Assise: Performance and Availability via Client-local NVM in a Distributed File System

Thomas E. Anderson[1] Marco Canini[2] Jongyul Kim[3†] Dejan Kostić[4] Youngjin Kwon[3]
Simon Peter[5] Waleed Reda[4,6⋆] Henry N. Schuh[1†] Emmett Witchel[5]

[1]*University of Washington* [2]*KAUST* [3]*KAIST* [4]*KTH Royal Institute of Technology*
[5]*The University of Texas at Austin* [6]*Université catholique de Louvain*

Abstract

The adoption of low latency persistent memory modules (PMMs) upends the long-established model of remote storage for distributed file systems. Instead, by colocating computation with PMM storage, we can provide applications with much higher IO performance, sub-second application failover, and strong consistency. To demonstrate this, we built the Assise distributed file system, based on a persistent, replicated coherence protocol that manages client-local PMM as a *linearizable* and *crash-recoverable* cache between applications and slower (and possibly remote) storage. Assise maximizes locality for all file IO by carrying out IO on process-local, socket-local, and client-local PMM whenever possible. Assise minimizes coherence overhead by maintaining consistency at IO operation granularity, rather than at fixed block sizes.

We compare Assise to Ceph/BlueStore, NFS, and Octopus on a cluster with Intel Optane DC PMMs and SSDs for common cloud applications and benchmarks, such as LevelDB, Postfix, and FileBench. We find that Assise improves write latency up to 22×, throughput up to 56×, fail-over time up to 103×, and scales up to 6× better than its counterparts, while providing stronger consistency semantics.

1 Introduction

Byte-addressable non-volatile memory (NVM), such as Intel's Optane DC persistent memory module (PMM) [14], is now commercially available as main memory. NVM provides high-capacity persistent memory with near-DRAM performance at lower cost. The promise of NVM as a low-cost main memory add-on is driving the adoption of node-local NVM at scale [43, 86, 87]. Remote direct memory access (RDMA) allows NVM access across the network without CPU overhead, raising interest in NVM for high-performance distributed storage.

A common paradigm in distributed file systems, like Amazon EFS [2], NFS [39], Ceph [82], Colossus/GFS [37], and NVM re-designs, like Octopus [58] and Orion [85], is to separate storage servers from clients. In this server-client design, files are stored by servers on machines physically separated from clients running applications. Client main memory is treated as a volatile block cache managed by the client's OS kernel. This design simplifies resource pooling by physically separating application from storage concerns with simple, server-managed data consistency mechanisms.

This simplicity comes at a cost, which becomes apparent as we move from SSD/HDD to NVM storage. In steady state, application performance is limited by the overhead to access kernel-level client caches. Upon a cache miss, multiple network round trips are needed to consult remote metadata servers and to fetch the actual data. On failure, client-server file systems must rebuild caches of failed clients from scratch, involving long fail-over times to re-establish application-level service and necessitating high network utilization during recovery. Third, managing client caches at fixed page-block granularity amplifies the small IO operations typical of many distributed applications and increases cache coherence overhead when IO is larger than the block size. These costs prevent NVM from reaching its performance potential and have led some within the storage community to advocate for a complete redesign of the file system API [54, 72, 73, 88].

We present Assise, a distributed file system designed to maximize the use of *client-local* NVM without requiring a new API for high performance. Assise unleashes the performance of NVM via pervasive and persistent caching in process-local, socket-local, and node-local NVM. Assise accelerates POSIX file IO by orders of magnitude by leveraging client-local NVM without kernel involvement, block amplification, or unnecessary coherence overheads. Assise provides near-instantaneous application fail-over onto a *hot replica* that mirrors an application's local file system cache in the replica's local NVM. Assise reduces node recovery time by orders of magnitude by locally recovering NVM caches with strong consistency semantics. Finally, Assise leverages cluster-wide NVM via *warm replicas* that provide lower latency reads than slower storage media, such as SSDs. In cascaded hot replica failure scenarios, warm replicas can become hot replicas to preserve near-instantaneous fail-over.

To enable these properties, we design and build to our knowledge the first crash consistent distributed file system cache coherence layer for replicated NVM (CC-NVM). CC-NVM serves cached file system state in Assise with strong consistency guarantees and locality. CC-NVM provides prefix crash consistency [80] by enforcing write order to local

[⋆]Lead student author.
[†]Co-student authors.

NVM via logging and to cross-socket and remote NVM by leveraging the write order of DMA and RDMA, respectively. CC-NVM provides linearizability for all IO operations via leases [38] that can be delegated among nodes, sockets, and processes for local management of file system state. CC-NVM consistently chain-replicates [77] all file system updates to a configurable set of hot and warm replicas for availability.

Using CC-NVM, Assise achieves the following goals:

- **Simple programming model.** Assise supports unmodified applications using the familiar POSIX API with strong linearizability and crash consistency [80].
- **Scalability.** Unlike NVM-aware distributed file systems that are limited to rack-scale [71, 85], Assise provides strong consistency but remains scalable using dynamic delegation of leases to nodes, sockets, and processes; local sharing uses CC-NVM for consistency without network, cross-socket, or kernel communication.
- **Low IO tail latency.** To efficiently support applications with low tail latency requirements, Assise allows kernel-bypass access to authorized local and remote NVM areas. To reduce write latency with replicated persistence, Assise provides an optimistic mode using asynchronous chain replication with prefix crash consistency.
- **High availability.** Assise provides near-instantaneous fail-over to a configurable number of replicas and minimizes the time to restore the replication factor after failure.
- **Efficient bandwidth use.** The high bandwidth provided by NVM means that communication can be a throughput bottleneck (cf. Table 1). Assise minimizes communication by eliminating redundant writes [52] and reducing coherence protocol overhead via logging.

We make the following contributions:

- We present the design (§3) and implementation (§4) of Assise, a distributed file system that fully utilizes NVM by persistent caching in client-local NVM as a primary design principle. Assise uses client-local NVM to recover the file system cache for fast fail-over and locally synchronizes reads and writes to file system state.
- We present CC-NVM (§3.3), the first persistent and available distributed cache coherence layer. CC-NVM provides locality for data and metadata access, replicates for availability, and provides linearizability and prefix crash consistency for all file system IO.
- We quantify the performance benefits of using local NVM versus remote NVM for distributed file systems (§5). We compare Assise's steady-state and fail-over behavior to RDMA-accelerated versions of Ceph with BlueStore [21] and NFS, as well as Octopus [58], a distributed file system designed for RDMA and NVM, using common cloud applications and benchmarks, such as LevelDB, Postfix, MinuteSort, and FileBench.

