SplitFT: Fault Tolerance for Disaggregated Datacenters via Remote Memory Logging

Xuhao Luo
University of Illinois Urbana-Champaign

Ramnatthan Alagappan
University of Illinois Urbana-Champaign and VMware Research

Aishwarya Ganesan
University of Illinois Urbana-Champaign and VMware Research

Abstract

We introduce SplitFt, a new fault-tolerance approach for storage-centric applications in disaggregated data centers. SplitFt uses a novel *split* architecture, where large writes are directly performed on the underlying disaggregated storage system, while small writes are made fault-tolerant within the compute layer. The split architecture enables applications to achieve strong durability guarantees without compromising performance. SplitFt makes small writes fault-tolerant using a new abstraction called *near-compute logs* or Ncl, which leverages underutilized memory on remote nodes to log small writes in a fast, cheap, and transparent manner. We port three POSIX applications (RocksDB, Redis, and SQLite) to SplitFt and show that they offer strong guarantees compared to weak versions of the applications that can lose data; SplitFt applications do so while approximating weak versions' performance (only 0.1%-10% overhead under YCSB). Compared to strong versions, SplitFt improves performance significantly (2.5× to 27× under write-heavy workloads).

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**; • **Information systems → Cloud based storage**.

Keywords: Fault-tolerance, Disaggregation, Remote Memory, Storage

ACM Reference Format:
Xuhao Luo, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. SplitFT: Fault Tolerance for Disaggregated Datacenters via Remote Memory Logging. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3627703.3629561

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629561

1 Introduction

Storage-centric applications such as key-value stores [52, 53], databases [17, 20], document stores [70], and data-structure stores [82] are increasingly deployed in modern cloud datacenters [3–5]. These applications must offer high availability, durability, and strong consistency for the data they store.

A traditional way of realizing this goal is to replicate the application using primary-backup (PB) [42], consensus [63, 76], or custom protocols [19, 69, 71, 72]. We refer to this approach as *application-level fault tolerance*. This approach offers high availability, durability, and strong consistency. However, it has two main drawbacks. First, it imposes a significant burden on developers as they must either make non-trivial changes to their application (e.g., to model it as a state machine when using consensus) or implement custom protocols, which is a difficult task. Second, it imposes significant resource overhead because the application uses $n\times$ CPU, network, and disk resources with n replicas.

Storage-centric applications today in the modern data center use an alternative approach. Here, the application runs on a stateless compute layer and persists its data on a storage layer disaggregated from the compute. The storage layer keeps the data highly available, durable, and consistent. The application instance can fail and restart (potentially on different physical hardware) yet seamlessly recover its state from the storage layer. We refer to this approach as *disaggregated fault-tolerance* or Dft (§2). Many storage-centric applications are built in this manner [5, 10, 12, 18, 44, 67]. Dft offers two main benefits for applications. First, it enables transparent fault tolerance: applications need not be modified or implement complex fault-tolerance protocols; the underlying storage layer provides the required fault-tolerance. Second, Dft is cheap: it does not require additional application-level resources, and the cost of running the storage service is amortized across all applications using it. This paper focuses on an important class of applications like key-value stores and databases that use the POSIX file interface for persistence. In the Dft paradigm, many such applications store their data on a disaggregated, distributed file system (dfs) [2, 9].

Unfortunately, applications in the Dft paradigm suffer from poor performance or weak guarantees. Applications that require strong durability guarantees must persist every update to the underlying dfs before acknowledging clients.

Such forced writes, even with batching, dramatically reduce throughput and increase latency. Despite efforts to improve performance with fast interconnects [8, 11] and local storage [7], distributed file systems suffer from poor performance due to software overheads [96]. Given the high overhead of synchronous dfs writes, many applications tradeoff guarantees for performance. In particular, applications do not make updates immediately durable on the dfs but do so only periodically. However, this results in poor semantics, as acknowledged updates can be lost upon application failures. Thus, applications atop Dft today are faced with a dilemma: choose strong guarantees and pay a high performance cost, or choose performance and settle for weak guarantees.

To address this problem, we introduce *split fault-tolerance* or SplitFt, a new approach that helps applications in the Dft paradigm realize strong guarantees without compromising on performance (§3). SplitFt achieves this goal via a novel *split* architecture. SplitFt realizes that most storage-centric applications perform two kinds of writes: small writes to logs for durability and crash recovery in the critical path, and large writes for compacting or checkpointing data in the background. While large writes can obtain high throughput from the underlying dfs, small writes severely limit throughput and increase latency. Thus, SplitFt splits and handles the writes differently: large, background writes are pushed directly to the underlying dfs, while small, synchronous writes are made fault-tolerant quickly within the compute layer.

To make small writes fault-tolerant, SplitFt introduces a new abstraction called *near-compute logs* (Ncl). Ncl makes an application's small log writes fault-tolerant by replicating them to the memory of a few compute nodes that we call *log peers*. Any node in the compute layer can lend its spare memory to a common pool and act as a log peer. Ncl manages the common pool and allocates memory regions in peers to applications. Ncl achieves fault-tolerance in a fast, cheap, and transparent manner. Ncl is fast: it leverages kernel bypass, and fast interconnects and protocols (e.g., RoCE, InfiniBand) ubiquitously available in modern data centers [55, 56] to replicate writes with low latencies. Ncl is cheap: memory is largely underutilized in modern data centers [1, 54, 87] and the peers only lend such spare memory. Further, Ncl performs writes on peers via 1-sided RDMA operations that require no or minimal CPU interruption at the peers, essentially treating the peers as passive memory units. Finally, Ncl is *transparent*: POSIX writes are transparently intercepted and replicated, requiring no to little changes to applications.

We have designed and implemented SplitFt and the Ncl abstraction (§4). A primary challenge in our design is to ensure correctness under a range of possible failures (e.g., application and peer crashes). To address this challenge, Ncl uses many techniques. For example, in-order majority-based replication ensures that completed writes can be recovered in the application-issued order despite application and peer failures. Second, Ncl enables safe application recovery by managing peer-allocation information in a controller and a peer-side allocation validation. Finally, an atomic catch-up mechanism helps safely replace failed peers with new ones.

We have ported three storage-centric applications to SplitFt: a key-value store (RocksDB [52]), a data-structure store (Redis [82]), and a relational database (SQLite [20]). Porting them required minimal changes: we modified 6 to 19 LOC. Our experiments (§5) show that applications in SplitFt offer the same high throughput as the weak versions of applications for a write-only workload (8% overhead in the worst case). Under YCSB workloads, SplitFt performs significantly better than the strong versions (by an order of magnitude under write-heavy workloads) and closely matches the performance of weak versions (10.8% worst-case overhead). We also show that applications in SplitFt can recover their data as quickly as applications in the Dft paradigm. Finally, we show that Ncl can gracefully handle peer failures with minimal impact on application performance.

This paper makes four contributions.

- We propose SplitFt, a new fault-tolerance approach for storage-centric applications in disaggregated data centers.
- We introduce Ncl, a new abstraction that uses remote memory within the compute cluster to log small writes in a fast, cheap, and fault-tolerant manner.
- We design and implement SplitFt and Ncl. Our implementation is available on GitHub at `dassl-uiuc/compute-side-log`.
- We experimentally show the benefits of SplitFt and Ncl by porting three applications.

2 Background and Motivation

We first provide a brief background on disaggregated fault-tolerance (Dft), focusing on POSIX applications that use disaggregated file systems as their storage backend. Then, we explain how these applications process updates and why applications atop Dft today can achieve strong guarantees *or* high performance but not both.

2.1 Disaggregated Fault Tolerance

Application-level fault tolerance offers strong guarantees, but it has critical drawbacks. First, it imposes significant burden on the developer. It either requires significant changes to the application or requires implementing custom protocols, which are difficult to get right and have led to many bugs in popular systems [13–16]. Second, it requires running multiple copies of the application, usually requiring $(f+1)\times$ or $(2f+1)\times$ resources to tolerate f failures, increasing costs. Finally, the system cannot support large datasets because the storage capacity is limited by the disks on each replica.

Dft offers an attractive alternative in disaggregated data centers. The application runs in a stateless compute layer and persists its state on disaggregated storage. The storage layer provides high availability, durability, and consistency guarantees for the data stored. Upon a failure, an application

Configuration	Throughput (KOps/s)	Avg. Latency (μs)
Weak	232	50
Strong	4.3 (54× ↓)	4625 (92× ↑)

Table 1. Cost of Strong Guarantees. *The table shows the overheads of achieving strong guarantees in a RocksDB application server with CephFS as the underlying dfs. The throughput shown is for a write-only workload with 12 clients. The arrows show the throughput drop and latency increase compared to the weak configuration.*

can be restarted on any physical machine in the compute layer. However, the application can still seamlessly recover its state from the disaggregated storage layer and continue to operate. For this reason, this approach is also sometimes called REDS (recovery from disaggregated storage) [67]. One difference from application-level fault tolerance is that there are no active backups in DFT; as a result, application fail-over in DFT might be slow. However, this problem can be solved with mechanisms such as speculative recovery [67]. DFT is transparent, cheap, and can support large datasets as the underlying dfs offers huge storage capacities.

Our focus is on storage applications that use the POSIX file interface for persistence. Many such applications store their data on a disaggregated, distributed file system such as CephFS [9] in the DFT setting. Access to the dfs is hidden underneath the VFS layer [85], enabling applications to transparently achieve fault tolerance. POSIX applications persist updates by issuing one or more write system calls followed by an fsync. With dfs, writes are locally buffered in client's (i.e., the application server's) memory, and then pushed to the underlying dfs upon an fsync. Internally, the dfs uses replication [92] or erasure coding [61] to keep the data safe. Thus, upon a crash, writes that were issued before the last successful fsync are guaranteed to be durable.

