutine yields its CPU core to another coroutine to process the next dtxn. A dedicated coroutine in each thread polls RDMA ACKs. If all ACKs of a coroutine arrived, FORD schedules this coroutine to occupy the CPU core to resume execution. The results presented later in Figure 22 show that using a proper number of coroutines improves the throughput without high latency.

6 PERFORMANCE EVALUATION

6.1 Experimental Setup

Testbed. We use three machines, each of which contains a 100-Gbps Mellanox ConnectX-5 InfiniBand RNIC. They are connected via a 100-Gbps Mellanox SB7890 InfiniBand switch. One machine equipped with the Intel Xeon Gold 6230R CPU and 8 GB of DRAM is leveraged as the compute pool to run coordinators. The other two machines form the PM pool, each of which contains six interleaved 128-GB Intel Optane DC PM modules. Each DB table is stored in the two PM machines to maintain a two-way replication (i.e., one primary and one backup).

Benchmarks. We leverage a **Key-Value Store (KVS)** as the *micro-benchmark* to analyze how different factors affect each design of FORD. `KVS` stores 1 million key-value pairs in one table, in which the key is 8B and value is 40B. The transaction in `KVS` accesses a specific number of objects with different read:write ratios and different access patterns as configured in Section 6.2. `KVS` supports the skewed and uniform access patterns, in which the skewed access uses the Zipfian distribution with the default skewness 0.99 [16]. We further adopt three OLTP benchmarks (i.e., `TATP` [70], `SmallBank` [71], and `TPCC` [27]) as the *macro-benchmarks* to examine the end-to-end performance. These benchmarks are widely used in prior studies [31, 46, 81, 82]. `TATP` models a telecom application and contains four tables, in which 80% of the transactions are RO and the record size is up to 48B. `SmallBank` simulates a banking application that includes two tables, in which 85% of transactions are read-write and the record size is 16B. `TPCC` models a complex ordering system that consists of nine tables, in which 92% of transactions are read-write and the record size is up to 672B. We generate eight warehouses in `TPCC`. We have implemented all the workloads of each macro-benchmark and run the standard transaction mix in Section 6.3. We run 1 million dtxns in each benchmark and report the throughput by counting the number of *committed* dtxns per second. We report the processing time of the committed dtxn as the latency, including the 50th and 99th percentile latencies.

Comparisons. We compare FORD with two state-of-the-art RDMA-based dtxn systems: FaRM [31] and DrTM+H [81] (called *DrTMH* in the rest of this article). We use one-sided RDMA to re-implement the dtxn protocols in FaRM and DrTMH to meet the requirement of disaggregated PM. We do not compare against FaSST [46] that fully uses two-sided RDMA, which is difficult to work in the disaggregated memory architecture due to consuming the remote CPUs in the memory pool throughout the entire process of running dtxns.

6.2 Micro-Benchmark Results and Analysis

6.2.1 Lock Duration. Locks are generally used to serialize dtxns. However, a long lock duration causes frequent aborts and leads to low throughputs. To obtain the lock duration, we configure the coordinator not to abort dtxn but to wait for the data to be unlocked if the locking fails. We compare the lock duration in FORD, FaRM, and DrTMH by using 64 coordinators to concurrently run dtxns in which each dtxn processes one data. Our localized validation scheme is not enabled. Figure 14 shows the average lock duration of each coordinator at different read:write ratios in the dtxn mix—for example, 25:75 means that 25% of the dtxns are RO while 75% are read-write. The results show that the reduction of lock duration is larger in the uniform access when reducing the write ratio, since the uniform access has lower locality than the skewed access, which decreases the data hotness. Hence, the total time for the coordinator to wait for unlocking the hot data significantly decreases. Compared with DrTMH and FORD, FaRM suffers from longer lock durations since the data are locked across four phases: locking, validation, commit backup, and commit primary. DrTMH reduces the lock duration by merging the locking and validation into one phase. Our FORD uses the hitchhiked locking scheme to lock the read-write data in the execution phase,

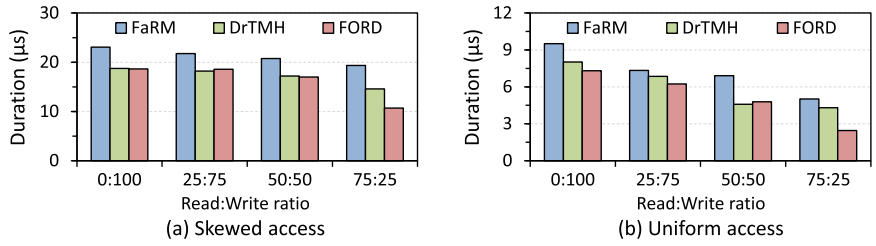

Fig. 14. The lock durations.

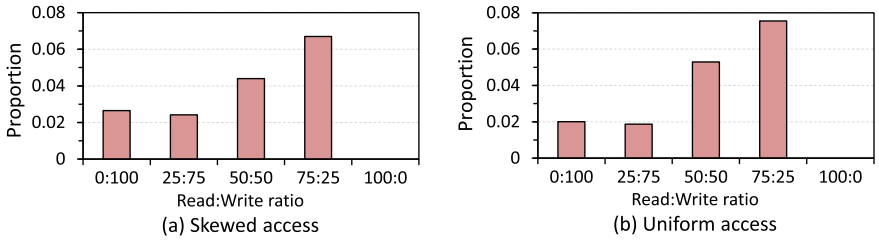

Fig. 15. The proportions of the invisibility durations.

but the lock duration does not become longer, since the read-write data do not need to be locked or validated again, and the dtxn commits earlier.