Our evaluation shows that Assise provides up to 22× lower write latency and up to 56× higher throughput than NFS and Ceph/BlueStore. Assise outperforms Octopus by up to an order of magnitude. Assise scales better than Ceph, providing 6× throughput for Postfix with 48 processes over 3 nodes. Assise is more available than Ceph, returning a recovering LevelDB store to full performance up to 103× faster. Demonstrating that strong consistency with the familiar POSIX API and high performance are not mutually exclusive, Assise finishes a local external sort 3% faster than a hand-tuned implementation using processor loads and stores to memory mapped NVM. Finally, Assise finishes the MinuteSort distributed sorting benchmark up to 2.2× faster than a parallel NFS installation.

Memory	R/W Latency	Seq. R/W GB/s	$/GB
DDR4 DRAM	82 ns	107 / 80	9.77 [19]
NVM (local)	175 / 94 ns	32 / 11.2	3.83 [20]
NVM-NUMA	230 ns	4.8 / 7.4	-
NVM-kernel	0.6 / 1 μs	-	-
NVM-RDMA	3 / 8 μs	3.8	-
SSD (local)	10 μs	2.4 / 2.0	0.32 [15]

Table 1: Memory & storage price/performance (October 2020).

Assise supports networked access to remote storage where it makes sense. Assise can automatically migrate cold data that does not fit in NVM to slower, network-attached storage devices, such as SSDs and HDDs. To do so, Assise's implementation builds on Strata [52] as its node-local store.

2 Background

Distributed applications have diverse workloads, with IO granularities large and small [56], different sharding patterns, and consistency requirements. All demand high availability and scalability. Supporting these properties simultaneously has been the focus of decades of distributed storage research [23, 39, 41, 58, 81, 82, 85]. Before NVM, trade-offs had to be made. For example, by favoring large transfers ahead of small IO, or steady-state performance ahead of crash consistency and fast recovery, leading to the common idiom of remote-storage file system design. We argue that with the arrival of fast NVM, these trade-offs need to be re-evaluated.

The opportunity posed by NVM is two-fold:

Cost/performance. Table 1 shows measured access latency, bandwidth, and cost for modern server memory and storage technologies, including Optane DC PMM (measurement details in §5). We can see that local NVM access latency and bandwidth are close to DRAM, up to two orders of magnitude better than SSD. At the same time, NVM's per-GB cost is only 39% that of DRAM. NVM's unique characteristics allow it to be used as the top layer in the storage hierarchy, as well as the bottom layer in a server's memory hierarchy.

Fast recovery. Persistent local storage with near-DRAM performance can provide a *recoverable* cache for hot file system data that can persist across reboots. The vast majority of system failures are due to software crashes that simply require rebooting [25, 36, 40]. Caching hot file system data in NVM allows for quick recovery from these common failures.

For these reasons, data center operators are deploying NVM

at scale [43, 86, 87]. However, to fully realize its potential, we have to efficiently use local NVM. NVM accessed via RDMA (NVM-RDMA), via loads and stores to another CPU socket (NVM-NUMA), or via the kernel on the same socket (NVM-kernel) can be an order of magnitude slower in terms of latency and bandwidth.

2.1 Related Work

We survey the existing work in distributed storage and highlight why it cannot fully utilize the storage system performance offered by local NVM.

Block and object stores, such as Amazon's EBS [1], S3 [3], and Ursa [56], provide a new API to a multi-layer storage hierarchy that can provide cheap, fault-tolerant access to vast amounts of data. However, block stores have a minimum IO granularity (16KB for EBS) and IO smaller than the block size suffers performance degradation from write amplification [56, 69]. For this reason, Dropbox uses Amazon S3 for data blocks, but keeps small metadata in DRAM for fast access, backed by an SSD [62]. Apache Crail [4] and Blizzard [63] provide file system APIs on top of block stores, but both focus on parallel throughput of large data streams, rather than small IO.

To realize the performance benefits of NVM for all IO, we need to abandon fixed block sizes and instead persist and track IO at its original operation granularity. Hence, Assise leverages logging to persist writes at their original granularity in NVM. A similar model is realized in the RAMcloud [66] key-value store. RAMcloud maintains data in DRAM for performance, using SSDs for asynchronous persistence. However, the capacity limits of DRAM mean that many RAMcloud operations still involve the network, and because DRAM state cannot be recovered after a crash, it is vulnerable to cascading node failures. Even after single node failures, state must be restored from remote nodes and RAMcloud requires a full-bisection bandwidth network for fast recovery. Assise leverages local NVM for recovery and does not require full-bisection bandwidth or asynchronous backup storage.

Client-server file systems, like Ceph [82], use distributed hashing over nodes to provide scalable file service for cloud applications. However, network and system call latency harms file IO latency, as shown in Table 1. Typical network access bandwidth to NVM is similarly surpassed by the higher bandwidth of local NVM.

To combat network overheads, several file systems have been built [58, 85] or retrofitted [13, 39, 44] to use RDMA. Octopus [58] and Orion [85] are redesigns that use RDMA for high performance access to NVM. Still, neither leverages kernel-bypass for low-latency IO (Octopus uses FUSE, Orion runs in the kernel) and both pool storage remotely. Like Ceph, Octopus uses distributed hashing to place files on nodes (Octopus does not replicate). Orion can store data locally via "internal clients," but uses a metadata server. Clover [76] is a key-value store that takes the opposite approach, locating metadata with applications, but storing data remotely. All systems perform remote operations in the common case to update data and/or metadata, increasing IO latency.

Concept	Explanation
LibFS	Per-process, user-level file system library
SharedFS	Per-socket system daemon; manages local leases
CC-NVM	Crash-consistent cache coherence with linearizability
Hot replica	Cache-hot replica for fast failover
Warm replica	Provides NVM for low-latency, remote, warm reads
Cluster manager	Fault-tolerant service for membership & leases

Table 2: Concepts used in Assise.

Network latency and limited bandwidth increase operation latency, reduce throughput, and limit scalability. For example, due to update contention at a central metadata server, Orion scales only to a small number of clients. Orion omits an evaluation of server fail-over and recovery (Assise's is in §5.4). Tachyon [55] aims to circumvent replication overhead by leveraging the concept of *lineage*, where lost output is recovered by re-executing application code that created the output. However, to do so, Tachyon requires applications to use a complex data lineage tracking API.

