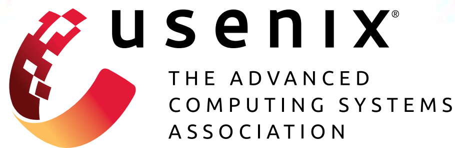


FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory

Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu,
Huazhong University of Science and Technology

https://www.usenix.org/conference/fast22/presentation/zhang-ming

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.
February 22–24, 2022 • Santa Clara, CA, USA
978-1-939133-26-7

Open access to the Proceedings
of the 20th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory

Ming Zhang, Yu Hua*, Pengfei Zuo, Lurong Liu
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

Persistent memory (PM) disaggregation improves the resource utilization and failure isolation to build a scalable and cost-effective remote memory pool. 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 in the disaggregated PM architecture. In this paper, we propose FORD, a **F**ast **O**ne-sided **R**DMA-based **D**istributed transaction system. FORD thoroughly leverages one-sided RDMA to handle transactions for bypassing the remote CPU in PM pool. To reduce the round trips, FORD batches the read and lock operations into one request to eliminate extra locking and validations. To accelerate the transaction commit, FORD updates all the 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. Experimental results demonstrate that FORD improves the transaction throughput by up to 2.3× and reduces the latency by up to 74.3% compared with the state-of-the-art systems.

1 Introduction

Memory disaggregation, which decouples the compute and memory resources from the traditional monolithic servers into independent compute and memory pools, has gained extensive interests in both industry [11, 14, 42] and academia [1, 20, 47, 57, 72]. By efficient resource pooling, the resource utilization, elasticity, failure isolation, and heterogeneity are significantly improved in datacenters [45]. The compute pool runs programs with a small DRAM buffer, and the memory pool stores application data with weak compute units only for memory allocations and interconnections [72]. Fast networks, e.g., RDMA, are generally adopted to connect the compute and memory pools [57]. Recently, the persistent memory (PM) is available on the market [12], which exhibits non-volatility and low latency with high density and low costs [67]. Hence, the efficient use of PM becomes important to build a persistent, large, and cost-effective disaggregated PM pool [54].

To ensure that the data are atomically and consistently accessed in the PM pool, the compute pool is required to leverage distributed transactions (dtxns) to read/write the 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, e.g., concurrency control in HTM [9, 61], data retrieving [60], locking [19, 29, 44], and busy buffer polling [18]. 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 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 the disaggregated PM due to the following challenges:

1) Long-latency processing. Legacy systems adopt the optimistic concurrency control (OCC) [32] to serialize dtxns, and the primary-backup replication for high availability. OCC is efficient for read-only dtxns due to no locks on read-only data. However, for the read-write dtxns, the data in read-write set consume 3 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 2 round trips to write remote replicas, i.e., writing redo logs to backups and updating primaries, which however delays the dtxn commit.

2) Limited PM bandwidth on the primary. When using the primary-backup replication, 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 256GB PM DIMMs [67]) than recent RDMA-capable NICs or RNICs (e.g., 25GB/s for a dual-port ConnectX-5 RNIC [52]). 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 [17, 23] 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 [29] uses the remote procedure call (RPC) to reduce round trips, but RPC requires the CPU in PM pool to frequently query, lock and update data. DrTM+H [60] 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 [68] decouples 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 operation logs. Once the operation logs fill up the memory, the system cannot serve writes. NAM-DB works on DRAM and disks, which is not designed for PM.

To tackle the above 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 paper 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 validations after the execution phase (§ 3.2). 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 (§ 3.3).
- ***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 (§ 3.4).
- ***Selective Remote Flush* to guarantee remote persistency with low overheads.** FORD leverages one-sided RDMA flush schemes to persist the written data from 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 primary-backup replication 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 (§ 3.5).

2 Background and Motivation

2.1 Disaggregated Persistent Memory

Traditional datacenters consist of a collection of monolithic servers, each of which hosts compute units and memory modules. However, such an architecture suffers from low resource utilization, poor elasticity, and coarse failure domain [57]. For example, even if only more CPU cores are needed, we have to add more servers that waste the memory/storage capacities.

To address these drawbacks, memory disaggregation decouples the compute and memory resources from monolithic servers to independent and RDMA-connected resource pools, in which each compute and memory pool is flexibly deployed and scaled, thus improving the resource utilization, elasticity, and failure isolation [72]. The compute pool contains substantial compute blades (e.g., CPU cores) to execute applications with a small memory as cache. The memory pool consists of many memory blades (e.g., DRAM DIMMs) to store the application data, and contain weak compute units only for memory allocations and network interconnections [57, 72].

The memory pool does not guarantee data persistence when using DRAM as memory blades. Plugging UPS [19, 61] 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 with the support of UPS, which incurs I/O overheads. Moreover, it is hard to increase the capacity of one DRAM DIMM due to the limited scalability [53], causing high costs to build a large memory pool.

Persistent memory (PM) addresses the above issues by providing persistence, high density (e.g., 512 GB/DIMM [13]), and low costs (e.g., 39.2% $/GB of DRAM [3]), while exhibiting DRAM-like latency [67]. As memory disaggregation meets the needs of datacenters, disaggregating PM also enjoys the same benefits [54]. Hence, we leverage PM as memory blades to build the disaggregated persistent memory (DPM), which forms a persistent and cost-effective memory pool.

