Dandelion: In-Memory Distributed Transactions with Few Machines

Vasilis Gavrielatos*
Huawei Research

Chris Jensen
University of Cambridge

Antonios Katsarakis*
Huawei Research

Nikos Ntarmos
Huawei Research

ABSTRACT

This paper presents an in-memory, RDMA-enabled, highly-available, transactional Key-Value Store (KVS), dubbed Dandelion, focusing on significantly improving performance in small deployments (e.g., 5 machines). The focus on small deployments is motivated by the anticipated memory expansion (e.g. through CXL), which enables the deployment of in-memory KVSes with few machines but lots of memory.

A small deployment presents locality opportunities that have not been examined by related work. Specifically, it is more likely that at any given time, we must send multiple messages to the same recipient. We leverage this by batching multiple requests in the same network packet. Similarly, it is more likely that at any given time, we have multiple requests that can be served by the local hashtable without going through the network. Sending all requests to the hashtable as a batch allows it to overlap their memory latencies through software prefetching. Finally, it is more likely that the node that requests a key, is itself a backup of that key. We leverage this by allowing local reads from backups.

Our evaluation shows that these optimizations result in a 3.3 - 6.5x throughput improvement over a state-of-the-art system, FaSST, in OLTP workloads in a 5-machine deployment. We characterize the impact and scalability of each of these optimizations with up to 10 machines – where Dandelion still offers up to 3.5× higher throughput than FaSST.

PVLDB Reference Format:
Vasilis Gavrielatos, Antonios Katsarakis, Chris Jensen, and Nikos Ntarmos. Dandelion: In-Memory Distributed Transactions with Few Machines. PVLDB, 18(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

This paper focuses on reliable distributed Key-Value Stores (KVSes). Modern KVSes shard and replicate the data in-memory of multiple servers and provide strongly consistent transactions with high availability. They leverage RDMA for efficient networking to deliver high throughput while scaling into big deployments (e.g., 90 machines). FaRM [11, 12, 33] was the first such work, which sparked a multitude of subsequent works [7, 18, 31, 34–37, 40, 42]. Unlike these works, we focus on small deployments (e.g., 3-10 machines).

Smaller deployments exhibit various forms of locality. For example, it is more likely that at any given time, multiple messages from different transactions must be sent to the same recipient. Similarly, it is more likely that a key-value pair is stored in the machine that is searching for it. Such locality opportunities have, for the most part, not been exploited in the context of rdma-enabled transactional KVSes, because of the assumption that the deployment must always be large. Instead, related work has focused mostly on debating the correct usage of the RDMA primitives [10, 17, 31, 35].

We focus on smaller deployments, partially in anticipation of CXL [1] memory expansion. In its first version, CXL-1 will enable scaling up a few servers by adding more memory at a significantly lower cost (than buying more new servers). Further down the line, it is expected that CXL-2 will enable the pooling of memory, entirely removing the coupling between compute and memory. In either case, we will no longer need a large number of servers simply to fit the dataset in-memory.

This work tackles this challenge by exploiting the locality opportunities presented in small deployments. Specifically, we build *Dandelion (DNL)*, a distributed, in-memory, highly-available, RDMA-enabled Key-Value Store, that achieves up to 6.5x (3.5x) higher throughput than a state-of-the-art system (FaSST [18]) with 5 (10) machines in popular OLTP workloads. Below, we introduce each of DNL's main components – networking, hashtable, and protocol – discussing the relevant locality opportunities we exploit.

Networking - § 5. The main locality opportunity that arises in small deployments is that there is a higher probability that multiple messages must be sent to the same node at the same time. We leverage this opportunity by batching multiple requests and responses in the same network packet. Batching multiple messages in the same packet amortizes the per-packet overheads incurred in CPU (software stack needed to transmit/receive), PCIe, and network (per-packet metadata in NIC caches, packet headers, routing, etc.).

Batching mandates the use of RPCs. RPCs can be implemented with either two-sided or one-sided RDMA. This choice is orthogonal to this work. As we will see in § 5, in small deployments with small messages, batching can yield up to a 10x throughput improvement.

We use eRPC [19], which is the state-of-the-art RPC library that encapsulates the community's RDMA expertise. Crucially, eRPC is a complete product in terms of features, offering support for multiple transports (Infiniband/RoCE, DPDK, UDP), large packets and packet re-transmission. However, eRPC does not support batching. We add a layer on top of eRPC that batches messages to the same packet.

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

*The two authors contributed equally to this work.

While network batching is by no means a new idea, this work is the first to highlight its importance for in-memory, RDMA-enabled, transactional KVSes and characterize its performance benefits.

Hashtable - § 6. In DNL, each server uses a hashtable to store and index key-value pairs. Again, locality facilitates batching, as it is more likely that at any given time, there are multiple requests that must be propagated to the local hashtable (e.g., , after receiving a batch of requests through the network). Batching in the hashtable has been shown to significantly increase throughput by overlapping the memory latencies of the different requests [25, 30].

We design a hashtable, DLHT, which implements a batching API along with a number of other features, such as lock-free operations (Gets, Puts, Inserts, Deletes), special support for transactions, non-blocking resizing, garbage collection, support for variable-size keys and values and iterator API. We could not use an existing hashtable because, to the best of our knowledge, no open-source hashtable includes all these features. We provide a summary of DLHT in section § 6, noting that we have also written a full paper on it (under submission – anonymously accessible in https://bit.ly/dlht).

Protocol - § 4. DNL features a customizable protocol skeleton, through which we implement three protocols. One of the protocols is very similar to FaRM's OCC protocol, while the other two have not yet been explored, as far as we know. Our results show that the three protocols provide very similar performance, even though they have many differences. We expect that the community can use the skeleton to explore more protocols. The protocol skeleton leverages locality, by allowing reads of local replicas, despite of whether the replica is a primary or a backup. This optimization has been explored before ([21]), but not in closely related work, as it provides limited benefit in large deployments.

Batching. Batching is the main performance driver across DNL. In the network, we leverage locality to do batching and amortize the costs per packet. At the hashtable, again, locality enables batching, which allows overlapping the memory accesses of multiple requests. On the protocol side, locality affords reading the local backup, which is beneficial because it increases the batching opportunity for the hashtable (shown in § 8). DNL is architected from the ground up to support and leverage batching across the stack. We discuss this more, along with its impact on latency in the next section.

Contribution Summary. This work anticipates that memory expansion (i.e., through CXL) will open the road for deploying in-memory, transactional KVSes in a small number of machines with access to lots of memory. We build DNL, an in-memory, RDMA-enabled, transactional KVS to leverage three locality opportunities found in small deployments. Specifically, DNL batches in the network to amortize the fixed per-packet costs, batches in the hashtable to enable software prefetching and can read from local backups in the protocol. Our evaluation shows that these optimizations result in a 3.3 - 6.5x throughput improvement in OLTP workloads in a 5-machine deployment over FaSST. We characterize the impact and scalability of each of these optimizations with up to 10 machines, where DNL still maintains a significant throughput benefit.

Limitations. Most notably, this work is not evaluated with CXL hardware. However, we note that due to hashtable batching, all accesses to the index and dataset are software prefetched. For that, we expect that with CXL, our approach will become even more favorable in the face of CXL's additional latency. Similarly to related work [11, 18, 36, 40, 42], DNL does not handle range queries. At the moment, DNL can remain available after a server crash (discussed in § 4.4), but does not replace the crashed node, nor re-replicates the lost replicas. This is currently a work-in-progress. A complete discussion of DNL recovery will merit its own paper in the near future. Related work has done the same [7, 11, 12, 36], or does not support recovery at all [18, 31, 42].

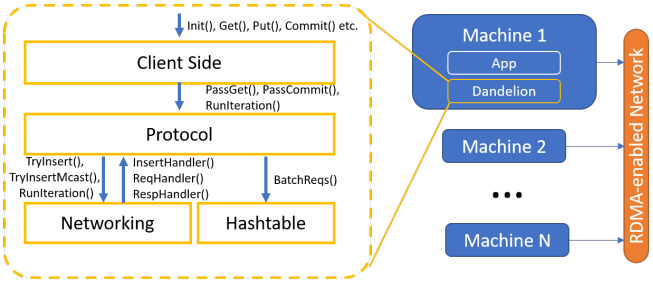

Figure 1: *DNL architecture*

2 PRELIMINARIES

Architecture and assumptions. We assume a symmetric architecture of N machines, where each machine hosts part of the dataset, and all machines execute transactions, communicating with each other through a local-area network. The dataset is comprised of key-value pairs. Each key-value pair is replicated in multiple machines to ensure availability in the face of failures. One of the replicas is typically denoted *primary*; the others are called *backups*. We call the machine executing a transaction the *coordinator* of this transaction.