6.2.2 Invisibility Duration. FORD leverages the coalescent commit scheme to update the primaries and backups together in one round trip. To avoid partial reads, the data are temporarily marked as invisible after commit until the background release phase. To analyze the overheads of the data invisibility, we record the total time spent for re-reading the invisible data until visible (i.e., invisibility duration) in 64 coordinators, then calculate the proportion of the invisibility duration in the entire dtxn running time.

As shown in Figure 15, the proportion slightly decreases when increasing the read ratio from 0% to 25%, since the invisible data are reduced when decreasing writes. As the read ratio continues to increase, the proportion increases, since there are substantial RO dtxns that wait for the data to be visible, which increases the total invisibility durations across all coordinators. When the read:write ratio is 100:0, all data are visible and the proportion becomes 0. The results show that the proportion is only less than 8% in different read:write ratios and access patterns, since the background release phase consumes at most 0.5 RTT to make data visible. Therefore, the data invisibility in our coalescent commit design exhibits low performance overheads.

6.2.3 Read from Backups. Due to the limited write bandwidth of PM, FORD enables the coordinators to read the RO data from backups to alleviate the load on the primary's PM. To demonstrate the benefits of this design, we run 224 coordinators to increase the load, and examine the dtxn throughput and latency when disabling/enabling the coordinators to read backups.

Figures 16 and 17 show that as the read ratio increases, enabling coordinators to read the backup replica improves the throughput by up to 1.5×, and reduces the 50th/99th percentile latencies by up to 31.7%/35.3%. The backup absorbs substantial read requests to prevent all the coordinators from competing for the primary's PM, thus improving the throughput. When increasing the number of backups, FORD will gain higher performance improvements since all the backups can be read to balance loads. Moreover, Figure 17 shows that enabling read backups

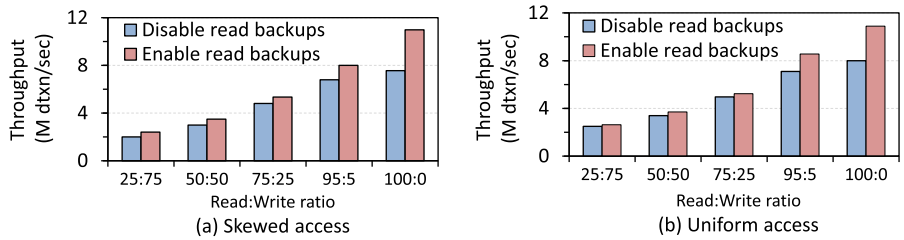

Fig. 16. The dtxn throughput when disabling/enabling coordinators to read the backup replicas.

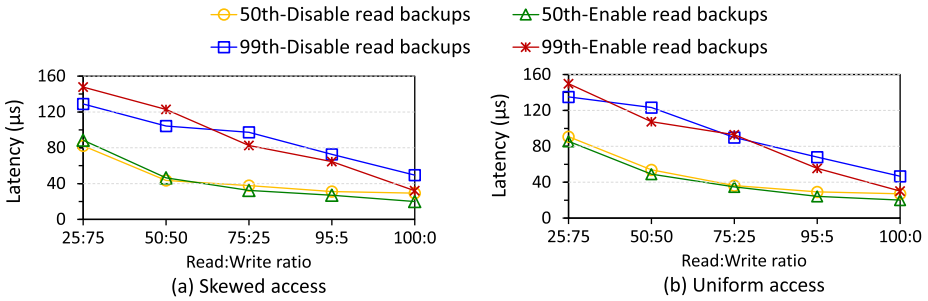

Fig. 17. The dtxn latency when disabling/enabling coordinators to read the backup replicas.

slightly increases the latency at a low read ratio, since the coordinator does not know the remote data addresses in the backup at the first read. In this case, the coordinator needs to perform hash reads to locate data. Nevertheless, as the read ratio increases, the coordinator could access the same key and leverage the local address cache to directly read the target data without consuming the hash reads. Moreover, the backups serve for more read requests to balance the loads. Hence, the latency of enabling read backups decreases as the read ratio increases.

Case Study. Figures 16 and 17 leverage the micro-benchmark that contains one table. A memory node hence stores either the primary or backup of the table. However, in practice, one memory node can store the primary for some data and the backup for other data. Therefore, to further demonstrate the benefit of using backups for reading data, we compare the transaction performance of "FORD-Disable read backups" and "FORD-Enable read backups" schemes on the `SmallBank` benchmark, which contains two tables: `Savings` and `Checking`. In our experiments, one memory node (i.e., `MN1`) stores the primary of `Savings` and backup of `Checking`, and the other memory node (i.e., `MN2`) stores the primary of `Checking` and backup of `Savings`. The "FORD-Disable read backups" scheme only leverages the primary to handle all the requests from coordinators, whereas the "FORD-Enable read backups" scheme leverages the backup to absorb the read requests. The performance results are shown in Figure 18. To draw a throughput-latency curve, we increase the number of concurrent coordinators to augment the loads and contentions. Figure 18 shows that "FORD-Enable read backups" improves the throughput by 20.7% over "FORD-Disable read backups" due to enabling the backups to serve read requests to balance the load.

