Fast, Highly Available, and Recoverable Transactions on Disaggregated Data Stores

Mahesh Dananjaya*
Huawei Research
Edinburgh, United Kingdom
Mahesh.Dananjaya@huawei.com

Vasilis Gavrielatos
Huawei Research
Edinburgh, United Kingdom
Vasilis.Gavrielatos@huawei.com

Antonios Katsarakis
Huawei Research
Edinburgh, United Kingdom
Antonios.Katsarakis@huawei.com

Nikos Ntarmos
Huawei Research
Edinburgh, United Kingdom
Nikos.Ntarmos@huawei.com

Vijay Nagarajan
University of Utah
Utah, USA
vijay@cs.utah.edu

ABSTRACT

Memory Disaggregation decouples memory from traditional datacenter servers, offering a promising pathway for achieving very high availability in transactional in-memory disaggregated Key-Value Stores (DKVSes). Achieving such availability hinges on transactional protocols that can efficiently handle failures in this setting where compute and memory are independent.

However, existing transactional protocols overlook the scenario where compute and memory fail independently. Exacerbating the problem, memory disaggregation relies on memory nodes with limited compute capacity, requiring one-sided RDMA-style protocols instead of traditional RPC-based approaches. This significantly complicates achieving a correct and recoverable protocol due to the limited semantics of one-sided RDMA. Moreover, the only state-of-the-art one-sided transactional protocol has overlooked recovery, jeopardizing correctness and performance.

We present Pandora, the first one-sided transactional protocol that is specifically designed to enable fast and correct recovery on disaggregated KVSes. Pandora's fast recovery hinges on two innovations: (a) the PILL (Pandora's Implicit Lock Logging), a novel technique for managing locks in the presence of compute failures; and (b) an RDMA-based recovery algorithm that detects and quickly recovers from failures. To validate that Pandora recovers correctly in the presence of failures, we introduce a new litmus-testing framework for end-to-end validation of transactional protocols. Our evaluation (and validation) reveals that Pandora achieves fast and correct recovery in the range of a few milliseconds without compromising the performance of failure-free runtime execution.

1 INTRODUCTION

Disaggregated Memory (DM) decouples application memory from datacenter servers and aggregates it into a network-attached memory pool [9, 31, 48, 61]. In this work, we explore the recovery of transactional in-memory key-value stores over disaggregated memory (DM) architecture. Transactions play a crucial role in modern cloud and HPC ecosystems, offering rich programmability and high performance in contrast to traditional global-checkpoint-based recovery [17, 29, 38, 53]. We argue that existing work has not studied how to recover correctly and efficiently under disaggregated memory. To address that, we propose a novel approach to tackling recovery in this new setting. But first, let us provide some context.

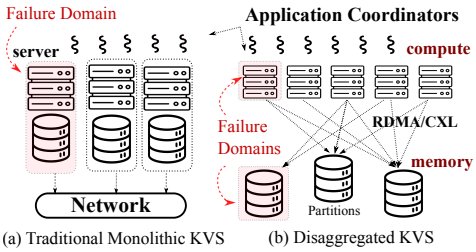

Figure 1: Independent failures in Disaggregated Memory. Because memory and compute are independent in DM, distributed applications could keep running even if a compute server fails.

Both academia [31, 32, 43, 46, 50, 57, 60, 61, 67, 69] and industry [5–7, 16, 18, 48] are exploring DM to mitigate the inefficiency caused by the fixed compute-to-memory ratio in traditional datacenter servers. This inefficiency arises when an application needs more compute than memory, or vice versa, but the server's fixed ratio doesn't match its requirements. DM decouples memory from compute by deploying two types of servers: 1) *memory servers* that provide ample memory with minimal compute and 2) *compute servers* that offer high compute capabilities with minimal memory. Memory servers are connected to compute servers through a high-speed RDMA network (or possibly through emerging CXL [31, 48]), which allows memory and compute to be efficiently provisioned according to the needs of the application. This one-sided communication enables dynamic scaling and elastic memory management, reducing cost and power consumption by minimizing reliance on expensive CPUs.

The primary advantage of DM is improved resource utilization, but there is another crucial benefit. With memory and compute now operating independently, distributed applications can keep running even if a compute server fails, since no memory is lost (Figure 1). Simply, recovery from compute failures can be *non-blocking*. This is in contrast to a traditional monolithic server architecture, where a server failure results in the automatic loss of a portion of memory.

In this work, we leverage this observation in the context of transactional in-memory key-value stores (dubbed KVSes). Such KVSes are a crucial cloud infrastructure [22, 23, 37, 41], and their availability in the face of faults is critical. Typically, when a KVS server fails, a portion of the objects becomes inaccessible. Although objects are replicated, the entire KVS must stop briefly to – at least – reconfigure itself and steer requests for the inaccessible objects to other replicas. In KVSes the ratio of compute (transactions per second) and memory (dataset size) can vary arbitrarily across the numerous use-cases. This makes them a perfect fit for DM.

*This work was conducted while the author was at the University of Edinburgh.

In DM-KVSes (dubbed DKVSes), the compute servers are solely responsible for coordinating transactions, while memory servers hold the dataset passively. In this architecture, there is no reason why the failure of a compute server should interrupt the operation of the DKVS.

Unfortunately, there is no existing DKVS that offers this capability. Providing this capability will require a new DM-based recovery protocol. The key challenge in designing this protocol is that compute servers can only access memory through the limited one-sided RDMA API (read, write, compare-and-swap, and fetch-and-add). This is in contrast to traditional, non-disaggregated architectures where servers can send arbitrary remote procedure calls (RPCs) to one another.

Over the past decade, researchers have studied this RPC-to-RDMA transformation in the context of transactional protocols [22, 41, 54, 74]. FaRM [22, 23] showed how to implement the execution phase of a transactional protocol over one-sided RDMA without RPCs. However, FaRM's commit phase relies on RPCs. FORD [74] took the next step, designing the execution, validation, and the commit/abort phases through one-sided RDMA. Problematically, none of the above works have addressed the problem of recovering from a compute failure without interruption. Exacerbating the issue, recovery in RPC-based protocols [23] necessitates strong CPUs on the memory side, undermining the benefits of disaggregation, and does not guarantee optimal performance for DKVSes [24, 74]. Meanwhile, weak CPUs struggle with RPC overhead due to the increasing gap between network speeds and CPU power, impacting latency and throughput.

There are two key challenges in coming up with a one-sided protocol for efficiently recovering from compute failures. First, the failing compute node may have grabbed locks and efficiently identifying and cleaning up these locks during recovery is challenging. Second, ensuring a correct recovery algorithm without compromising steady-state performance is inherently challenging. This difficulty becomes even more pronounced in DKVSes, where recovery is constrained by the limited capabilities of one-sided RDMA primitives.

1.1 Pandora: Fast, Seamless and Safe Recovery

We propose Pandora, a fully one-sided transactional protocol that ensures memory is always in a recoverable state and includes special handling of compute failures to avoid unnecessary interruption. Starting with FORD as the steady-state protocol, we design an RDMA-based recovery protocol that detects and recovers from a compute failure while eliminating interruption. To ensure correctness, we introduce an end-to-end litmus testing framework that revealed a number of bugs on FORD, which prohibited recovery from being fast, correct, or non-blocking. Finally, we thoroughly validate and evaluate our proposal in comparison with FORD [74].

Pandora's Implicit Lock Logging (PILL). One problem with recovering after a compute failure is that the failing compute node may have grabbed locks, and cleaning up these locks during recovery is challenging. Should the locks be reinstated to a consistent state, transactions require a mechanism to record the owner of each lock before it is locked, which is typically achieved through logging. The FORD protocol, however, locks keys before logging. Thus, post-failure, the entire memory must be scanned to discover and undo any not-yet-logged locks taken by the failed compute server. This operation can take multiple seconds: e.g., scanning 100 GiBs through a 100Gbps network link will require at least 8 seconds. During this period, all compute servers accessing these locked objects get blocked, adversely impacting the overall performance. This is because, in the absence of owners, recovery would otherwise erroneously unlock keys locked by active compute nodes (Section 3.1.1).

Crucially, to solve this, we cannot simply reorder logging and locking, because that will either require a heavy redesign of the protocol or impose overheads by adding extra messages. Instead, we propose a new RDMA-friendly technique called *Pandora's Implicit Lock Logging* (PILL), where we extend the lock structure to also include the id of the compute server. When failing to lock, a compute server inspects the lock to see if the current server holding the lockis a failed compute server. If so, the lock can be *stolen*.

RDMA-based Recovery Protocol. To detect and handle compute failures, we propose an RDMA-based recovery protocol that works in four steps. First, it uses heartbeats to detect failures. Second, it revokes the RDMA rights of a server deemed failed to ensure safety even under false positives of failure detection. Third, it reads the logs of the failed compute server (which are stored in the memory servers) and either rolls forward or rolls back all of its logged transactions. Finally, it notifies the remaining compute servers of the failure so that they can acquire any stray locks of the failed server. Crucially, during this process, the alive compute servers can seamlessly continue executing transactions. Note that while we rely solely on non-RPC accesses in the data path, RPCs are used to a limited extent in the control path (e.g., to setup and manage network connections) similar to other works over disaggregated memory [24].

End-to-end Litmus Testing. There are several factors that render the recovery protocol particularly error-prone. Firstly, recovery is a complicated distributed algorithm that must be able to detect failures and roll back and forward uncommitted transactions. The recovery is only executed once per failure, limiting the ability to uncover corner cases; contrast this with the rest of the transactional protocol, which is executed millions of times per second. Secondly, it does not suffice for only the recovery protocol to be correct; for recovery to work, the transactional protocol must ensure that memory is always in a recoverable state. However, this aspect of the protocol is not tested during failure-free operation, and thus is especially error-prone.