2.2 Strong Guarantees *Or* Performance

To achieve strong guarantees, storage-centric applications must flush updates to the underlying dfs before acknowledging external clients (i.e., they must issue a write followed by an fsync). To amortize the cost of flushes, applications batch several update requests into one or few write calls and then issue one fsync. However, such synchronous flushes, even with batching, lead to poor performance. Synchronous flushes are prohibitively expensive, which has led many applications to provide configuration options to only lazily flush updates to storage. However, such a configuration leads to an obvious problem: acknowledged updates can be lost after application failures, resulting in poor semantics.

To illustrate the performance gap between strong and weak application guarantees, we compare the performance of RocksDB in the DFT setting with CephFS as the underlying dfs. We deploy a CephFS cluster with three replicas; each replica has a SATA SSD. We mount the file system on an application server where we run RocksDB (please see §5 for more details about the setup). We run a write-only workload in two modes of RocksDB: one where updates are acknowledged only after they are flushed to a write-ahead log (strong) and another where updates are acknowledged before a flush (weak). Table 1 shows the result. As shown, the strong mode offers about 50× lower throughput and incurs roughly 90× higher latency compared to the weak mode (even when RocksDB batches update requests). While the above experiment uses CephFS (a disaggregated file system) as the backing store, we observed similar trends when the application server uses a local file system backed by CephRBD [6] (a remote block device). Thus, storage-centric applications atop DFT today must choose either strong guarantees or high performance. We next explain how these seemingly conflicting goals can be achieved together.

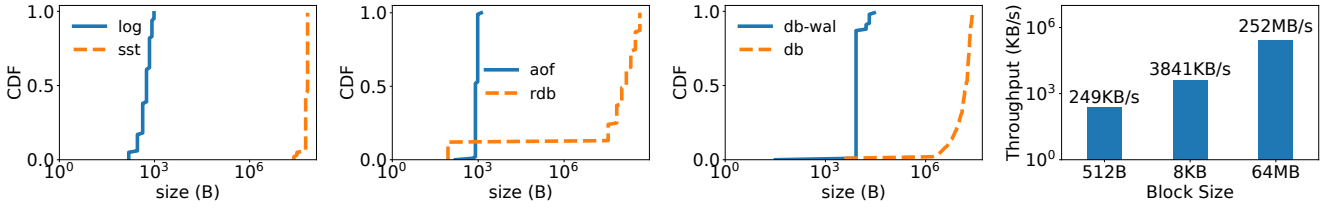
App	Small, sync writes	Large, bg writes	Reclaim policy
RocksDB	write-ahead log (`log`)	sorted-string tables (`sst`)	delete
LevelDB	write-ahead log (`log`)	sorted tables (`ldb`)	delete
Redis	append-only file (`aof`)	redis database (`rdb`)	delete
SQLite	write-ahead log (`db-wal`)	database (`db`)	overwrite*
Postgres	write-ahead log (`pg_wal`)	database (`base`)	overwrite*
HyperSQL	redo log (`log`)	database (`data`)	overwrite
MariaDB	redo log (`ib_logfile`)	tablespace file (`ibd`)	overwrite
MongoDB	journal (`WiredTigerLog`)	WiredTiger Store (`wt`)	delete

Table 2. Writes in Storage-Centric Applications. *The second and the third columns show the files that mostly receive small writes in the critical path and the ones that receive bulk writes in the background, respectively. The last column shows how logs are reclaimed. * – also supports delete.*

3 SplitFT Idea and Design Rationale

Our goal is to enable storage-centric applications to achieve both strong guarantees and high performance in the DFT paradigm. To this end, we first observe that most storage-centric applications perform two kinds of writes. First, they issue small, synchronous writes to a log for durability and crash recovery in the critical path. Then, they periodically compact or checkpoint data by performing bulk asynchronous writes, after which the logs are garbage collected.

We find that performing writes in this manner is pervasive in key-value stores, databases, and other systems. We analyzed eight widely used storage-centric applications including popular ones such as RocksDB, SQLite, and PostgreSQL. As shown in Table 2, all of these applications perform small, synchronous writes to a log and periodically perform large writes in the background. The large writes capture the information contained in the small log writes and thus applications garbage collect the logs after the large writes. Applications use two different policies to garbage collect logs. As shown in the last column, some applications delete a log file and create a new one for absorbing further updates, while some of them create one log file and keep reusing it (treating the log file as a circular buffer). As an example, consider RocksDB, where all updates are synchronously written to an on-disk log for durability and crash recovery (in addition to



(a) RocksDB **(b)** Redis **(c)** SQLite **(d)** Write Throughput

Figure 1. IO Sizes and Effect on Throughput. *(a)-(c) show the sizes of log writes, and compaction or checkpointing writes in RocksDB, Redis, and SQLite under a write-only workload. We configure the applications to offer strong guarantees, i.e., updates are flushed before they are acknowledged. The underlying dfs is CephFS. RocksDB and Redis perform batching: several requests are grouped in a large* `write` *system call and then an* `fsync` *is issued. The plotted write sizes indicate size of data submitted to CephFS upon an* `fsync`*. (d) shows the sequential write throughput with different IO sizes in CephFS.*

the in-memory memtable). Periodically, the memtable is compacted and written as large sstable files in the background, after which RocksDB garbage collects the log by deleting it.

To understand the difference in write IO sizes, we run a write-only workload with small key-value pairs (24B keys and 100B values) typical in many application workloads [43]. We measure the size of writes issued to the logs and the ones issued in the background to compact or checkpoint data. Figure 1 shows the result for three applications (RocksDB, Redis, and SQLite). As shown, the size of the writes to logs is significantly smaller than the background writes issued for compacting or checkpointing data. For example, in RocksDB, synchronous log writes are five orders of magnitude smaller than background writes to sstable files. We find a similar pattern in other applications shown in Table 2 as well.

Large writes issued to dfs can achieve high throughput; further, since such writes are issued during compaction or checkpointing in the background, they rarely affect request latencies. In contrast, small writes severely limit throughput and dramatically increase the latency as they are issued in the critical path of a client request. As shown in Figure 1(d), small 512-byte writes lead to roughly three orders of magnitude lower throughput than large 64-MB writes.

Based on these observations, SplitFt splits and handles the writes differently. Large, background writes take the usual path: they are directly performed on the underlying dfs as they can extract good performance from the dfs and they rarely affect request latencies. Small writes in the critical path, however, are made fault-tolerant quickly within the compute layer, reducing latency and improving throughput. To enable fast fault-tolerance for small writes, SplitFt uses a new abstraction that we call *near-compute logs* (Ncl).

Ncl makes an application's small log writes fault-tolerant by replicating them to the memory of a few compute nodes called *log peers*. Three trends in data centers make this design possible and effective.

- First, low-latency networking (enabled by kernel bypass and fast protocols and interconnects) is ubiquitous in the modern data center [55, 56]; thus, writes can be replicated with minimal latency.
- Second, memory is largely underutilized in modern data centers; for example, as much as 50% of server memory is unutilized in Google and Alibaba data centers [1, 54, 87]. A recent study from Microsoft [93] also shows that about 45% of memory is not even allocated to any VM or container in Azure Compute clusters; a fraction of this unallocated memory is also stranded where the local CPUs are saturated and cannot use the memory. The log peers only lend such spare memory to absorb small writes without requiring additional resources. Further, because SplitFt predominantly keeps log files in Ncl and logs are garbage collected after checkpointing writes, the memory required for fault-tolerance is small and fixed.
- Finally, the memory on the log peers can be efficiently accessed with 1-sided RDMA operations; such 1-sided writes require no CPU interruption at the peers.

As a result, Ncl achieves fault-tolerance in a fast and cheap manner. Ncl does so in a transparent fashion, requiring little modifications to applications.

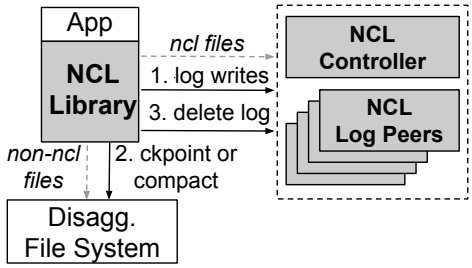


Figure 2. SplitFt High-Level Architecture. *The figure shows the high-level architecture of SplitFt. Ncl components are shown in grey.*

4 SplitFT Design and Implementation

We now describe the design and implementation of SplitFt. We first provide an overview (§4.1) and explain the Ncl abstraction and its operation in failure-free cases (§4.2-§4.4). Then, we explain how failures are handled (§4.5), and show that Ncl preserves correctness under failures (§4.6). We finally describe SplitFt and Ncl's implementation (§4.7).

```
initialize() //called upon open; get
peers and set up memory regions
record(offset, data) //called upon
write; records data on regions at offset
release() //called upon unlink; releases
and resets mem region on peers
recover() //called upon open during app
recovery; reads the mem regions
```

Figure 3. Ncl Operations. *The figure shows Ncl's main operations.*

4.1 SplitFt Overview

Figure 2 shows the architecture of SplitFt. At a high level, SplitFt intercepts file-system operations and transparently directs them either to the underlying dfs or Ncl. SplitFt does this classification at the file level and in a static manner. The file-level, static approach lends simplicity and works well in practice because in most applications, certain files (e.g., write-ahead logs in RocksDB) almost always receive only small writes, while certain files (e.g., sstables) mostly receive large writes as shown in Figure 1(a)-(c). Applications indicate that a file will receive small, synchronous writes via a new O_NCL flag in the open system call; we call such a file an *ncl* file. Determining whether a file should be classified as an ncl file is an one-time effort and is straightforward.

Figure 2 also shows a typical workflow in SplitFt. In most applications, logs are the ones that are classified as ncl files. First, small, synchronous writes to such logs are sent to the Ncl layer. The application then usually checkpoints and compacts the data; these non-ncl file operations are directly sent to the dfs. Once the data is safe on the dfs, the log is garbage collected (usually by deleting the log, removing the file from the Ncl layer). The operations to non-ncl files are essentially the same as in Dft and thus we do not describe them further. The rest of the section focuses on how Ncl manages and handles operations to ncl files.