We now analyze the detailed reasons for the preceding throughput improvement. In the standard transaction mix of `SmallBank`, there are six types of transactions with specified running frequencies [71], as listed in Table 2. Without loss of generality, suppose that 100 dtxns have been processed following the standard running frequencies, the total number of reads and updates on `Savings` and `Checking` tables are counted in Table 3, and hence the request loads on `MN1` and

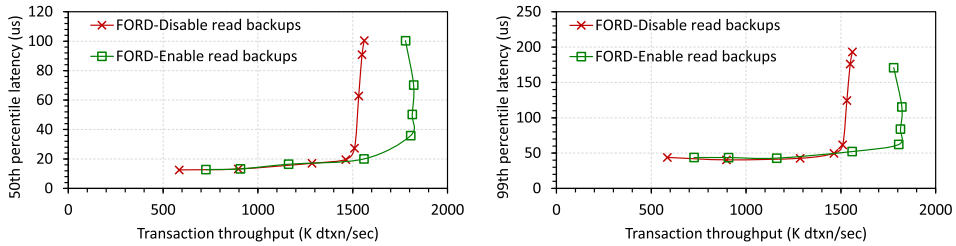

Fig. 18. The dtxn throughput and the 50th and 99th percentile latencies on the SmallBank benchmark.

Table 2. Transaction Frequencies and Data Accesses to Tables in SmallBank

Transaction	Frequency (%)	Data Access	
		Savings Table	Checking Table
AMALGAMATE	15	Update 1 record	Update 2 records
BALANCE	15	Read 1 record	Read 1 record
DEPOSIT_CHECKING	15	—	Update 1 record
SEND_PAYMENT	25	—	Update 2 records
TRANSACT_SAVINGS	15	Update 1 record	—
WRITE_CHECK	15	Read 1 record	Update 1 record

Table 3. Total Number of Reads and Updates on Savings and Checking Tables After Processing 100 Dtxns

Savings Table	Read	1×15(BALANCE) + 1×15(WRITE_CHECK) = 30
	Update	1×15(AMALGAMATE) + 1×15(TRANSACT_SAVINGS) = 30
Checking Table	Read	1×15(BALANCE) = 15
	Update	2×15(AMALGAMATE) + 1×15(DEPOSIT_CHECKING) + 2×25(SEND_PAYMENT) + 1×15(WRITE_CHECK) = 110

Table 4. Request Loads on MN1 and MN2 by using the "FORD-Disable Read Backups" and "FORD-Enable Read Backups" Schemes

Memory Node	Role of Table	"FORD-Disable Read Backups"	"FORD-Enable Read Backups"
MN1	(*Primary*) Savings	30 (Read) + 30 (Update) = 60	30 (Read) + 30 (Update) = 60
	(*Backup*) Checking	—	15 (Read)
MN2	(*Primary*) Checking	15 (Read) + 110 (Update) = 125	110 (Update)
	(*Backup*) Savings	—	—

MN2 are calculated in Table 4. The results show that when using the "FORD-Disable read backups" scheme, the load of MN1 : MN2 = 60 : 125 = 1 : 2.1, since the Checking table is a hotspot. However, when using the "FORD-Enable read backups" scheme, MN1 is able to serve the read requests for the Checking table, and the load of MN1 : MN2 = 75 : 110 = 1 : 1.5. Therefore, the loads on memory nodes become more balanced. This is why the "FORD-Enable read backups" scheme achieves better throughput in Figure 18. In summary, using backup for reading data is effective to balance load and improve the dtxn throughput especially when data hotspots exist.

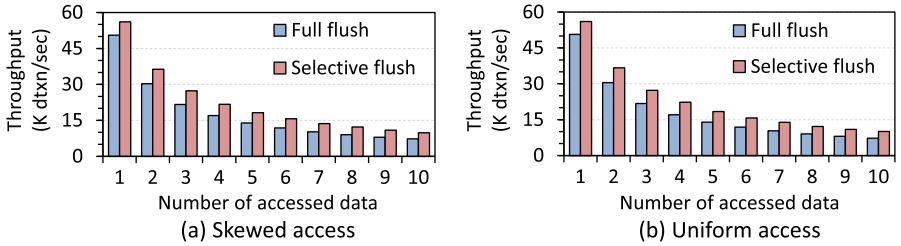

Fig. 19. The dtxn throughput using full/selective flush when accessing different numbers of data per dtxn.

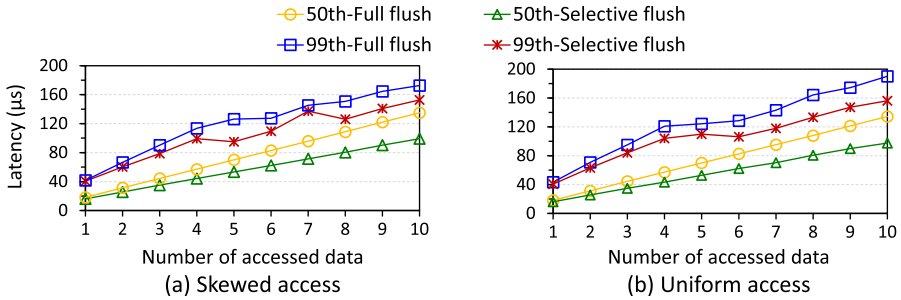

Fig. 20. The dtxn latency using full/selective flush when accessing different numbers of data per dtxn.