2.2 RDMA-based Distributed Transactions

Due to the benefits of bypassing remote CPU and traditional TCP/IP stack, recent studies leverage RDMA to run distributed transactions (dtxns) [19, 29, 44, 60, 61]. Specifically, a coordinator is leveraged to read remote data, run dtxn logic, and commit the updated data back to remote machines. The concurrency control schemes, such as two-phase locking (2PL) [4] and optimistic concurrency control (OCC) [32], 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 read-only data, which benefits read-only dtxns. Moreover, the primary-backup replication (PBR) [33] is incorporated in dtxn processing for high availability [19, 60, 70]. The $(f+1)$-way PBR contains 1 primary and f backups for each data shard, and tolerates at most f replica failures. We assume that the fail-stop failures [25] can occur in arbitrary replicas at any time. The failed replica can be quickly detected and recovered by using RDMA [19]. Like FaRM [18, 19, 44], DrTM [9, 60, 61] and FaSST [29], we currently do not consider the byzantine failures [26].

Our paper focuses on the efficient use of OCC and PBR. Fig. 1 presents how existing RDMA systems [19, 60] process dtxns over OCC and PBR. Without loss of generality, we use 2-way replication as an example. In general, there are 5 phases: 1) 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. 2) Locking. After execution, the coordinator locks the write set in primaries to serialize dtxns. If locking fails, the coordinator aborts the dtxn. 3) Validation. If 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. 4) Commit backup. If the validation succeeds, the coordinator sends redo logs to remote backups. 5) Commit primary. After receiving all ACKs from backups, the coordinator updates and unlocks the primaries to commit the dtxn.

2.3 Distributed Transactions on DPM

System Model. In the disaggregated PM architecture, PM is used as remote memory with persistence to durably store the application data (including the primary and backups). The PM pool contains a small number of weak compute units only for memory allocations and RDMA connections during the *initialization* [57, 72]. Afterwards, these compute units are not used during the *execution* since they are too weak to frequently and efficiently handle substantial tasks. Moreover, there is no PM in the compute pool that uses RDMA to access the data stored in remote PMs at the byte granularity (no page swap). To ensure the atomicity, the compute pool uses coordinators to run transactions that read/write data across remote PMs. All transactions are hence distributed, and the

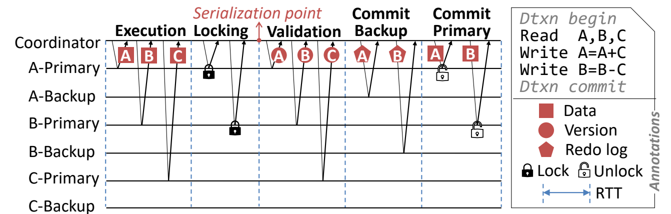


Figure 1: Using OCC and PBR to process dtxns.

replication is accessed by multiple coordinators, which use RDMA to commit each dtxn. Two-sided RDMA-based RPC reduces the network round trips by consuming remote CPUs to handle multiple operations in one round trip [29]. But the PM pool does not contain CPUs to process requests during execution, and RPC fails to work. Hence, the coordinators need to use one-sided RDMA to bypass remote CPUs.

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:

1) As shown in Fig. 1, due to no locks in the execution phase, the intersected data between read and write sets (i.e., read-write set = {A, B}) are operated in execution, locking, and validation phases, which consume 3 round trip times (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 [29]. Hence, these round trips widely exist in read-write dtxns, 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 [60] merges the locking and validation phases, but still consumes an RTT to validate the read-write set.

2) Fig. 1 shows that existing systems [19, 29, 44, 60] consume 2 RTTs to first write backups (redo logs) and then write primaries (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 co-locate with a primary or backup, and hence the local commit can save an RTT. But in the disaggregated architecture, the compute pool does not store any replica. Hence, each read-write dtxn inevitably spends 2 RTTs to commit, which incurs high latency.

Moreover, prior systems work on DRAM+SSD. FaRM [19] and DrTM [61] regard the battery-backed DRAM as PM, but the bandwidth and persistence properties of real PM are overlooked, causing inefficiency on the disaggregated PM:

1) Prior systems [9, 19, 44, 60] 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 [44]. As a result, all requests from coordinators are sent to the primary, causing a high load on the primary's

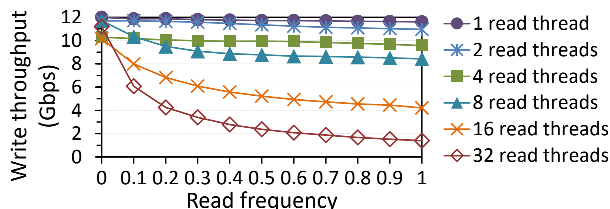


Figure 2: The throughput of RDMA writes to remote PM when mixing different frequencies of RDMA reads, e.g., 0.5 means that 5 reads are mixed with every 10 writes.

PM. However, PM shows lower write bandwidth than the new generations of RNICs, as mentioned before. We use 128GB Optane PM DIMMs and ConnectX-5 RNIC with 100Gbps InfiniBand to evaluate the throughput of RDMA writes when mixing different frequencies of RDMA reads. As shown in Fig. 2, 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.

2) Lack of remote persistency guarantee. Current RDMA verbs have no persistency semantic [17]. For RDMA writes, the data are first buffered in a volatile cache in remote RNIC, which acknowledges (ACK) the writes once validated [59]. Hence, even if the client receives all ACKs, some data may not be persisted to 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 due to using 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. Our paper proposes FORD, an efficient one-sided RDMA-based dtxn processing system for the new disaggregated PM architecture.

3 The FORD Design

3.1 Overview

Fig. 3 shows the overview of FORD. The compute blades run dtxns and access application data in PM blades. The compute and PM pools communicate using connection managers (CMs), which maintain the RDMA queue pair connections.