While our discussion focuses on disaggregated file systems as the storage backend, we note that our approach and design also readily apply to applications that use a disaggregated block store (e.g., CephRBD [6]). With a block store, applications run atop a local file system. Since SplitFt intercepts operations at the system-call level, it can transparently direct small, synchronous writes to Ncl, while allowing bulk writes to go to the local file system backed by the block store.

4.2 Ncl Components, Operations, and Guarantees

Ncl has three important components as shown in Figure 2: *ncl-lib*, a *controller*, and the *log peers*. ncl-lib is a library that applications link to; all operations to ncl files are handled by ncl-lib. The controller is a fault-tolerant service that stores important metadata and manages several control-plane operations. In particular, it maintains the list of available log peers, and aids during peer and application server recovery. Finally, any node in the compute layer willing to lend its spare memory can register itself as a peer on the controller.

Figure 3 shows the main Ncl operations. First, when a new ncl file g is created, upon open, initialize is invoked. In this call, ncl-lib contacts the controller to get available peers and establishes connections to peers, requesting each peer to set up an RDMA memory region for g. The peers' CPUs are involved only during this initial setup phase. Upon a write (or variants such as pwrite) to g, record is invoked. In this call, ncl-lib uses 1-sided RDMA writes to record the data to the memory regions on enough peers. When g is deleted (e.g., when the application checkpoints or compacts the corresponding data), the release call is invoked. ncl-lib releases and resets the memory regions associated with g on the log peers. As we mentioned, in most applications, logs are the ones that will be classified as ncl files and so they are not read during normal operation. However, they will be read when the application server recovers from a crash; an open to a ncl file g during application recovery invokes the recover call. This call internally constructs the most up-to-date content of g by issuing 1-sided RDMA reads to enough peers. Note that applications do not invoke the Ncl operations directly. Instead, ncl-lib intercepts the POSIX file operations and invokes the appropriate Ncl operations.

Failure model and Guarantees. Ncl assumes a fail-recover failure model and is not designed to tolerate Byzantine failures [64]. In this model, both log peers and application servers can fail (either due to a crash or network partition) at any time and then recover later. When a log peer crashes, it loses the data in its memory regions and all of its in-memory state. When an application server crashes, it loses all its state and must recover it from Ncl and the underlying dfs. An application can recover on any physical machine after a failure and thus an application cannot expect to recover data from its local disk. Upon a network partition, a log peer does not lose its state but may be lagging due to missed writes.

Each application specifies a failure budget, f, to Ncl. To tolerate f failures, Ncl allocates $n = 2f + 1$ peers for each ncl file and replicates writes to these n peers. Ncl guarantees that if a write w_i is acknowledged to the application, then w_i and all preceding writes will be recovered as long as not more than f log peers fail simultaneously. In the unfortunate case where more than f log peers fail simultaneously, Ncl correctly makes the file unavailable. While such a failure scenario is rare, the possibility can be further minimized by placing the peers in different failure domains (e.g., across racks). The above guarantees are similar to that of standard fault-tolerance protocols [68, 75].

4.3 Peer Registration, Allocation, and Deallocation

Any node in the compute layer can lend its spare memory to act as a log peer. A peer is bound to a physical machine and it initially registers itself on the controller by specifying its unique name (derived from the physical machine's identifier) and the amount of memory that it is willing to lend.

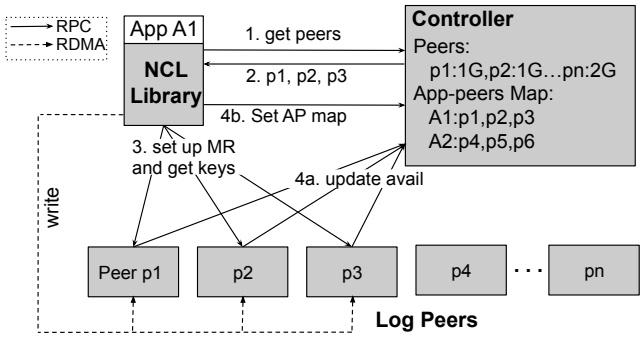


Figure 4. Ncl Log Peer Allocation. *The figure shows how log peers are allocated to applications and how memory regions are set up. Once set up, applications use 1-sided RDMA writes to write to the regions.*

The controller records this information and uses it when an application requests for new peers.

When an application creates an ncl file, ncl-lib contacts the controller, requesting n peers (see step-1 in Figure 4). The application specifies the amount of memory m that it needs. This is easy to obtain as applications usually specify the log size as a configuration parameter. The controller returns n peers whose available memory is at least m (step-2). ncl-lib then contacts the peers, requesting each of them to set up an RDMA memory region for the ncl file. The peers set up the memory region and return a key with RDMA write-read permissions to the region (step-3). ncl-lib uses these keys to subsequently perform RDMA writes on the peers.

The peers maintain a data structure called *mr-map*, which maps an application's unique identifier and ncl filename to a memory region. The *mr-map* is used to find the memory region when an application tries to recover a ncl file (after a failure). When a peer allocates a memory region for an ncl file, it decrements its available memory on the controller (step-4a). The log peers run a lightweight process to handle this initial setup. After this point, the log peers are treated as passive memory units that are written to without any CPU involvement. Finally, ncl-lib also adds an entry into a data structure called *app-peers map* (*ap-map*) on the controller (step-4b). The *ap-map* maps the application identifier and the ncl filename to the associated peers; this information is used when the application server recovers from a crash. It is possible that application server could crash before it updates the *ap-map*, leaking the reserved space on the peers. We discuss how Ncl handles space leaks in §4.5.1.

The available memory on a peer maintained at the controller might not be accurate because of the delay between when a memory region is allocated and when the peer updates the available memory on the controller. Thus, a peer returned by the controller is only a hint and the peer could reject an allocation request. In this case, the application obtains another peer from the controller and retries the allocation.

When an application deletes an ncl file, then the associated memory regions are freed. The peers remove the ncl file mapping from their *mr-map*, invalidate the keys, and recycle the memory region for future use. The peers also contact the controller to update their available memory. Finally, ncl-lib also deletes the entry for the ncl file from the *ap-map*.

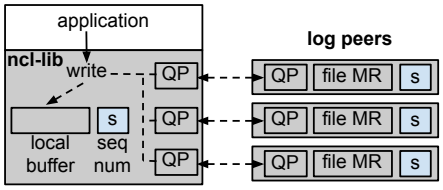


Figure 5. Ncl RDMA Replication. *The figure shows Ncl's RDMA-based replication architecture and the structures used by ncl-lib.*

4.4 Replication Protocol: Failure-free Operation

Once the memory regions are set up, the application can perform 1-sided RDMA writes. We now describe how writes are replicated in the failure-free case. The next subsection explains how failures are handled.

Applications perform writes and reclaim log files in two ways as we discussed in §3. First, some applications (e.g., RocksDB, Redis) only append to the log file; when the data is checkpointed or compacted, the log is deleted and a new file is created to handle further updates. Some applications (e.g., SQLite), however, create one log file and keep reusing it, treating the log as a circular buffer; in such cases, the file contents are overwritten. To support both kinds of applications, Ncl replicates the physical contents of the log file.

Figure 5 shows Ncl's RDMA-based replication architecture. Ncl maintains a local buffer for a ncl file. Each write is performed on the local buffer first and then replicated to the log peers. When the write is replicated on at least a majority (i.e., $f + 1$) of log peers, ncl-lib returns success to the application for the write. This guarantees that all completed writes will be recoverable even if f log peers fail. To replicate data, Ncl maintains an RDMA queue pair (QP) for each peer. To write to a peer p's memory region, Ncl creates a work request (WR) and puts it in p's queue. To find if a WR has completed, Ncl polls p's RDMA completion queue (CQ). When a majority of WRs for a particular write request completes, the write is returned as success to the application.

Ncl must ensure that write-s are completed in the correct, application-issued order on each peer. Ncl ensures this by leveraging RDMA's send queue (SQ) ordering guarantee: RDMA write WRs will be completed in the order in which they are added to the SQ [81]. Therefore, Ncl adds WRs for write-s to work queues in the order issued by the application. Upon a current write W_i, for each peer p, Ncl waits for completion of all write-s up to W_i (by polling the CQs). W_i is considered complete when at least a majority of peers have completed all write-s up to W_i; Ncl then returns success for W_i. This ensures that when Ncl returns success for a write W_i, all write-s up to W_i are recorded in the correct order in the memory regions of at least a majority of peers.

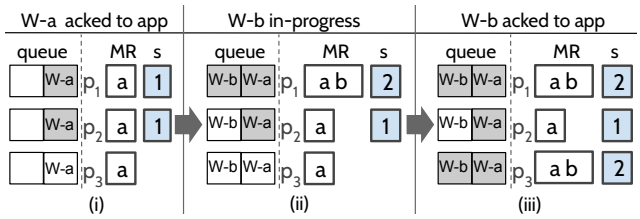


Figure 6. Memory Region States. *The figure shows the status of writes in the RDMA queue at the application server for two write-s W_a and W_b. It also shows the memory regions of the peers. MR: memory region; s: sequence number. A shaded entry for a write W in the queue of a peer p_i indicates that both data and sequence-number for W have completed on p_i.*

When an application crashes and restarts, it recovers its ncl files by contacting the corresponding logs peers. For correctness, Ncl must recover all writes that were returned as success to the application. Since Ncl does not wait for all peers to acknowledge (but only a majority), some peers may be lagging. Ncl thus must differentiate lagging peers from up-to-date ones and recover the most up-to-date data.

To identify which peer is up-to-date, Ncl assigns a *sequence number* to every write to an ncl file. The sequence number is written to a fixed location within the memory regions. Thus, Ncl converts every application-level write into two RDMA writes on each peer: one for the actual data and another for the sequence number. For correctness, the sequence number must be written only after the data write. This ordering is ensured by adding the WR for sequence-number to the queue after the WR for the data. The RDMA writes on a peer for an application-level write is considered complete only when the WRs of both data and sequence-number are complete. The next subsection discusses how the sequence numbers are used during recovery.