6.2.4 Remote Flush. FORD guarantees the remote persistency in dtxn processing by flushing the data from the remote RNIC cache to PM. We compare the full flush and selective flush (with request batching) schemes discussed in Section 4.5 in terms of dtxn throughput and latency. To show the overheads of remote data flush, we use one coordinator to avoid extra contention overheads. We increase the number of written data per dtxn to add the flush operations. The results in the skewed and uniform accesses exhibit similar trends. Figure 19 shows that our selective flush scheme improves the throughput by 28.7%/29.5% over the full flush scheme in skewed/uniform access. Figure 20 shows that the selective flush mitigates the 50th/99th percentile latencies by 22.5%/12.4% (22.8%/14%) in the skewed (uniform) access. Our selective flush scheme performs better due to only issuing flushes to the backups after the final RDMA `WRITE`, thus reducing the number of flush operations. Moreover, we observe that the performance of using the selective flush decreases when the number of accessed data increases. This is because other operations in the dtxn increase (e.g., data reads, validations, and remote writes), thus decreasing the overall performance.

6.2.5 Address Cache. The coordinator has a local address cache to buffer remote data addresses for efficient one-sided RDMA operations. To evaluate the overheads (including the size and miss rate) of the address cache, we change the maximum number of accessible keys from 1k to 512k to obtain the average sizes of buffered addresses, and the average miss rates during address lookups.

Figure 21(a) shows that the buffered addresses only consume 6.8 MB even if uniformly accessing 512k keys with poor locality. Hence, a small cache is sufficient for a coordinator to buffer remote addresses. Since a gigabyte-scale DRAM is leveraged in the compute pool to store the metadata [66], it is unnecessary to limit the size of the coordinator-local address cache in practice. Figure 21(b) shows that the miss rate is 18.2% (or 44.6%) when accessing 512k keys in the skewed (or uniform) access. For a cache hit, the coordinator uses the buffered address to directly read the record. However, if a cache miss (or a hash bucket collision) occurs, the coordinator needs to calculate the

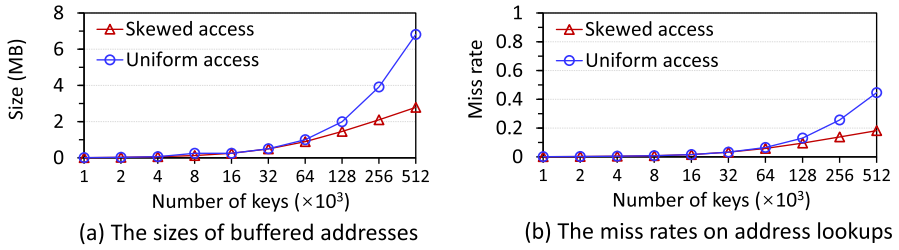

Fig. 21. The size and miss rate of the address cache.

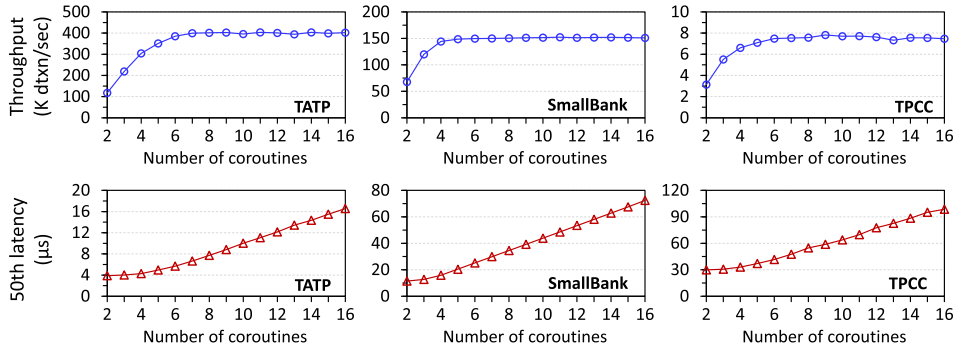

Fig. 22. The dtxn throughput and 50th percentile latency of different numbers of coroutines in one thread.

remote bucket address, read a bucket in one round trip, and finally find the record. Such a long processing flow incurs more latency. In general, the miss rate depends on the locality of workloads. If some remote addresses are not buffered, the cache misses are inevitable in dtxn processing. However, FORD provides a sufficiently large local address cache for each coordinator to avoid evicting the buffered addresses, thus reducing the miss rate as much as possible.

6.3 Macro-Benchmark Results and Analysis

6.3.1 Coroutine Execution. To improve the throughput, FORD leverages coroutines to process dtxns to avoid CPU idling. A thread generates at least two coroutines, since one coroutine is dedicated to poll the RDMA ACKs. Figure 22 shows the dtxn throughput and median latency in macro-benchmarks when changing the number of coroutines in one thread. The throughputs increase by 3.4×/2.2×/2.5× on TATP/SmallBank/TPCC until the CPU is saturated. However, the latency continues to increase when using more coroutines, since the execution pipeline becomes deeper, and the coroutines are scheduled to wait for occupying the CPU to resume execution.

From the experimental results, we observe that using a small number of coroutines (e.g., 5–8) is helpful to significantly improve the throughput without heavily increasing the latency. In general, read-intensive workloads require more coroutines to saturate the CPU since an RO dtxn has less computation tasks than a read-write dtxn. For example, SmallBank and TPCC respectively require five and six coroutines to reach the maximum throughput, whereas TATP requires eight coroutines.