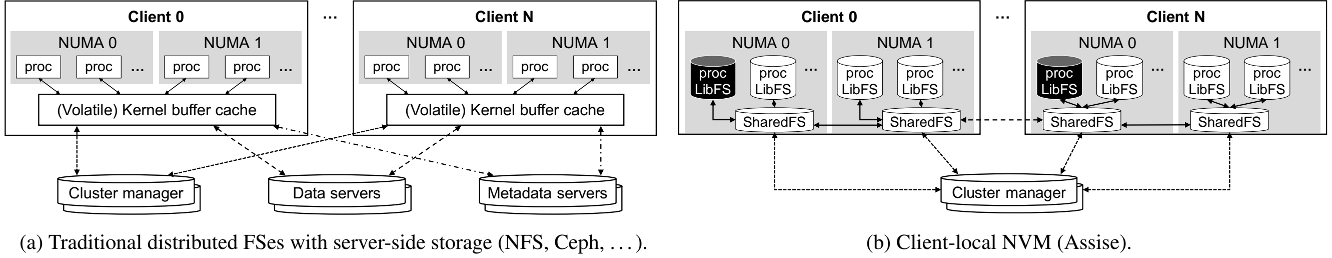
To maximize NVM utility, we need to design for a scenario where kernel and networking overheads are high compared to storage access. Hence, Assise eliminates kernel overhead for local IO operations and remote IO incurs a single operation to the nearest replica in the common case, without requiring dedicated metadata servers or a distributed hash to balance load. For scalability, we need to enforce data and metadata consistency locally, which CC-NVM tackles with the help of leases. Unlike Tachyon, Assise supports the classic POSIX file API and is fully compatible with existing applications.

Leases [38, 57] have long been integral to performance in distributed file systems, by allowing local operations to leased portions of the file system name space, with linearizability. Read-only leases are a common design pattern [12, 27, 39, 42], but some research systems have explored using both read and write leases in a similar manner to Assise. A prominent example is Berkeley xFS [23], which maintained a local block-level update log at each node, written as a software RAID 5/6 partitioned across other nodes. Assise differs from xFS by using an operational log, replicating rather than striping the log, and by doing update coalescing.

2.2 Remote Storage versus Local NVM

Figure 1 contrasts the IO architecture of traditional client-server file systems and Assise. Each subfigure shows two dual-socket nodes executing a number of application processes sharing a distributed file system. Both designs use a replicated cluster manager for membership management and failure detection, but they diverge in all other respects.

Traditional distributed file systems first partition available cluster nodes into clients and servers. Clients cache file system state in a volatile kernel buffer cache that is shared by processors across sockets (NVM-NUMA) and accessed via expensive system calls (NVM-kernel). Persistent file system state is stored in NVM on remote servers. For persistence and



(a) Traditional distributed FSes with server-side storage (NFS, Ceph, ...). (b) Client-local NVM (Assise).

Figure 1: Distributed file system IO architectures. Arrow = RPC/system call. Cylinder = persistence. Black = hot replica.

consistency, clients thus have to coordinate updates with replicated storage and metadata servers via the network (NVM-RDMA) with higher latency than local NVM. The cluster manager is not involved in IO. Data is typically distributed at random over replicated storage servers for simplicity and load balance [82]. The overhead of updating a large set of storage nodes atomically means that (crash) consistency is often provided only for metadata, which is centralized.

3 Assise Design

Assise avoids remote storage servers and instead uses CC-NVM to coordinate linearizable state among processes. Processes access cached file system state in local NVM directly via a library file system (LibFS), which may be replicated for fail-over (two LibFS hot replicas shown in black in Figure 1). CC-NVM coordinates LibFSes hierarchically via per-socket daemons (SharedFS) and the cluster manager. Table 2 explains several Assise-related concepts.

Crash consistency modes. Assise supports two crash consistency modes: optimistic or pessimistic [30]. Mount options specify the chosen crash consistency mode. When pessimistic, `fsync` forces immediate, synchronous replication and all writes prior to an `fsync` persist across failures. When optimistic, Assise commits all operations in order, but it is free to delay replication until the application forces it with a `dsync` call [30]. Optimistic mode provides lower latency persistence with higher throughput, but risks data loss after crashes that cannot recover locally (§3.4). In either mode, Assise guarantees a prefix crash-consistent file system [80]—all recoverable writes are in order and no parts of a prefix of the write history are missing.

We now describe cluster coordination and membership management in Assise (§3.1). We then detail the IO paths (§3.2) and show how CC-NVM interacts with them to provide linearizability and prefix crash consistency (§3.3). Finally, we describe recovery (§3.4) and warm replicas (§3.5). We close with a discussion of connected design questions (§3.6).

3.1 Cluster Coordination and Failure Detection

Like other distributed file systems, Assise leverages a replicated cluster manager for storing the cluster configuration and detecting node failures. Assise uses the ZooKeeper [10] distributed coordination service as its cluster manager.

Cluster coordination. Each SharedFS in Assise registers with the cluster manager. In our prototype, the system administrator decides which SharedFS replicates which parts of the cached file system namespace and the caching policy (hot or warm replica) for arbitrary subtrees; the cluster manager records this mapping. When a subtree is first accessed, LibFSes contact their local SharedFS, which consults the cluster configuration and sets up an RDMA replication chain from LibFS through the subtree's hot replicas. For each chain, hot replicas preallocate a configurable amount of NVM for replication (sensitivity evaluated in §5.2). It is future work to implement a distributed replica discovery service (e.g., using CC-NVM). LibFSes on any node are already able to cache any (meta-)data with linearizability.

Failure detection. The cluster manager sends heartbeat messages to each active SharedFS once every second. If no response is received after a timeout, the node is marked failed and all connected SharedFS are notified. When the node comes back online, it contacts the cluster manager and initiates recovery (§3.4).

3.2 IO Paths

Application IO interacts first with Assise caches. To keep tail latency low, Assise does not use a shared kernel buffer cache. Instead, LibFS caches file system state first in process-local memory. The LibFS cache uses both NVM and DRAM. NVM stores updates, while DRAM caches reads. LibFS implements the POSIX API at user-level. We now discuss cache operation upon IO, including replication, eviction, and access permissions. Figure 2 shows these mechanisms for two hot replicas and one warm replica, using SSDs for cold storage. Cache coherence is discussed in §3.3.

3.2.1 Write Path

Writes in Assise involve three mechanisms that operate on different time scales:

1. To allow for persistence with low latency, LibFS directly writes into a process-local cache in NVM (W). To efficiently support writes of any granularity, the write cache is an *update log*, rather than a block cache.
2. To outlive node failures, the update log is chain-replicated, with kernel-bypass, by LibFS (S1, S2).
3. When update logs fill beyond a threshold, evictions are initiated (E2), moving their contents to SharedFS. We describe replication and eviction next.