Figure 6 illustrates how two application-level write-s W_a and W_b complete in Ncl. First, in (i), W_a is returned success as the data and the sequence numbers are successfully written to a majority of peers (p_1 and p_2). Although the data is written to p_3, the write on p_3 is not considered complete because the sequence number is not written yet. In (ii), W_b is complete on p_1 and is being written to p_2 and p_3. (iii) shows that W_b completes on p_3 (and by induction the previous write W_a is also complete on p_3). Thus, all writes upto W_b are safe on a majority of peers and so W_b can be returned as success.

4.5 Failures and Recovery

So far, we have discussed the failure-free case. We now discuss failures and how Ncl handles them.

4.5.1 Application Failures and Recovery

The application can fail at any point and restart later, possibly on a different physical machine. Once restarted, the application must be able to recover all its data. Data written to the dfs is guaranteed to be safe and is directly recovered from there. The data written to ncl files must also be safely recovered. To do so, the application first contacts the controller to get the ncl files that it had before it crashed and also the peers associated with each of the file (from the *ap-map*).

The application then contacts the peers. A peer checks its *mr-map* to see if it holds the memory region corresponding to the ncl file of the application. If the entry is present, the peer returns a key that allows the application to perform RDMA operations on the memory region. If not, it rejects the request; this case is possible if the peer had crashed and recovered (and thus had lost its *mr-map*).

The application then reads the sequence number of the ncl file on the log peers and waits to get responses from at least a majority ($f + 1$) peers. The application then chooses the maximum sequence number among the $f + 1$ responses as the most up-to-date sequence number for the file. This is correct because writes are considered complete when the data and their sequence numbers have been safely recorded on at least $f + 1$ peers. Because the application contacts $f + 1$ peers during recovery, at least one peer within the contacted majority must contain the latest sequence number.

The application then contacts the peer that has the maximum sequence number to retrieve the data from its memory region; we call this peer the *recovery peer*. The recovery peer's memory region is retrieved as the data for the file and returned to the application. However, before returning to the application, Ncl must ensure that enough peers (specifically, a majority) are caught up with this latest data. If not, Ncl may return the most up-to-date data to the application now and subsequently may fail to do so if the application crashes again; this can happen if the recovery peer was the only peer that contained the latest data. This is unsafe because the application can externalize outputs based on this recovered data and if the same latest data is not recovered after the subsequent failure, then it will cause inconsistencies.

To understand the above problem, consider the example scenario in Figure 7(i). Here, p_1 can be selected as the recovery peer; p_1 has a greater sequence number because the application crashed just after writing to p_1 but before writing to p_2 and p_3. Now, assume the latest data (i.e., *ab*) is returned after recovery and the application externalizes outputs based on this returned data. If p_1 and the application crash subsequently, then only an older version of the data (*a*) will be recovered in the future from p_2 and p_3, leading to an application inconsistency. For this reason, before returning the recovered data to the application, Ncl must ensure that enough peers are caught up with that data.

For log files that are append-only, the catch-up procedure is fairly straightforward. The application can simply transfer the missing bytes at the end of the log to a lagging peer. Note that it is also possible for a peer p to be more up-to-date than the recovery peer; for example, in Figure 7(i), p_2 could have also been chosen as the recovery peer (considering responses from p_2 and p_3), in which case p_1 will be more up-to-date than the recovery peer. In this case, the application could

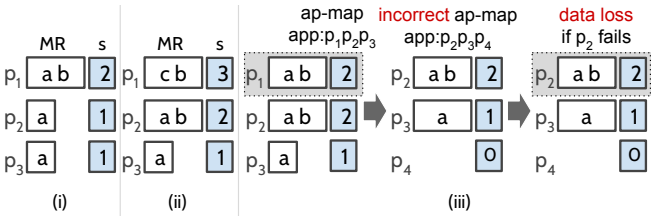


Figure 7. Example Application- and Peer-Failure Scenarios. *(i) shows a possible memory-region state when the application recovers after a crash when processing write* b. *In (ii), the application uses the log file as a circular buffer; in this case, it is not possible to catch up a lagging peer by only transferring the tail of the log. (iii) shows the case where a peer fails and a new peer is added by the application and how a data loss is possible if the* ap-map *is updated before catching up the new peer.*

instruct the more up-to-date peer to truncate the log tail and update its sequence number.

However, logs that are used as a circular buffer cause challenges, where a simple transfer as discussed above is not possible. For example, consider the case in Figure 7(ii), where the data written by the first write *a* is overwritten by the third write with *c*. Here, simply transferring the tail of the log from p_1 to the other peers would be incorrect. NCL thus copies the entire recovered memory region to all the peers. However, this step must be done atomically. Thus, NCL copies the data to new memory regions on the peers; the peers then atomically switch their *mr-map* entry to point to the newly copied region. For simplicity, NCL adopts this design for both append-only and circular logs uniformly. Note that if the sequence number of a peer p is equal to the recovery peer's sequence number, it does not mean that p's memory region will be identical to that of the recovery peer. This is because the next data write might have succeeded on one of them, but the next sequence number write may not have. Thus, NCL safely performs the data catch-up step even when the sequence numbers are equal. As an optimization, instead of sending the entire file content, a bytewise difference could be shipped during the catch-up.

Application failures at an inopportune moment can also lead to partial writes on the peers. However, this is not a problem as most POSIX applications understand and expect that writes can be non-atomic and use application-level mechanisms such as checksums to ensure atomicity [78]. NCL's responsibility is to correctly recover the data (even if it is only partially written) and pass it on to the application without interpreting the data in any way.

Note that NCL can recover incomplete, unacknowledged writes. While not recovering acknowledged writes is unsafe, recovering unacknowledged writes does not impact correctness. This behavior is no different from when applications perform writes on dfs. Specifically, most dfs persist data on a close or an explicit fsync, and therefore they also can potentially recover unacknowledged writes.

Handling Space Leaks. Application failures can sometimes lead to a space leak on the peers. For example, an application could allocate space on peers on initialization and crash right after that (with no information recorded in the *ap-map*). NCL handles such leaks as follows. The application maintains an epoch number which is incremented whenever the application intends to update its *ap-map* entry (i.e., during initialization or when replacing peers). The application provides this epoch number to the peers when requesting an allocation; the peers store the epoch number in their *mr-map*-s. The application then stamps the *ap-map* entry with the epoch number before writing it to the controller. Periodically, for each memory region with epoch e_p a peer maintains, it queries the controller to get the epoch number e of the application's current *ap-map* entry. If $e > e_p$, then it means that the application has moved to a new epoch and the peer can free its allocation; on the other hand, if $e < e_p$, then the peer's allocation might be still in progress and so the allocation cannot be freed. If $e = e_p$, the peer checks if it is part of the *ap-map* entry. If it is, the allocation cannot be freed; if not, the allocation is freed.

4.5.2 Log Peer Failures and Recovery

Log peers can fail at any time. In such cases, RDMA writes performed on a failed peer will return an error. After retrying (configurable) number of times without success, the application declares that the log peer has failed. However, as long as f or fewer peers fail, write-s can still proceed without any impact on availability.

Upon a peer failure, the application asks the controller for a new peer. The application then establishes connection to the peer and obtains the keys to write to the memory region. However, before updating the *ap-map* on the controller with the new peer information, the application must ensure that the new peer is caught up. If not, it might result in a data loss. Consider the case in Figure 7(iii). Here, two write-s *a* and *b* have completed because they are safely recorded on p_1 and p_2. Now p_1 fails and as a result, the application adds a new peer p_4. If the application incorrectly updates the *ap-map* immediately, then a data loss can result. This is because the application could fail after updating the *ap-map* but before catching p_4 up. Then, upon a subsequent recovery, the application may only receive recovery responses from p_3 and p_4, causing it to recover only write *a*, which is unsafe.

To catch up a new peer, ncl-lib copies the contents of the ncl file from its local buffer to the peer's memory region. Once the new peer is up-to-date, the *ap-map* is updated and then subsequent write-s are normally replicated to the new peer. If more than f log peers fail, then the application cannot process write-s without blocking. In such a situation, the application server waits until the new log peers are brought up-to-date before processing new writes.

Maintaining FT Level. NCL must maintain the desired

fault-tolerance level despite peer failures. When the application is active, it ensures the required fault-tolerance level by allocating and catching up new peers to replace failed ones as described above. Nonetheless, when the application has failed, peer failures cannot be immediately detected. However, this is not a problem because a failed application can be quickly re-launched; the application can then detect failed peers and replace them.

Memory Revocation. A log peer uses the same failure-recovery mechanism described above to revoke memory at its will (for example, when there is a memory pressure on the physical machine). When a log peer decides to revoke its memory, it simply resets the RDMA permissions of the memory region. The reclaimed memory can then be reused for other VMs or applications on the physical machine. Note that memory reclamation is local and instantaneous without any distributed coordination. After a region is reclaimed, subsequent writes to it will fail. The application server then treats that as a log-peer failure and contacts the controller to get a new log peer assigned as described above.

4.6 Correctness

Ncl must satisfy the following correctness condition: all write-s returned as success to the application must be recovered and in the order in which the write-s completed. Unacknowledged write-s may or may not be recovered; however, that doesn't affect correctness as we discussed earlier. We now provide a proof sketch for how Ncl ensures this condition despite application and peer failures. Let W be the latest write that completed and s be its sequence number. Ncl guarantees that the application will recover W and all prior completed write-s as long as more than f peers do not fail simultaneously.

We first consider the case where the *ap-map* has not changed after initialization, i.e., no peers have failed or the application has not yet replaced the failed peers in the *ap-map*. During normal operation, W and s must have been written successfully on at least $f + 1$ peers. During recovery, Ncl contacts $f + 1$ peers to retrieve the latest sequence number. First, if no peers had failed, because of the intersection between the write and recovery quorums, at least one peer will return a sequence number that is at least s. Second, if some peers had failed, then the application will not receive a response from those peers; if a peer had crashed and recovered, such a peer will correctly reject the recovery request. As long as the application receives $f + 1$ recovery responses, one among them is guaranteed to be at least s.