FORD's workflow contains two stages. 1) 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 [57, 72], and then load database (DB) tables. The DB tables are organized by indexes (§ 4.1). ❷ The compute and PM pools build RDMA connections using CMs. To calculate the remote address for one-sided RDMA in the compute pool, the CM in PM pool sends the metadata of all the indexes to each compute blade. These metadata only consume several MBs and are buffered in the compute pool (§ 4.1). Moreover, each memory blade notifies the compute blade about the roles (i.e., primary or backup) of its stored tables, so that the coordinator can correctly access the data during processing. 2) The

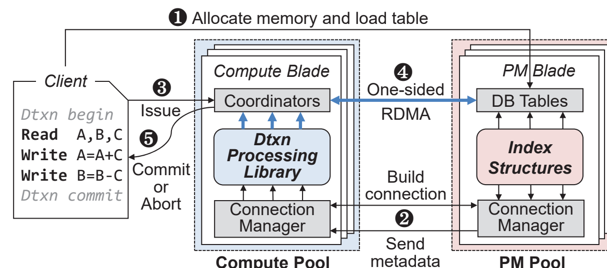


Figure 3: The system overview of FORD.

`Run` stage: ❸ The clients issue substantial dtxns to the compute blades, which spawn threads as coordinators to leverage our runtime library for fast dtxn processing. This library contains our novel designs in § 3.2–§ 3.5, and exposes easy-to-use interfaces (§ 4.2). 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 blades. ❺ After processing, the coordinators report "tx_commited" or "tx_aborted" to clients. The `Init` stage performs only once before the `Run` stage, and the weak compute units in PM pool are not involved in the `Run` stage.

3.2 Hitchhiked Locking

As analyzed in § 2.3 and shown in Fig. 4a, prior works consume 3 RTTs to separately read, lock, and validate data to process a general read-write dtxn in Fig. 3.

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 using the CPUs in 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 [28] 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 shown in Fig. 4b. The RDMA `CAS` first tries to lock the remote data, and RDMA `READ` further fetches the data. Since the transport mode is *reliable connection*, the two RDMA operations are reliably delivered to the remote RNIC in order [51]. Then the batched operations are executed by RNIC as the delivering order to ensure 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 success. If the locking fails, the coordinator aborts the dtxn and unlocks the previously locked data to avoid deadlocks. Fig. 4c shows our hitchhiked locking scheme, which locks and reads the read-write data (e.g., {`A`, `B`}) using one-sided RDMA in one round trip, thus reducing the latency.

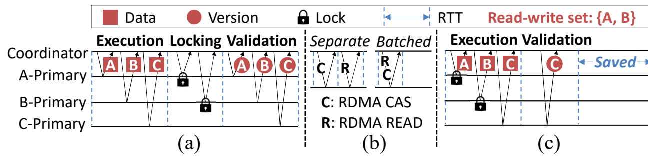


Figure 4: (a) Legacy schemes serially read, lock, and validate the read-write set. (b) Adopting doorbell batching reduces the number of round trips. (c) Our hitchhiked locking avoids extra round trips for locking and validating the read-write set.

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 read-only data (e.g., data C in Fig. 4c) are not locked in the execution phase, and the locked read-write data can still be read by other coordinators (but cannot be locked). There is a validation phase to guarantee the version correctness for the read-only data. If a dtxn does not have the read-only data, the validation is eliminated.

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 (§ 4.1). 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 first read. After reading, the coordinator obtains the remote data addresses and buffers them in its local cache. Each time the previous data are read again, the local 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 places), 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 [18, 19, 44] and DrTM+H [60], that consume a dedicated RTT to lock data. 2) DrTM+R [9], that exclusively locks all the data in the read and write sets. 3) FaSST [29], that 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 above systems that support OCC and primary-backup replication, we summarize the *lock duration*: 1) **FaRM** [18, 19, 44]. 4 phases = lock + validate + commit backup + commit primary&unlock. 2) **FaSST** [29]. 5 phases = lock + validate + log + commit backup + commit primary&unlock. 3) **DrTM+R** [9]. 4 phases = lock + validate + update + unlock. 4) **DrTM+H** [60]. 3 phases = lock&validate + commit backup + commit primary&unlock. 5) **Our FORD**. 3 phases (or 4 phases) w/o (or w/) read-only data = read&lock read-write set (or + validate read-only set) + commit all replicas (§ 3.3) + background unlock (§ 3.3.2).

Though our lock duration experiences 4 phases w/ read-only 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 [9, 19, 60]. This trade-off is beneficial for improving the transaction throughput.

3.3 Coalescent Commit

As analyzed in § 2.3 and shown in Fig. 5a, existing dtxn systems spend 2 RTTs to separately write redo logs to the backup and then update the primary to finish commit. Hence, if the primary crashes, the dtxn can roll forward by using the redo logs in backups. However, this incurs high network overheads on the disaggregated PM, since each read-write dtxn needs 2 RTTs to replicate the updated data.

To reduce latency, FORD proposes a *coalescent commit* protocol to update the primaries and backups together in only one round trip. 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 (i.e., 2 RTTs + roll forward, or 1 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 common cases, 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 [19, 44] and DrTM+H [60]. 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.

3.3.1 Parallel Undo Logging

In general, it is challenging to in-place update the backups and primaries in one round trip. Because in case of a crash, the remote old data could 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, i.e., not involving the remote CPU to move data.

The next question is how to send undo logs to remote replicas. One solution is to spend a dedicated round trip to

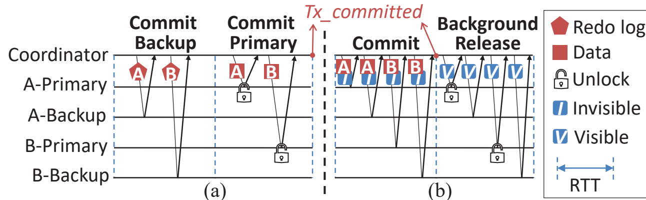


Figure 5: The comparisons between (a) separate commit, and (b) coalescent commit.

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, e.g., the `New Order` transaction in TPCC [15].