Further, we make several assumptions, which all exist in closely related work (i.e., FaRM[11], FaSST[18] and DrTM[35]). We assume that client applications run on the same machines as DNL. We target OLTP workloads with short-lived transactions and small values (< 100 bytes). Clients use the KVS as a library, issuing *interactive* transactions; i.e., applications do not a priori know the shape of transactions, which may vary at runtime. We focus on the strongest consistency (strict serializability [32]), and can recover in the event of a machine crash (§ 4.4). We do not target durability, but note that the technique used in FaRM is applicable to DNL.

API. Transactions are issued through the multi-key API, whose core functions are: Init(), Get(), Put(), Insert(), Delete(), Commit() and Abort(). We also offer a single-key API for transactions that access only a single key; the core functions are GetOne(), PutOne(), InsertOne(), DeleteOne().

Transactional protocols. A transaction is logically split in two phases, the *execution phase* in which requests are issued and the *commit phase* which validates that reads were consistent and applies all updates atomically. In DNL, during the execution phase, reads are executed but all other requests are buffered. The commit phase is configurable, offering actions such as locking, logging and read validation.

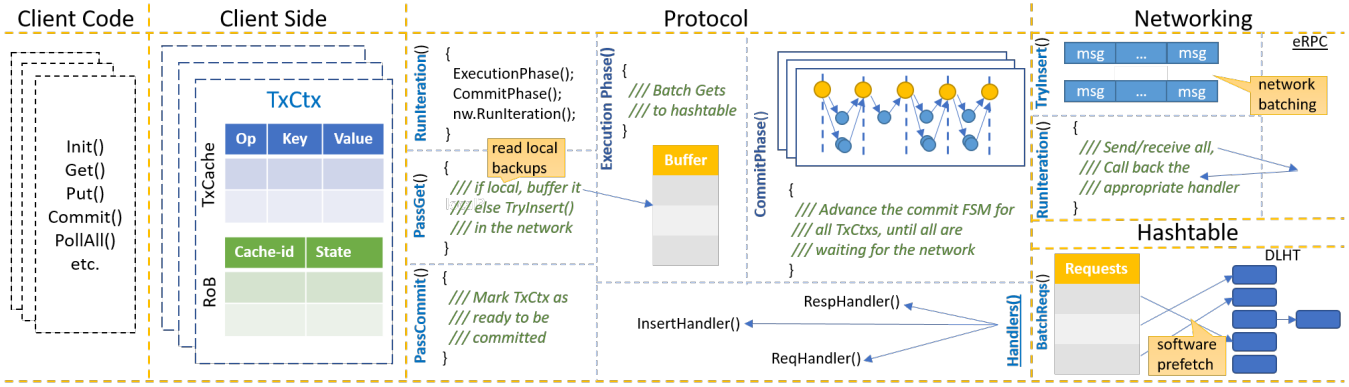

Figure 2: *An overview of the DNL architecture*

Servers	10 x Two-socket 18-core Intel Xeon Gold 6254 (72 h/w threads in total incl. hyper-threads)
Hardware Caches	36× 1MB L2 | 2× 24.8MB L3
System Memory	4× 32GB DDR4-2933 (128GB in total)
OS / Kernel	Ubuntu 20.04.3 | Linux 5.4.0-90-generic
NICs	Mellanox ConnectX-5 100GbE (single-port QSFP28 PCIe3x16)

Table 1: *Cluster configuration*

Threading model. DNL runs in the same threads as the client code. I.e. we do not spawn any additional DNL threads. Crucially, in each thread, DNL only communicates with one thread in each remote machine of the cluster. The only shared data between the threads is the hashtable. Each thread works on its own transactions, oblivious of the existence of other threads. This model is scalable and is also used in FaRM and FaSST. Our design description focuses on a single thread of client code which uses the DNL API to issue transactions.

Testbed. Table 1 describes our 10-machine cluster.

Software Stack. Figure 1 shows the DNL software stack, including how components communicate with each other. Client code uses the DNL API to issue transactions. The Client Side propagates the transactional requests to the Protocol, which executes them using the Hashtable and the Networking in the process.

Batching and Latency. Batching is a common theme across the design of DNL. Requests are buffered by the protocol and batched to the networking or the hashtable. Crucially, we never wait to fill a quota. Instead, we work in each module until there is no more work to be done. Then we move to the next module. Client code executes until it cannot make any progress. At this point, the protocol takes over and works until it cannot make any more progress; then, the networking takes over. Each time a module takes over, it executes its batches. Similarly to prior work [17, 18], we refer to this pattern as *opportunistic batching*. Because we never wait to fill a quota this has a minimal impact on latency. When the load is low the batching is low. At higher load, we use batching to increase throughput and thus keep latency from exploding. We corroborate this in § 8.

Deployment size with CXL. One of the purposes of the CXL spec [1] is to loosen the tight coupling of memory and compute. Initially, type-3 of CXL-1 will allow for memory expansion. I.e. we will be able to plug DIMMs in PCIe slots[1]. Consider the example of FaRM, which requires 90 machines for a dataset of 4.9TB (14.7TB with 3-way replication). With 5 machines with 4TB each, we can fit this dataset (plus the indexes and other overheads). This is possible today without CXL, but will require very expensive large-capacity DIMMs. (For reference, a 64GB DIMM may cost around 200 USD and a 128GB DIMM around 1k USD and a 256GB DIMM more than 3k USD). Even buying the most expensive DIMMs will be a lot cheaper than buying, maintaining, and cooling another 85 machines. With the additional DIMM slots provided by CXL-1, we can reach the goal of 4TB per machine, through smaller, cheaper DIMMs.

To avoid this scale-up approach, CXL-2 specifies a much more aggressive scheme, where multiple servers can have access to a pool of DIMMs through a CXL switch. In this case, the memory will still be statically allocated to one server at a time. Sharing the pooled memory is left to CXL-3.

Recent industry-led works suggest that CXL will come into the mainstream [2, 13, 24, 27]. Adding to their arguments, we show that emerging memory expansion can enable a range of locality-based optimizations that significantly increase the transaction load that each server can sustain.

In the next four sections, we go through each of the four main modules in our software stack: Client Side, Protocol, Networking and Hashtable using Figure 2 to demonstrate various aspects of the design.

3 DNL CLIENT-SIDE

Applications use DNL as a library. Client code interfaces with DNL through a construct called *transaction context (TxCtx)*. Client code uses the TxCtx to issue API calls, such as Init, Get, Put, Insert, Delete, Commit, and so on. Calls are asynchronous by default. For instance, after issuing a Get, the value will not be available after the Get call has returned. A set of API calls allows the client to poll

[1]As of submission, there are no type-3 CXL-1 commercial products.

for the completion of a specific request or all previous requests. All API calls have a synchronous flavor.

As shown in Figure 2, a TxCtx encapsulates a *transaction cache (TxCache)* and a *reorder buffer (RoB)*. A TxCache consists of an array of entries acting as a cleverer log of a transaction. In this array, each entry holds a key-value pair plus some necessary metadata, such as the type of operation (Get, Put, etc.) and the state of the request (e.g., `Success`, `NotYetDone`, `KeyNotFound`, etc.). Each cache has a compile-time configurable height (by default 16 entries) and width (by default 128 bytes per entry); when more space is needed in either direction, we allocate it on the spot.

The RoB is a lock-free ring buffer that maintains the order of requests within a given transaction. Each entry in the RoB is a pointer to a TxCache entry. Respecting the order of requests is not required for correctness, but improves the usability of the API. Specifically, after the client polls for the completion of a Get, using the RoB we guarantee that all previous Gets have also completed. This relieves the client from having to poll for every Get. It also enables the efficient implementation of the PollAll API call. Notably, only the Get and Commit calls allocate an RoB entry, as the rest of the calls do not need to be propagated to the protocol explicitly.