In this work, we introduce a new litmus-testing framework for end-to-end validation of transactional protocols in general, and Pandora in particular. Litmus tests are small transactions that are designed to expose bugs. To the best of our knowledge, this is the first work to create litmus tests, and a framework for deploying these to validate DKVS protocols. Our validation revealed multiple subtle bugs in the state-of-the-art FORD, which can – in rare cases – leave the memory in an unrecoverable state, all of which have been fixed in Pandora.

1.2 Contributions

- We observe that memory disaggregation from compute presents an opportunity for highly available transactional KVSes. However, the limited one-sided RDMA semantics and the absence of remote procedure calls (RPCs) between compute and memory servers pose a challenge to fast recovery. (§2)
- We propose Pandora, the first one-sided transactional protocol specifically designed to provide correct and fast recoverable transactions on DKVSes. Pandora consists of two innovations: the PILL, an RDMA-friendly technique for

making locks recoverable after failures without compromising the performance of the fault-free operation, and a novel RDMA-based recovery protocol for quickly detecting and safely recovering from failures in the DKVS setting. (§3)
- To validate correctness, we introduce a new litmus-testing framework. Our validation reveals multiple subtle bugs in the FORD protocol, all of which are addressed in Pandora. (§5)
- Our experiments show that Pandora offers orders of magnitude faster recovery (just a few milliseconds) than the state-of-the-art DKVS protocol, while also allowing live compute servers to proceed with their transactions without blocking. Critically, Pandora's recovery mechanism comes at negligible overhead during fault-free operation, unlike naive recovery approaches, which can degrade performance by as much as 35%. (§6)

2 PRELIMINARIES

We begin with a brief background on disaggregated key-value-stores (DKVS). We then briefly discuss FORD– the state-of-the-art DKVS.

2.1 Disaggregated KVS (DKVS)

Researchers from academia and industry are advocating for the adoption of disaggregated memory (DM), arguing that it improves scalability, power utilization and cost efficiency [9, 15, 18, 32, 50, 55, 68, 73]. In a DM architecture, servers are divided into *compute* and *memory*. Compute servers have the compute capabilities of today's commodity servers, but limited memory (i.e., a few GiB) for caching but not in-memory storage. Memory servers have a lot of memory for storage but near-zero compute [46, 67, 74]. As in recent DM works [25, 70], we assume that memory servers have a small set of wimpy cores (1 - 2) to support lightweight connection management and initialization but do not traverse indexes or apply transactional logic. Instead, compute servers perform those over the memory servers through one-sided RDMA.

This paper focuses on Key-Value Stores deployed over DM, or simply DKVSes. Specifically, we focus on DKVSes that replicate and distribute their data *in-memory* across multiple memory servers. A set of compute servers run the DKVS compute-side library, which offers a simple transactional API. Applications express their transactions through requests that include calls to BeginTx, Write, Read, ReadRange, Insert, Delete, and CommitTx.[1] An application can run on the same servers as the DKVS compute-side library or on remote servers. In either case, the applications' requests are routed to the DKVS compute-side library, which executes a *transactional protocol*, accessing and replicating DKVS data as needed. While in this work we focus on non-persistent compute and (replicated in-) memory servers; Pandora, like state-of-the-art DKVSes, is compatible with non-volatile memory (NVM) and can easily support efficient persistence mechanisms (Section 7).

Architecture. The compute server executing the protocol for a transaction is called its *coordinator*. Each object is stored in multiple memory servers. Every object is assigned a *primary* memory server, with the remaining servers designated as *backups*. An object can only be accessed through its primary. The backups are kept consistent with the primary so that they can take over in the event of a failure.

[1] Such systems [22, 74] form the foundation for relational databases and distributed systems. More complex data structures are built on top of this simple, fault-tolerant API.

RDMA Primitives. RDMA offers the following primitives: Read, Write, Send/Receive, Compare-And-Swap (CAS) and Fetch-And-Add (FAA). Send/Receive are used to facilitate RPCs; hence are not useful for DKVSes. Read, Write, CAS and FAA directly access remote memory; for that reason they are called *one-sided*. Programming using these primitives is very challenging as opposed to RPCs [65]. This is because simple algorithms, such as accessing a hashtable, require multiple one-sided RDMAs, each of which includes a non-trivial programming overhead (e.g. polling completions, asynchronous programming to tolerate network latencies etc.). Most critically, the programmer must reason about the possibility of the compute node failing before the algorithm is executed to its completion.

RPCs. As in prior work [24, 74], we do not consider the usage of RPCs in the data path. We assume that the compute on the memory nodes is too slow for it. Admittedly, it is conceivable that memory nodes have more compute than we assume, such that a hybrid design with both one-sided RDMA and RPCs is viable. This paper does not investigate this possibility. Instead, we maintain that memory nodes have slow compute, hence presenting a stronger motivation for DKVSes while facilitating a fair comparison with related work.

Consistency and Failure Model. As in prior works in distributed replicated transactions [22, 37, 71, 74], we focus on transactions that provide the strongest consistency guarantee (i.e., *strict serializability* [58]). We consider a non-byzantine partially synchronous model [27] with crash-stop compute and (up to $f + 1$) memory server failures, as well as network faults, including message reordering, duplication, and loss. In short, our failure model aligns with prior work [74]. We address message loss between compute and memory nodes by leveraging the reliable connection guarantees of one-sided RDMA primitives, which handle retransmissions transparently at the transport layer. Partial synchrony is necessary to safely manage leases, which are used for failure detection (Section 3.2.4).

2.2 Recoverable Transaction Protocol

A recoverable transactional protocol is responsible for ensuring consistency (i.e., strict serializability) and handling failures under the aforementioned failure model. We find it useful to classify the actions of the protocol under three categories.

C1. Online-failure-free. This category encompasses all actions necessary to ensure transaction correctness (i.e., strict serializability) *when there are no faults*.

C2. Online-recovery. This category includes the actions required by the protocol to *maintain the state needed to enable recovery in the event of a failure*. Typically, these actions involve the logging of both data and metadata to preserve transactional integrity.

C3. Recovery. This category covers all protocol actions related to the detection and recovery from a compute server failure. Specifically, it includes mechanisms for detecting failures and preventing the server deemed as failed from further impacting the system (i.e., in the case of a false positive). It also ensures that transactions from the failed server are either fully rolled back or rolled forward to completion.

Key features of a recoverable transactional protocol. An ideal recoverable protocol should have four key features.

(1) **Correctness.** First and foremost, the protocol must ensure correctness (i.e., strict serializability) both in the absence and presence of compute failures.

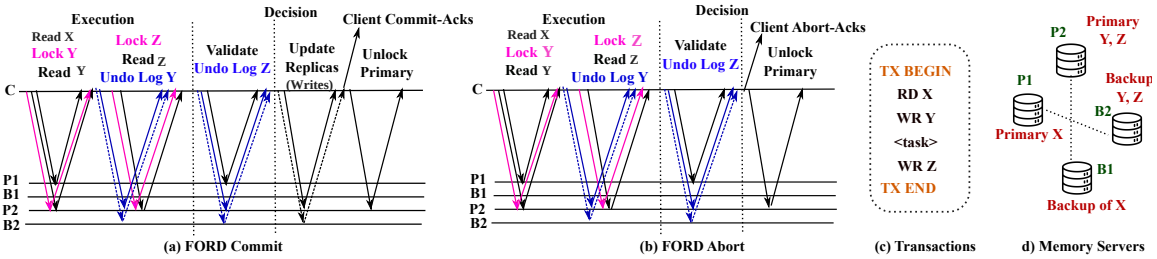

Figure 2: FORD's commit path is shown in (a), and the abort path in (b) – which occurs if any lock or read-validation fails before the decision. In (c), an example transaction reads object X, writes object Y, and then writes object Z. In (d), X has P1 as its primary, while Y and Z have P2. P1 and P2 are replicated in B1 and B2. In (a), the coordinator first reads X from P1, then locks and reads Y from P2, performing a task locally before locking and reading Z from P2. It then reads X's version from P1 for read-validation, while writing undo logs to P2 and its backup B2 in the background. Finally, it updates Y and Z in-place in P2 and B2, before unlocking them in P2. (b) is similar.

(2) **Minimal online overhead.** The online-recovery component of the protocol should ideally add as little overhead as possible to the online-failure-free component; i.e., the overhead of logging should be minimal.

(3) **Fast recovery.** The recovery component should be fast; i.e., it must quickly roll back or forward any pending transactions on the failed compute server, so that these transactions (and any other transactions from other compute servers conflicting with those transactions) can make progress as soon as possible.

(4) **Non-blocking.** Recovery must not block in-flight non-conflicting transactions from other compute servers; these transactions must continue to make forward progress despite the failure.

The combination of the non-blocking property (for non-conflicting transactions) and fast recovery (which affects the latency of conflicting transactions) is what makes transactions *highly available*.

2.3 FORD

This section presents FORD [74], the only published transactional DKVS to date[2]. FORD executes distributed transactions using a variant of an optimistic transactional protocol [44], which offers strict serializability [59]. Specifically, FORD's protocol consists of three phases: execution, validation and commit/abort.

1. Execution. During execution, the coordinator reads all objects in its read-set. It also reads and eagerly locks all objects in its write-set. The execution phase fails if any accessed object is already locked. If it succeeds (fails), the protocol moves to the validation (abort) phase.