6.3.2 End-to-End Performance. We leverage three macro-benchmarks (i.e., TATP, SmallBank, and TPCC) to compare the end-to-end dtxn performance of FaRM, DrTMH, FORD, and FORD-LV. FORD incorporates our schemes in Sections 4.2 through 4.5 to perform all transaction operations

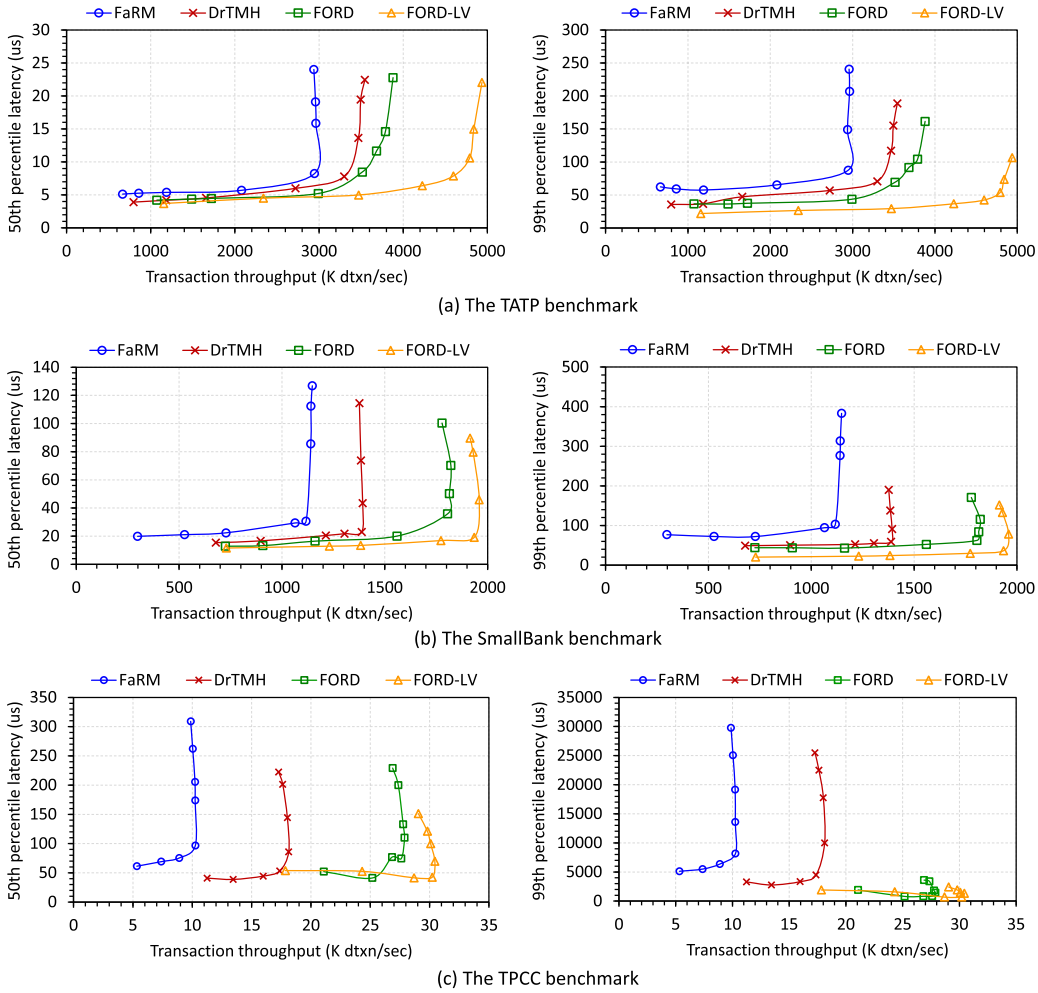

Fig. 23. The end-to-end dtxn performance of different systems on three macro-benchmarks.

through RDMA. FORD-LV further leverages our *L*ocalized *V*alidation scheme in Section 4.7 to validate data versions in the local Vcache of all coordinators.

The performance metrics include the dtxn throughput, the 50th percentile latency, and the 99th percentile latency. To compare the performance among all these systems, we increase the number of threads from 8 to 20, and set the number of coroutines of each thread from 2 to 12, to augment the loads and contentions (i.e., supporting up to $20 \times (12 - 1) = 220$ concurrent coordinators). Due to different system scales (e.g., 90 machines in the work of Dragojevic et al. [31]), the overall throughput in our small-scale testbed becomes lower than that in other works [31, 81]. However, our testbed can accurately evaluate the performance gap between different systems.

Figure 23 shows that compared with FaRM/DrTMH, FORD improves the throughput by 1.3×/1.1× and reduces the 50th (99th) latency by 36.8%/13.2% (50.2%/23.3%) in TATP, improves the throughput by 1.6×/1.3× and reduces the 50th (99th) latency by 43.9%/28.3% (54.6%/26.7%) in SmallBank, and improves the throughput by 2.7×/1.5× and reduces the 50th (99th) latency by 23%/13.2% (84.2%/68.8%) in TPCC. DrTMH outperforms FaRM by merging the locking and

validation phases. FaRM and DrTMH show high throughput in TATP, since 80% of the dtxns are RO and the usages of one-sided RDMA READs accelerate dtxn processing. However, in SmallBank and TPCC that contain extensive read-write dtxns, the performance decreases due to their long dtxn processing paths. Unlike them, our FORD efficiently mitigates the round trips to shorten the processing path, and balances loads on the replicas, thus improving the performance.