Next, we consider the case where the *ap-map* has changed since initialization. Let P be the set of peers that completed the latest write W and thus have a sequence number that is at least s. Thus, $|P| >= f + 1$. Suppose a set of peers, P_{old}, fails and the application replaces them with new peers, P_{new}, by modifying the *ap-map*; $|P_{old}| = |P_{new}|$. Two sub-cases are possible. First, no peer in P_{old} is part of P, i.e., no peer in P_{old} was part of the $f + 1$ quorum that completed W. Then, at least $f + 1$ nodes in the remaining nodes ($P - P_{old}$) will have W and s, ensuring that s will be retrieved during recovery (due to quorum intersection). Second, some peers in P_{old} are part of P. Before replacing the *ap-map*, Ncl brings all peers in P_{new} up-to-date. Therefore, P_{new} will have a sequence number that is at least s. Since W was committed in P, all nodes in $P - P_{old}$ will also have W. Since $|P| >= f + 1$ and $|P_{old}| = |P_{new}|$, $|P - P_{old} + P_{new}| >= f + 1$. Thus, at least $f + 1$ in the modified *ap-map* will a have sequence number that is at least s and thus will be safely retrieved during recovery.

We now show that Ncl recovers all completed write-s up to W. Let W_s be the sequence of write-s up to W. First, assume the application never failed during W_s. Because Ncl replicates write-s in order, if W is present on at least $f + 1$ peers then all prior completed write-s will also be present on at least $f + 1$ peers. Thus, the current recovery will recover W_s. Next, any future application recovery after this will also recover at least up to W_s. Since Ncl brings at least $f + 1$ peers up-to-date with W_s before proceeding to normal operation and any peer that commits a new update must have seen up to W_s, at least $f + 1$ that are committing new updates will have W_s followed by the new updates. This ensures that W_s survives in all future application recoveries.

Model Checking. We have modeled Ncl's replication and recovery protocols, and model-checked them. We explored over 4 million states; in each state, a few writes are performed to ncl files and then peer and application failures are injected at various points. Upon each state, the checker asserts that the above correctness condition is met. We also introduced subtle bugs in the model and the checker correctly flags these bugs. For example, when a peer incorrectly writes the sequence number before the data, a data loss can arise if the peer becomes the recovery peer; the checker catches this violation. Similarly, when the *ap-map* is incorrectly updated before catching up a new peer or if the lagging peers are not caught up during application recovery, a data loss is possible and the checker flags these cases.

4.7 Implementation

We have implemented Ncl in C++ in about 2877 LOC. ncl-lib intercepts all POSIX file operations and invokes the appropriate Ncl operations. It uses the Infinity ibverbs-based library [37, 38, 47] for RDMA over Converged Ethernet [74]. The O_NCL flag is understood only by ncl-lib and requires no kernel modifications. We implement the controller using a fault-tolerant ZooKeeper [34] instance. The controller maintains the information about log peers under a directory called `Peers`; each peer creates a znode inside this directory with its unique name as the key and the available memory as the value. The controller walks these znodes to assign peers

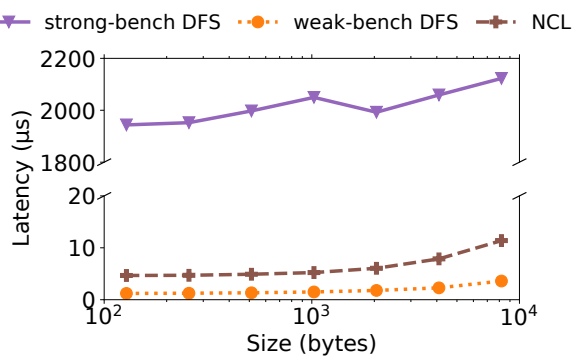


Figure 8. Write Latency – Embedded Mode. *The figure shows the latency to perform writes of different sizes in Ncl, weak-bench dfs, and strong-bench dfs. Strong-bench issues a fdatasync after every write. Note that the y-axis has a break to accommodate the huge disparity in latencies.*

to an application. The controller maintains the *ap-map* structure in a directory called `Apps`; each application maintains a znode in this directory with its identifier as the key and its assigned peers and epoch number as the value.

Ncl requires that, for each Ncl-backed file, only one application instance has access to it (although there can be multiple writer threads within the application). Ncl guarantees this by ensuring that only one instance of the application runs at a time as follows. An application creates an additional ephemeral znode (inside a directory called `Servers`) that is removed by ZooKeeper upon application failures. When an application instance recovers, it recreates the znode. If many instances try to recreate the znode, ZooKeeper guarantees that only the first creation will succeed.

Application Integration. We have ported three applications – RocksDB, Redis, and SQLite – to use Ncl. To port these applications, we first identify in the source code the function calls that open or create log files, and then pass an additional O_NCL flag to the underlying POSIX open call. This was a fairly straightforward exercise and expect it to be even easier for application developers because they understand the source code well already. Porting required very minimal code change in all three applications: 10, 19, and 6 LOC in RocksDB, Redis, and SQLite, respectively.

5 Evaluation

To evaluate Ncl and SplitFt, we ask the following questions:

- How does Ncl perform on a write microbenchmark? (§5.1)
- How do applications perform in the SplitFt paradigm compared to Dft for a write-only workload? (§5.2)
- How do applications perform in SplitFt under different YCSB workloads [49]? (§5.3)
- How does Ncl perform on reads (upon recovery)? (§5.4.1)
- How quickly do applications in SplitFt recover? (§5.4.2)
- How quickly does Ncl replace failed peers and what is impact of peer failures on application performance? (§5.4.3)

Setup. We conduct our experiments on a cluster in CloudLab. Each machine in the cluster has a 2.4GHz 10-Core E5-2640v4 CPU, 64GB DDR4 memory, a 25Gb Mellanox ConnectX-4 NIC, and a 480GB SATA SSD. We use CephFS as the underlying dfs and host it on three machines. The application server runs on a single machine. We use the remaining machines as Ncl log peers and application clients. CephFS is mounted on the application server via a CephFS kernel driver.

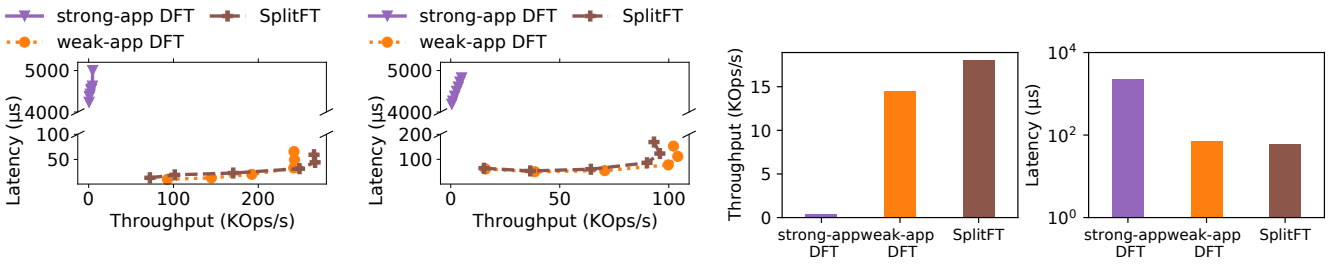
Application Configuration and Workloads. We have ported three applications – RocksDB, Redis, and SQLite – to use SplitFt. Each application is configured with a failure budget of $f = 1$, so Ncl assigns three $(2f + 1)$ log peers to each application. We run the applications as servers. Redis is already built as a server and we use it as-is; we built a eRPC-based [60] server for RocksDB and SQLite. Based on typical cache sizes for key-value stores and databases [48, 79], we set the cache size of RocksDB and SQLite application servers to 30% of the dataset size; we use `cgroup` to limit the memory. We do not limit Redis' memory since it is designed to keep the entire dataset in memory. RocksDB and Redis employ application-level batching to batch concurrent update requests into a single log write, while SQLite does not use such a batching mechanism. We set SQLite's locking mode to `exclusive` since the database is accessed from a single application-server process, reducing its locking overhead.

For both write-only and YCSB workloads, we use small key-value pairs (24B keys and 100B values) typical in many workloads [43]. We use 20 application-server threads to run YCSB workloads for RocksDB and Redis, and show single-threaded performance for SQLite since its performance does not scale with increasing threads. We plot latency vs. throughput for the write-only workload and the throughput for YCSB workloads. Each data point runs for at least 120s and is repeated three times; we plot the average of the three runs, with error bars showing the standard deviation.

Baselines. We compare SplitFt against two baselines: *weak-app Dft* and *strong-app Dft*. In weak-app Dft, the applications run in a weak configuration, where log writes are buffered in the application server's OS buffer cache and only asynchronously flushed to CephFS. Thus, when the application server crashes, acknowledged data in the cache is lost. In strong-app Dft, the application flushes every log write synchronously to CephFS. We configured CephFS to return an fsync/fdatasync when the data is replicated to the buffer cache of the CephFS servers. Ncl synchronously replicates every write system call to ncl files on to the peers.