As an example, assume the following client code that issues a transaction to DNL.

```
1  TxCtx tx;
2  tx.Init();
3  tx.Get(key1, val1);
4  tx.Get(key2, val2);
5  coroutine_yield;
6  tx.PollAll();
7  if (val1 == val2) {
8    tx.Put(key1, 0);
9  }
10 auto ret = tx.Commit();
11 coroutine_yield;
12 PollRequest(ret);
```

On a Get, we first probe the TxCache. If we do not find an entry for the requested key, we allocate a new entry in the TxCacheand another entry in the RoB. In the latter case, we then notify the protocol of the new request (through the `PassGet()` function call, discussed in the next section). The protocol buffers the request (i.e. a pointer to the TxCache entry), but does not execute it yet. We then do the same for the second Get. At this point, the client yields to another coroutine. We discuss the use of coroutines immediately after this example. Before using `val1` and `val2`, we call `PollAll()`, which returns only after all previous requests have been completed. `PollAll()` calls on the protocol to run an iteration, executing all buffered requests, including the two Gets. After executing the Gets, the protocol copies the values for `val1` and `val2` in their respective entries in the TxCache and tags them completed in the RoB. The Gets return pointers to the TxCache entries, so no further copies are needed. Suppose the two values are equal. The first Put request will find that there is already a TxCache entry allocated for `key1` and will fuse its operation on it. It will change the type of the entry to `GetPut`, overwrite the value in the same TxCache entry and return to the client. Notably, the Put need not allocate a RoB entry or be propagated to the protocol yet. Finally, the Commit allocates a RoB entry and gets communicated to the protocol. The protocol buffers this information and returns. When we poll for the commit, we trigger the protocol to run an iteration, during which it will run its commit protocol.

Parallelism within a client thread. To increase parallelism, clients must be able to issue multiple transactions from each thread. We facilitate this through the asynchronous API. The clients can take advantage through the use of coroutines. Specifically, in the above code example, before polling for the Gets, the client yields to a different coroutine. After cycling through all of the coroutines, the client can then poll. This allows for a bigger batch to be created on the protocol side. Coroutines are a standard practice in related work (e.g., FaRM, FaSST, DrTM) to increase parallelism.

Summary. DNL offers an asynchronous API. A small TxCache holds the state of the transaction and an RoB maintains ordering so that clients need not poll for every request explicitly. Client code can leverage the asynchronous API to issue multiple transactions concurrently through coroutines.

4 DNL PROTOCOL

We start our discussion with an overview that describes the actions taken by the protocol when probed by the client-side. Then, we dive into the specification of the protocol, describing the execution phase and the customizable commit phase. Finally, we discuss the locality optimization and the recovery mechanism.

4.1 Overview

The client-side module interfaces with the protocol via three functions: 1) `PassGet()`, which is called after a Get request is issued during execution; 2) `PassCommit()`, which is called after a Commit request and 3) `RunIteration()` which is called when the client polls for a request that has not yet been completed. This interface is illustrated in Figures 1 and 2.

1. `PassGet()`. First, we identify the primary and backup nodes of the requested key. If the key exists locally, either as primary or backup, we buffer the request so that it can be batched to the hashtable later. Otherwise, we call `TryInsert()` on the networking, which inserts the Get request in a network packet. This packet will be sent when `RunIteration()` is called, to facilitate batching at the network.

2. `PassComit()`. A client thread can have up to a maximum of N ongoing transactions (N is a configuration parameter). The protocol maintains metadata for each of these transactions. On `PassCommit()` we simply mark the transaction as ready to be committed and return.

3. `RunIteration()`. This function takes three steps: 1) batches all Gets that can be executed locally to the hashtable and writes back the values to the corresponding TxCaches. 2) for each transaction that must be committed, it makes progress until it reaches a point where further progress requires a response from a remote node 3) it notifies the networking layer that it should send all buffered packets and poll for remote messages (through `network.RunIteration()`).

4.2 Protocol Specification

A transaction is logically split into two phases, the *execution phase* in which requests are issued, and the *commit phase*, which is initiated when the client code calls `Commit()`. Before we delve into the two phases, note that the protocol requires that each key-value pair

Version	29-bytes, used by the protocol to validate reads
Lock	1-bit, used by the protocol commit phase to lock objects
Readable	1-bit, used in backups to denote that a Tx holds the protocol lock on the primary
Deleted	1-bit, used to denote whether the key has been deleted

Table 2: *Key-value metadata used by the protocol(s)*

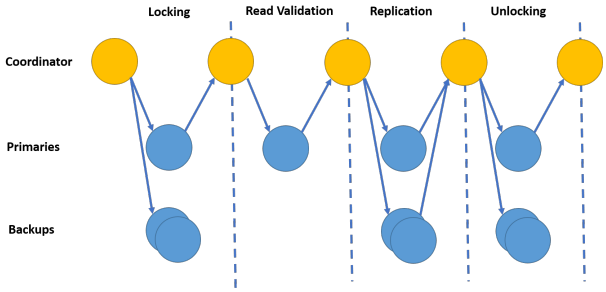

Figure 3: *Skeleton of the DNL commit-phase protocol. We create 3 concrete protocols by specializing this skeleton.*

Protocol	Locking	Read Validation	Replication	Unlocking
Dnl-4	P W	Yes	B W	P W
Dnl-2	P R & W	No	P & B W	P R & W
Dnl-1	P & B R & W	No	No	P & B R & W
Dnl-RO	No	Yes	No	No

Table 3: *Customizing the commit phase to create three different OCC protocols committing in 4,2 and 1 rtt. Read-only transactions are committed through Dnl-RO in all cases. (P = primary, B = backup, R= read-set, W = write-set)*

is accompanied by certain metadata (Table 2). The "Lock" is orthogonal to thread-safety. Instead it is a protocol-level lock ensuring isolation for distributed transactions. For thread-safety, we use a spin-lock for each key-value pair.

4.2.1 Execution Phase. During the execution phase, only Gets are executed while the rest of the requests are buffered in the TxCaches. To perform the Get, the protocol identifies the set of machines that replicate the requested key. If the coordinator is one such machine, then the Get will be executed locally; otherwise, a Get RPC is sent to the key's primary replica.

If the key was not found or has been marked deleted, then an error code is reported to the client. Otherwise, the value is copied in the corresponding TxCache entry, to be used by the client. Note that we return the value, even if it is locked by the protocol or it is tagged *not readable*. During the commit phase, we will check if the value was legally read, and if not, we will abort the transaction.

Recall that we also offer single-key transactions. In this case, a single-key `GetOne()` transaction does not have a commit phase. For this reason, in GetOnes, we return an error code if we find that the key is locked by the protocol, or is tagged as not-readable. Therefore, the readable flag is needed only to support GetOnes.

4.2.2 Commit Phase. Figure 3 outlines the commit phase skeleton. It entails 4 steps: 1) Locking, 2) Read Validation, 3) Replication and 4) Unlocking. The four steps can be customized to create different protocols. We first describe the four steps, and then the three protocols we implement by customizing the steps.

1. Locking. For each key in the transaction, we lock it in all of its replicas. If the access is not a Get, we also write it in a log. Locking will fail in two cases: 1) if the key is already locked or 2) if the version does not match the version we read during the execution phase. The latter is only applicable if the key was read during the execution phase. If locking fails, we transition to the Unlocking step with the intention to abort.

2. Read Validation. For each read in the read-set of the transaction, we validate that the key is unlocked and still has the same version in its primary replica. If it is not, we transition to the Unlocking step with the intention to abort.

3. Replication. If we have reached this step, then the transaction can commit successfully. This step ensures that we replicate the transaction state before reporting success to the client. For each key in the transaction, we access all of its replicas and log the corresponding operation. For each key in the write-set, we also lower the Readable flag in each of its backup replicas. This ensures the correctness of GetOne operations (as discussed in § 4.2.1).

4. Unlocking. In this step, we issue unlocks to all replicas that were locked during the Locking step. We transition to this step, either when we must commit or abort. The unlock messages include this information. We can notify the client about the result of her transaction either after sending all unlock messages, or after receiving all acks. Notifying the client early, before receiving the acks for the unlocks saves a network round-trip time in the latency perceived by the client, but may have repercussions on the recovery. We discuss this in more detail in § 4.4.