By localizing the validation phase in compute pool, FORD-LV further reduces the RDMA round trips. Specifically, compared with FaRM/DrTMH/FORD, FORD-LV improves the throughput by 1.7×/1.4×/1.3× and reduces the 50th (99th) latency by 39.4%/16.8%/4.1% (66.4%/48.2%/32.5%) in TATP, improves the throughput by 1.7×/1.4×/1.1× and reduces the 50th (99th) latency by 54.3%/41.6%/18.5% (74.7%/59.1%/44.3%) in SmallBank, and improves the throughput by 3×/1.7×/1.1× and reduces the 50th (99th) latency by 57.2%/51.7%/44.3% (87.4%/75%/19.9%) in TPCC. We observe that FORD-LV achieves more throughput improvements over FORD in TATP. This is due to the fact that the write contention in TATP is low, and hence the utilization of Vcache largely mitigates the needs of fetching versions from remote replicas. Note that FORD-LV exhibits the performance when accessing the data versions from an in-compute-node global Vcache. In other system configurations, if some communications between the compute nodes occur when accessing the Vcache, our solutions discussed in Section 4.7 efficiently mitigate the overheads, and the performance is expected to remain higher than FORD, which fully accesses the data versions in the remote memory pool.

Finally, we evaluate the size of Vcache in macro-benchmarks and observe that Vcache only consumes 1 GB for a high hit ratio. Given that caching remote data addresses and the metadata of remote hash tables only consumes several megabytes of space, the total size of all caches in the compute pool does not exceed the DRAM capacity of the compute pool (e.g., 4 GB [66]).

7 DISCUSSION

Recently, Intel discontinued the business of Optane PM for commercial reasons [19]. However, the impact of our work is not significantly affected by this issue. We present the reasons in the following.

Our work targets on the common features of different PMs rather than only relying on Intel Optane PM. The Optane PM is merely one available PM product in the market, and we leverage it to conduct experiments. In general, different types of PMs (e.g., **Compute Express Link (CXL)**-based PM [7], CXL-based NAND flash supporting memory-semantic SSD [43], non-volatile dual in-line memory module (NVDIMM) [51], phase-change memory (PCM) [84], resistive random access memory (ReRAM) [2], and spin-transfer torque magnetic random access memory (STT-MRAM) [5]) share the common features of byte-addressability, persistence, and asymmetric read and write bandwidth. Hence, our proposed five schemes are able to be applied on different types of PMs, as analyzed next:

- Our *hitchhiked locking*, *coalescent commit*, and *localized validation* schemes aiming to mitigate the network round trips are not affected by the specific type of PM in the memory pool.
- Our *selective remote flush* scheme still guarantees the RDMA-written data to be persisted from the remote NIC to PM, regardless of the PM types, due to sharing the "persistence" property of PM.
- Our *backup-enabled read* scheme is still efficient to balance the request loads on the primary and backup replicas to improve the performance. In practice, many real-world OLTP and DB applications perform skewed accesses, as reported in prior studies [12, 13, 39, 42, 90, 92]. Small amounts of hotspot data are frequently accessed, whereas other data remain cold. In

this case, the primary replica storing the hotspot data quickly becomes the performance bottleneck when the backup replicas cannot serve requests. As the InfiniBand network bandwidth increases at twice the speed (i.e., 100 Gbps/port in ConnectX-5 [24], 200 Gbps/port in ConnectX-6 [26], and 400 Gbps/port in ConnectX-7 [25]), substantial requests simultaneously arrive at the primary replicas to wait for being served. However, the write bandwidth is generally lower than the read bandwidth for different PMs, since the write operation needs to modify the bit states in the underlying physical media of PM. As a result, the read requests are blocked and the throughput decreases. Our backup-enabled read scheme efficiently allows the coordinators to read data from backup replicas at in-place locations, thus alleviating the pressure on primary replicas to improve the overall throughput, as demonstrated in Figures 16 and 18. Such benefit of load balancing applies to different types of PMs.

It is worth noting that PM is still an important type of memory device due to meeting necessary requirements (e.g., fast persistence, byte addressability, large capacity, and ns-scale low latency) for many important application scenarios, such as OLTP, DBs, file systems, and key-value stores. The academia and industry continue to explore and exploit various forms of PMs to provide fast and direct persistence. For example, CXL [15] opens new opportunities for byte-level persistence based on NAND flash [43] and Nantero's NRAM [23].

In summary, our work targets on the common features of different types of PM devices and adapts to other types of PMs. As a memory device, PM still plays an important role in many fundamental application scenarios. Therefore, the impact of our work is not significantly affected.

8 RELATED WORK

8.1 Fast Dtxn Processing

Many systems have been proposed for efficient dtxn processing. Some designs leverage RDMA to handle transactions [14, 30, 31, 46, 48, 60, 65, 81, 82]. Storm [60] proposes a transactional API to operate remote data based on one-sided reads and write-based RPCs. HyperLoop [48] offloads some computations to RNIC and requires the remote CPU to operate the metadata. Moreover, application locality [10, 47, 56] is exploited to convert a dtxn to a local one, which however sacrifices the generality. New transaction abstractions [85], replication protocols [93], and concurrency controls [59, 79, 86, 92] are also proposed to improve the performance. Some works offload locking and indexing to SmartNIC to accelerate dtxns [64]. All the preceding systems work on the monolithic architecture, whereas our FORD focuses on the new disaggregated PM architecture and fully leverages one-sided RDMA to process transactions.

8.2 Distributed PM

PM has been recently exploited in the distributed environments. These studies employ PM in a symmetric way, where each server in a cluster hosts the PM that can be accessed locally or remotely by other servers. Some designs expose interfaces of file system [4, 57, 87, 88] and memory management [67, 95]. Some studies provide optimization hints on system implementations when using RDMA and PM [44, 83]. In general, the symmetric deployment supports fast local accesses but suffers from poor resource scalability and coarse failure domain due to using monolithic servers. Unlike these works, FORD provides transaction interfaces and deploys PM in the disaggregated way to improve the scalability and failure isolation.