Replication and crash consistency. When pessimistic, `fsync` forces immediate, synchronous replication. The caller

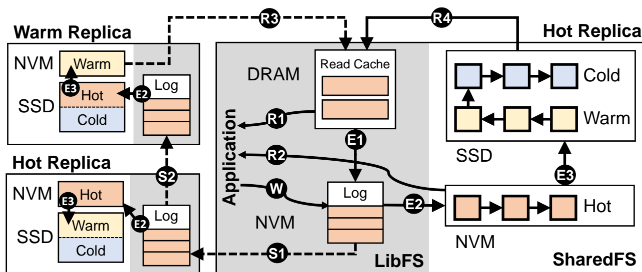


Figure 2: Assise IO paths. Dashed line = RDMA operation, solid line = local operation. Shaded areas are per process.

is blocked until all writes up to the `fsync` have been replicated. Thus, all writes prior to an `fsync` outlive node failures.

When optimistic, Assise is free to delay replication. This provides Assise with an opportunity to *coalesce* [52] temporary durable writes (i.e., overwritten or deleted files), a workload pattern seen in application-level commit protocols [67]. Eliminating these writes allows Assise to conserve network bandwidth. In optimistic mode, Assise initiates replication on `dsync` or upon log eviction.

In both cases, the local update log contents are written to preallocated NVM on the first replica along the replication chain via RDMA (S1). The replica continues chain replication to the next replica (S2), and so on. The final replica in the chain sends an acknowledgment back along the chain to indicate that the chain completed successfully.

Cache eviction. When a LibFS update log fills, it replicates any unreplicated writes and initiates eviction. Eviction is done in least-recently-used (LRU) fashion through the SharedFS shared caches to cold storage (E2, E3). Hot replicas keep *hot* data in NVM, while moving *warm* and *cold* data to cold storage. Warm replicas (§3.5) keep hot and cold data in cold storage, while warm data resides in NVM to accelerate warm reads (§3.5). Cold storage may be remote (e.g., via NVMe-over-Fabrics [17]). Each replica along the chain evicts in parallel and acknowledges when eviction is finished. This ensures that all replicas cache identical state for fast failover.

For log eviction (E2), issuing direct stores to NVM shared caches on another socket has overhead due to cross-socket hardware cache coherence, limiting throughput [83]. Since CC-NVM provides cache coherence, Assise can bypass hardware cache coherence by using DMA [53] when evicting to NVM-NUMA. This yields up to 30% improvement in cross-socket file system write throughput (§5.5).

3.2.2 Read Path

LRU cache eviction guarantees that the latest version of all data is always available in the fastest cache. Thus, upon a read, LibFS (1) checks the process-private write and read caches (via a *log hashtable* and *read cache*, shown in Figure 2) for the requested data (R1). If not found, LibFS (2) checks the node-local hot SharedFS cache (R2) (via an *extent tree* used to index the SharedFS cache [52]). If the data was found in either of these areas, it is read locally. If not found, LibFS (3) checks the warm replica's SharedFS cache (R3), if it exists, and, in parallel, checks cold storage (R4).

Read cache management. Recently read data is cached in process-local DRAM, except if it was read from local NVM, where DRAM caching does not provide benefit. LibFS prefetches up to 256KB from cold storage and up to 4KB from remote NVM. For remote NVM reads, LibFS first fetches the requested data and then prefetches the remainder. This minimizes small read latency while improving the performance of workloads with spatial locality. Data from remote NVM and cold storage is evicted from the read cache to the process-local update log (E1).

3.2.3 Permissions and Kernel Bypass

Assise assumes a single administrative domain with UNIX file and directory ownership and permissions. SharedFS enforces that LibFS may access only authorized data, by checking permissions and metadata integrity upon cache eviction and enforcing permissions on reads. To minimize latency of node-local SharedFS cache reads, Assise allows read-only mapping of authorized parts of the SharedFS cache into the LibFS address space. LibFS caches and mappings are invalidated when files or directories are closed and whenever the update log is evicted.

The metadata integrity of the file system is ensured by SharedFS. LibFS operations do not prevent one thread from corrupting another's data in the process-local update log, but SharedFS verifies that all metadata operations are valid before they become visible to other processes. This implies that processes can corrupt *their* own data in their private update log, even after it was written (memory protection keys can mitigate inter-thread data corruption [34]). However, only process-local writes go to the process-local update logs. Multi-process access to any filesystem object (including a subtree) is linearizable and access-controlled via leases. Processes cannot corrupt shared file system (meta-)data.

3.3 Crash Consistent Cache Coherence with CC-NVM

CC-NVM provides distributed cache coherence with linearizability when sharing file system state among processes; it provides prefix semantics upon a crash.

Prefix crash consistency. To provide prefix crash consistency, CC-NVM tracks write order via the update log in process-local NVM. Each POSIX call that updates state is recorded, in order, in the update log. When chain-replicating, CC-NVM leverages the write ordering guarantees of (R)DMA to write the log in order to replicas. In optimistic mode, CC-NVM wraps coalesced file system operations in a Strata transaction [52]. This ensures that file system updates are persisted and replicated atomically and that a prefix of the write history can be recovered (§3.4).

Sharing with linearizability. CC-NVM serializes concurrent access to shared state by multiple processes and recovers the same serialization after a crash via leases [38]. Leases

provide a simple, fault-tolerant mechanism to delegate access. Leases function similarly to reader-writer locks, but can be revoked (to allow another process access) and expire after a timeout (after which they may be reacquired). In CC-NVM, leases are used to grant shared read or exclusive write access to a set of files and directories—multiple read leases to the same set may be concurrently valid, but write leases are exclusive. Reader/writer semantics efficiently support shared files and directories that are read-mostly and widely used, but also write-intensive files and directories that are not frequently shared. CC-NVM also supports a *subtree lease* that includes all files and directories at or below a particular directory. A subtree lease holder controls access to files and directories within that subtree. For example, a LibFS with an exclusive subtree lease on `/tmp/bwl-ssh/` can recursively create and modify files and directories within this subtree.

Leases must be acquired by LibFS from SharedFS via a system call before LibFS can cache the data covered by the lease. Assise does this upon first IO; leases are kept until they are revoked by SharedFS. This occurs when another LibFS wishes access to a leased file or when a LibFS instance crashes or the lease times out. Lease revocation latency is bounded by a grace period, within which the current lease holder can finish its ongoing IO before releasing contended leases. If LibFS fails to surrender the lease after the grace period, the lease is revoked by SharedFS and any subsequent IO on the leased file is rejected as invalid. SharedFS enforces that the lease holder's read and write caches are cleaned and evicted of the covered data before the lease is transferred. The time taken to do so is bounded by the holder's update log size. SharedFS logs and replicates each lease transfer in NVM for crash consistency. A LibFS may overlap IO with SharedFS lease replication until `fsync`/`dsync`.