Logging. We use undo logging in the backups. Hence, we apply any updates during the Replication step and log the previous value. This is beneficial compared to the redo logs used by FaRM because, if the transaction has reached this step, it means it will commit (unless a failure occurs). Using undo logging means that we need not communicate with the backup again, since we have already applied the update.

Raising readable flag. During the Replication step, we lower the readable flag, denoting that the key is locked and GetOnes should not return its value. We raise the flag, when we know that the transaction has committed. This information is piggybacked in messages of subsequent transactions.

4.2.3 Customizing the Commit Phase. We customize the commit phase by creating three protocols which we call *Dnl-4*, *Dnl-2*, and *Dnl-1*. The names refer to the number of network round-trip times (rtts) needed before we can report commit to the client. For instance, in Dnl-1, we can report commit after 1 rtt (but need 2 rtts in aborts).

Table 3, describes how each step of the commit phase is customized for each protocol. Specifically, the Locking step can be customized to be performed for the write-set (W) only or to also include the read-set (R). And it can visit only the primaries (P) or include the Backups (B). Similarly, for the Replication step. Unlocking always mirrors the Locking step. Any step can be elided.

Dnl-4 is largely the protocol used in FaRM. It locks the primaries of the write-set, validates reads, logs the write-set in the backups and unlocks the primaries. Dnl-2 locks the primaries of both reads and writes, logs the write-set in both backups and primaries, and finally unlocks. Dnl-1 locks both reads and writes in all replicas, and if successful, it skips to the Unlocking step. In all protocols, read-only transactions are handled specially through Dnl-RO which only performs read-validation. This is what happens by default in Dnl-4.

4.3 Locality: Read from local backups

To take advantage of the small deployment, the protocol allows Gets to read from a backup, when a backup is found locally. This removes the overhead of sending a remote RPC to execute the Get. In a deployment with N machines, the probability that a Get executes locally increases from $1/N$ to R/N, where R is the replication degree. For instance, in a deployment with 5 machines and $R = 3$, our optimization increases the chance that a Get executes locally from 20% to 60%. But in a deployment with 50 machines, the same percentage increases from 2% to 6%. The performance penalty for this optimization is that we must set and reset the *readable* flag in all backups, every time we lock a primary. Based on this we hypothesize that the optimization will be more impactful on read-intensive workloads and will not scale well in write-intensive workloads. We investigate this in § 8.2.

4.4 Recovery

At the moment, DNL offers a limited form of fault tolerance. It can recover after a server crash and continue operation, but does not replace the crashed node nor does it re-replicate the lost data replicas.

On a crash, all operation is halted and control transfers to the recovery protocol. Each server examines its own coordinated transactions and its logs with transactions coordinated by remote servers. For each transaction, a decision is reached: commit or abort. Once all decisions for all transactions are applied, recovery has completed and operation can resume.

In each server, we maintain an epoch-id counter. When initiating recovery we increment this counter in all live nodes. This allows servers to disregard messages from previous epochs (messages are tagged with an epoch-id). We must also ensure that the decision reached for each transaction does not contradict any decision that has already been reported to the client (abort or commit) e.g., if the client believes its transaction is committed, we cannot abort it during recovery. This entails a synergy between the transactional and the recovery protocol. When the transactional protocol notifies the client of a decision, then it must be that after any f crashes (with replication degree $f+1$), the recovery protocol can implement this decision.

> ***Correctness***: For all three protocols, we have formally verified in TLA+, that they provide strict serializability and can recover from crashes without contradicting the decisions that have been reported to the client.

4.5 Summary

The protocol creates batches of requests by buffering requests and executing them all together, when probed by the client. The execution phase only issues Gets, reading local backups if they exist. The commit phase can be customized to implement different protocols. We implement and present three protocols. We can recover after a crash, but do not currently handle all aspects of recovery.

5 DNL NETWORKING

5.1 Overview

As shown in Figure 1, the networking library exposes three functions to the protocol: `TryInsert()`, `TryInsertMcast()` and `RunIteration()`. In turn, the protocol must also implement three handlers: `InsertHandler()`, `ReqHandler()` and `RespHandler()`.

TryInsert(). Whenever the protocol reaches a state where it must send a message it calls `TryInsert()`. The network identifies the buffer that holds the appropriate network packet and calls the protocol's `InsertHandler()` passing it the pointer where it must write its message. If there is no buffer available it returns false, and the protocol must retry in the future.

TryInsertMcast(). Same as `TryInsert()`, but the message is written in in multiple buffers that are later sent as unicasts to the multicast group.

RunIteration(). This function first sends all buffered messages and then checks if any new messages have been received. For each received message it calles the appropriate handler, `ReqHandler()` or `RespHandler()`.

Flows. The networking library exposes the abstraction of *flows*. A flow is a group of messages that can be batched together in the same network packet. Separate handlers must be implemented and registered for each flow. DNL uses three flows, one for the execution phase, one for the commit phase and one for the recovery. Using multiple flows reduces the opportunity for batching, but simplifies the software.

5.2 eRPC

The Networking layer is implemented over the eRPC networking library [19]. eRPC implements RPCs over UD RDMA, DPDK or UDP. We use RDMA over Converged Ethernet (RoCE). eRPC is a feature-complete library, supporting features such as packet retransmission and packet fragmentation. It also makes use of commonly known RDMA optimizations such as doorbell batching [3, 17]. Furthermore, its is well-documented with high-quality code, easy to use and its many features do not hinder performance in the common case. We performed a thorough exploration of eRPC performance and found

four performance bugs, which we fixed. Specifically, we removed an unnecessary header in RoCE messages, increased the RDMA inlining limit to 188 bytes (from 60),[2] spawned the buffers in the socket that the thread is pinned rather than the socket that the NIC is attached to, and finally, we removed the 4-byte immediate from the header, which was not being used. We discuss the impact of these fixes in the next section.

5.3 Batching layer

Over eRPC, we implement a layer that batches multiple messages in a single packet. Each batch of requests, results in a batch of responses. Batching is not a new idea. However, it is typically not used in modern rdma-enabled OLTP KVSes despite its large benefits in cases where small messages abound. The rest of this section analyzes the impact of batching.

To do this, we perform an experiment where we use DNL as a single-machine KVS and two other machines issue Get requests for 8B keys and 16B values. The KVS holds 10 million keys, but all requests are for the same key. This is done in order to isolate the performance of the networking library. Figures 4-6 characterize the effect of batching through measurements on the machine that acts as the KVS. Specifically, Figure 4 shows the throughput of the KVS in millions Gets per second while scaling the threads. We run three configurations 1) DNL: is regular DNL which can batch up to 100 read requests/responses in every network packet. 2) DNL-no-batch: we disable batching and 3) eRPC: we also remove the fixes for the performance bugs that we mentioned in the previous section. Figure 5 focuses on DNL (64 threads) and shows three statistics as we increase batching from 1 to 64: the throughput, the send network bandwidth (Gbps) and the number of packets sent per second. Finally, Figure 6 breaks down the send network bandwidth into goodput and headers when increasing the batching degree (using 64 threads).

We make the following observations. First note that our optimizations on eRPC yield a throughput improvement from 20% with 1 thread to 5x with 71 threads. From this point on, we always use the optimizations and do not discuss them again. Batching yields from 2.8x improvement up to more than 10x at 71 threads. As batching increases we cannot sustain the same packet rate, because packets become larger and thus more expensive to create, DMA, transfer etc. Adding to this, when batching more than 6 responses, the payload surpasses the inlining threshold (188 bytes) and thus the NIC must do a second DMA per packet to fetch the payload (the first DMA fetches the work request). This is why increasing batching from 4 to 8 is underwhelming. However, with bigger packets, we use more of the available network bandwidth (100Gbps) because we send fewer packets amortizing the required costs per packet, such as the NIC DMAs to fetch the packet payload or the required per-packet metadata that are cached in the NIC. Crucially, a lot more of that bandwidth is spent on application data (rather than on headers).

[2]This is not part of the specification, even though the programmer must decide at a per-packet basis if a packet can be inlined. To find the inlining limit, we modify the driver, such that on initialization of DNL we can query it. We assume eRPC uses 60 Bytes as a lower bound for safety.