3.3.2 Visibility Control

In order to ensure consistency, our coalescent commit protocol guarantees that the data that being updated in the replicas are not partially read. Since our hitchhiked locking scheme does not block read-only 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 Fig. 5b. The idea is to batch an invisible request followed by an RDMA `WRITE` into one request to update the remote replicas: 1) 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`. 2) The RDMA `WRITE` in-place updates the remote replica. After receiving *all* ACKs, the coordinator reports "tx_committed" to clients. Otherwise, e.g., 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 round trip time (RTT) or can be fully overlapped: 1) 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. 2) 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.

3.4 Backup-enabled Read

As discussed in § 2.3, 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 read-only (RO) data, i.e., the coordinators are allowed to read the RO data from backups. 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 the RO data that are read from backups. 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 validates the versions of all `dtxn1`'s RO data before `dtxn1` commits, as guaranteed by our hitchhiked locking scheme in Fig. 4c. If the validation fails, the coordinator aborts `dtxn1` to ensure correctness.

Existing systems unfortunately fail to efficiently read data from backups: 1) For legacy database systems, e.g., Microsoft Azure [16] and Amazon Aurora [56]. 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. Moreover, 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. 2) For prior RDMA-based dtxn systems [19, 44, 60]. The coordinator writes updated data to the primaries (i.e., in-place updates) and backups (i.e., redo logs) 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 PM pool during dtxn processing. Unlike the above systems, our coalescent commit protocol in-place updates the backups and primaries together without involving the CPU in 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 primary's PM to improve the throughput.

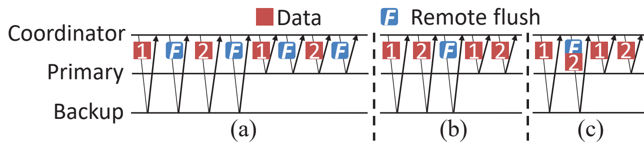


Figure 6: Ensuring remote persistency using (a) full flush, (b) selective flush, and (c) selective flush with request batching.

3.5 Selective Remote Flush

It is important to guarantee the remote persistency when committing the data updates to the PM pool, which is however overlooked in prior dtxn systems that use DRAM as the memory. Recently, the one-sided RDMA `FLUSH` [23, 48] is being proposed to persist data from remote RNIC to PM. However, flushing each data to each remote replica (i.e., full flush) consumes many round trips. As shown in Fig. 6a, updating 2 data incurs 8 round trips after using remote flushes.

In order to guarantee remote persistency with low network overhead, we propose a *selective remote flush* scheme, as shown in Fig. 6b. The idea is to issue an RDMA `FLUSH` after the final RDMA `WRITE` and only to backups, since: 1) 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. 2) In the $(f+1)$-way primary-backup replication, 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 [19]. FORD guarantees remote persistency with 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 [48], we leverage one-sided RDMA `READ`-after-`WRITE` to flush the data in RNIC to memory like [27, 31]. 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`, i.e., replacing RDMA `READ` with RDMA `FLUSH`, as shown in Fig 6c. 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, e.g., using the future RDMA `FLUSH` primitive or current `READ`-after-`WRITE` method.

3.6 Failure Tolerance

The replica fails in 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: i) Before commit, the coordinator aborts the transaction and unlocks the data. ii) During commit, the coordinator aborts the transaction, reads the remote undo logs to revoke data updates and unlocks data. iii) In the release phase, the transaction has already committed. The coordinator clears the `VLock` in the replicas. If some replicas that 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 compute pool. Due to writing undo logs to remote replica, FORD handles coordinator failures by rolling back dtxns. Like FaRM [19] and DrTM [61], FORD supports to use leases [22] to detect failures. After the leases expire (e.g., 5 ms [19]), a failure possibly occurs. However, once 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.

The network communication fails. Due to network partitions, either availability or consistency is sacrificed based on the CAP theorem [6, 21]. In this case, FORD only allows the primary partition [5] to serve requests, which guarantees the strong consistency of ACID dtxns for OLTP workloads.

3.7 Put It All Together

Fig. 7 illustrates how our designs (§ 3.2–§ 3.5) work together to process dtxns by using one-sided RDMA primitives. 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 read-only 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 read-only data (if any) using RDMA `READ`, and verifies that the data versions are not modified by other coordinators. If a version changes, the coordinator aborts the dtxn and unlocks the remote data.

3) Commit. After validation, the coordinator checks that all the ACKs of undo logs are received, and then commits the updated data to all the replicas in one round trip. The data in primaries are marked to be invisible and updated with the 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. Afterwards, the coordinator starts processing the next dtxn.

4) Release. After the dtxn commits, the coordinator uses RDMA `CAS` to release the remote data by setting them visible and unlocking them in the background.

FORD efficiently handles different types of dtxns. 1) For read-only dtxns, FORD reads remote data and validates ver-

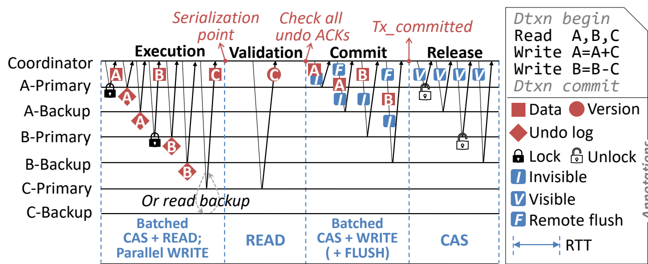


Figure 7: Distributed transaction processing in FORD.

sions before commit. Prior systems [19, 60] adopt similar operations. However, the difference is that FORD supports coordinators to read backups to improve the throughput while prior systems do not support this. 2) For read-write dtxns, only 2 RTTs (i.e., Execution+Commit, w/o read-only data) or 3 RTTs (i.e., Execution+Validation+Commit, w/ read-only data) are on the critical path. Compared with existing designs that require 5 RTTs [19,29] or 4 RTTs [60] to process a read-write dtxn (as analyzed in § 2.3), FORD significantly improves the performance for the disaggregated PM architecture.