Hierarchical coherence. To localize coherence enforcement, leases are delegated hierarchically. The cluster manager is at the root of the delegation tree, with SharedFSes as children, and LibFSes as leaves (cf. Figure 1b). LibFSes request leases first from their local SharedFS. If the local SharedFS is not the lease holder, it consults the cluster manager. If there is no current lease holder, the cluster manager assigns the lease to the requesting SharedFS, which delegates it to the requesting LibFS and becomes its *lease manager*. If a lease manager already exists, SharedFS forwards the request to the manager and caches the lease manager's information (leased namespace and expiration time of lease). The cluster manager expires lease management from SharedFSes every 5 seconds. This allows CC-NVM to migrate lease management to the local SharedFS, while preventing leases from changing managers too quickly, facilitating scalability.

Hierarchical coherence minimizes network communication and thus lease delegation overhead. LibFSes on the same node or socket require only local SharedFS delegation in the common case. This structure maps well to the data sharding patterns of many distributed applications (§5.5).

3.4 Fail-over and Recovery

Assise caches file system state with persistence in local NVM, which it can use for fast recovery. Assise optimizes recovery performance for the most common crash types.

LibFS recovery. An application process crashing is the most common failure scenario. In this case, the local SharedFS simply evicts the dead LibFS update log, recovering all completed writes (even in optimistic mode) and then expires its leases. Log-based eviction is idempotent [52], ensuring consistency in the face of a system crash during eviction. The crashed process can be restarted on the local node and immediately reuse all file system state. The LibFS DRAM read-only cache has to be rebuilt, with minimal performance impact (§5.4).

SharedFS recovery. Another common failure mode is a reboot due to an OS crash. In this case, we can use NVM to dramatically accelerate OS reboot by storing a checkpoint of a freshly booted OS. After boot, Assise can initiate recovery for all previously running LibFS instances, by examining the SharedFS log stored in NVM.

Cache replica fail-over. To avoid waiting for node recovery after a power failure or hardware problem, we immediately fail-over to a hot replica. The replica's SharedFS takes over lease management from the failed node, using the replicated SharedFS log to re-grant leases to any application replicas. The new instances will see all IO that preceded the most recently completed `fsync`/`dsync`.

Writes to the file system can invalidate cached data of the failed node during its downtime. To track writes, the cluster manager maintains an epoch number, which it increments on node failure and recovery. All SharedFS instances are notified of epoch updates. All SharedFS instances share a per-epoch bitmap in a sparse file indicating what inodes have been written during each epoch. The bitmaps are deleted at the end of an epoch when all nodes have recovered.

Node recovery. When a node crashes, the cluster manager makes sure that all of the node's leases expire before the node can rejoin. When rejoining, Assise initiates SharedFS recovery. A recovering SharedFS contacts an online SharedFS to collect relevant epoch bitmaps. SharedFS then invalidates every block from every file that has been written since its crash. This simple protocol could be optimized, for instance, by tracking what blocks were written, or checksumming regions of the file to allow a recovering SharedFS to preserve more of its local data. But the table of files written during an epoch is small and quickly updated during file system operation, and our simple policy has been sufficient.

3.5 Warm Replicas

To fully exploit the memory hierarchy presented in Table 1, remote NVM can be used as a third-level cache, below local DRAM and local NVM. To do so, we introduce *warm replicas*. Like hot replicas, warm replicas receive all file system updates via chain-replication, but leverage a different update

log eviction policy. Warm replicas track the LRU chain for a specified portion of "warm data" beyond the LibFS and SharedFS caches. Warm replicas do not impact the latency of replicated writes, but they reduce read latency for warm data by serving these reads from NVM, rather than cold storage.

LibFSes can read from warm replicas via RDMA with lower latency and higher bandwidth than cold storage (NVM-RDMA versus SSD in Table 1). Applications do not run on warm replicas in the common case. In the rare case of a failure cascade crashing all hot replicas, processes can fail-over to warm replicas, albeit with reduced short-term performance. After fail-over, warm replicas become hot replicas and hot data must be migrated back into local NVM.

3.6 Discussion

Assise may be deployed at scale. The use of local NVM together with hierarchical lease delegation aligns well with datacenter server, rack, and pod architecture [22]. We discuss factors of Assise's design that impact such a deployment. In particular, the memory overhead of per-process and per-replica update logs, the use of NVM and RDMA at scale, and security.

Update log scalability. Assise uses per-process and per-replica update logs for efficient chain-replication with kernel-bypass. These update logs are preallocated on process creation in our prototype. While update logs can support high performance at moderate size (§5.2), a deployment at scale might be concerned with the memory consumption of update logs. In this case, the per-process and per-replica update log size can be adapted dynamically to momentarily available NVM capacity and per-process IO demand. SharedFS can resize logs upon eviction. The most significant overhead for log resizing is memory registration for RDMA. It requires pinning the memory and mapping it in the RDMA NICs. This operation can be overlapped with the log eviction itself. To help reduce the need for frequent resizing, logs can be resized multiplicatively, similar to resizing approaches in prior work [84].

RDMA scalability. Assise uses RDMA reliable connections (RCs) for each process and replica. RCs require the NIC to create and maintain connection state. For larger clusters, maintaining a large number of connections can stress the NIC's limited memory and degrade performance. Several proposals have been made to reduce NIC cache thrashing [29, 68] and Mellanox introduced dynamically-connected (DC) transports [70], which allows connection-sharing and enables a high degree of scalability. Assise can leverage these approaches to scale the use of RDMA.

NVM wear-out. Assise uses local NVM extensively. This use can lead to the wear-out of NVM. To prevent frequent NVM replacement at scale, it is important to minimize writes to the NVM media. Assise's update logs minimize write amplification, but update log eviction in causes a 2× write amplification in the worst case. This write amplification can be partially eliminated via coalescing as seen in workloads, like Varmail (§5.3). To further reduce write amplification, update log pages may be remapped to the SharedFS shared cache, without introducing any additional writes [48]. We leave this as future work.

Security. In a large-scale public cloud scenario, data from each tenant is usually encrypted for security. For this purpose, both NVM and RDMA support encryption of data at rest and in-flight. Intel's Optane DC PMMs support transparent hardware encryption of data stored in NVM and modern RDMA NICs [61] support transparent encryption of RDMA operations.