2. Validation. For validation, the coordinator checks that all objects in its read-set are still in the same state, i.e., have the same version and have not been locked. This ensures that the transaction is working over a consistent view. When the validation phase completes, then we have reached a *decision*: the transaction either commits or aborts.

3. Commit/Abort. Commit entails two steps: 1) all writes are applied to both the primary and backups of each object and 2) locked objects are unlocked. The client is notified after the first step with either a *commit-ack* or an *abort-ack*. Conversely, to abort, we simply unlock all locked objects and then notify the client.

[2]While other RDMA-based transactional KVSes (e.g., FaRM) use one-sided operations for performance, they still rely on a symmetric monolithic architecture where each server possesses sufficient compute to handle RPCs (e.g., for the commit phase). In contrast, a DKVS involves memory nodes with limited compute, hence cannot rely on RPCs.

Undo Logging. During the first two phases, the protocol writes an undo log in the primary and every backup of each object in its write-set. The purpose of this is to facilitate recovery under faults.

Figure 2 illustrates FORD. Figure 2(a) and (b) show the commit and abort path of the transaction in Figure 2(c), which reads object X and writes Y and Z. Figure 2(d) shows the four memory servers, Each of which serves as primary or backup for objects X, Y and Z.

FORD summary. Going back to our classification, the execution, validation, and commit/abort phases of the protocol comprise the online-failure-free (C1) component of FORD. The Undo logging comprises the online-recovery (C2).

In the next section, we discuss the limitations of FORD's logging component in the face of compute server faults and how it is addressed in Pandora. We also delve into Pandora's recovery algorithm, which addresses the lack of a recovery component in FORD.

3 PANDORA

This section details Pandora, a highly-available transactional protocol that recovers efficiently on independent compute and memory failures in DKVSes. In Section 2, we split a transactional protocol into three distinct categories: online-failure-free (C1), online-recovery (C2), and recovery (C3). Based on this classification, Pandora adapts C1 from FORD;[3] Pandora also adapts C2 from FORD but significantly reworks it, making it efficiently recoverable. One of the other limitations of FORD is that it lacks a recovery component (C3). In Pandora, we introduce a recovery algorithm that works over one-sided RDMA.

3.1 Efficient Recoverable Steady-State

In Section 1, we asserted that the FORD protocol prohibits fast recovery on a compute failure because it first locks objects and later it writes logs for said locks. We elaborate on this issue and present *Pandora's Implicit Lock Logging* to resolve it.

3.1.1 ***Problem: Stray locks.*** On a compute failure, it is possible that several objects are locked but there is no log that points to these locks. We call these *stray locks*. First, we discuss why stray locks prohibit fast recovery. Then, we delve more into the problem, arguing that it is more subtle than it looks, with its roots stemming from adopting disaggregated memory.

Impact on Recovery. Stray locks create two related problems. First, we cannot simply unlock while other compute servers are executing transactions, as we cannot differentiate between the stray locks and the regular locks of the live servers. Second, we need

[3]Our validation revealed a few bugs in C1 of FORD, which we fixed in Pandora (§ 5.1).

to scan the entire memory to find the stray locks, which can take seconds: e.g., scanning 100 GiBs through a 100Gbps network requires at least 8 seconds. Hence, we must block the entire system for several seconds.

What's the problem?. To modify an object, FORD issues an RDMA Compare-And-Swap (CAS) to lock it first, and then a RDMA Read to read it. Because RDMA *reliable connection* mode guarantees that the two messages will be delivered in order, we are certain that we read the object only after locking it. Moreover, only after the RDMA Read has returned we can perform the undo logging. This is because undo logs store the previous value (so that they can "undo" the modification). Note that, had we read the value without first locking it, we would not be able to log it, as it would be possible to log a different value than the one we locked. Thus, we need to lock before reading and read before logging. This is why there are stray locks.

The role of RDMA. This ordering conundrum does not exist in the traditional non-disaggregated architecture, because an RPC can execute all three tasks – locking, reading, and logging – in the same step. Crucially, this step is atomic with respect to failures. I.e., if the server that executes the RPC fails, then neither the log nor the lock will be visible. In contrast, with one-sided RDMA-access to a remote memory server, we do not have the luxury of performing these multi-step functions in a failure atomic manner. We expect that failure atomicity will become a recurring problem as more and more applications are ported to disaggregated memory.

Summary. Stray locks is a problem that occurs in a disaggregated architecture where it is not possible to perform multi-step functions in a failure atomic manner. Crucially, this prohibits fast recovery.

3.1.2 ***Solution: Pandora's Implicit lock logging.*** To solve this problem, we assign a unique 16-bit coordinator-id to each coordinator and mandate that locks include the coordinator-id of their owner coordinator. The unique coordinator-id is generated by an independent service, the *failure detector* (FD), which we detail in Section 3.2.2. The FD increments the coordinator-id counter when a new coordinator is spawned and sends the coordinator-id along with initial configurations to the server before executing transactions. Each compute server's spawn is strictly serialized, ensuring that no two servers are assigned the same coordinator-ids thereby preserving the uniqueness of the coordinator-id. In the event of FD failures, these steps can be repeated without violating correctness (Section 3.2.4).

On a compute server failure, we need not scan the entire memory in a blocking manner to release its stray locks. Instead, we enable other transactions to *steal* these locks. We call this technique *Pandora's Implicit Lock Logging (PILL)* because we have repurposed the coordinator-id (added to the lock) to signify whether or not the lock is stale, avoiding the need for explicit logging.

How does stealing work? Recall that the coordinator issues an RDMA CAS to lock an object. When the RDMA CAS fails, it returns the value of the lock, which includes its owner coordinator-id. We check this coordinator-id against a series of the *failed-ids*, i.e., an array that contains the coordinator-ids of all previously failed compute servers. If we discover that the lock is stray, we execute one more RDMA CAS to steal it. Notably, stray locks can also cause Reads to abort during both the execution and validation phases. To avoid this, we again check the failed-ids, and if the lock is found to be stray, we proceed as if the object was not locked at all.

Failed-ids. The FD maintains a list of all *failed-ids* and includes it in the initial configuration message sent to the compute server. After a compute server failure, the FD is responsible for notifying all alive compute servers, so that they can update their failed-ids. We discuss this further in a later section (Section 3.2.2).

Overhead. The overhead of this approach is: 1) a check against the failed-ids, incurred only when accessing a locked object and 2) an extra RDMA CAS when finding a stray lock. Note that actually finding a stray lock is extremely rare, because we execute millions of transactions per second, while we may only get one failure every a few hours (depending on the number of compute servers [11]).

Recycling coordinator-ids. We must ensure that a failed coordinator-id cannot be assigned to another compute server, until all of its stray locks are unlocked. We ensure this as follows. We use 16 bits to represent coordinator-ids, allowing for 64K compute servers to join over the lifetime of the system. Although we expect 64K coordinator-ids to be plenty, they might outlast the utility of a long-running system. As such, we implemented a background mechanism that scans the memory and unlocks all stray locks, allowing to recycle failed coordinator-ids. FD triggers this mechanism if more than 95% of available coordinator-ids are used. Additionally, our recycling mechanism unlocks all *stray locks* using CAS operations, which is sufficient to resolve race conditions with in-flight transactions.

Notably, as more compute servers fail over time, we must ensure that the overhead of checking the failed-ids stays constant. We achieve this by implementing failed-ids as a compact bitset with 64K entries.

Summary. We presented PILL, a technique that links each lock with the unique coordinator-id of its owner compute server. This allows us to detect which locks are stray and steal them. We use a large enough number for coordinator-ids to ensure that we will not need to recycle them but have a contingency plan for that. PILL enables recovery from a compute fault without interrupting the rest of the system.

3.1.3 ***Problem: Logging aborted transactions.*** In FORD it is possible for a logged transaction to be aborted. The problem is that at recovery-time it is impossible to differentiate between committed and aborted logged transactions. This prevents correct recovery. For instance, consider a failed compute server C, which has logged a write to object X in one of its transactions. Also, assume that during recovery, we see that X has been modified. It is impossible to know whether X has been modified by C or not. This is because it is possible that C's transaction aborted, unlocking X, and then a different compute server locked and updated X. As we will see when discussing our recovery protocol (§3.2.2), recovery hinges on knowing whether C is the one that modified X so that we can undo the modification if needed. Notably, we realized this issue after our validation revealed three bugs caused by this problem (Section 5.1).

3.1.4 ***Solution: Logging phase.*** First, note that we cannot solve this problem by relying on the fact that locks include the coordinator-id of their owner because we do not lock backups. Therefore, if the memory server that serves as the primary for object X fails, we will still face the same problem.

To resolve this problem, we add an extra *logging* phase in between validation and commit/abort. This phase is executed only if validation succeeds. To minimize the performance overhead of this extra logging phase, we incorporate it with the main undo logging scheme used in the transaction protocol. Pandora

further optimizes this phase by limiting extra logging to aborts and enforcing lock-to-log order.

FORD writes a log in each replica of each object in its write-set. For example, assume a transaction that writes X and Y with a replication degree of 3 (i.e., $f + 1 = 3$), where X is replicated in memory servers 0,1,2 and Y is replicated in memory servers 3,4,5. FORD will log X in 0,1,2 and Y in 3,4,5. We take a different approach. For each compute server, we specify f+1 memory servers that hold its logs. Therefore, in the above example, both writes to X and Y will be logged in the same three servers. This is a well-established technique [66]. This approach significantly reduces the number of log copies and minimizes logging overhead during the commit phase.

As we log after validation, at which point we know the entire write-set, we can log all writes with the same single RDMA Write amortizing its overheads. Therefore, the total cost of logging in our technique is always $f + 1$ RDMA Writes as opposed to FORD's $f + 1$ RDMA Writes per object in the write-set. This technique also simplifies the recovery protocol, as all the logs of a compute server are gathered in the same $f + 1$ memory servers.