5.4 Locality: Network batching

In our Introduction, we hypothesized that network batching is useful in small deployments because of locality. Simply put, it is more likely to find two messages for the same recipient at any given time. However, in Figure 5, we observe that even batching two requests yields almost a 2x improvement in throughput. This hints that batching may be useful even in the presence of lower locality in bigger deployments. We explore this further in our Evaluation (§8).

5.5 RDMA primitives debate

Related work has extensively debated the use of two-sided vs one-sided RDMA. One-sided RDMA cannot execute in a gather/scatter fashion; i.e., we can only read/write at a single memory location at a time. This means that we need RPCs in order to do batching in the network, as reading/writing multiple locations directly with RDMA Reads/Writes would require multiple network packets. Therefore, this work takes the position that in a scenario where small messages are exchanged at a high rate we must use batched RPCs; the primitive through which we implement the RPCs, be it one-sided or two-sided, is of secondary importance.

5.6 Summary

We identified eRPC [19] as a feature-complete, high performance networking library for RPCs over RDMA. We fix a few performance bugs in eRPC that yield 1.2x - 5x throughput improvement, and implement a batching layer on top of it, which yields another 2.8x - 10x throughput improvement.

6 DNL HASHTABLE (DLHT)

We design a new hashtable called DLHT. In this section, we provide a summary of DLHT, noting that we have also written a full paper on it (will appear in HPDC'25 – accessible via https://bit.ly/dlht).

We design a new hashtable because fast, open-source hashtables lack many important features, that are necessary to make them usable in practice. Specifically, DLHT supports a very high-throughput CRUD API and comes with its own garbage collector. It can handle keys and values of any size in the same instance and it implements a very fast resizing algorithm that only blocks operations to a single bucket at a time. It offers significant optimizations for small values that can be inlined in the index and it can handle non-unique keys. Most of these features are not found in open-source hashtables [9, 25, 26, 30].

Besides sacrificing core functionality, state-of-the-art designs still incur multiple memory accesses per request and block request processing in three cases. First, most hashtables block while waiting for data to be retrieved from memory. Second, open-addressing designs, which represent the current state-of-the-art, either cannot free index slots on deletes or must block all requests to do so. Third, index resizes block every request until all objects are copied to the new index.

Defying folklore wisdom, DLHT forgoes open-addressing and adopts a fully-featured and memory-aware closed-addressing design based on bounded cache-line-chaining. This design offers ❶ lock-free operations and deletes that free slots instantly, ❷ completes most requests with a single memory access, ❸ utilizes

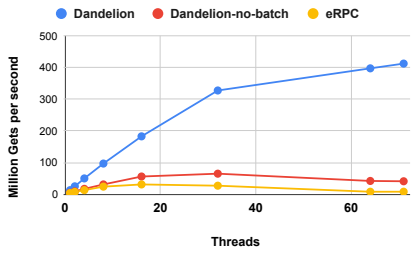

Figure 4: *Get Throughput of single-machine KVS (8B keys 16B values).*

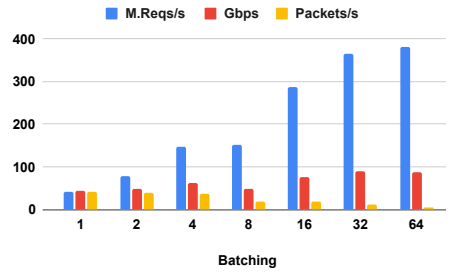

Figure 5: *Get Throughput of single-machine KVS while varying batching.*

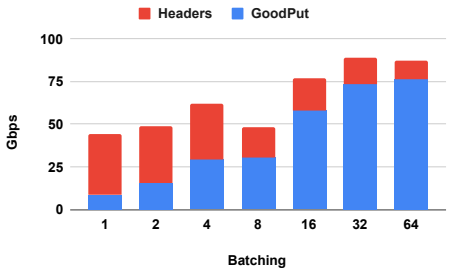

Figure 6: *Percentage of goodput in single-machine while varying batching.*

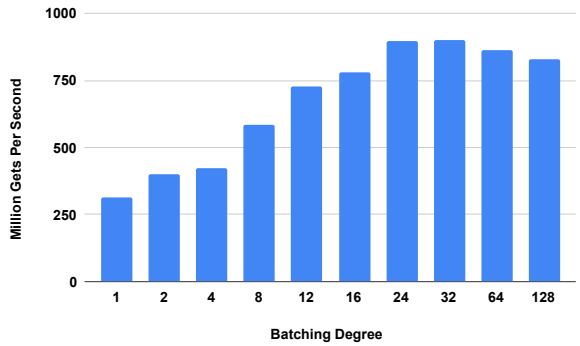

Figure 7: *DLHT Get throughput varying the batching degree*

software prefetching to hide memory latencies, and ❹ employs a novel non-blocking and parallel resizing.

Locality. Our DLHT paper compares with eight other high-performance hashtables (the fastest we could find). Here we focus solely on how we can take advantage of the locality that is present in smaller clusters. Specifically, with a small cluster, it is more likely that at any given time, there are multiple requests that must be propagated to the local hashtable.

Having a batch of requests allows us to overlap their memory latencies. As shown in Figures 1 and 2, DLHT offers a batching API (`BatchReqs()`), that allows clients to propagate a batch of hashtable requests. Before executing the requests, DLHT loops through the batch issuing a software prefetch for the hashtable location of each request. Thus the memory accesses of the batched requests are overlapped. This is a crucial optimization because hashtable accesses are random by nature and thus cannot be prefetched by the hardware. This technique has been shown by DLHT but also prior work [25, 30] to increase throughput by more than 2x.

Figure 7, shows the throughput of DLHT for Gets that access random keys in a 4GB index that holds 100m 8-byte keys (with 8-byte values). Note that (unlike previous Figures) there is no network involved in Figure 7; this is within a single-node. We simply measure the number of Gets we can do to the hashtable. The measurement is taken with 16 threads. With 24 requests, we can achieve an improvement of more than 3x in throughput. However, when only few requests are available, e.g., four, the improvement is not very big. This hints that this optimization will work best in small deployments, but may not scale as well. We corroborate this hypothesis in our evaluation (§ 8).

7 EXPERIMENTAL METHODOLOGY

Table 1 summarizes the cluster hardware. By default, we run with 5 machines and use a replication degree of 3 in all experiments. We use 2 MiB hugepages and we pin threads to cores. We briefly discuss our workloads next.

Smallbank. Smallbank is a write-intensive OLTP benchmark that simulates bank accounts. 15% of transactions read a single key.. Keys are 8 bytes and values are 8 bytes. Each machine is the primary for 18 million keys and the backup for 36 million keys. We chose this number, because it is the maximum number of keys we could use in FaSST.

Tatp. Tatp is a read-intensive OLTP benchmark that simulates a telecommunication workload. In Tatp, 70% of transactions read a single key Keys are 8 bytes and values are 48 bytes. Each machine is the primary for 11 million keys and backup for 22 million keys. We chose this number, because it is the maximum number of keys we could use in FaSST.

Microbenchmarks. Similarly to FaSST we use the notation O(G,P) to specify a microbenchmark with G Gets and P Puts. We run O(1,0), O(0,1), O(4,0) and O(4,2). Keys are 8 bytes and values are 16 bytes. We select keys at random. Each machine is the primary for 18 million keys and the backup for 36 million keys (again, this is the maximum in FaSST).

Evaluated Systems. We will evaluate the three protocols implemented in DNL (Dnl-4, Dnl-2 and Dnl-1). The most relevant systems we can compare against are FaSST [18], DrTM [35] and FaRM [31]. They are all in-memory, RDMA-enabled transactional systems, whose main differences are in the choice of RDMA primitives. FaSST argues for 2-sided, FaRM argues for 1-sided, and DrTM argues for a hybrid approach. FaSST and DrTM show significant improvements over FaRM, so, we do not discuss FaRM in our evaluation (also FaRM is closed source). We could not get DrTM to run in our setup. However, the infrastructure used in the DrTM evaluation [35] is very similar to ours, so we compare with the numbers from their paper. We run FaSST, observing that we measured very similar throughput/latency as reported in their paper. Similarly, to DNL, FaSST uses 2-sided RDMA and employs batching in the hashtable.

Metrics. In our graphs, we use the term *Total Mtps*, to stand for the total system throughput (from all machines) in million committed transactions per second.