8.3 Disaggregated Memory

Disaggregated memory becomes popular in datacenters due to high resource utilization and elasticity. Existing works explore memory disaggregation in hardware architectures [38, 54, 55],

networks [32, 68], operating systems [66], key-value stores [75], hash indexes [96], data swapping [3, 11, 36, 63], and memory management [1, 52, 58, 76, 78]. Our proposed FORD is orthogonal to these systems to build a fast transaction processing system for the disaggregated PM.

9 CONCLUSION

Our paper proposed FORD, a fast distributed transaction processing system that fully leverages one-sided RDMA for the the new disaggregated PM architecture. To accelerate transaction processing, FORD explores and exploits the request batching and parallelization to eliminate extra locking and validation round trips for the read-write set, and commit all remote replicas together in a single round trip. Moreover, to efficiently utilize the remote PM, FORD enables the backup replicas to serve read requests to balance loads and guarantees the remote persistency with low network overheads. By transferring the validations for the RO set from remote to local, FORD further reduces the RDMA round trips to improve the performance. Experimental results demonstrate that FORD significantly outperforms state-of-the-art systems in terms of transaction throughput and latency.

ACKNOWLEDGMENTS

We are grateful to anonymous reviewers for their feedback.

REFERENCES

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, et al. 2018. Remote regions: A simple abstraction for remote memory. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 775–787.

[2] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proceedings of the IEEE* 98, 12 (2010), 2237–2251.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput? In *Proceedings of the 15th EuroSys Conference (EuroSys'20)*. ACM, New York, NY, Article 14, 16 pages.

[4] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostic, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. 2020. Assise: Performance and availability via client-local NVM in a Distributed File System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 1011–1027.

[5] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, et al. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems* 9, 2 (2013), Article 13, 35 pages.

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

[7] Ankit Bhardwaj, Todd Thornley, Vinita Pawar, Reto Achermann, Gerd Zellweger, and Ryan Stutsman. 2022. Cache-coherent accelerators for persistent memory crash consistency. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*. ACM, New York, NY, 37–44.

[8] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (2012), 23–29.

[9] Eric A. Brewer. 2000. Towards robust distributed systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, Vol. 7. 343477–343502.

[10] 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.

[11] 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 (ASPLOS'21)*. ACM, New York, NY, 79–92.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. 209–223.

[13] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A hotspot-aware in-memory key-value store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. 239–252.

[14] 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 11th European Conference on Computer Systems (EuroSys'16)*. ACM, New York, NY, Article 26, 17 pages.

[15] CXL Consortium. 2022. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. Retrieved February 2, 2023 from https://www.computeexpresslink.org.

[16] 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 (SoCC'10)*. ACM, New York, NY, 143–154.

[17] Edgecore Networks Corporation. 2022. DCS800 Data Center Switch. Retrieved February 2, 2023 from https://www.edge-core.com/_upload/images/2022-051-DCS800_Wedge100BF-32X-R10-20220705.pdf.

[18] Hewlett Packard Corporation. 2022. The Machine: A New Kind of Computer. Retrieved February 2, 2023 from https://www.hpl.hp.com/research/systems-research/themachine/.

[19] Intel Corporation. 2022. Intel Reports Second-Quarter 2022 Financial Results. Retrieved February 2, 2023 from https://www.intc.com/news-events/press-releases/detail/1563/intel-reports-second-quarter-2022-financial-results.

[20] Intel Corporation. 2022. Intel®Optane™Persistent Memory. Retrieved February 2, 2023 from https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[21] Intel Corporation. 2022. Intel®Optane™Persistent Memory 200 Series (512GB PMEM Module). Retrieved February 2, 2023 from https://www.intel.com/content/www/us/en/products/sku/203880/intel-optane-persistent-memory-200-series-512gb-pmem-module/specifications.html.

[22] Intel Corporation. 2022. Intel®Rack Scale Design (Intel®RSD). Retrieved February 2, 2023 from https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[23] Nantero Corporation. 2022. Nantero's NRAM®. Retrieved February 2, 2023 from https://www.nantero.com/technology/.

[24] NVIDIA Corporation and Affiliates. 2021. NVIDIA CONNECTX-5. Retrieved February 2, 2023 from https://nvdam.widen.net/s/pkxbnmbgkh/networking-infiniband-datasheet-connectx-5-2069273.

[25] NVIDIA Corporation and Affiliates. 2021. NVIDIA CONNECTX-7. Retrieved February 2, 2023 from https://nvdam.widen.net/s/m6pt7j5rlb/networking-datasheet-infiniband-connectx-7-ds---1779005.

[26] NVIDIA Corporation and Affiliates. 2022. NVIDIA CONNECTX-6. Retrieved February 2, 2023 from https://nvdam.widen.net/s/5j7xtzqfxd/connectx-6-infiniband-datasheet-1987500-r2.

[27] The Transaction Processing Council. 2022. TPC-C Benchmark. Retrieved February 2, 2023 from http://www.tpc.org/tpcc/.

[28] Azure SQL Database. 2022. Use Read-Only Replicas to Offload Read-Only Query Workloads. Retrieved February 2, 2023 from https://docs.microsoft.com/en-us/azure/azure-sql/database/read-scale-out.

[29] Chet Douglas. 2015. RDMA with PM: Software mechanisms for enabling persistent memory replication. In *Proceedings of the 2015 Storage Developer Conference*.

[30] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. 401–414.

[31] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 54–70.

[32] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 249–264.