3.1.5 ***Protocol Summary.*** This section summarizes Pandora's Online-failure-free (C1) and Online-recovery (C2) protocols.

(1) Execution. The coordinator reads all objects in the read-set and write-set, eagerly locking write-set objects if their exact addresses are known. If any write-set object is already locked, the coordinator aborts the transaction. Unlocked write-set objects and reads are retried later. Pandora logs write-set objects after successful locking, enforcing a lock-to-log order before validation.

(2) Validation. Pandora validates only after completing execution. The coordinator checks that all read-set versions match and unlocks the objects. This phase also ensures the write-set is logged.

(3) Abort. If validation fails, the coordinator aborts the transaction. First, the coordinator logs the decision by truncating logs. Then it unlocks every lock using an RDMA write.

(4) Commit. Coordinator applies writes on primary and backups of each object, sends client acknowledgments, and unlocks the write-set.

3.2 Recovery Protocol

In the previous section, we ensured that fast and non-blocking recovery is possible. In this section, we guarantee that it is also correct. We start by specifying four correctness criteria. Then, we describe non-blocking recovery from compute failures. Finally, we provide a brief description of how we handle memory failures.

3.2.1 ***Correctness criteria.*** Before we state the correctness criteria, we first introduce some definitions. A failed compute server, C, may have been working on a number of transactions before failing. We refer to these as *stray transactions (Stray-Txs)*. There are three side-effects of Stray-Txs that must be addressed: 1) stray locks on objects 2) updates on objects (during commit phase) and 3) communication with the client to notify it of commit or abort. We differentiate between two types of Stray-Txs, the *Logged-Stray-Txs* for which C has written a log, and the *NotLogged-Stray-Txs*, for which C had not yet reached the point of writing a log. The dichotomy is crucial, as transactions with a written log can have all three of the side-effects, while the NotLogged-Stray-Txs can only have stray locks.

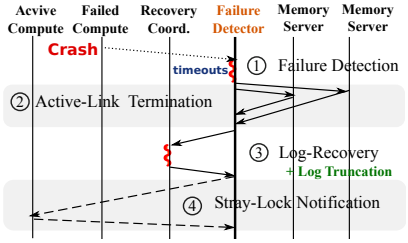

Figure 3: Recovery Protocol

We next list four correctness criteria for the recovery algorithm after the failure of compute server C.

(Cor1) Before trying to recover the Stray-Txs of C, we must ensure that C cannot access memory anymore. This ensures that memory will not be compromised by unreliable failure detection.

(Cor2) We must either roll back or forward all Logged-Stray-Txs, to ensure that all or none of their updates are applied to objects.

(Cor3) We must not roll back a Logged-Stray-Tx, if it has notified its client that it has committed. And vice versa, we cannot roll forward if the client has been notified of an abort.

(Cor4) We can only steal the stray locks from NotLogged-Stray-Txs. This is because Logged-Stray-Txs may have also updated some of the locked objects, and thus, stealing could leave the memory in an inconsistent state.

3.2.2 ***Recovering from compute failures.*** Figure 3 illustrates the protocol, which comprises four steps that are overviewed below.

(1) Failure Detection. The first step of the protocol is initiated by a fault detector (FD), which detects a crash on a compute server. Our protocol can work with any off-the-shelf FD. For our evaluation, we have implemented a heartbeat-based FD, which exchanges heartbeats with compute servers, and reports failure after a 5ms time-out.

(2) Active-Link Termination. Notably, any FD can have false positives, i.e., it can mistakenly deem compute server C as failed. Recall correctness criterion (Cor1): before recovering the Stray-Txs of C, we must ensure that C can no longer access memory. In a non-disaggregated system, this is typically achieved by rejecting RPCs from servers that are not in the stable configuration [40]. To achieve the same effect, we revoke C's RDMA rights, ensuring that any future requests from C will get dropped. We call this *active-link termination*. Recall that memory servers have some low-power, cheap compute for network management. We implement active-link termination by sending a link-termination RPC to this compute.

(3) Log Recovery. Again, assume that compute server C has failed. For each of failed C's Logged-Stray-Txs we will make a decision, either roll it forward or backward, satisfying criterion (Cor2). Recall that in its commit phase (Figure 2), the transaction will update all replicas of an object (i.e., it applies its writes). To also address criterion (Cor3), we make the following two assertions on the protocol 1) if all replicas of all objects in the write-set are updated, then it is possible that the client has received a commit-ack. 2) If any of the objects in the write-set are updated, then it is impossible that the client has received an abort-ack. Based on these, we can assert that we can safely roll-forward all Logged-Stray-Txs that have updated all objects in their write-set in all replicas, because the commit-ack is possible, but the abort-ack is impossible. We roll-back all other Logged-Stray-Txs. This is

correct, because it is impossible that we have sent a commit-ack for these transactions.

In practice, we implement log recovery as follows. First, we spawn a thread which we call Recovery Coordinator (RC). Recall that for any compute server, we write its transaction logs in $f+1$ specific memory servers (§ 3.1.4). Thus, the RC can read all logs by issuing $f+1$ RDMA Reads. Using the logs, the RC recreates the write-set of each Logged-Stray-Tx. Then, for each Logged-Stray-Tx it issues an RDMA Read on every replica of each object in its write-set to check if it has been updated. Specifically, each object has a version, so we simply read the version and compare it with the version in the undo logs. Then, for each transaction that has updated all replicas of all writes in its write-set, we simply unlock its locks with an RDMA Write to each replica. For the rest of the transaction, we also unlock all objects, but also use the undo log to roll back any updated objects.

Note here the importance of the second fix in FORD's online-recovery. Had we not done the logging after validation, it would be impossible to differentiate between an object that is updated by a Stray-Tx or by a live transaction from an alive compute server. This is because in FORD it is possible to log an object, and then later abort and unlock it. However, the log would remain.

F+1 Log Reads. In most cases, F+1 reads are sufficient to efficiently retrieve the logs. Each coordinator is allocated 32KB for logs, meaning each RDMA read returns a few MBs of contiguous memory for failed coordinators. RDMA hardware typically supports read sizes of up to 1 GB. With the standard 4KB MTU, those RDMA reads are split into multiple packets, but these packets are bundled together, minimizing latency compared to multiple round trips.

(4) Stray lock notification. Finally, we notify all compute servers of a failure so that they can start stealing the stray locks of the failed server. Recall the first correctness criterion: we can only steal stray locks of not-logged stray transactions. For this reason, it is crucial that we only perform the stray lock notification after log recovery.

3.2.3 ***Idempotent Recovery.*** Pandora ensures idempotent recovery, enabling any step of the end-to-end recovery algorithm to be re-executed. This capability is key to tolerating failures during the recovery phase, given that the recovery coordinator operates within a standard compute server. For instance, in cases where compute failures can cause log recovery to stall, necessitating re-execution, Pandora allows for the re-execution of the log-recovery step until the final acknowledgment is received from the recovery coordinator. To guarantee idempotent correctness, Pandora truncates all logs from the failed compute server before sending the Stray-Lock notifications (refer to Figure 3). Note that RC truncates logs by simply setting an invalid bit in each coordinator's log header using an RDMA write.

3.2.4 ***Failure Detector Availability.*** The availability of the failure detector (FD) is critical to the end-to-end recovery algorithm. Pandora ensures FD availability by replicating its state across a quorum of replicas, using a flexible approach that can be implemented in various ways. In our design, we use Zookeeper as the replication layer, leveraging its proven reliability as a coordination service [33].

First, we decouple the FD's program state and migrate it to Zookeeper. Second, we modify compute servers to send RDMA-based heartbeat messages to all Zookeeper replicas, hoping to reach at least a majority of them in the event of failures.

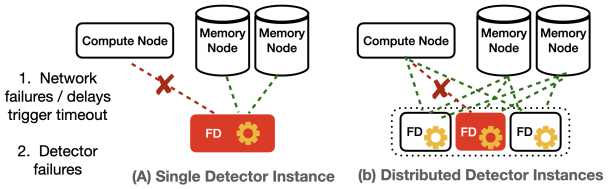

Figure 4: Failure detector reliability

Figure 4 shows our standalone FD (a) and distributed FD (b). The distributed FD (b) reduces failure detection times by replicating the failure detector across multiple servers, ensuring that delays in compute nodes or network issues do not cause false negatives. When this is combined with the low-latency of RDMA-based heartbeats, failure detection times can be reduced to the order of milliseconds [24]. In an RDMA-based setup, a timeout of a few ms can be practical, as RDMA round-trip times are in the low µs range [22]. This combination allows transient network hiccups to be absorbed, minimizing false positives and ensuring that a node is only considered failed when it is disconnected from the majority of FD replicas.

3.2.5 ***Recovering from memory failures.*** We have thus far focused on compute failures because they present the opportunity for non-blocking recovery. Additionally, Pandora can recover from f memory failures using $f+1$ memory replicas.

Pandora handles memory server failures in three steps. First, we notify all compute servers of the failure. Then, for each object whose primary is lost, compute servers deterministically calculate the new primary using metadata, which includes the location of each data partition and its replicas. We use consistent hashing [39] to statically partition data across memory servers, avoiding resizing when new replicas are added or removed.

In the event of a memory failure, we do not initiate the full recovery protocol if all compute servers remain operational. This is because each compute server retains a small amount of local memory (typically a few GBs) to store transaction states and has complete knowledge of the state of its transactions. After learning of the memory failure, each compute server makes a decision for each of its transactions, using the same criterion as log recovery; committing transactions that have updated all live replicas and aborting the rest. Once this process is complete, compute servers resume initiating new transactions. In the case where memory and compute servers fail together, we execute both protocols independently.