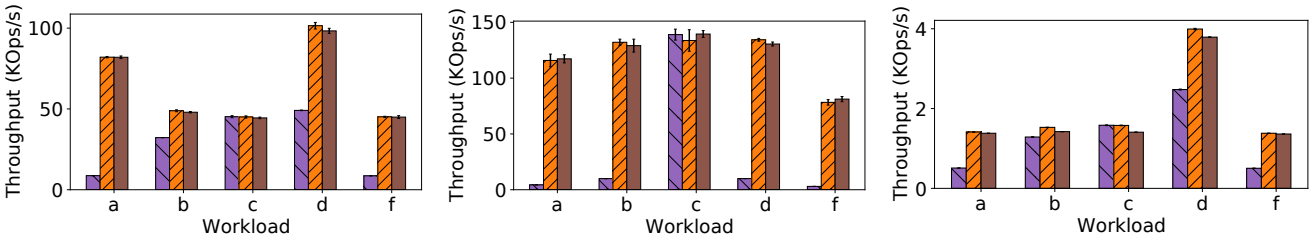
5.1 Microbenchmark: Ncl Write Latency

First, we measure Ncl's write latency using a microbenchmark that sequentially writes a 100MB file. The benchmark issues write system calls of different sizes ranging from 128



(a) RocksDB **(b)** Redis **(c)** SQLite

Figure 9. Latency vs. Throughput. *(a) and (b) plot the latency versus throughput of strong-app DFT, weak-app DFT, and NCL for a write-only workload in RocksDB and Redis, respectively. (c) shows the throughput and latency for SQLite.*



(a) RocksDB **(b)** Redis **(c)** SQLite

Figure 10. YCSB. *(a), (b), and (c) show the throughput under the YCSB workloads in RocksDB, Redis, and SQLite, respectively.*

bytes to 8KB in single thread. Unlike most subsequent experiments, this experiment runs the benchmark in an embedded mode, i.e., the benchmark process directly links against ncl-lib and hence does not incur network latency to submit operations. We run two versions of the benchmark: strong and weak. In strong, the benchmark issues an fdatasync after every write, flushing the data to CephFS. In weak, the writes are not flushed and so can be lost upon failures. In NCL, every write is synchronously replicated to the log peers, offering strong guarantees. We measure and plot the average latency of writes for each size. Figure 8 shows the result.

As shown, the strong version is significantly (two orders of magnitude) slower than the weak and NCL versions as it flushes every write. NCL's latency is close to that the weak configuration. For instance, with 128-byte writes NCL's latency is 4.6 μs, while weak's latency is 1.2 μs. However, while the weak version risks data loss in case of a crash, NCL provides stronger guarantees at almost the same level of latency.

5.2 Application Write-Only Benchmark

We next evaluate application performance in the SPLITFT paradigm. We first compare the performance of SPLITFT against weak-app DFT and strong-app DFT for a write-only workload in RocksDB, Redis, and SQLite.

Figure 9(a) and 9(b) show the results for RocksDB and Redis. For both applications, we vary the number of clients and plot the throughput and latency. As shown, in both applications, strong-app DFT offers roughly two orders of magnitude lower throughput and higher latency. With SPLITFT, RocksDB achieves a peak throughput of 266 KOps/s. We find that SPLITFT's throughput is slightly better than weak-app DFT. Deeper measurements revealed that with SPLITFT, RocksDB performs fewer IOs to CephFS, which reduces the writes stalls, improving throughput. Redis in SPLITFT achieves a peak throughput of 100 KOps/s, closely approximating the throughput of weak-app DFT (only 8% overhead), while offering stronger guarantees. SPLITFT offers the same level of latency (a few 10s of μs) as weak-app DFT but with stronger guarantees. Figure 9(c) shows the result for SQLite. Similar to RocksDB, SPLITFT slightly improves over weak-app DFT as it does fewer IOs on CephFS, and is much faster than strong-app DFT.

5.3 Application YCSB Benchmark

We next evaluate application performance under YCSB workloads. We first load the application with key-value pairs and then run different workloads. For RocksDB and Redis, we load 100M key-value pairs. For SQLite, we load 10M key-value pairs; we convert each YCSB operation into a SQLite transaction. Figure 10 shows the result.

As shown, for all three applications, SPLITFT's throughput on all workloads is almost the same as that of weak-app DFT. For RocksDB, SPLITFT's performance overhead compared to weak-app DFT is minimal: 0.1% to 3.2%, a small price to pay for the stronger guarantees. Similarly, in Redis and SQLite, the overheads are small (2.9% and 10.8% in the worst case).

In all three applications, strong-app Dft offers significantly lower throughput than SplitFt. In RocksDB, strong-app Dft has low throughput on write-heavy workloads (YCSB-A and YCSB-F) due to the high overhead of writes. Under read-heavy workload (YCSB-B and YCSB-D), as expected, the performance gap between strong-app Dft and SplitFt reduces, and the gap closes for the read-only workload (YCSB-C). We observe similar trends in SQLite as well. With Redis, strong-app Dft has low throughput on all workloads except for the read-only workload. This is due Redis' single-threaded architecture, which results in head-of-line blocking. Read operations, even though they do not require writes to CephFS, are blocked by write operations before them. This reduces the performance even for read-heavy workloads.

5.4 Recovery and Performance Under Failures

Thus far, we have analyzed failure-free performance. Next, we examine recovery and performance under failures.

5.4.1 Microbenchmark: Ncl Read Latency

In most storage-centric applications, logs are the ones that are classified as ncl files. As a result, these files are not read during normal operation but only during recovery. When an application recovers, it must recover the log from the Ncl peers. We thus first measure the read latency of Ncl for sequentially reading a file and compare it with reading the file from underlying CephFS. We use a single thread to sequentially read a 100MB file by issuing read-s of different sizes ranging from 128 bytes to 8KB and plot the average latency of a read operation. Figure 11(a) shows the result.

Ncl prefetches the file from the peers and serves subsequent read-s from the local buffer (Ncl). The latency for Ncl in the figure includes the prefetch time; the prefetch time is amortized among all reads. Prefetching helps Ncl reduce the latency and thus is smaller than that of CephFS. For example, Ncl is 4× and 28% faster than CephFS with 128-byte and 8KB reads, respectively. We also show the performance for an Ncl variant that does not prefetch the file. This variant performs worse than CephFS (4.51× higher latency for 128-byte reads), showing that Ncl's prefetching is essential. Note that CephFS client also prefetches data from the underlying file system. Without prefetching, CephFS reads incur significantly higher latency as shown by the direct IO line.

5.4.2 Application Failure Recovery

We next analyze how fast applications in SplitFt can recover after a failure. During recovery, storage-centric applications recover the log to reconstruct the state. In Dft, the application recovers the log from CephFS, while in SplitFt, the logs are recovered from Ncl. We measure the time to recover a 60MB log in RocksDB, Redis, and SQLite. We do not distinguish between weak- and strong-app configurations in this experiment, as both recover the log from CephFS. We also compare to recovering from an ext4 partition on local disk; this baseline is used only as a comparison point and is not realistic as the application is not guaranteed to have access to the local disk upon a restart. Figure 11(b) shows the result.

Step	Time (us)
Get new peer from controller	3586
Connect to new peer and set up MR	64871
Catch up new peer	23368
Update *ap-map* on controller	4734
Total	**96559**

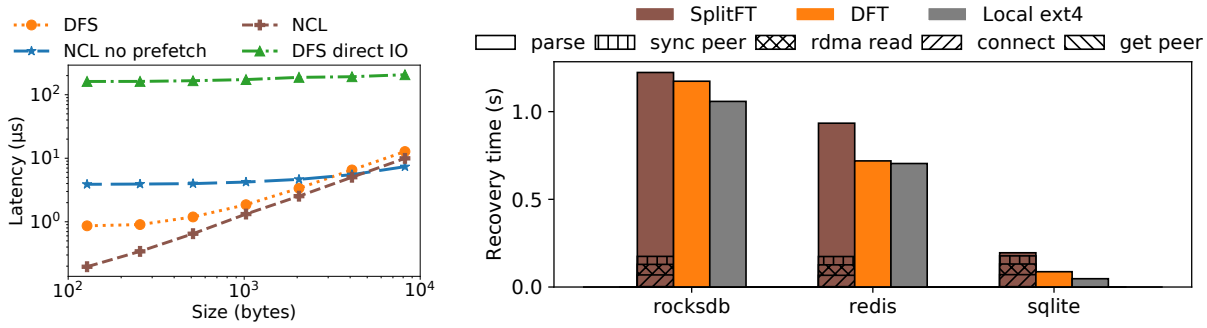
Table 3. Peer Recovery. *Latency breakdown of recovering a new peer.*

First, the time to recover from CephFS is not very high compared to local ext4 because CephFS prefetches data effectively. Second, in all three applications, the time to recover from Ncl is comparable to that of CephFS; both are in the order of several hundred ms. Most of this time is spent in application-level reading and parsing the log. Ncl's recovery is slower than CephFS (by 4%, 15%, and about 2× in RocksDB, Redis, and SQLite, respectively). The slowdown in Ncl is expected as Ncl must perform several steps in addition to reading and parsing the log at the application-level. We breakdown the recovery latency in Ncl into five parts: getting peer information from the controller (denoted as *get peer* in the figure), connecting to peers (*connect*), prefetching the log (*rdma read*), synchronizing the peers with latest data (*sync peer*), and finally reading and parsing the log at the application level (*parse*). The increased recovery time in Ncl comes from the first four parts; application-level parsing time is similar to that of CephFS. However, the additional latency in Ncl is small and often is a tiny portion of the overall time. Given that failures are rare and the log must be recovered once during startup, Ncl's increased recovery time is a small cost to pay for its improved guarantees.

5.4.3 Peer Recovery

We now analyze how Ncl handles peer failures and the time it takes to replace a failed peer with a new one. Table 3 shows the latency breakdown of replacing a failed log peer that had 60MB of log before it crashed. The peer-recovery time has four constituents: getting a new peer from the controller, connecting to the new peer, catching up the new peer, and updating the *ap-map* on the controller. Among these parts, connecting to the new peer takes the most time, as the new peer allocates a new memory region and registers it with the RDMA stack, which can take about 50ms. However, this is the worst-case scenario: in most cases, we expect a peer to have a memory region that is already allocated and registered, and thus overall latency would be much lower.

We next examine to what extent applications are affected by peer failures and recovery. To do so, we run RocksDB in SplitFt with three Ncl peers (i.e., $f = 1$). In the unlikely scenario where two peers fail simultaneously, writes can block until Ncl finds a new peer. However, a single failed peer must have little impact on availability or throughput.