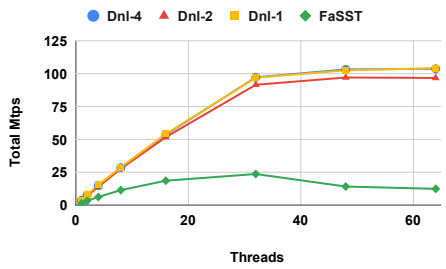

Figure 8: *Smallbank throughput*

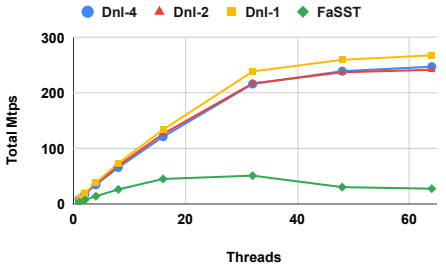

Figure 9: *Tatp Throughput*

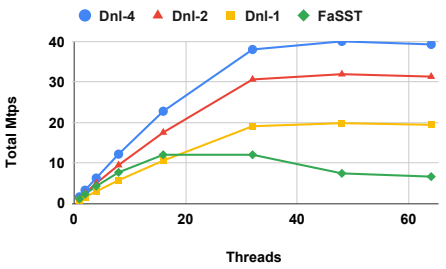

Figure 10: *O(4,2) throughput*

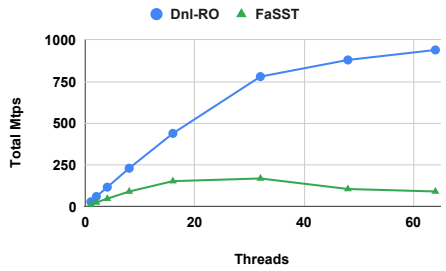

Figure 11: *O(1, 0) throughput*

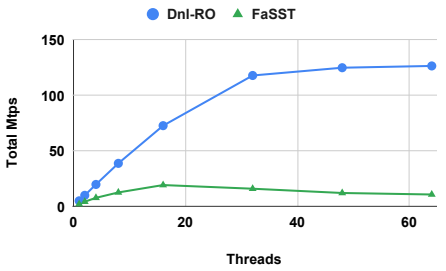

Figure 12: *O(4, 0) throughput*

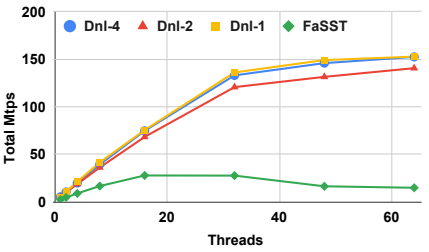

Figure 13: *O(0, 1) throughput*

8 EVALUATION

The main thesis of this paper is that we can leverage the locality found in small deployments to increase throughput. To corroborate this, we first compare the throughput and latency of DNL with FaSST with 5 machines (§ 8.1).

Our thesis raises a crucial question. *How big is a "small deployment"?* In previous sections, we hypothesized that the three locality optimizations scale differently. To characterize this we do a breakdown of the impact of each optimization as we increase the cluster size to 10 machines (§ 8.2).

Then we perform two additional studies. Firstly, we focus on network batching to better understand how it impacts performance as a function of the load (§ 8.3). Finally, in § 8.4, we vary the number of keys that are stored in each machine. This allows us to emulate the system's behaviour with a skewed workload (with very few keys) and ensure that performance does not degrade with more keys.

8.1 Performance in a small deployment

Figures 8 to 13 show the throughput of the four evaluated systems across all six benchmarks when varying the number of threads. The experiments are run with 5 machines. In the read-only benchmarks, (O(1,0) and O(4,0)) the DNL protocols are the same, i.e., Dnl-RO. Figures 14 and 15 show throughput vs latency of all evaluated systems as the load increases.

In Smallbank, DNL protocols achieve about 100 Million transactions per second (Mtps). FaSST achieves up to 23 Mtps. From [35], DrTM achieves up to 35 Mtps, but with 2-way replication (we use 3), using two 100 Gbps NICs per node (we use one) and assuming that 4% of the keys are accessed by 90% of transactions. Crucially 4% is not a small enough number to cause a high abort rate. In § 8.4 we show that DNL performance in Smallbank improves when using only 4% of our default number of keys. Crucially, 4% is small enough so that DrTM can cache all indexes for only these keys in all nodes. This is very beneficial for DrTM, because servers can Get the keys with a single RDMA Read. The question of how this hot 4% is identified remains unanswered. In Tatp, DNL protocols achieve up to 268 Mtps. FaSST achieves up to 51 Mtps. DrTM does not run Tatp.

When we compare the three protocols of DNL, we observe that their performance is very similar with the exception of O(4,2). This is because the protocols take very similar steps unless the transaction has Gets and Puts to different keys. This pattern exists only in O(4,2). The difference is that on a Get, Dnl-4 performs read-validation for the key, Dnl-1 locks all replicas of the key and Dnl-2 locks its primary. These differences are not visible when we do only Gets, because then all protocols execute Dnl-RO or when Gets and Puts are to the same key, because then this operation is treated as a Put (more precisely a GetPut).

With respect to latency in Figures 14 and 15, we observe that in Tatp, the 50th percentile latencies are very small in all systems, because 70% of all transactions are Gets to a single key. The latencies are similar in DNL with FaSST, for the the same throughput, despite the batching on DNL. This is because the batching is opportunistic and thus, when the load is small the batching is also small.

8.2 How big is a small deployment?

In this section we investigate the scalability of our three locality optimizations when increasing the size of the deployment up to 10 machines. We focus on Dnl-4 and we run Smallbank and Tatp with 64 threads.