4 Implementation

Assise uses *libpmem* [11] for persisting data on NVM and *libibverbs* for RDMA operations in userspace. Assise intercepts POSIX file system calls and invokes the corresponding LibFS implementation of these functions in userspace [8]. The Assise implementation consists of 28,982 lines of C code (LoC), with LibFS and SharedFS using 16,515 and 6,563 LoC, respectively. The remaining 5,904 LoC contain utility code, such as hash tables and linked lists. SharedFS communicates with LibFSes via shared memory [24]. Assise uses Strata code (LoC not counted) for cold storage in SSD and HDD.

Assise uses Intel Optane DC PMM in App-Direct mode. App-Direct exposes NVM as a range of physical memory. It is the most efficient way to access NVM, but it requires OS support. OS-transparent modes have weaker persistence or performance properties [45]. For example, memory mode integrates NVM as *volatile* memory, using DRAM as a hardware-managed level 4 cache. Sector mode exposes NVM as a disk, with attendant IO amplification and disk driver overheads.

4.1 Strata as a Building Block

Assise builds upon Strata's local file system functionality and augments it with the CC-NVM cache coherence layer and RDMA to create a replicated and highly efficient distributed file system with prefix crash conistency. Assise inherits several components from Strata, including its use of extent trees to index storage managed by SharedFS (in turn based on Ext4 [60]), the LibFS update log, and log coalescing. We enhance Strata's extent trees to manage directories and Strata's leases to support delegation.

4.2 Efficient Network IO with RDMA

Assise makes efficient use of RDMA. For lossless, in-order data transfer among nodes, Assise uses RDMA reliable connections (RCs). RCs have low header overhead, improving throughput for small IO [49, 59]. RCs also provide access to one-sided verbs that bypass CPUs on the receiver side, reducing message transfer times [35, 64] and memory copies [74].

Log replication. Logs are naturally suited for one-sided RDMA operations. Replication typically requires only one RDMA write, reducing header and DMA overheads [59]. As-

sise uses RDMA *write-with-immediate* for log replication. This operation performs a write and also notifies the remote replica to forward the data to the next replica in the chain. The only exceptions are when the remote log wraps around or when the local log is fragmented (due to coalescing), such that it exceeds the NIC's limit for scatter-gather DMA.

Persistent RDMA writes. The RDMA specification does not define the persistence properties of remote NVM writes via RDMA. In practice, the remote CPU is required to flush any RDMA writes from its cache to NVM. Assise flushes all writes via the CLWB and SFENCE instructions on each replica, before acknowledging successful replication. In the future, it is likely that enhancements to PCIe will allow RDMA NICs to bypass the processor cache and write directly to NVM to provide persistence without CPU support [50].

Remote NVM reads. Assise reads remote data via RPC. To keep the request sizes small, Assise identifies files using their inode numbers instead of their path. As an optimization, DRAM read cache locations are pre-registered with the NIC. This allows the remote node to reply to a read RPC by RDMA writing the data directly to the requester's cache, obviating the need for an additional data copy.

5 Evaluation

We evaluate Assise's common-case as well as its recovery performance, and break down the performance benefits attained by each system component. We compare Assise to three state-of-the-art distributed file systems that support NVM and RDMA. Our experiments rely on several microbenchmarks and Filebench [75] profiles, in addition to several real applications, such as LevelDB, Postfix, and MinuteSort. Our evaluation answers the following questions:

- **IO latency and throughput breakdown (§5.2).** What is the hardware IO performance of a storage hierarchy with local NVM (Table 1)? How close to this performance do the file systems operate under various IO patterns? What are the sources of overhead?
- **Cloud application performance (§5.3).** What is the performance of cloud applications with various consistency, latency, throughput, and scalability requirements? What is the overhead of Assise's POSIX API implementation versus hand-tuned, direct use of local NVM? By how much can a warm replica improve read latency? By how much can optimistic crash consistency improve write throughput for real applications?
- **Availability (§5.4).** How quickly can applications recover from various failure scenarios?
- **Scalability (§5.5).** How well does Assise perform when multiple processes share the file system? By how much can Assise's hierarchical coherence improve multi-process, multi-socket, and multi-node scalability?

Feature	Assise	Ceph	NFS	Octopus	Orion
Cache recovery	✓				
Local consistency	✓				
Kernel-bypass	✓				
Linearizability	✓				✓
Data crash consistency	✓				✓
Byte-oriented	✓			✓	✓
Replication	✓	✓			✓
RDMA	✓	✓	✓	✓	✓

Table 3: Features of the evaluated distributed file systems.

Testbed. Our experimental testbed consists of 5× dual-socket Intel Cascade Lake-SP servers running at 2.2GHz, with a total of 48 cores (96 hyperthreads), 384GB DDR4-2666 DRAM, 6TB Intel Optane DC PMM, 375GB Intel Optane DC P4800X series NVMe-SSD, and a 40GbE ConnectX-3 Mellanox InfiniBand NIC, connected via an InfiniBand switch. Exploiting all 6 memory channels per processor, there are 6 DIMMs of DRAM and NVM per socket. NVM is used in App-Direct mode (§4). All nodes run Fedora 27 with Linux kernel version 4.18.19.

Hardware performance. We first measure the achievable IO latency and throughput for each memory layer in our testbed server. We do this by using sequential IO and as many cores of a single socket as necessary. We measure DRAM and NVM (App-Direct) latency and throughput using Intel's memory latency checker [5]. NVM-RDMA performance is measured using RDMA read and write-with-immediate (to flush remote processor caches) operations to remote NVM. SSD performance is measured using `/dev/nvme` device files. The IO sizes that yielded maximum performance are 64B for DRAM, 256B for NVM, and 4KB for SSD. Table 1 shows these results. The measured IO performance for DRAM, NVM, and SSD matches the hardware specifications of the corresponding devices and is confirmed by others [45]. NVM-RDMA throughput matches the line rate of the NIC. NVM-RDMA write latency has to invoke the remote CPU (to flush caches) and is thus larger than read latency. We now investigate how close to these limits each file system can operate.

State-of-the-art. Table 3 shows performance-relevant features of the state-of-the-art and Assise. We can see that no open-source distributed file system provides all of Assise's features. Hence, a direct performance comparison is difficult. We perform comparisons against the Linux kernel-provided NFS version 4 [39] and Ceph version 14.2.1 [82] with BlueStore [21], both retrofitted for RDMA, as well as Octopus [58]. We cannot directly compare with Orion [85] as it is not publicly available, but we emulate its behavior where possible. Only Ceph provides availability via replicated object storage daemons (OSDs), delegating metadata management to a (potentially sharded) metadata server (MDS). Octopus and NFS do not support replication for availability and thus gain an unfair performance advantage over Assise. However, Assise beats them even while replicating for availability, showing that both features can be had when leveraging local NVM.