3.8 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 read-only 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*, i.e., 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 dtxns: (1) **A**tomicity. FORD records undo logs, which are used to revoke the partial updates if a failure occurs before commit. (2) **C**onsistency. All the data versions are consistent before the dtxn starts and after it commits. (3) **I**solation. FORD uses locks and version validations to guarantee the serializability among the read-write and read-only dtxns. (4) **D**urability. The updated data are persistently stored in PM after commit.

The Number of RDMA Operations. Due to fully using one-sided RDMA to bypass the CPUs in 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 [52]) are efficient to handle one-sided RDMA operations including `CAS` [60]. Hence, slightly increasing the number of RDMA operations has negligible impacts on performance. In fact, FORD focuses on reducing the number of RDMA round trips, which is more important to improve the performance since the RDMA round trip still suffers from higher latency (e.g., 3–8 µs [3]) than local access (e.g., 62–305 ns [67]).

4 Implementations

4.1 Data Store in Memory Pool

FORD supports different indexes to organize database (DB) tables in PM pool, e.g., 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 round trips 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 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 [18, 54, 61, 72], we use 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, e.g., RACE hashing [72]. 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, and 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. Like FaSST [29], the record consists of an 8B key and a maximum sized value (e.g., 1KB). Such record meets many OLTP workloads (e.g., TPCC [15]). 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): DB table that this record belongs to.
- `Version` (8B): 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 PM pool, FStore enables clients to load records into hash tables before running dtxns. Fig. 8 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 a 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 MR. The size of 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 RS. Moving the RS-Ptr forward to a bucket size will generate a new bucket in RS. If the RS is exhausted but the hash collision still occurs, the client re-allocates memory to load tables.

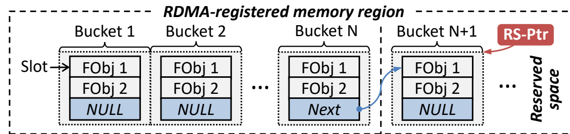


Figure 8: The hash table structure in FStore.

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

- `TableID` (8B): Global unique database table id.
- `Addr` (8B): Virtual start address of this hash table.
- `Off` (8B): Offset between `Addr` and MR's start address.
- `BucketNum` (8B): Bucket number of the hash table.
- `BucketSize` (4B): Size of a bucket (in bytes).
- `SlotNum` (4B): Number of slots per bucket.

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`

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. Or else go to S5.

S5: If the `next` field of *bkt* is `NULL`, there is no such remote record. Then go to S8. Or else go to S6.

S6: Calculate the next bucket offset as below and go to S3.
`bucket_off` = `bkt.next` − `Addr` + `Off`

S7: Exit if the record is visible. Or else re-read it until 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 in Fig. 15a. Caching metadata is scalable, because the compute blades do not need to synchronize their metadata with each other: 1) The metadata of index does not change. 2) If the cached addresses are stale, FORD enables the coordinator to detect this and update its own cached addresses, as discussed in § 3.2.

4.2 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 read-only set.
- `AddRW`: Add an initialized FObj to the read-write set.
- `TxExecute`: The coordinator reads the remote data specified in read-only and read-write sets, and then executes the dtxn logic. Our hitchhiked locking and backup-enabled read schemes are leveraged.
- `TxCommit`: After execution, the coordinator commits the updated data to remote primaries and backups using our coalescent commit and selective remote flush schemes.

```
1  bool WriteCheck(uint64_t dtxn_id, DTXN* dtxn) {
2    // The `dtxn` invokes FLib interfaces
3    dtxn->TxBegin(dtxn_id);
4    // Use a random account as the key
5    uint64_t acct_id = RandomAccount();
6    FObj* sav_obj = new FObj(SavingsTableID, acct_id);
7    FObj* chk_obj = new FObj(CheckingTableID, acct_id);
8    dtxn->AddRO(sav_obj);
9    dtxn->AddRW(chk_obj);
10   if (!dtxn->TxExecute()) return false;
11   // Get record values and run transaction logic
12   sav_val_t* sav = (sav_val_t*) sav_obj->value;
13   chk_val_t* chk = (chk_val_t*) chk_obj->value;
14   if (sav->balance + chk->balance < PredefinedAmount)
15     chk->balance -= (PredefinedAmount + 1);
16   else chk->balance -= PredefinedAmount;
17   bool status = dtxn->TxCommit();
18   delete sav_obj; delete chk_obj;
19   // Report commit (true) or abort (false) to client
20   return status;
21 }
```

Figure 9: The example of C++ code using FLib interfaces.

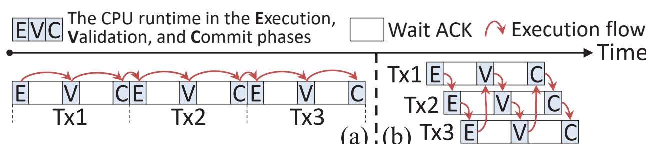


Figure 10: The comparisons between (a) sequential processing, and (b) interleaved processing, in one thread.

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. Fig. 9 illustrates an example of using our interfaces in the `Write Check` transaction of the SmallBank benchmark [50]. This transaction reads the balances from the `Savings` and `Checking` tables, and updates the balance in the `Checking` table. It shows that our interfaces are easy-to-use.

4.3 Interleaved Transaction Processing

As shown in Fig. 10a, sequentially processing dtxns in a thread wastes the CPU cycles due to waiting for RDMA ACKs, which significantly decreases the throughput. 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 pipeline, as presented in Fig. 10b. In this way, the network RTTs are efficiently hided and the CPU cores in the compute pool are fully utilized to improve the throughput.

We use coroutines [29, 60] 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 in Fig. 16 show that using a proper number of coroutines improves the throughput without heavily increasing the latency.