(a) Microbenchmark: Read Latency

(b) Application Recovery Time

Figure 11. Recovery Performance. *(a) shows the average read latency in Ncl and CephFS. (b) shows the recovery time for RocksDB, Redis, and SQLite.*

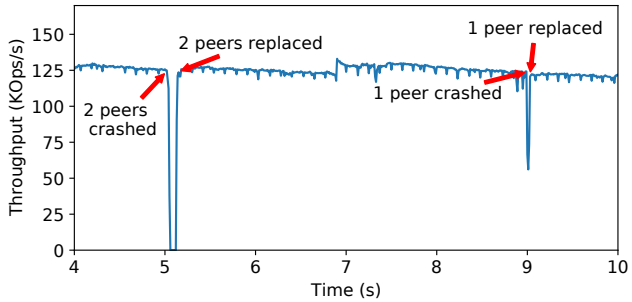


Figure 12. Application Performance Under Peer Failures.

We run a write-only workload and sample the real-time throughput every 10ms. Figure 12 shows the result. Initially all three peers are up and so RocksDB runs at full throughput. After a while, two peers are simultaneously crashed, causing the writes to stall. However, Ncl quickly finds a new peer and the throughput returns to the normal level. We find that the stall time is about 100ms, as the new peer allocates and registers a new region, matching the data in Table 3. The other failed peer is also replaced soon. When one peer is crashed after that, there is no impact on availability as a quorum is still available. The failure and subsequent catch up causes a small performance blip for a very short duration because of the bulk RDMA transfer to the newly added peer.

6 Discussion

We now discuss a few alternative design choices that can be used to build the Ncl abstraction and argue why our choices might be more desirable. We also discuss how Ncl can be extended to other use cases and scenarios.

Choice of Replication Protocol. In principle, one could use standard replication protocols like Paxos [63] or its variants [68, 76] to replicate small writes. We chose to implement our own protocol because standard replication protocols are not optimal for our use case. First, standard protocols run the entire application on all replicas. In contrast, our protocol treats the peers as passive memory units that do not run the application. Further, we perform replication using 1-sided RDMA-writes, without any CPU involvement.

Persistent Memory (PM) Logging. Persistent memory (PM) devices attached to the application servers can offer low-latency and high-throughput logging for applications. However, Ncl's design to replicate writes to remote peers is crucial even with PMs. This is because even with PM, a single hardware failure can render the logged data (on local PM) unavailable. Further, in modern datacenters, an application might be scheduled on a different physical hardware after a failure and thus may be unable to access the PM.

Supporting Non-Log Files and Applications. In our current port of applications, only log files utilize Ncl. If non-log files have a frequent, small-write access pattern, they can also benefit from Ncl. As discussed in §4.7, developers have more knowledge of such files and can decide whether or not a file is a good Ncl candidate. This is a one-time effort and requires just adding a flag to open.

Our focus in this work was on widely used storage-centric applications. Most of them employ logging for durability and crash recovery (§3). However, a few stores like KVell [66] do not log. Ncl can still be useful for such applications. Applications that do not log tend to issue many random writes. While such random writes might perform well with NVMe-SSDs, it can hurt performance in the Dft setting. Ncl can act as a faster tier to absorb the random writes and then write large chunks to dfs, improving performance.

Support for Fine-Granular Write Splitting. In our current implementation, we distinguish files that receive small writes from files that receive mostly large bulk writes. However, some applications may perform both small and large writes to the same file. To support such applications, Ncl must split writes in a more fine-granular manner (than at a file level). To do so, one solution would be to set a size threshold such that writes smaller than the threshold go to Ncl, while writes larger than the threshold go to dfs. In order to recover the file, SplitFt would track where (Ncl or dfs) the latest data resides for different byte ranges. This metadata can be conveniently stored in the Ncl layer.

7 Related Work

Fault-Tolerance for Storage Applications. In addition to traditional fault-tolerance approaches such as consensus [63, 68, 76] and primary-backup [42], several optimized protocols have also been proposed [40, 62, 88–90]. However, most of these protocols still suffer from high resource costs and require modifications to applications. A few approaches reduce cost [88] but still require application changes. Approaches such as whole VM replication [41, 50, 86] offer full application transparency but often have huge overhead. Further, these approaches cannot utilize the large storage capacity of a shared storage service unlike the DFT paradigm.

Shared-Log-based Fault-Tolerance. Shared logs [35, 36, 51, 59] offer an alternative way to realize fault-tolerance. However, most applications must be redesigned to use shared logs; thus, legacy POSIX applications cannot readily benefit. Further, while shared-log designs help applications make updates fault-tolerant efficiently, they do not help with storing large blobs of data (e.g., sstables in RocksDB), a key requirement in storage-centric applications. SPLITFT uses NCL to offer functionality similar to shared logs but also leverages the underlying dfs to store large blobs of data. Kafka [32] and other message queues [33] also can be used to log updates in a fault-tolerant manner. However, they incur additional CPUs and disks unlike SPLITFT that uses only spare memory.

Re-architecting Applications for the Cloud. Recent work has proposed to re-architect storage applications for cloud-native settings; these approaches often decompose monolithic applications for better elasticity and resource management [18, 31, 45, 95]. Similarly, prior work also extensively modify storage applications for disaggregated settings or to leverage RDMA [39, 57, 94]. While such complete redesigns might be possible in some cases, it is often difficult to change existing POSIX applications. SPLITFT offers a way for these applications to realize stronger guarantees and higher performance in the data center in a transparent fashion.

RDMA-based Replication. Recent systems leverage RDMA for fast replication [27–29, 65]. However, prior systems still use the application-level fault-tolerance paradigm and leverage RDMA to make it faster, unlike DFT and SPLITFT. NCL's replication protocol bears similarities to Mu's replication protocol [28]. A key difference is that prior RDMA-replicated systems run the full application on all replicas, incurring significantly high resource costs. In contrast, NCL treats the log peers as passive memory units.

Remote and Disaggregated Memory. While NCL uses the underutilized or unallocated memory on compute nodes to build a fault-tolerant-log abstraction, prior systems have proposed to use the spare memory for other purposes. For example, Infiniswap [54] and others [30] use it as swap space, enabling higher performance for applications with large working sets. Similarly, other systems use the spare memory as a cache [93, 97] to improve performance. Finally, prior systems have proposed better abstractions than RDMA ibverbs to export memory on one server to others [26].

Message Logging. SPLITFT has some similarities to message-logging systems [80, 91] that log messages or state on other nodes while only asynchronously persisting to stable storage. However, SPLITFT's context (disaggregated datacenters) and use case (storage-centric applications) are different from this body of work. Further, SPLITFT's design to transparently and CPU-efficiently (via RDMA) replicate application writes are different from prior approaches.

Durability vs. Performance in Local Storage. The trade-off between durability guarantees and performance exists in local storage systems as well. A few file systems offer high performance with prefix semantics, where recent writes may be lost [46, 77]. Weak configurations of applications in DFT offer a similar guarantee, which SPLITFT improves without performance overhead. Other systems that defer durability until externalization [73] cannot be readily applied to the DFT setting as clients must be acknowledged in a timely manner for low latencies while offering strong durability.

Optimizing Small Writes. Local file systems have optimized small writes by using a write-optimized index as a file system [58] (which converts small writes to large sequential transfers) or by storing metadata (which usually receives small writes) in a key-value store that converts small writes to sequential writes [83]. Our approach of treating small writes differently has a similar flavor.

8 Conclusion

This paper introduces SPLITFT, a new fault-tolerance approach for storage-centric applications in disaggregated data centers. SPLITFT performs bulk, background writes directly on the underlying disaggregated storage, while making making small writes fault-tolerant using a new *near-compute log* abstraction. NCL replicates writes to remote memory within the compute cluster efficiently via 1-sided RDMA writes. SPLITFT offers a way for storage-centric applications in disaggregated data centers to realize strong guarantees without compromising performance.

Acknowledgments

We thank Lorenzo Alvisi (our shepherd) and the anonymous EuroSys '24 reviewers for their insightful comments and suggestions. We thank the members of DASSL for their discussions. We also thank CloudLab [84] for providing a great environment to run our experiments.

References

[1] Alibaba Cluster Trace 2018. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md.

[2] Amazon Elastic File System. https://aws.amazon.com/efs/.

[3] Amazon RDS for MariaDB. https://aws.amazon.com/rds/mariadb/.

[4] Amazon RDS for PostgreSQL. https://aws.amazon.com/rds/postgresql/.

[5] Amazon RDS for PostgreSQL. https://www.nutanix.dev/2021/10/04/chakrdb-a-distributed-rocksdb-born-in-the-cloud-part-2/.

[6] Ceph Block Device. https://docs.ceph.com/en/quincy/rbd/index.html.

[7] Ceph Bluestore. https://docs.ceph.com/en/latest/rados/configuration/storage-devices/#bluestore.

[8] Ceph over Accelio. https://tracker.ceph.com/projects/ceph/wiki/Accelio_RDMA_Messenger.

[9] CephFS. https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/2/html/ceph_file_system_guide_technology_preview/what_is_the_ceph_file_system_cephfs.

[10] Components in Azure Database for PostgreSQL. https://learn.microsoft.com/en-us/azure/postgresql/single-server/concepts-high-availability#components-in-azure-database-for-postgresql--single-server.

[11] GlusterFS on RDMA. https://glusterdocs.readthedocs.io/en/latest/Administrator%20Guide/RDMA%20Transport/.

[12] HBase Region Servers. https://nag-9-s.gitbook.io/hbase/hbase-architecture/region-servers.

[13] MySQL Bug ID 103084. https://bugs.mysql.com/bug.php?id=103084.

[14] MySQL Bug ID 103334. https://bugs.mysql.com/bug.php?id=103334.

[15] MySQL Bug ID 106918. https://bugs.mysql.com/bug.php?id=106918.

[16] MySQL Bug ID 92949. https://bugs.mysql.com/bug.php?id=92949.

[17] PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.

[18] RocksDB-Cloud: A Key-Value Store for Cloud Applications. https://github.com/rockset/rocksdb-cloud.