Other file systems do not support persistent caches and their consistency semantics are often weaker than Assise's. Assise provides data crash consistency, while both Ceph/BlueStore

and Octopus provide only metadata crash consistency [31]. For NFS, crash consistency is determined by the underlying file system. We use EXT4-DAX [9], which also provides only metadata crash consistency. When sharing data, NFS provides *close-to-open consistency* [39], while Octopus and Ceph provide "stronger consistency than NFS" [28], and Assise provides linearizability, which is stronger than Octopus' and Ceph's guarantees. In all performance comparisons, Assise provides stronger consistency than the alternatives. Ceph is the closest comparison point.

File system compliance tests. We tested Assise using xfstests [18] and CrashMonkey [65]. Assise passed all 75 generic xfstests that are recommended for NFS [16]. NFSv4.2 and Ceph v14.2.1 pass only 71 and 69 of these tests, respectively. In part, this is due to their weaker consistency model. Assise also successfully passes CrashMonkey tests, runs all existing Filebench profiles, passes all unit tests for the LevelDB key-value store, and passes MinuteSort validation.

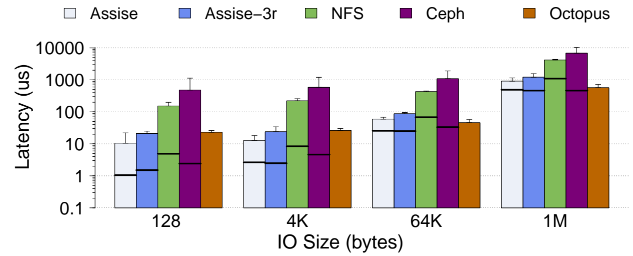
5.1 Experimental Configuration

Machines. Each experiment specifies the number (≥ 2) of testbed machines used. By default, machines are used as hot replicas in Assise, as a pool of storage nodes in Octopus, and as OSD and MDS replicas in Ceph. NFS uses only one machine as server, the rest as clients. We place applications on hot replicas for Assise, on OSD replicas for Ceph, on storage nodes for Octopus, and on clients for NFS. Assise's and Ceph's cluster managers run on 2 additional testbed machines (NFS and Octopus do not have cluster managers). The colocated deployment of applications and OSDs for Ceph is due to the small size of our cluster. It gives Ceph a potential performance advantage over an all-remote OSD deployment.

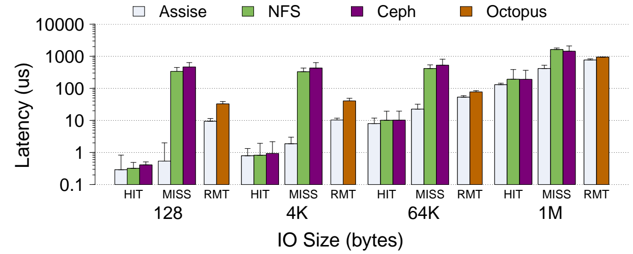
Network. We use RDMA for the NFS client-server connection. Ceph provides its client-side file system via the Ceph kernel driver and uses IP over InfiniBand, which was the fastest configuration (we also tried FUSE and Accelio [13]). Assise and Octopus use RDMA with kernel-bypass.

Storage and caches. For maximum efficiency, all file systems use NVM in App-Direct mode to provide persistence (cylinders in Figure 1) and DRAM when persistence is not needed (e.g., kernel buffer cache). We investigate Ceph and NFS performance using NVM in memory mode for volatile caches and find it to degrade throughput by up to 25% versus DRAM. For efficient access to NVM, Ceph OSDs use BlueStore and NFS servers use EXT4-DAX. Octopus uses FUSE to provide its file system interface to applications in *direct IO* mode to NVM, bypassing the kernel buffer cache [6].

To evaluate a breadth of cache behaviors with limited application data set sizes, we limit the fastest cache size for all file systems to 3GB. For Ceph and NFS, we limit the kernel buffer cache to 3GB. For Assise, we partition the LibFS cache into a 1GB NVM update log and a 2GB DRAM read cache (the SharedFS second-level cache may use all NVM available), and we run Assise in pessimistic mode.



(a) Sequential write. `write` latency is solid line, `fsync` is bar height.



(b) Read latencies for cache hits, misses, and remote (RMT) misses.

Figure 3: Avg. and 99%ile (error bar) IO latencies. Log scale.

5.2 Microbenchmarks

Average and tail write latency. We compare unladen synchronous write latencies with 2 machines (except Assise-3r which uses 3 machines). Synchronous writes involve `write` calls (fixed-width font identifies POSIX calls) that operate locally (except for Octopus where `write` may be remote), and `fsync` calls that involve remote nodes for replication (Assise, Ceph) and/or persistence (Ceph, NFS). Each experiment appends 1GB of data into a single file, and we report per-operation latency. In this case, the file size is smaller than each file system's cache size, so no evictions occur—with gigabytes of cache capacity, this is common for latency-sensitive write bursts.

Figure 3a shows the average and 99th percentile sequential write latencies over various common IO sizes (random write latencies are similar for all file systems). We break writes down into `write` (solid line) and `fsync` call latencies (bar). Octopus' `fsync` is a no-op. Assise's local write latencies match that of Strata [52]. Assise's average write latency for 128B two-node replicated writes is only 8% higher than the aggregate latencies of the required local and NVM-RDMA writes (cf. Table 1). Three replicas (Assise-3r) increase Assise's overhead to 2.2× due to chain-replication with sequential RPCs. The 99th percentile replicated write latency is up to 2.1× higher than the average for 2 replicas. This is due to Optane PMM write tail-latencies [45]. The tail difference diminishes to 19% for 3 replicas due to the higher average.

Ceph and NFS use the kernel buffer cache and interact at 4KB block granularity with servers. For small writes, the incurred network IO amplification during `fsync` is the main reason for up to an order of magnitude higher aggregate write latency than Assise. In this case, their `write` latency is up

to 3.2× higher than Assise due to kernel crossings and copy overheads. For large writes (≥ 64KB), network IO amplification diminishes but the memory copy required to maintain the kernel buffer cache becomes a major overhead. The latency of large `writes` is higher than Assise's replicated write latency (and up to 2.7× higher than Assise's non-replicated `write` latency), while aggregate write latency is up to 7.2× higher than Assise. Ceph has higher `fsync` latency than NFS due to replication.