5 Performance Evaluation

5.1 Experimental Setup

Testbed. We use three machines, each of which contains a 100Gbps Mellanox ConnectX-5 IB RNIC. They are connected via a 100Gbps Mellanox SB7890 IB switch. One machine equipped with the Intel Xeon Gold 6230R CPU and 8GB DRAM is leveraged as the compute pool to run coordinators. Other two machines form the PM pool, each of which contains 6 interleaved 128GB Intel Optane DC PM modules. Each database table is stored in the two PM machines to maintain a 2-way replication, i.e., one primary and one backup.

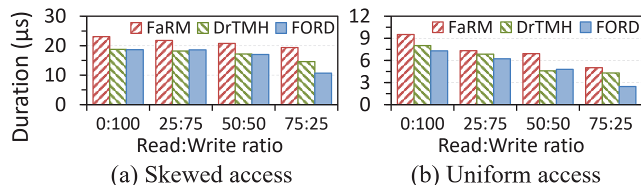
Benchmarks. We leverage a key-value store (`KVS`) as the ***micro-benchmark*** to analyze how different factors affect each design of FORD. `KVS` stores 1 million key-value pairs in one table, in which the key is 8B and value is 40B. The transaction in `KVS` accesses a specific number of objects with different read:write ratios and different access patterns as configured in § 5.2. `KVS` supports the skewed and uniform access patterns, in which the skewed access uses the Zipfian distribution with the default skewness 0.99 [10]. We further adopt three OLTP benchmarks, i.e., `TATP` [49], `SmallBank` [50], and `TPCC` [15], as the ***macro-benchmarks*** to examine the end-to-end performance. These benchmarks are widely used in prior studies [19, 29, 60, 61]. `TATP` models a telecom application and contains 4 tables, in which 80% of the transactions are read-only, and the record size is up to 48B. `SmallBank` simulates a banking application that includes 2 tables, in which 85% of transactions are read-write, and the record size is 16B. `TPCC` models a complex ordering system that consists of 9 tables, in which 92% of transactions are read-write, and the record size is up to 672B. We generate 8 warehouses in `TPCC`. We have implemented all the workloads of each macro-benchmark and run the standard transaction mix in § 5.3. 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 implement FORD[1] using C++ (13.1k lines of codes) and compare it with two state-of-the-art RDMA-based dtxn processing systems, i.e., FaRM [19] and DrTM+H [60] (called DrTMH). We use one-sided RDMA to re-implement their dtxn processes for the disaggregated PM. Moreover, our selective remote flush scheme is applied to FaRM and DrTMH to make them compatible with remote PM. We do not compare against FaSST [29] that fully uses two-sided RDMA, which is difficult to work in the disaggregated memory architecture due to consuming the remote CPUs in memory pool throughout the entire dtxn processing.

5.2 Micro-Benchmark Results and Analysis

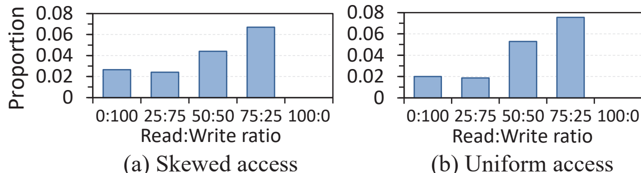
Lock Duration. Locks are generally used to serialize dtxns. However, a long lock duration causes frequent aborts and

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



(a) Skewed access (b) Uniform access

Figure 11: The lock durations.



(a) Skewed access (b) Uniform access

Figure 12: The proportions of invisibility durations.

leads to low throughputs. To obtain the lock duration, we configure the coordinator to not abort dtxn but 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. Fig. 11 shows the average lock duration of each coordinator at different read:write ratios in the dtxn mix, e.g., 25:75 means that 25% of the dtxns are read-only 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 4 phases, i.e., 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, 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.

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, and then calculate the proportion of the invisibility duration in the entire dtxn running time. As shown in Fig. 12, 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 read-only dtxns that wait for the data to be visible, which increases the total invisibility durations in 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

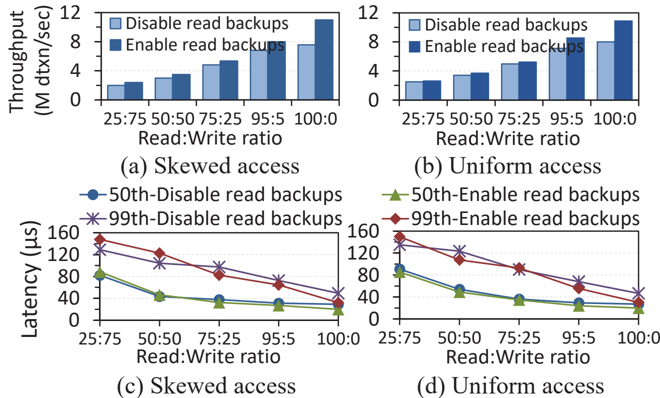


Figure 13: The dtxn throughput and latency when disabling/enabling coordinators to read the backup replicas.

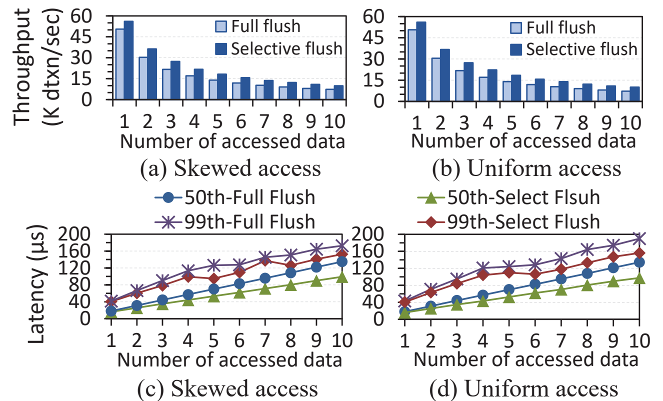


Figure 14: The dtxn throughput and latency using full/selective flush when accessing different numbers of data per dtxn.