Pandora adds new memory servers if there are more than f replica failures. For this, we stop the DKVS, re-replicate all the partitions, and then resume the compute server. In practice, re-replication overhead can be reduced by selecting a sufficiently large f value.

4 METHODOLOGY

Pandora's goal is to achieve fast and correct transactional recovery on DKVSes. We conduct experiments to answer four key questions.

Validation. Recovering transactions correctly on DKVSes involves subtlety. Is Pandora actually recoverable? We validate Pandora as well as the state-of-the-art protocol FORD [74] using our litmus-test-based validation framework.

Recovery latency. Reducing recovery latency is one of the key goals of our work. What do we mean by recovery latency precisely? Recall that our proposed techniques do not stop the entire KVS on a failure, but transactions whose coordinators fail are affected. The

recovery latency refers to the delay seen by such transactions that are affected by failures. We show the recovery latency of Pandora.

Fail-over throughput. When a failure does happen, can our techniques ensure minimal disruption? We show the fail-over throughout – the throughput of Pandora when it is recovering from a failure.

Steady-state throughput. How much overhead does Pandora impose on steady-state failure-free execution compared to the existing state-of-the-art? We show the steady-state throughput of Pandora and compare it with the FORD baseline.

Before diving into our experimental evaluation, we first explain our experimental setup, workloads, and methodology.

4.1 Testbed

Setup. We conducted our experiments on a cluster of 5 servers in CloudLab [26]. Each server is an r650 node in the Clemson cluster. A server can play the role of either a compute or a memory server. The configuration is different in different experiments. We use a dedicated server for our failure detector and recovery manager. We will explain the different configurations separately in each experiment.Each machine in our setup runs Ubuntu 18.04 and is equipped with two 36-core Intel Xeon Platinum 8360Y at 2.4GHz with two hardware threads per core. Furthermore, each machine has 256GB of 3200MHz DDR4 memory and a 100Gbps PCIe4 Mellanox ConnectX-6 NIC.

Protocols: Baseline vs Pandora. For our evaluation, we have adopted the in-memory version of FORD KVS [74] as the system for deploying the protocols. Recall that FORD is the only fully one-sided transactional DKVS in the literature. Because FORD misses the recovery part of the protocol, we integrated our recovery algorithm to FORD to make it our *Baseline*. We compare this Baseline protocol against Pandora, which adapts the online-failure-free component from FORD, but significantly improves upon the online-recovery component to speed up recovery by introducing PILL and a novel fast RDMA-based recovery component.

Workloads. To validate the Baseline and Pandora, we used our litmus tests, which we describe in the next section. For performance evaluation, we use the same three standard OLTP benchmarks that were used by FORD: TPC-C [3], TATP [1], and SmallBank [2]. These benchmarks have 8B keys. The values are 672B, 48B, and 16B, respectively. Besides these benchmarks, we used a microbenchmark with 8B keys and 40B values in which write ratios are adjusted.

It is worth noting that each workload runs a different number of transaction coordinators (which we explicitly specify). Unless mentioned otherwise, each of our workloads runs on 128 coordinators, and we use the same dataset sizes as the ones used by FORD [74].

Workloads characteristics. TATP, SmallBank, and TPC-C consist of 4, 2, and 9 tables, respectively. In TATP, 80% of the transactions are read-only. In contrast, both SmallBank and TPC-C have high write ratios – 85% and 95%, respectively.

5 END-TO-END LITMUS TESTS FOR CORRECTNESS VALIDATION

Verifying transactional protocols is notoriously hard. Despite a rich literature on formally verifying *models* of transactional protocols using manual and automated techniques [14, 45], our focus is on validating the actual *implementations* of the protocols. End-to-end testing with randomly injected faults has proven to be very effective in revealing bugs in transactional protocols of databases [12, 34, 42, 52].

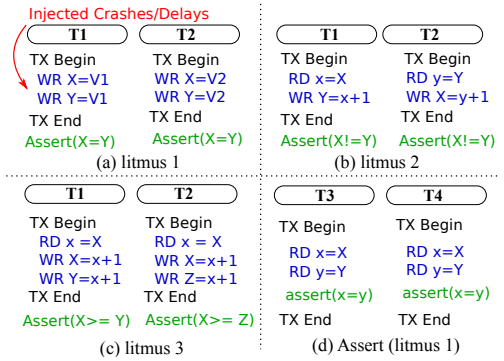

Figure 5: Basic litmus tests with application-observable states.

Method. The common technique to test databases is Adya's Histories [4, 42]. The idea is to run a number of randomly generated transactions, collect a rich trace of data and metadata for each run (the history), and use the history to determine (violations in) the consistency model of the database. However, existing frameworks – because they need to collect histories – tend to be heavyweight, hard to integrate and scale. There is an alternate method to validating databases – one based on *application-observable state* [19] rather than histories. The idea is to carefully construct transactions so that the values of the objects reveal the consistency model, and consequently reveal protocol bugs (if it does not match the intended consistency model). Crooks et al. [19] showed theoretically how the application-state-based approach can be as effective as the histories-based approach while being significantly more lightweight and less costly.[4] However, being a conceptual framework, it does not contain a suite of tests or a tool that can be readily used for testing protocols.

To the best of our knowledge, this is the first work to create transactional (litmus) tests for black-box testing of real-world protocols. Figure 5 lists 3 basic litmus tests that we have developed in our framework for validating strict serializability. These litmus tests cover all potential dependency cycles in serializable transactions [4]. For even greater test coverage, we have extended our tests with additional variables. In addition to the litmus tests, we also generate their matching application-centric assertions which we describe next.

Assertions. We use special read-only transactions for realizing the assertions. For example, the first litmus (Figure 5(a)) assigns a value of V1 to both variables X and Y in the first transaction, and assigns a value of V2 to both variables in the second transaction. Strict serializability mandates that X and Y should be equal at the end of each transaction. This is what we assert with our read-only transaction (Figure 5(d)). To test the steady-state and the recovery protocol together, we randomly inject crashes after any operation.

Compound Tests. Besides our litmus tests, we used extended tests with corresponding assertions. These new tests were created by either stretching or combining the basic litmus tests. However, no new bugs were discovered, and thus, they are not discussed in this paper.

5.1 Litmus Tests, Bugs, and Fixes

In this section, we detail our litmus tests and the bugs we have found in both Baseline and Pandora using our litmus testing framework.

[4] It is also worth noting an analogous approach of *litmus testing* [8, 21, 51] has been extremely effective for validating consistency models in shared-memory multiprocessors.

Litmus	Bugs (Category-Source)	Description	Fix(es)
Litmus-1 (Direct-Write Cycles)	Complicit Aborts (C1 - Baseline/Pandora)	Releasing unset locks in the abort path	Unlock only the acquired locks during execution in the abort path
	Missing Actions (C2 - Baseline)	Omitting logging of inserts	Add inserts into undo logs.
Litmus-2 (Read-Write Cycles)	Covert Locks (C1- Baseline/Pandora)	Not checking the lock value in the validation phase	Read and check locks of read-only data during the validation phase
	Relaxed Locks (C1 - Baseline/Pandora)	Relaxing the order of locks and validation in the commit path	Grab all locks before validation
Litmus-3 (Indirect-Write Cycles)	Lost Decision (C2 - Baseline/Pandora)	Logs for a transactions that aborted without being able to tell if it hs updated its objects	Add log phase if validation succeeds (Section 3.1.4)
	Logging without locking (C2 - Baseline/Pandora)	Loging a lock that was never grabbed	Add log phase if validation succeeds (Section 3.1.4)

Table 1: Three categories of bugs found in Baseline (FORD) and Pandora: online-failure-free (C1), online-recovery (C2) and recovery (C3).

In order to pinpoint where the bugs are, we classify the actions of each of these protocols into three distinct categories: online-failure-free (C1), online-recovery (C2), and recovery (C3). Recall from Section 2.2 that Baseline inherits C1 and C2 from FORD and C3 from Pandora. Pandora also inherits C1 from FORD. Table 1 summarizes the bugs that we have found, listing the protocol where we found the bug (Baseline or Pandora) and the appropriate category.

Litmus 1. This checks *Direct-Write* dependency cycles between two transactions. As we discussed above, there are two transactions in the litmus test, with the first transaction assigning value V1 to objects X and Y, and the second transaction assigning value V2 to the same two objects. We then assert that the two objects have the same value. Different values imply a strict serializability violation. We also ran variants of this litmus test, replacing writes with inserts and deletes.

Bug: Complicit Abort In this bug, FORD releases every lock in its write-set when it decides to abort. Thus, it also releases some locks that were never actually acquired by the transaction during execution. Crucially, this can cause a transaction to release a lock grabbed by a different transaction. This is an online-failure-free (C1) bug that affects both the Baseline and Pandora as it exists in FORD.

Fix: We fix this bug by releasing only the locks that have been actually acquired during execution.

Bug: Missing Actions We have found a bug in FORD because logging is omitted for inserts. This is an online-recovery (C2) bug in FORD that affects only the baseline.

Fix: We also add undo logs for inserts (besides writes and deletes).

Litmus 2. This checks *Read-Write* dependency cycles (or violations) (Figure 5(b)). Transaction T1 reads the value of X while updating the value of Y, and T2 reads Y while updating X. Let us assume that T1 reads the old value X=0 and writes Y=1. Since T1 does not see the write of T2, it must be that T2 sees the write of T1. Specifically, if T1 reads X=0 then T2 must read that Y=1. If t2 reads Y=0 and proceeds to commit X=1, the final outcome would be X=1, Y=1, which violates (strict) serializability [13, 19, 59].