[33] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2 (2002), 51–59.

[34] Cary Gray and David Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review* 23, 5 (1989), 202–210.

[35] Paul Grun, Stephen Bates, and Rob Davis. 2018. Persistent memory over fabrics. In *Proceedings of the 2018 Persistent Memory Summit*.

[36] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient memory disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. 649–667.

[37] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. 2022. uKharon: A membership service for microsecond applications. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC'22)*. 101–120.

[38] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. ACM, New York, NY, 417–433.

[39] 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 International Conference on Management of Data (SIGMOD'21)*. ACM, New York, NY, 658–670.

[40] Doug Hakkarinen, Panruo Wu, and Zizhong Chen. 2015. Fail-stop failure algorithm-based fault tolerance for Cholesky decomposition. *IEEE Transactions on Parallel and Distributed Systems* 26, 5 (2015), 1323–1335. https://doi.org/10.1109/TPDS.2014.2320502

[41] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. 2008. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. 175–188.

[42] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. 2022. Aurogon: Taming aborts in all phases for distributed in-memory transactions. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*. 217–232.

[43] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets Computer Express Link for memory expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*. ACM, New York, NY, 45–51.

[44] Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'20)*. ACM, New York, NY, 105–119.

[45] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX'16)*. 437–450.

[46] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 185–201.

[47] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. 2021. Zeus: Locality-aware distributed transactions. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys'21)*. ACM, New York, NY, 145–161.

[48] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*. ACM, New York, NY, 297–312.

[49] H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems* 6, 2 (1981), 213–226.

[50] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical Paxos and primary-backup replication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC'09)*. ACM, New York, NY, 312–313.

[51] Changmin Lee, Wonjae Shin, Dae Jeong Kim, Yongjun Yu, Sung-Joon Kim, Taekyeong Ko, Deokho Seo, et al. 2020. NVDIMM-C: A byte-addressable non-volatile memory module for compatibility with standard DDR memory interfaces. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, Los Alamitos, CA, 502–514.

[52] SeungSeob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. ACM, New York, NY, 488–504.

[53] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. ACM, New York, NY, 462–477.

[54] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 267–278.

[55] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA'12)*. IEEE, Los Alamitos, CA, 189–200.

[56] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a non-2PC transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*. ACM, New York, NY, 1659–1674.

[57] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: An RDMA-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 773–785.

[58] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. ACM, New York, NY, 757–773.

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

[60] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, et al. 2019. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*. ACM, New York, NY, 97–108.

[61] Oracle. 2022. What Is OLTP? Retrieved February 2, 2023 from https://www.oracle.com/database/what-is-oltp/.

[62] Vmware Research. 2021. Remote Memory. Retrieved February 2, 2023 from https://research.vmware.com/projects/remote-memory.

[63] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 315–332.

[64] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. ACM, New York, NY, 740–755.

[65] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2019. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*. ACM, New York, NY, 433–448.

[66] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 69–87.

[67] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. ACM, New York, NY, 323–337.

[68] 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 *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 255–270.

[69] Tom Talpey, Tony Hurson, Gaurav Agarwal, and Tom Reu. 2020. RDMA Extensions for Enhanced Memory Placement. Retrieved February 2, 2023 from https://tools.ietf.org/html/draft-talpey-rdma-commit-01.

[70] TATP. 2011. Telecom Application Transaction Processing Benchmark. Retrieved February 2, 2023 from http://tatpbenchmark.sourceforge.net/.

[71] The H-Store Team. 2022. SmallBank Benchmark. Retrieved February 2, 2023 from https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[72] NVIDIA Corporation. 2023. RDMA Aware Networks Programming User Manual v1.7. Retrieved February 2, 2023 from https://docs.nvidia.com/networking/display/RDMAAwareProgrammingv17/RDMA+Aware+Networks+Programming+User+Manual.

[73] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. 2008. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA'08)*. IEEE, Los Alamitos, CA, 51–62.

[74] Muhammad Tirmazi, Adam Barker, Nan Deng, Md. E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next generation. In *Proceedings of the 15th EuroSys Conference (EuroSys'20)*. ACM, New York, NY, Article 30, 14 pages.

[75] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. 33–48.

[76] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 306–324.

[77] 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 (SIGMOD'17)*. ACM, New York, NY, 1041–1052.

[78] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A memory-disaggregated managed runtime. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 261–280.

[79] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-performance transactions via learned concurrency control. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. 198–216.

[80] Live Webcast. 2018. Extending RDMA for persistent memory over fabrics. In *Proceedings of the SNIA Networking Storage Forum*.

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

[82] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 87–104.

[83] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Characterizing and optimizing remote persistent memory with RDMA and NVM. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*. 523–536.

[84] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase change memory. *Proceedings of the IEEE* 98, 12 (2010), 2201–2227.

[85] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. 2014. Salt: Combining ACID and BASE in a distributed database. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 495–509.

[86] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 279–294.

[87] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 221–234.

[88] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA networking for scalable persistent memory. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. 111–125.

[89] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. 169–182.

[90] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 191–208.

[91] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The end of a myth: Distributed transactions can scale. *Proceedings of the VLDB Endowment* 10, 6 (Feb. 2017), 685–696.

[92] 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 (SIGMOD'20)*. ACM, New York, NY, 511–526.

[93] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 263–278.

[94] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*. 51–68.

[95] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 3–18.

[96] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-conscious extendible hashing for disaggregated memory. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*. 15–29.

Received 13 June 2022; revised 29 October 2022; accepted 6 January 2023