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.

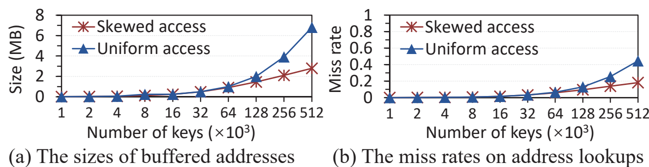
Read from Backups. Due to the limited write bandwidth of PM, FORD enables the coordinators to read the read-only data from backups to alleviate the load on the primary's PM. To demonstrate the benefits of this design, we run 224 coordinators to increase the load, and examine the dtxn throughput and latency when disabling/enabling the coordinators to read backups. Fig. 13 shows 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 backup nodes, FORD will gain higher performance improvements since all the backups can be read to balance loads.

Remote Flush. FORD guarantees the remote persistency in dtxn processing by flushing the data from the RNIC cache to PM. We compare the dtxn throughput and latency when adopting the full flush and selective flush schemes discussed in § 3.5. To show the overheads of remote data flush, we use one coordinator to avoid extra contention overheads. We increase the number (1–10) of written data per dtxn to add the flush operations. The results in the skewed and uniform accesses exhibit similar trends. Fig. 14a and 14b show that our selective flush scheme improves the throughput by 28.7%/29.5% over the full flush scheme in skewed/uniform access. Fig. 14c and 14d show 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, Fig. 14 shows 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.

Local Cache. The coordinator has a local cache to buffer remote data addresses for efficient one-sided RDMA operations. To evaluate the overheads (including the size and miss rate) of the local 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. Fig. 15a shows that the buffered addresses only consume 6.8 MB even if uniformly accessing 512k keys with poor locality. Hence, a small MB-scale cache is sufficient for a coordinator to buffer remote addresses. Since a GB-scale DRAM is leveraged in the compute pool to store the metadata [45], it is unnecessary to limit the size of the coordinator-local cache in practice. Fig. 15b shows that the miss rate is 18.2%/44.6% when accessing 512k keys in skewed/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 remote bucket address and read a bucket to find the record, which 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 cache for each coordinator to avoid evicting the buffered addresses from the cache, thus reducing the miss rate as much as possible.

5.3 Macro-Benchmark Results and Analysis

Coroutine Execution. To improve the throughput, FORD leverages coroutines to process dtxns to avoid CPU idling. A thread generates at least 2 coroutines since a specified coroutine in each thread is used to poll the RDMA ACKs. Fig. 16 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. On the other hand, 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 learn that using 6–8 coroutines is helpful to significantly improve the throughput without heavily increasing the latency.



(a) The sizes of buffered addresses (b) The miss rates on address lookups

Figure 15: The size and miss rate of the local cache.

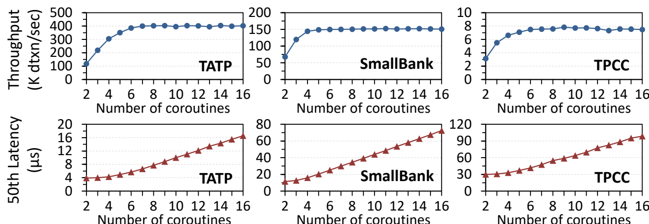


Figure 16: The dtxn throughput and 50th percentile latency in one thread when using different numbers of coroutines.

End-to-End Performance. We use 16 threads and each thread generates 8 coroutines (1 coroutine polls ACKs), as a total of $16 \times (8-1) = 112$ coordinators, to evaluate the end-to-end throughput and latency in FaRM, DrTMH, and FORD using the macro-benchmarks. In real experiments, due to different system scales (e.g., 90 machines [19]), the overall throughput in our small-scale testbed becomes lower than [19, 60]. However, our testbed can accurately evaluate the performance in different system configurations. Our selective remote flush scheme is applied to three systems to ensure remote persistency. As shown in Fig. 17, compared with FaRM/DrTMH, FORD improves the throughput by 1.4×/1.3×, and reduces the 50th (99th) latency by 12%/9.1% (54.8%/46.8%) in `TATP`, improves the throughput by 1.6×/1.3×, and reduces the 50th (99th) latency by 34.3%/30.9% (64.6%/32.4%) in `SmallBank`, and improves the throughput by 2.3×/1.4×, and reduces the 50th (99th) latency by 74.3%/66.2% (63.8%/28.7%) In `TPCC`. DrTMH outperforms FaRM by merging locking and validation phases. FaRM and DrTMH show high performance in `TATP`, since 80% of the dtxns are read-only and the uses of one-sided RDMA `READ`s accelerate the 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.

6 Related Work

Fast Distributed Transactions. Many systems have been proposed for efficient distributed transaction processing. Some designs leverage RDMA to handle transactions [9, 18, 19, 29, 31, 41, 44, 60, 61]. Storm [41] proposes a transactional API to operate remote data based on one-sided reads and write-based RPCs. HyperLoop [31] offloads some computations to RNIC and requires remote CPU to operate the metadata. Moreover, application locality [7, 30, 37] is exploited to convert a distributed transaction to a local one, which however sacrifices the generality. New transaction abstractions [63], replication protocols [70], and concurrency controls [40, 58, 64, 69] are

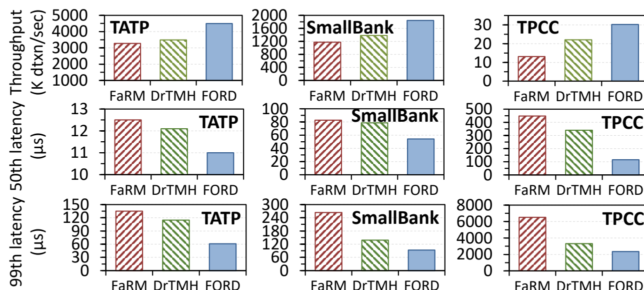


Figure 17: The end-to-end performance.

also proposed to improve the performance. The above systems work on the monolithic architecture, while our FORD focuses on the new disaggregated PM architecture and fully leverages one-sided RDMA to process transactions.