Bug: Covert Locks In its validation phase, FORD does not check if the read objects are locked. Recall that FORD checks versions of all the read-only objects in the validation phase. However, it must also ensure that the objects are not locked. Specifically, what happens in the litmus test is that transactions T1 and T2 concurrently read X=0 and Y=0 and then lock Y and X. Because the transaction protocol only checks the version numbers during the validation phase, without considering whether they have been locked, both T1 and T2 can progress, leaving objects in an inconsistent state (X=1, Y=1).

Fix: We fetch both the lock value and version for each read-only object in a single round trip. This is possible because the lock and version for each object in FORD's KVS are stored together. Then, in the validation phase, before comparing versions, we check whether the object is locked; if the object is locked, we abort the transaction.

Bug: Relaxed Locks Litmus Test 2 revealed another online-failure-free (C1) bug in FORD, where in rare cases, validation starts before ensuring all locks have been grabbed. Thus, the execution overlaps with the validation phase; affecting both Baseline and Pandora.

Fix: We enforce that validation happens strictly after locking.

Litmus 3. We use litmus 3 to check *Indirect-Write* dependency cycles (Figure 5(c)). Transaction T1 reads and increments X and writes the new X into Y. T2 reads and increments X but writes the new X into Z. Therefore, at any given time, the values of Y and Z cannot be larger than the value of X; this is checked by the assertions.

Bug: Lost Decision As we discussed in Section 3.1.3, FORD writes logs for transactions that may later abort. Crucially, it may be impossible for the recovery protocol to tell whether the transaction has aborted or it has updated all of its objects and thus must be rolled forward. In the litmus test, T1 logs the writes to X and Y, but it aborts. The bug occurs when the recovery protocol reads the log and infers that the write to X has been applied because it sees that X has been modified. However, the modification was done by T2, not T1. Because Y has not been modified, the recovery protocol rolls back the update to X. In doing so, it partially undoes T2 and leaves memory in an inconsistent state, as Z has been updated by T2.

Fix: As discussed in Section 3.1.4, we add a logging phase after validation, which is executed only if validation succeeds.

Bug: Logging without locking This bug is caused by the problem discussed above (logging and then aborting), in conjunction with a corner case in FORD where a log is written before the lock is actually grabbed. Similarly, to the above, the recovery protocol can either erroneously roll forward or undo the write of another transaction.

Fix: The logging phase after validation suffices to solve both issues, as it ensures we log after locking. Pandora, however, implicitly enforces lock-to-log order, eliminating the additional round trip. This approach employs unique coordinator IDs for locking (Section 3.1.2).

6 EXPERIMENTAL EVALUATION

The goal of our experimental evaluation is to compare the Baseline against Pandora on the following performance metrics: recovery latency, fail-over throughput, and steady-state throughput.

6.1 Recovery Latency

In this section, we report recovery latency for Baseline and Pandora. Recall that FORD's design incurs a significant overhead on

recovery. On a failure, the entire KVS is stopped and searched to detect stray locks. We observe that these overheads are in the order of seconds. Specifically, in our measurements, a recovery process that runs on one thread while searching over the KVS using one-sided reads takes around 5 seconds for 1 million keys. While more threads could be used, the latency grows linearly with the number of keys. This is impractical for today's KVSes, which store much more than 1M keys and demand high availability; hence, we do not explore the Baseline's recovery latency any further. Instead, we focus on Pandora's recovery latency. But before this, we briefly describe how we emulate failures.

Emulating Failures. In this experiment, we emulate a failure by stopping a process at a selected point in time, which implicitly stops all the in-flight transactions running within that process. The failure detector (FD) identifies these failures using timeouts and requests the recovery coordinator to perform recovery for each of the failed coordinators. We use *5ms* timeouts in the FD.

Pandora. Recall that Pandora introduced PILL, a fast recovery technique that moves the recovery for stray locks out of the critical path of failures. Table 2 shows the recovery latency for each benchmark with respect to different numbers of in-flight transactions (i.e., transaction coordinators) per compute node. Specifically, in TPC-C, SmallBank, and TATP, recovery takes 5ms, 5.3ms, and 2.2ms, respectively (with 512 outstanding transactions). In addition, our micro-benchmark with 100% writes shows 2ms latency. The latency represents the time spent in the log recovery step of the recovery protocol.

The table demonstrates that Pandora achieves recovery latency within a few milliseconds. As expected, the latency increases with the number of transaction coordinators, since a larger number of outstanding transactions must be recovered on each compute server. Pandora's recovery latency is three orders of magnitude lower than that of the Baseline (FORD). Specifically, the Baseline exhibits recovery times in the range of seconds (e.g., ~5 seconds per million keys) due to the need for scanning the entire KVS for lock recovery, which blocks all outstanding transactions during the process.

Traditional Logging Scheme. In addition to our main techniques, we also evaluate a more traditional scheme that adds extra logging to the protocol to allow locks to be recovered in the recovery phase (without scanning the whole KVS). Recall that the undo logging scheme used in FORD is not sufficient for recovering locks. This traditional scheme adds extra logging before the lock operation is executed by the transaction coordinator. We extend the Baselines recovery protocol with this extra logging to recover from locks. With the highest number of outstanding transactions (512 in this experiment), recovery latency for the TATP, TPC-C, SmallBank, and MicroBench reaches 10ms, 13ms, 2.7ms, and 2.5 ms, respectively. This recovery is not only approximately up to 2× slower than that of Pandora, but as we will see in the next section, it slows down the steady-state performance by as much as 35%.

Summary. Overall, these results show that PILL drastically reduces recovery latency, substantiating our argument that we can recover fast from a compute failure on DKVSes. Next, we look into how much overhead these protocols impose on fault-free stable-state execution.

6.2 Sensitivity study of PILL

Pandora's implicit lock logging (PILL) offloads the recovery of stray locks to the transaction's execution phase, which raises the question of how much overhead this introduces to the steady-state performance. Recall that PILL adds three extra steps to the steady-state protocol: (1) locking with coordinator-ids, (2) a check against the failed-ids, and (3) releasing stray locks. Notably, the overhead of the last operation is only visible when there are actual failures. First, we evaluate the steady-state overhead of PILL (only (1) and (2)). Second, we measure the overhead of PILL under failures.

Bench \ Coord. per node	1	8	64	128	256	512
TPC-C	8 us	22 us	158 us	272 us	563 us	4951 us
SmallBank	8 us	139 us	232 us	424 us	876 us	5272 us
TATP	9 us	20 us	131 us	513 us	1039 us	2236 us
MicroBench	10 us	21 us	119 us	474 us	1001 us	2043 us

Table 2: Recovery latency of Pandora (in microseconds) while increasing the number of outstanding coordinators per compute node.

PILL under no failures. For this experiment, we use our micro-benchmark with 128 transaction coordinators. Figure 6 shows the (average of 5 runs) throughput over time without PILL (blue) and with PILL (red). Note that the throughput difference is negligible. The MTps between 10s-30s is 0.919 and 0.912, respectively. This is because the failed-id list is empty; hence, Pandora does not incur any extra round trip overhead for stealing locks. Notably, each failed-id bitfield lookup (with $O(1)$ complexity) only adds a few nanoseconds on every failed lock (and reads), which is insignificant compared to the round trip latencies that are in the order of microseconds.

PILL under failures. In this experiment, we measure the end-to-end steady-state overhead of PILL under failures. Recall that after failures, stealing the lock adds an extra round trip. To measure the overhead, we ran the same experiment with failures that stopped (then recovered) half of the coordinators in the setup. We then reduced the Mean Time To Failure (MTTF) and reran the experiment. Lower MTTF means that the number of stray locks in the DKVS is higher, and the time to recover these locks before the next failure is lower.

Figure 7 shows the (average over 5 runs) transaction throughput without failures (blue), with *MTTF=10s* (red), *MTTF=2s* (yellow), and *MTTF=1s* (green). The throughput between 10s-30s is 0.911, 0.912, 0.901, and 0.911 MTps, respectively. It is worth noting that the typical MTTF in the datacenter is in the range of minutes [11], and MTTF < 10s is highly unlikely. As we can see, it is clear that PILL adds insignificant overhead under failures. This is because only just a few stray locks must actually be recovered and that overhead is amortized over the entire run.

6.2.1 Traditional Logging Scheme. Besides the proposed technique, we measured the steady-state overhead of the traditional scheme that we previously discussed. Recall that for this scheme to work, we need an additional logging round trip for each lock in the steady-state execution phase. Hence, unsurprisingly, the steady-state throughput is lower than that of the baseline FORD's throughput. Smallbank, TPC-C, and TATP incur average throughput overhead of **35%,14%, and 2%**, respectively. Similarly, this approach adds **21%** overhead on our microbenchmark with 100% writes. We observe that logging overhead generally increases with increasing write ratios (e.g., TATP, which is mostly read-only, shows lesser overhead than write-intensive workloads like SmallBank). On the other hand, some write-intensive workloads like TPC-C show lesser overhead than anticipated because of one-sided read overhead that is not proportional to the actual read/write ratio present in the benchmarks.

Summary. Unlike traditional logging, PILL adds negligible overhead over FORD's steady-state while ensuring safety and drastically reducing the recovery latency. Next, we look at the fail-over throughput.