Firstly in Figures 16 and 17, we compare the total throughput in

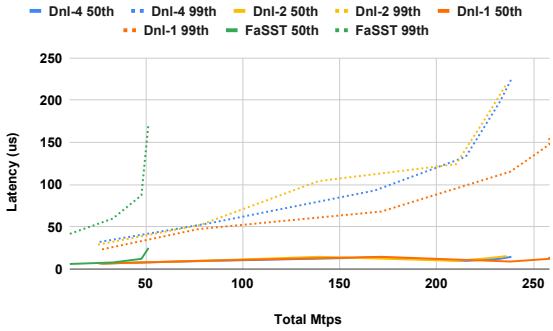

Figure 14: *Tatp: Latency vs Throughput*

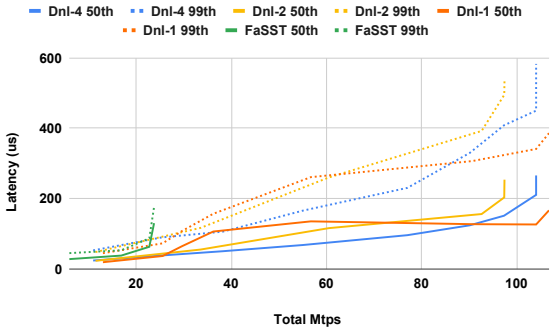

Figure 15: *Smallbank: Latency vs Throughput*

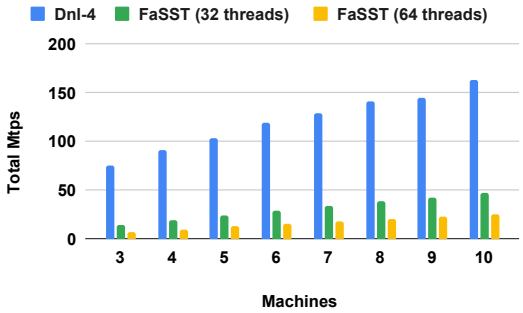

Figure 16: *Smallbank: scaling up to 10 machines*

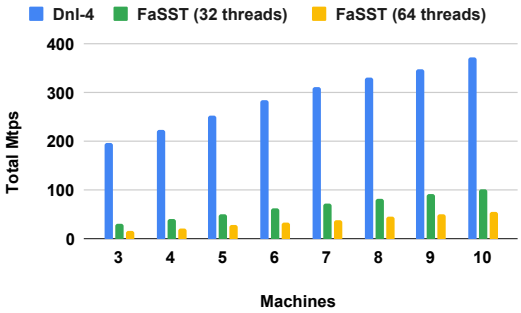

Figure 17: *Tatp: scaling up to 10 machines*

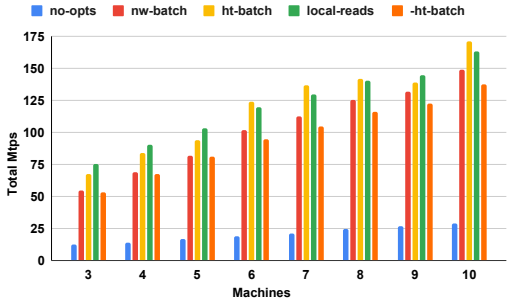

Figure 18: *Smallbank: throughput vs machines with opts*

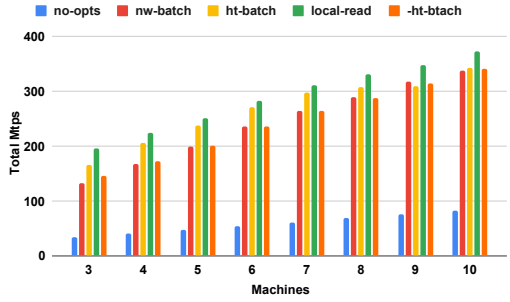

Figure 19: *Tatp: throughput vs machines with opts*

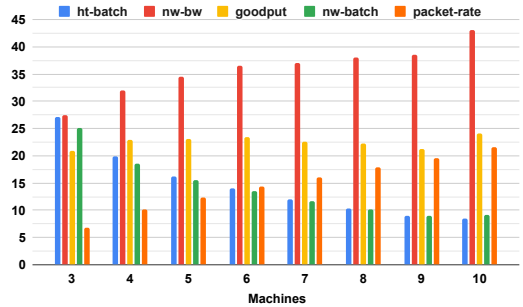

Figure 20: *Smallbank: analysis of batching*

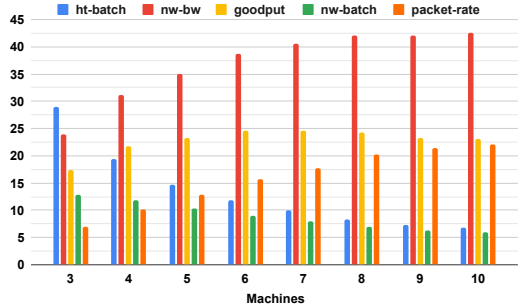

Figure 21: *Tatp: analysis of batching*

million transactions per second (Mtps) of Dnl-4 with FaSST as we increase the number of machines. Note that we show FaSST with both 64 and 32 threads. This is because 64 threads is our default, but FaSST maximizes throughput with 32 threads. As the number of machines increases the gap between Dnl-4 and FaSST closes because locality decreases. However at 10 machines DNL offers more than 3x higher throughput in both workloads. Figures 18 and 19 show the total throughput of Dnl-4 as we increase the cluster size and as we add and remove optimizations. Specifically, *no-opts* includes none of the optimizations. We incrementally add optimizations: *nw-batch* adds network batching; *ht-batch* adds hashtable batching; *local-reads* adds the ability to read from local backups. Finally, -*ht-batch* removes hashtable batching, but leaves local-reads and nw-batch.

Without any of the optimizations DNL's performance is similar to FaSST. Network batching is by far the most influential optimization and scales well up to 10 machines. Ht-batch and local-reads work best when employed together. This is shown in the last bar when we remove hashtable batching, where a lot of the benefit of the local-reads disappears. This is because with local reads we have a bigger opportunity to batch in the hashtable. When we remove batching, we can no longer prefetch the keys for the local reads; hence we must wait for memory in each request. In Smallbank, the cumulative benefit of the two optimisations ranges from 38% with 3 machines to 10% with 10 machines. In Tatp, the same benefit ranges from 50% with 3 machines to 10% with 10 machines. Tatp benefits more, as it is read-intensive.

Recall that, when we employ local reads, we pay an overhead on writes which must set and reset the *readable* flag on backups. As the number of machines increases, the percentage of reads that can be executed locally decreases. In Smallbank which is write-intensive, the overhead surpasses the benefit with 6 machines.

Figures 20 and 21 offer a deeper dive in the impact of network and hashtable batching. All optimizations are open for these figures. Note that they y-axis shows a different metric for each of the bars. For *ht-batch* and *nw-batch* it shows the average number of requests that are being batched to the hashtable and network respectively. For *nw-bw* it shows the Gigabits per second (Gbps) that are sent by each node; goodput is the nw-bw (in Gbps) minus the packet headers. Finally, for *packet-rate*, the y-axis shows millions packets per second sent by a single node.

Note that, as the number of machines increases both types of batching decrease. As network batching decreases the packet rate increases, as we need to send more packets. For this reason the difference between nw-bw and goodput also increases, as more of our nw-bw is spent on packet headers.

Finally, we note that even though nw-batch steadily decreases with more machines, it still provides a 5x improvement with 10 machines. This is because, as we saw earlier in Figure 5, even a very low batching degree has a very significant impact on throughput.

8.3 Impact of network batching

In this section we further study network batching by varying the amount of ongoing transactions in each machine. Specifically, in Figure 22 we plot the Mtps for Dnl-4 when running Smallbank, while increasing the number of ongoing transactions per thread

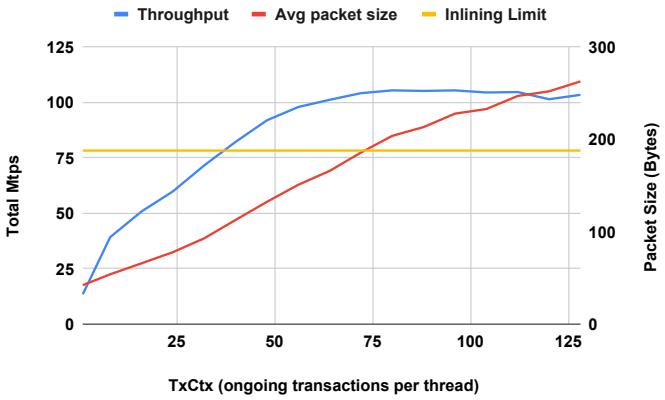

Figure 22: *Smallbank: Throughput vs average packet size.*

(i.e., the TxCtxs). We use 64 threads and 5 machines. We also plot the average payload size in the right y-axis and the inlining limit (188 Bytes). Note that "payload" here refers to a network packet's payload. Recall that when the payload of a packet exceeds the inlining limit, then it cannot be inlined in the descriptor (called "Work Request") that the NIC reads from the send queue. Instead, the descriptor will contain a pointer to the payload, which the NIC must read through a second DMA.

The first observation is that as we increase the number of TxCtxs, the average packet size is increased. This is because, with more TxCtxs, network batching increases. The second observation is that while initially throughput scales with the packet size, the scaling stops at around 75 TxCtxs, even though the average packet size continues to grow. This is because of the inlining limit: when a packet's payload exceeds 188 bytes the NIC needs to perform a second DMA, significantly hindering throughput.

8.4 Varying keys and contention study

In Figure 23, we run Smallbank in 5 machines with 64 threads and we vary the number of keys. Note that the x-axis is in log scale. The figure also points to the default number of keys used in previous experiments (52m). It also explicitly points to 4% of that (2m), because both DrTM and FaSST assume that 90% of accesses go to 4% of the keys to represent skew (discussed in § 8.1). We start with 3072 keys per machine (1024 primaries and 2048 replicas) up to 1 billion. With fewer keys, we simulate skewed workloads. For instance, with 3072 keys per machine, we see that the abort rate is very high because Smallbank is write-intensive. As the number of keys increases, the throughput oscillates around 100 Mtps. The oscillations are due to the different occupancy in the hashtable. For example, when moving from 800m keys to 1b, throughput improves, because a bigger index is needed for the 1b, which however has a lower occupancy.

The fact that the performance stabilizes around 100m Mtps also points to the fact that with 2MiB hugepages, the pressure on the virtual memory subsystem (TLBs, walkers, and MMU caches) remains manageable.