[19] RocksDB Replication. https://github.com/pinterest/rocksplicator.

[20] What Is SQLite? https://www.sqlite.org/index.html.

[21] Seattle, WA, November 2006.

[22] Farmington, Pennsylvania, October 2013.

[23] Huntsville, ON, Canada, October 2019.

[24] Crete, Greece, April 2020.

[25] Virtual, June 2021.

[26] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote Regions: A Simple Abstraction for Remote Memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, July 2018.

[27] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The Impact of RDMA on Agreement. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC '19)*, Toronto, Canada, August 2019.

[28] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond Consensus for Microsecond Applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*, Banff, Canada, November 2020.

[29] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. UBFT: Microsecond-Scale BFT Using Disaggregated Memory. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*, Vancouver, BC, Canada, 2023.

[30] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proceedings of the EuroSys Conference (EuroSys '20)* [24].

[31] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD '19)*, Amsterdam, Netherlands, June 2019.

[32] Apache. Kakfa. http://kafka.apache.org/.

[33] Apache. Pulsar. https://pulsar.apache.org/.

[34] Apache. ZooKeeper. https://zookeeper.apache.org/.

[35] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[36] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)* [22].

[37] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, pages 1463–1475, 2015.

[38] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.

[39] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-Based Databases. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, Lausanne, Switzerland, March 2020.

[40] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[41] Thomas C Bressoud and Fred B Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1), 1996.

[42] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The Primary-backup Approach. *Distributed systems*, 2, 1993.

[43] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, February 2020.

[44] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. pages 15–15, 2006.

[45] Zongzhi Chen, Xinjun Yang, Feifei Li, Xuntao Cheng, Qingda Hu, Zheyu Miao, Rongbiao Xie, Xiaofei Wu, Kang Wang, Zhao Song, Haiqing Sun, Zechao Zhuang, Yuming Yang, Jie Xu, Liang Yin, Wenchao Zhou, and Sheng Wang. CloudJump: Optimizing Cloud Databases for Cloud Storages. *Proc. VLDB Endow.*, 15(12):3432–3444, August 2022.

[46] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating*

Systems Principles (SOSP '13) [22].

[47] Claude Barthels. A lightweight C++ RDMA library for InfiniBand networks. https://github.com/claudebarthels/infinity.

[48] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, Online, July 2020.

[49] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IN, June 2010.

[50] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI '08)*, San Francisco, CA, April 2008.

[51] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, February 2020.

[52] Facebook. RocksDB. http://rocksdb.org/.

[53] Sanjay Ghemawhat, Jeff Dean, Chris Mumford, David Grogan, and Victor Costan. LevelDB. https://github.com/google/leveldb, 2011.

[54] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, March 2017.

[55] Chuanxiong Guo. Rdma in data centers: Looking back and looking forward. *Keynote at APNet*, 2017.

[56] 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*, New York, NY, United States, August 2016.

[57] Haoyu Huang and Shahram Ghandeharizadeh. Nova-LSM: A Distributed, Component-Based LSM-Tree Key-Value Store. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)* [25].

[58] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A Right-optimized Write-optimized File System. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST '15)*, Santa Clara, CA, February 2015.

[59] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*, Virtual, October 2021.

[60] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '19)*, Boston, MA, February 2019.

[61] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012.

[62] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving Scalability and Fault-tolerance for Microsecond-scale Datacenter Services. In *Proceedings of the EuroSys Conference (EuroSys '20)* [24].

[63] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[64] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[65] Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo Coelho, and Fernando Pedone. RamCast: RDMA-Based Atomic Multicast. In *Proceedings of the 22nd 22nd International Middleware Conference (Middleware'21)*, Quebec, Canada, Dec 2021.

[66] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* [23].

[67] Nanqinqin Li, Anja Kalaba, Michael J Freedman, Wyatt Lloyd, and Amit Levy. Speculative Recovery: Cheap, Highly Available Fault Tolerance with Disaggregated Storage. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022.

[68] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. 2012.

[69] Microsoft. Microsoft SQL Server Replication. https://learn.microsoft.com/en-us/sql/relational-databases/replication/sql-server-replication?view=sql-server-ver16.

[70] MongoDB. MongoDB. https://www.mongodb.org/.

[71] MongoDB. MongoDB Replication. https://docs.mongodb.com/manual/replication.

[72] MySQL. MySQL Replication. https://dev.mysql.com/doc/refman/8.0/en/replication.html.

[73] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)* [21].

[74] Nvidia. RDMA Over Converged Ethernet (RoCE). https://docs.nvidia.com/networking/pages/viewpage.action?pageId=19807245.

[75] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.

[76] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014.

[77] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February 2017.

[78] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[79] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building Key-value Stores using Fragmented Log-structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[80] S. Rao, L. Alvisi, and H.M. Vin. The Cost of Recovery in Message Logging Protocols. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):160–173, 2000.

[81] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. RDMA is Turing Complete, we just did not know it yet! In *Proceedings of the 19th Symposium on Networked Systems Design and Implementation (NSDI '22)*, Renton, WA, April 2022.

[82] Redis. Redis. http://redis.io/.

[83] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efficiency

in the Local File System. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, San Jose, CA, June 2013.

[84] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *USENIX ;login:*, 39(6), 2014.

[85] Richard Gooch. Overview of the Linux Virtual File System. `https://www.kernel.org/doc/html/latest/filesystems/vfs.html`.

[86] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. The Design of a Practical System for Fault-tolerant Virtual Machines. *ACM SIGOPS Operating Systems Review*, 44(4), 2010.

[87] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, November 2018.

[88] Rong Shi and Yang Wang. Cheap and Available State Machine Replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016.

[89] Jeff Terrace and Michael J Freedman. Object Storage on CRAQ: High-throughput Chain Replication for Read-mostly Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '09)*, San Diego, CA, June 2009.

[90] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, December 2004.

[91] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: Fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)* [23].

[92] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)* [21].

[93] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. Redy: Remote Dynamic Memory Cache. *Proc. VLDB Endow.*, 15(4), December 2021.

[94] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Xinjun Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proc. VLDB Endow.*, 14(10):1900–1912, jun 2021.

[95] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)* [25].

[96] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. Octopus+: An Rdma-Enabled Distributed Persistent Memory File System. *ACM Transactions on Storage (TOS)*, 17(3), 2021.

[97] Tobias Ziegler, Carsten Binnig, and Viktor Leis. ScaleStore: A Fast and Cost-Efficient Storage Engine Using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD '22)*, Philadelphia, PA, June 2022.

A Artifact Appendix

A.1 Abstract

The near-compute log (Ncl) library replicates write-ahead logs of storage-centric POSIX applications (e.g., databases, key-value stores) to underutilized memory on remote nodes. This artifact contains the source code to build Ncl library, as well as the benchmarks to evaluate Ncl.

A.2 Description & Requirements

The source code of Ncl is available at https://github.com/dassl-uiuc/compute-side-log. The source code of the benchmark is available at https://github.com/dassl-uiuc/YCSB-cpp. To test the basic functionalities, one needs at least two machines equipped with RDMA NICs. The full evaluation requires eight machines and at least five of them need to be equipped with RDMA NICs:

- one machine as the application client node (RDMA required)
- one machine as the application server node (RDMA required)
- three machines as the Ceph cluster and has CephFS deployed
- three machines as the memory replicas for Ncl (RDMA required)

Testing the basic functionalities is relatively simple and evaluators may use their own hardware if they have RDMA machines. If you don't have RDMA machines and would like to setup by your own we recommend using CloudLab xl170.

The full evaluation requires more dependencies including setting up a Ceph cluster. To ease the procedure of setting up the environment, during the artifact evaluation period, we provided the already-setup machines and evaluators can ssh into them to run the evaluation. Our evaluation cluster consists of eight CloudLab xl170 machines. We recommended the evaluators use our provided machines.

A.2.1 How to Access

During the artifact evaluation period, we set up an 8-node cluster on CloudLab for reviewers to run the experiments. The cluster is not available now. Please see benchmark document for instructions on how to set up your own experimental cluster.

A.2.2 Hardware Dependencies

Machines equipped with RDMA NICs (Mellanox ConnectX-4 25 Gb or better to match the performance in our paper).

A.2.3 Software Dependencies

Ubuntu 20.04 operating system. For complete software dependencies, please refer to our Ncl document and benchmark document.

A.2.4 Benchmarks

We use YCSB benchmark. During the artifact evaluation, any data required was provided on our machines. To run YCSB benchmark on your own cluster, you need to first generate the base database for each application evaluated. The instructions to generate base databases is in the benchmark document.

A.3 Setup

Our evaluation cluster needs no additional setup. To setup your own cluster, see Ncl document and benchmark document.

A.4 Evaluation Workflow

For kick-the-tires, please see Ncl document for steps to test basic functionalities. This section discusses the comprehensive evaluation.

A.4.1 Major Claims

- (C1): Ncl has good write latency. This is proved by the write microbenchmark described in §5.1. Results are shown in Figure 8 in the paper.
- (C2): Ncl has good write performance for three applications (RocksDB, Redis, and SQLite). It achieves the same level of fault tolerance as strong apps while offering the same level of latency and throughput as weak apps. This is proved by the write-only workload described in §5.2. Results are shown in Figure 9 in the paper.
- (C3): Ncl has good performance in YCSB benchmark. It offers same level of throughput as weak apps and is much better than strong apps in update-heavy benchmarks. This is proved by YCSB benchmark described in §5.3. Results are shown in Figure 10 in the paper.
- (C4): Ncl can fast recover log files from replicas. This is proved by read benchmark and measuring recovery time for 3 applications. Results are shown in Figure 11 in the paper.

A.4.2 Experiments

Please see benchmark document for detailed instructions for running the experiments.

A.5 Notes on Reusability

For usage of Ncl see General Usage for directions. For development upon the code base, the code is documented. You can run `doxygen ./Doxyfile` to generate html document of Ncl. After generation, the document can be viewed from `./html/index.html`.