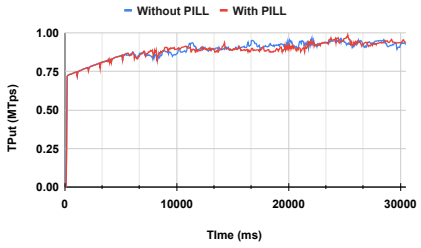

Figure 6: Steady-state of non-recoverable FORD (blue) vs recoverable Pandora (red).

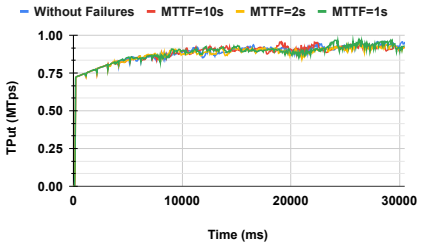

Figure 7: Steady-state throughput of Pandora while varying mean time to failures (MTTF).

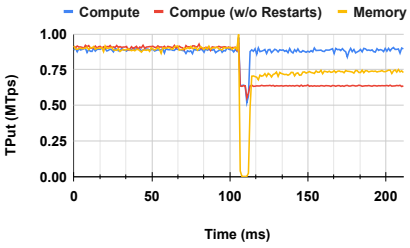

Figure 8: Average Microbenchmark fail-over throughput of Memory and Compute faults.

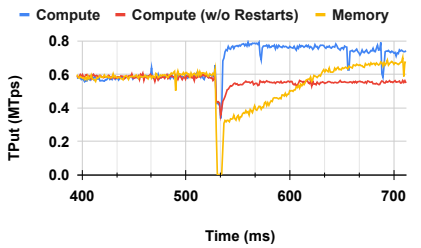

Figure 9: Average Smallbank average fail-over throughput of Memory and Compute faults.

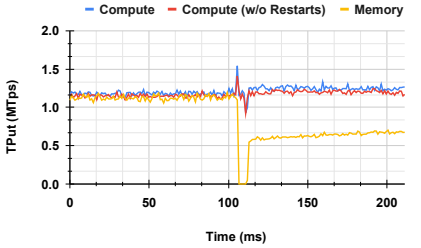

Figure 10: Average TATP fail-over throughput of Memory and Compute faults.

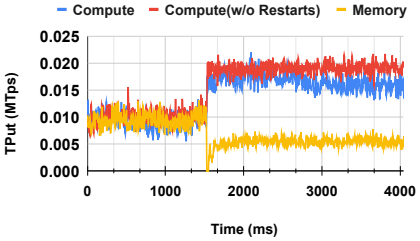

Figure 11: Average TPC-C fail-over throughput of Memory and Compute faults.

Figure 12: Smallbank fail-over throughput of Memory and Compute faults [low contention].

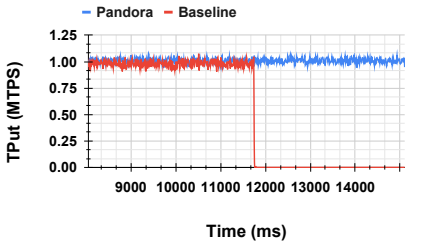

Figure 13: Microbenchmark with contention [hot objects=1000]. Baseline throughput recovers but after seconds (not shown in the plot).

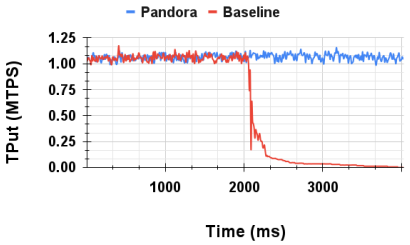

Figure 14: Microbenchmark with contention [hot objects=100000]. Baseline throughput recovers but after seconds (not shown in the plot).

6.3 Fail-over Throughput

The fail-over throughput is the difference between the throughput while recovering from a failure and the fault-free steady-state throughput. To measure this for Pandora, we conducted an end-to-end experiment to show the impact of our fast recovery that does not stop the operation of the entire KVS.

For this experiment, we set up a cluster of five machines with two memory nodes and two compute nodes; the fifth server runs the failure detector (we report experiments with distributed FD later). We use our standard benchmarks with 128 transaction coordinators. We emulate a failure by crashing one compute node while measuring the throughput of the rest of the KVS. Recall that the failure detector identifies the failure after waiting for the timeout (i.e., 5ms in our case) and initiates the recovery coordinator. For this experiment, we use the same failed machine to run the recovery coordinator.

Unlike blocking (i.e., "stop-the-world") type recoveries as in the Baseline (or traditional monolithic server deployments [22, 23, 28, 64]), our recovery need not stop the entire KVS for compute failures. Indeed, our microbenchmark in Figure 8 (blue line) shows that Pandora's throughput does not drop to zero, but drops to about two-thirds of the original throughput after the emulated crash. Similarly, Figures 9, 10, and 11 respectively show the fail-over throughput of Smallbank, TATP and TPC-C. Recall that Pandora handles memory failures as an all-compute failure that requires stopping the entire KVS to update the new replica configuration. Thus, in our benchmarks (yellow line), fail-over throughput drops to zero but rapidly recovers.

6.4 Post-failure throughput

Recovery impacts not only the fail-over transactional throughput but also the post-failure throughput. If the recovery cannot restore the lost compute resources, the post-failure throughput may drop in proportion to the percentage of lost coordinators. In some cases, the post-failure throughput hinges on the ability to restore the failed coordinators after the recovery process. In such a scheme, the KVS can either use the freed-up resources from failed coordinators or standby backup resources. Reusing resources from failed coordinators is possible for software crashes. Figure 8 shows two scenarios: the red line denotes the case when there is a fault followed by recovery, but the failed resource is not reused. The blue line shows the case in which the failed resources are reused, and hence, the post-recovery throughput matches the pre-failure throughput. Notably, the failed coordinators are brought back in less than 10ms after the fault.

Moreover, post-failure throughput is impacted by oversubscription and the system's bandwidth limitations. For example, in scenarios with high load where transactions operate significantly below optimal throughput, due to a multitude of coordinators competing for network bandwidth, a loss of a compute node might paradoxically elevate the application's throughput. This

phenomenon manifests in certain workloads, resulting in post-failure throughput exceeding the steady-state throughput observed pre-failure. However, this elevation is primarily attributed to bandwidth constraints and can be mitigated by reducing the number of coordinators active on the compute nodes. For instance, Figure 12 depicts the Smallbank benchmark with half the number of coordinators. In this configuration, Pandora effectively restores the post-failure throughput to its pre-failure levels.

Distributed FD. Recall that replicating the FD using Zookeeper quorums impacts recovery (Section 3.2.4). Crucially, even with three FD replicas (managed via Zookeeper), Pandora recovers in under 20ms, which is still orders of magnitude faster than the Baseline.

Sensitivity to stalls. Let us consider the case in which a transaction T1 locks an object X during execution and then is forced to abort due to a fault. The failure would trigger a recovery operation (section 3.2). But before the recovery can be completed, suppose another transaction T2 accesses the same object X during its execution. At this point, there are two options: abort transaction T2 or stall T2 until recovery is complete. Thus far, we have assumed the former, letting other transactions that do not conflict with object X to execute and commit.

This experiment investigates the stalling path, where a transaction accessing an object that needs recovery is delayed until recovery completes. This approach naturally affects fail-over throughput, with the impact being proportional to the recovery latency. To demonstrate the sensitivity of fail-over throughput to varying recovery latencies, we conducted two experiments using our microbenchmark with 100% writes, on the same setup as the last experiment. To simulate failures, we randomly crashed half of the coordinators at different times.

In the first experiment, we used 1,000 hot objects/keys (Figure 13). As expected, without the fast recovery of Pandora, throughput rapidly drops to zero. The combination of high recovery latency and a high conflict rate quickly blocked all coordinators. Conversely, Pandora's fast recovery resulted in an initial drop in throughput, but once recovery completed, throughput quickly stabilized.

We ran a similar experiment with 100,000 hot objects to reduce the number of conflicting transactions (Figure 14). With fewer conflicts, even under slow recovery, non-conflicting transactions could still execute, leading to a gradual decline in throughput rather than an immediate drop to zero. However, with fast recovery, throughput remained steady, and the fail-over throughput was primarily influenced by the number of failed coordinators, which could be restored later.

7 RELATED WORK

Pandora introduces a new end-to-end transaction protocol for disaggregated-memory key-value stores, where compute and memory are decoupled. Specifically, we address transactional recovery in one-sided RDMA-based disaggregated key-value stores, focusing on ensuring a fast non-blocking recovery that is correct and does not compromise the steady-state performance.

Disaggregated Memory (DM). The advent of fast networking technologies, such as RDMA and CXL, has sparked interest in Disaggregated Memory (DM) [9, 61]. DM decouples memory from conventional monolithic servers, enabling applications to perform one-sided operations for direct access to remote memory [31, 48]. However, DM has introduced new challenges in runtime systems [16, 60], memory management [47, 61, 62], remote data structures [6, 10, 43, 49, 70], concurrency control [63], and fault tolerance [63, 74, 75].

Disaggregated Key-Value Store (DKVS). Key-value stores (KVS) are a critical component of cloud services. Thus far, KVSes adhere to a monolithic architecture, where computation and memory are tightly integrated within servers [20, 30, 35, 36, 56, 72]. Conversely, DM has paved the way for the concept of disaggregated-memory key-value stores (DKVSes) [46, 67]. DKVS stores its key-value dataset in passively distributed DM servers (memory nodes), reserving compute servers solely to execute application logic [32, 46, 49, 63, 67]. Compute servers can directly access the key-value pairs via one-sided RDMA and issue operations such as get and put requests [46, 63].