Summary. We evaluated the impact of our optimizations with up

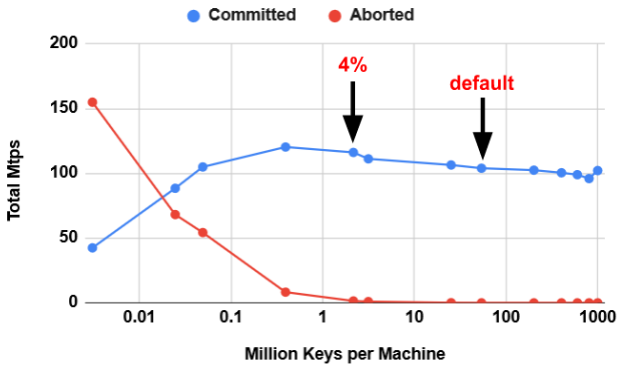

Figure 23: *Smallbank: Committed and Aborted Mtps of Dnl-4 varying the number of keys stored in each machine.*

to 10 machines and concluded that network batching continues to be very impactful with 10 machines, but there is a trend to hint that its impact will be reduced with a higher number of machines. Batching in the hashtable is most beneficial when used in conjunction with local reads from the backups, however its benefits decrease as we increase the size of the deployment. Local reads to backups scale with more machines in read-intensive workloads (e.g., Tatp) but may hurt performance in write-intensive workloads (e.g., Smallbank) on deployments of 6 or more machines.

9 RELATED WORK

FaRM [11, 12, 33] sparked the research interest in in-memory, RDMA-enabled, transactional KVSes. Since then, a number of works have improved on various aspects of its design.

Most of the discussion has been focused on RDMA [10, 34]. FaSST [18] exposed a number of performance issues caused by one-sided RDMA and argued for two sided. Storm [31] argues that the issues can be solved and in-memory transactional KVSes should use one-sided RDMA. The DrTM series [7, 35–37] has combined hardware transactional memory with RDMA, arguing for using both one-sided and two-sided. RCC [34] evaluates a number of protocols and also argues for a hybrid approach [34]. Other works [15, 17, 22, 23, 28] have focused on the low-level details of RDMA, creating guidelines on how to use it best. RIMA [38] exposed an important issue when dealing with variable size messages, and proposed a microarchitectural solution.

In addition, there are several in-memory transactional KVSes that target specific use-cases. Zeus [21] focuses on transactions that exhibit locality. Note that this is an orthogonal kind of locality to the one we have exploited. NAM-DB [5, 40] offers Snapshot Isolation over disaggregated memory which can be accessed through one-sided RDMA. FORD [42] offers Strict Serializability over the same architecture, while G-Tran [6] focuses on graphs.

Crucially, none of these systems have studied optimizations for small deployments, such as network batching.

There have also been other RDMA networking libraries. Flock [29] offers RPCs and batches messages from different threads in the same network packet. Crucially, Flock does not require any collaboration from the above layers, hence can be easily adopted by existing systems. While an essential feature for a library, this is significantly limiting for performance. We opted on the side of performance. ScaleRPC [8] focuses on scalability with respect to one-sided RDMA. The solutions provided could be used in an RPC library on top of one-sided (we use two-sided). However, with network batching, we send significantly fewer packets per second, substantially decreasing the pressure from NIC resources that can otherwise hinder scalability. Odyssey [14] contains an RDMA-based networking library that uses two-sided similarly to eRPC and also supports network batching similarly to DNL. However, it does not provide most of the features found in eRPC that are crucial for a complete system. Notably, high-performance implementations of single-key replication protocols have used network batching [14, 20], but have not highlighted its impact in their implementations. Finally, a number of systems have studied transactional protocols, without a focus on the networking [4, 16, 39, 41].

10 DISCUSSION

Why not use CXL directly instead of RDMA?. CXL does not provide fault tolerance or transactional guarantees – not even coherence for objects that span more than a single cacheline. Protocols like FORD implement fault-tolerant transactions in via one-sided RDMA that resembles the CXL setting. However, comparing Dandelion to FORD would be unfair, as FORD is significantly slower than Dandelion (e.g., in 3 servers running TATP and Smallbank, Dandelion achieves 200M and 75M txs/second, respectively, while FORD is an order of magnitude slower). This is because FORD's protocol is limited to the cumbersome one-sided RDMA semantics, which includes the inability to batch any requests. Note that the load batching is also not exploitable by CXL accesses.

OLAP workloads. Dandelion primarily targets fault-tolerant OLTP workloads (similar to prior work—FaRM, FaSST, DrTM). Yet, parts of Dandelion have been used by another internal product to implement high-performance OLAP operators, including joins and aggregations. However, in this case, fault tolerance through replication was stripped down as it was not a requirement of the OLAP solution.

Protocols and concurrency control. Prior work has emphasized the importance of protocols and concurrency control in the performance of distributed transactions. We found that within a local-area network deployment, although different protocols play a role in performance, other factors (e.g., batching here) can be equally or more important.

11 CONCLUSION

In this paper, we presented DNL, an in-memory, RDMA-enabled, distributed, replicated, transactional KVS. We focused on smaller deployments and explored system- and protocol-level optimizations to take advantage of the locality found on such a scale. Namely network batching, hashtable batching, and reading backups. We showed that the optimizations provide substantial performance improvements (3.3-6.5x over a state-of-the-art system), we characterized their individual and combined benefits and stressed their scalability limits. In addition, we provide a framework to implement new protocols, implementing and testing three protocols two of which we had not seen before.

REFERENCES

[1] Compute express link (cxl). https://www.computeexpresslink.org/.

[2] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling cxl memory expansion for in-memory database management systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*, DaMoN '22, New York, NY, USA, 2022. Association for Computing Machinery.

[3] Dotan Barak. Tips and tricks to optimize your RDMA code. https://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/, June 2013. (Accessed on 10/10/2023).

[4] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores. *Proc. VLDB Endow.*, 12(13):2325–2338, sep 2019.

[5] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[6] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jie Zhang. G-tran: A high performance distributed graph database with a decentralized architecture. *Proc. VLDB Endow.*, 15(11):2545–2558, 2022.

[7] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 631–644, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. RDMA Reads: To Use or Not to Use? *IEEE Data Eng. Bull.*, 40(1):3–14, 2017.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, 2015. ACM.

[13] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. Odyssey: The impact of modern hardware on strongly-consistent replication protocols. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 245–260, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, USA, 2016. ACM.

[16] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A. Bernstein. Cornus: Atomic commit for a cloud dbms with storage disaggregation. *Proc. VLDB Endow.*, 16(2):379–392, oct 2022.

[17] Anuj Kalia, Michael Kaminsky, and David Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 437–450, Berkeley, CA, USA, 2016. USENIX Association.

[18] Anuj Kalia, Michael Kaminsky, and David Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation*, OSDI'16, pages 185–201, USA, 2016. USENIX.

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

[20] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. ASPLOS '20, page 201–217, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: Locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 145–161, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 31–48, Boston, MA, April 2023. USENIX Association.

[23] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, Renton, WA, April 2022. USENIX Association.

[24] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 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*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Hyeontaek Lim, Dongsu Han, David Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th Networked Systems Design and Implementation*, NSDI'14, pages 429–444, USA, 2014. USENIX Association.

[26] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general(?)! *ACM Trans. Parallel Comput.*, 5(4), feb 2019.

[27] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 313–326, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling rdma rpcs with flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 212–227, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Vikram Narayanan, David Detweiler, Tianjiao Huang, and Anton Burtsev. Dramhit: A hash table architected for the speed of dram. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 817–834, New York, NY, USA, 2023. Association for Computing Machinery.

[31] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 97–108, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, oct 1979.

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

[34] Chao Wang and Xuehai Qian. Rdma-enabled concurrency control protocols for transactions in the cloud era. *IEEE Transactions on Cloud Computing*, 11(1):798–810, 2023.

[35] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, Carlsbad, CA, October 2018. USENIX Association.

[36] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 335–347, Santa Clara, CA, July 2017. USENIX Association.

[37] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 87–104, New York,

NY, USA, 2015. Association for Computing Machinery.

[38] Jiachen Xue, T. N. Vijaykumar, and Mithuna Thottethodi. Network interface architecture for remote indirect memory access (rima) in datacenters. *ACM Trans. Archit. Code Optim.*, 17(2), may 2020.

[39] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, nov 2014.

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

[41] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Trans. Comput. Syst.*, 35(4):12:1–12:37, December 2018.

[42] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, Santa Clara, CA, February 2022. USENIX Association.