Distributed Persistent Memory. 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 [3, 38, 65, 66] and memory management [46, 71]. Some studies provide optimization hints on system implementations when using RDMA and PM [27, 62]. 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.

Disaggregated Memory. The disaggregated memory becomes popular in datacenters due to high resource utilization and elasticity. Existing works explore memory disaggregation in hardware architectures [35, 36], networks [20, 47], operating systems [45], KV stores [54], hash indexes [72], data swapping [2, 8, 24, 43], and memory managements [1, 34, 39, 55, 57]. Our proposed FORD is orthogonal to these systems to build a fast transaction processing system for the disaggregated PM.

7 Conclusion

Our paper proposes FORD, a fast distributed transaction processing system that leverage one-sided RDMA for the new disaggregated persistent memory (PM) architecture. To accelerate transaction processing, FORD explores and exploits the request batching and parallelization to eliminate extra locking and validations, 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. Experimental results demonstrate that FORD significantly outperforms the state-of-the-art systems in terms of transaction throughput and latency.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant 62125202. We are grateful to our shepherd, Rong Chen, and anonymous reviewers for their feedbacks and suggestions.

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 775–787. USENIX Association, 2018.

[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 14:1–14:16. ACM, 2020.

[3] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostic, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1011–1027. USENIX Association, 2020.

[4] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

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

[6] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

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

[8] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 79–92. ACM, 2021.

[9] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 26:1–26:17. ACM, 2016.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[11] Hewlett Packard Corporation. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/, 2021.

[12] Intel Corporation. Intel optane persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2021.

[13] Intel Corporation. Intel optane persistent memory 200 series (512gb pmem) module. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory/optane-persistent-memory-200-series-512gb-pmem-module.html, 2021.

[14] Intel Corporation. Intel rack scale design architecture. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html, 2021.

[15] The Transaction Processing Council. Tpc-c benchmark. http://www.tpc.org/tpcc/, 2021.

[16] Azure SQL Database. Use read-only replicas to offload read-only query workloads. https://docs.microsoft.com/en-us/azure/azure-sql/database/read-scale-out, 2021.

[17] Chet Douglas. Rdma with pm: Software mechanisms for enabling persistent memory replication. In *Storage Developer Conference (2015)*, 2015.

[18] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[19] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70. ACM, 2015.

[20] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.

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

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

[23] Paul Grun, Stephen Bates, and Rob Davis. Persistent memory over fabrics. In *Persistent Memory Summit (2018)*, 2018.

[24] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 649–667. USENIX Association, 2017.

[25] Doug Hakkarinen, Panruo Wu, and Zizhong Chen. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1323–1335, 2015.

[26] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. Nysiad: Practical protocol transformation to tolerate byzantine failures. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 175–188. USENIX Association, 2008.

[27] Anuj Kalia, David G. Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 105–119. ACM, 2020.

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 185–201. USENIX Association, 2016.

[30] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: locality-aware distributed transactions. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 145–161. ACM, 2021.

[31] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 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 2018, Budapest, Hungary, August 20-25, 2018*, pages 297–312. ACM, 2018.

[32] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[33] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 312–313. ACM, 2009.

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

[35] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 267–278. ACM, 2009.

[36] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 189–200. IEEE Computer Society, 2012.

[37] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1659–1674. ACM, 2016.

[38] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 773–785. USENIX Association, 2017.

[39] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 757–773. ACM, 2020.

[40] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 479–494. USENIX Association, 2014.

[41] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pages 97–108. ACM, 2019.

[42] Vmware Research. Remote memory. https://research.vmware.com/projects/remote-memory, 2021.

[43] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: high-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.

[44] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.

[45] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.

[46] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 323–337. ACM, 2017.

[47] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki-Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 255–270. USENIX Association, 2019.

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

[49] TATP. Telecom application transaction processing benchmark. http://tatpbenchmark.sourceforge.net/, 2011.

[50] The H-Store Team. Smallbank benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/, 2021.

[51] Mellanox Technologies. Rdma aware networks programming user manual. rev 1.7. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015.

[52] Mellanox technologies. Connectx-5 vpi card. https://www.mellanox.com/files/doc-2020/pb-connectx-5-vpi-card.pdf, 2020.

[53] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pages 51–62. IEEE Computer Society, 2008.

[54] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 33–48. USENIX Association, 2020.

[55] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 306–324. ACM, 2017.

[56] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1041–1052. ACM, 2017.

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

[58] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: High-performance transactions via learned concurrency control. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 198–216. USENIX Association, 2021.

[59] Live Webcast. Extending rdma for persistent memory over fabrics. In *SNIA Networking Storage Forum (2018)*, 2018.

[60] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 233–251. USENIX Association, 2018.

[61] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 87–104. ACM, 2015.

[62] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and optimizing remote persistent memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 523–536. USENIX Association, 2021.

[63] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 495–509. USENIX Association, 2014.

[64] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 279–294. ACM, 2015.

[65] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 221–234. USENIX Association, 2019.

[66] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Filemr: Rethinking RDMA networking for scalable persistent memory. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 111–125. USENIX Association, 2020.

[67] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.

[68] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[69] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 511–526. ACM, 2020.

[70] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 263–278. ACM, 2015.

[71] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 3–18. ACM, 2015.

[72] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 15–29. USENIX Association, 2021.