Transactional Recoverable KVS. Transactions offer multiple gets and puts in a single operation, making them the de facto programming model for KVSes. Traditional monolithic KVS architectures fail to fully leverage the advantages of disaggregation [22, 23, 37]. For instance, FaRM [22], a state-of-the-art recoverable transactional KVS, does not account for the decoupled nature of memory and compute. Hence, despite allowing for faster RDMA-based recovery than prior works, it imposes a stop-the-world recovery mechanism [23]. Additionally, the reliance on traditional RPC-based recovery during fault-free and recovery operations undermines the disaggregation's benefits, like flexible compute-memory scaling. Hence, these protocols are neither optimal nor applicable for DKVSes. FORD [74] is the first transactional DKVS designed for a full one-sided RDMA-based architecture. However, FORD overlooks recovery, a crucial challenge in DM systems where memory and compute fails independently.

Pandora. Pandora is the first transaction protocol specifically designed to handle recovery in one-sided RDMA-based DKVSes. It offers an end-to-end solution with fast recovery and minimal steady-state performance overhead while ensuring correctness. While Pandora is a non-persistent DKVS, it is compatible with non-volatile memory (NVM) devices. It supports FORD's selective one-sided RDMA flush scheme for efficiently persisting data from the RNIC cache to NVM when needed. Moreover, with battery-backed DRAM, no flushing is required on the critical path to achieve persistence.

8 CONCLUSION

This work reveals that memory disaggregation presents an opportunity to enhance the availability of transactional key-value stores (KVSes), but it also introduces challenges due to the limited one-sided RDMA semantics. To address these challenges, we proposed Pandora, the first one-sided transactional protocol designed for correct, seamless, and fast recovery in disaggregated KVSes.

We introduce two key innovations: Pandora's Implicit Latch Logging (PILL), which ensures latches remain recoverable without impacting performance during fault-free operation, and a novel RDMA-based recovery protocol for fast failure recovery.

Through a litmus-testing framework, we validated Pandora's correctness and uncovered bugs in the state-of-the-art (FORD) protocol, all of which Pandora resolves. Our experimental results show that Pandora recovers from failures in milliseconds, without blocking live transactions or degrading performance in normal, fault-free execution—unlike traditional approaches that can impose significant performance overhead. Our work underscores the importance of integrating thorough testing and careful engineering into the design of recovery protocols to ensure both correctness and high performance.

REFERENCES

[1] 2011. TATP. Telecom application transaction processing benchmark. http://tatpbenchmark.sourceforge.net/

[2] 2021. Smallbank benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/

[3] 2021. Tpc-c benchmark. http://www.tpc.org/tpcc/

[4] Atul Adya. 1999. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. Ph.D. Dissertation.

[5] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. 2018. Remote regions: a simple abstraction for remote memory. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 775–787.

[6] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 120–126.

[7] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 120–126. https://doi.org/10.1145/3317550.3321433

[8] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. [n.d.]. Litmus: Running Tests against Hardware. Springer.

[9] 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 Fifteenth European Conference on Computer Systems*. 1–16.

[10] Hang An, Fang Wang, Dan Feng, Xiaomin Zou, Zefeng Liu, and Jianshun Zhang. 2023. Marlin: A Concurrent and Write-Optimized B+-tree Index on Disaggregated Memory. In *Proceedings of the 52nd International Conference on Parallel Processing*. 695–704.

[11] Luiz Andr Barroso, Jimmy Clidaras, and Urs Hlzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (2nd ed.). Morgan & Claypool Publishers.

[12] Ali Basiri, Niosha Behnam, Ruud De Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos engineering. *IEEE Software* 33, 3 (2016), 35–41.

[13] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10. http://scholar.google.com/scholar.bib?q=info:RPxI-4mKQyEJ:scholar.google.com/&output=citation&hl=en&as_sdt=0,5&ct=citation&cd=0

[14] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[15] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2015. The end of slow networks: It's time for a redesign. *arXiv preprint arXiv:1504.01048* (2015).

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

[17] Zizhong Chen. 2011. Algorithm-Based Recovery for Iterative Methods without Checkpointing. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC '11)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/1996130.1996142

[18] Brian Cho and Ergin Seyfe. 2019. Taking advantage of a disaggregated storage and compute architecture. *Spark+ AI Summit* (2019).

[19] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '17)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/3087801.3087802

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating System Principles*. https://www.amazon.science/publications/dynamo-amazons-highly-available-key-value-store

[21] diy. 2020. diy. https://diy.inria.fr/. [Online; accessed 19-July-2022].

[22] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[23] Aleksandar Dragojević, 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)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[24] Jingwen Du, Fang Wang, Dan Feng, Changchen Gan, Yuchao Cao, Xiaomin Zou, and Fan Li. 2023. Fast One-Sided RDMA-Based State Machine Replication for Disaggregated Memory. *ACM Trans. Archit. Code Optim.* 20, 2, Article 31 (April 2023), 25 pages. https://doi.org/10.1145/3587096

[25] Jingwen Du, Fang Wang, Dan Feng, Changchen Gan, Yuchao Cao, Xiaomin Zou, and Fan Li. 2023. Fast One-Sided RDMA-Based State Machine Replication for Disaggregated Memory. *ACM Trans. Archit. Code Optim.* (mar 2023). https://doi.org/10.1145/3587096

[26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, USA, 1–14.

[27] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (apr 1988), 288–323. https://doi.org/10.1145/42282.42283

[28] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 299–313. https://doi.org/10.1145/2723372.2723726

[29] Bo Fang, Qiang Guan, Nathan Debardeleben, Karthik Pattabiraman, and Matei Ripeanu. 2017. LetGo: A Lightweight Continuous Framework for HPC Applications Under Failures. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. Association for Computing Machinery, New York, NY, USA, 117–130. https://doi.org/10.1145/3078597.3078609

[30] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux J.* 2004, 124 (Aug. 2004), 5.

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

[32] 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)*. Association for Computing Machinery, New York, NY, USA, 417–433. https://doi.org/10.1145/3503222.3507762

[33] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, USA, 11.

[34] Jepsen. 2020. Jepsen. https://jepsen.io/. [Online; accessed 19-July-2022].

[35] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 295–306.

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

[37] 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 Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 185–201.

[38] Prajakta Kalmegh and Shamkant B. Navathe. 2012. Graph Database Design Challenges Using HPC Platforms. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 1306–1309. https://doi.org/10.1109/SC.Companion.2012.160

[39] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. Association for Computing Machinery, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[40] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 201–217. https://doi.org/10.1145/3373376.3378496

[41] 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*. 297–312.

[42] Kyle Kingsbury. 2018. Jepsen: A framework for distributed systems verification, with fault injection.

[43] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. 2021. Farview: Disaggregated memory with operator off-loading for database engines. *arXiv preprint arXiv:2106.07102* (2021).

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

[45] Leslie Lamport. 2002. Specifying systems: the TLA+ language and tools for hardware and software engineers. (2002).

[46] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory (Extended Version). *arXiv preprint arXiv:2209.08743* (2022).

[47] Seung-seob 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)*. Association for Computing Machinery, New York, NY, USA, 488–504. https://doi.org/10.1145/3477132.3483561

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

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

[50] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–12.

[51] litmus7. 2020. litmus7. https://diy.inria.fr/doc/litmus.html. [Online; accessed 19-July-2022].

[52] Rupak Majumdar and Filip Niksic. 2017. Why is Random Testing Effective for Partition Tolerance Bugs? *Proc. ACM Program. Lang.* 2, POPL, Article 46 (dec 2017), 24 pages. https://doi.org/10.1145/3158134

[53] Abdullah Al Mamun, Feng Yan, and Dongfang Zhao. 2021. BAASH: Lightweight, Efficient, and Reliable Blockchain-as-a-Service for HPC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 17, 18 pages. https://doi.org/10.1145/3458817.3476155

[54] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided {RDMA} Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 103–114.

[55] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage.. In *NSDI*, Vol. 17. 17–33.

[56] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, USA, 385–398.

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

[58] Ravi Sethi. 1982. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM (JACM)* 29, 2 (1982), 394–403.

[59] Ravi Sethi. 1982. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM (JACM)* 29, 2 (1982), 394–403.

[60] 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 Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, USA, 69–87.

[61] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. 2022. Towards a fully disaggregated and programmable data center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*. 18–28.

[62] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R Lyu. 2023. Ditto: An elastic and adaptive memory-disaggregated caching system. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 675–691.

[63] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 81–98. https://www.usenix.org/conference/fast23/presentation/shen

[64] Sijie Shen, Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. 2022. DrTM+B: Replication-Driven Live Reconfiguration for Fast and General Distributed Transaction Processing. *IEEE Transactions on Parallel and Distributed Systems* 33, 10 (2022), 2628–2643. https://doi.org/10.1109/TPDS.2022.3148251

[65] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 29, 16 pages. https://doi.org/10.1145/3342195.3387519

[66] James W Stamos and Flaviu Cristian. 1993. Coordinator log transaction execution protocol. *Distributed and Parallel Databases* 1 (1993), 383–408.

[67] 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 Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 3, 16 pages.

[68] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage.. In *NSDI*, Vol. 20. 449–462.

[69] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *Companion of the 2023 International Conference on Management of Data (SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 37–44. https://doi.org/10.1145/3555041.3589403

[70] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1033–1048. https://doi.org/10.1145/3514221.3517824

[71] 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)*. Association for Computing Machinery, New York, NY, USA, 87–104. https://doi.org/10.1145/2815400.2815419

[72] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412. https://doi.org/10.1109/ICDE.2018.00044

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

[74] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 51–68. https://www.usenix.org/conference/fast22/presentation/zhang-ming

[75] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 658–674. https://doi.org/10.1145/3600006.3613135