Octopus eliminates the kernel buffer cache and block orientation, which improves its performance drastically versus NFS and Ceph. However, Octopus still treats all NVM as remote and uses FUSE for file IO. Octopus exhibits up to 2.1× higher latency than Assise for small (< 64KB) writes. This overhead stems from FUSE (around 10μs [78]) and writing to remote NVM via the network. Large writes (≥ 64KB) amortize Octopus' write overheads. Assise has up to 1.7× higher write latency due to replication. Octopus does not replicate.

Average and tail read latency. We compare unladen read latencies across different cache configurations. To do this, we read a 1GB file using various IO sizes, once with a warm cache (to report cache hits) and once with a cold cache (to report misses). The results are shown in Figure 3b. Assise has a second-layer cache in SharedFS before going remote, and we report three cases for Assise. Reads in Octopus are always remote.

We first compare cache-hit latencies (HIT), where Assise is up to 40% faster than NFS and 50% faster than Ceph. Assise serves data from the LibFS read cache, while NFS and Ceph use the kernel buffer cache. If Assise misses in the LibFS cache, data may be served from the local SharedFS (MISS). Assise-MISS incurs up to 3.2× higher latency than Assise-HIT due to reading the extent tree index, especially for larger IO sizes that read a greater number of extents. If Assise misses in both caches, it has to read from a remote replica (RMT). Assise-RMT incurs the latency of an RPC using RDMA. When NFS and Ceph miss in the cache, their clients have to fetch from remote servers, which incurs up to orders of magnitude higher average and tail latencies than Assise-RMT and Assise-MISS. Ceph performs worse than NFS due to a more complex OSD read path.

The elimination of a cache hurts Octopus' read performance, because it has to fetch metadata and data (serially) from remote NVM (RMT). Octopus' read latency is up to two orders of magnitude higher than the other file systems hitting in the cache, and up to an order of magnitude lower than NFS and Ceph missing in the cache. Octopus does not handle small (≤ 4KB) reads well due to FUSE overheads, with up to 3.54× Assise-RMT read latency. This overhead is amortized for larger reads (≥ 64KB), where Octopus incurs up to 1.46× the read latency of Assise-RMT. By configuring FUSE to use the kernel buffer cache for Octopus, we reduce Octopus' read hit latency to 1.8× that of Assise-HIT, with the remaining overhead due to FUSE. However, using the kernel buffer cache inflates write latencies for Octopus by up to an order of magnitude due to additional buffer cache memory copies.

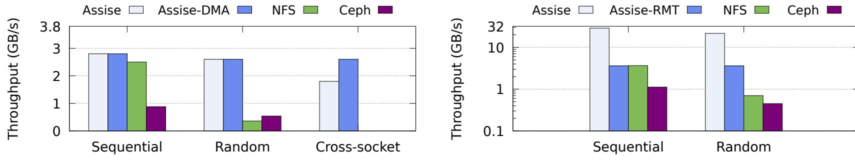
Peak throughput. Figure 4 shows average throughput of sequential and random IO to a 120GB dataset (on the local socket) with 4KB IO size from 24 threads (all cores of one socket). To evaluate a standard replication factor of 3, we use 3 machines for Assise and Ceph. The dataset is sharded over 24 files, and 5GB of data is written per thread. For random writes, a random offset is generated for every IO. `write` calls are not followed by `fsync` and the total amount of accessed data is larger than the cache size, causing cache eviction on write. The cache is initially cold so reads miss in the cache. For Assise, we show cache miss performance from a local and remote SharedFS. Octopus crashes during this experiment and is not shown.

For sequential writes, Assise and NFS achieve 74% and 66% of the NVM-RDMA bandwidth (cf. Table 1), respectively, due to protocol overhead for NFS and log header overhead for Assise. Chain-replication in Assise affects throughput only marginally. Ceph replicates in parallel to 2 remote replicas, consuming 3× the network bandwidth. This reduces its throughput to 31% of Assise and 35% of NFS. Assise achieves similar performance for sequential and random writes, as Assise's writes are log-structured. NFS and Ceph perform poorly for random writes due to cache block misprefetching incurring additional reads from remote servers, causing Assise to achieve 4.8× Ceph's throughput. NFS throughput is at only 67% that of Ceph, which is due to kernel locking overhead.

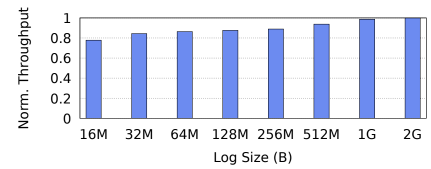
To quantify the benefit of bypassing hardware cache coherence for cross-socket writes with DMA, we repeat the benchmark, placing all files on the remote socket. We can see that Assise-DMA attains 44% higher cross-socket throughput than non-temporal processor writes (Assise). Sequential and random writes provide comparable performance. NVM-NUMA writes occur during eviction from the LibFS update log (local socket) to the NVM shared cache (remote socket). When writing to the local socket, Assise-DMA attains identical throughput to Assise, regardless of pattern.

For local sequential and random reads from the local SharedFS cache, Assise achieves 90% and 68%, respectively, of local, sequential NVM bandwidth. The 10% difference for sequential reads to local NVM bandwidth is due to metadata lookups, while random reads additionally suffer PMM buffer misses [45]. Assise remote reads (Assise-RMT) attain full NVM-RDMA bandwidth (3.8GB/s), regardless of access pattern. NFS and Ceph are limited by NVM-RDMA bandwidth for all reads and again have worse random read performance due to prefetching.

Log size sensitivity. To evaluate the impact of log size on write throughput, we conduct a sensitivity analysis. We run a single-process microbenchmark that writes a 1GB file sequentially at 4KB IO granularity. This experiment models a



(a) Write. 3.8GB/s is NVM-RDMA bandwidth.



(b) Read. 32GB/s is NVM read bandwidth.

Figure 4: Average throughput with 24 threads at 4KB IO size.

Figure 5: Worst-case throughput versus update log size, normalized to 2GB.

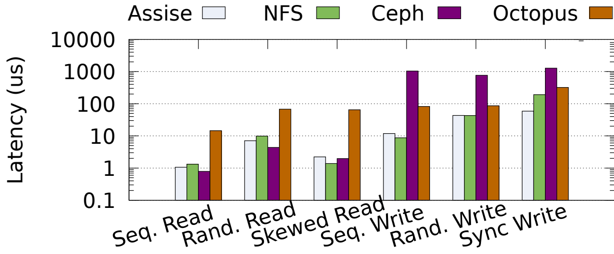


Figure 6: Average LevelDB benchmark latencies. Log scale.

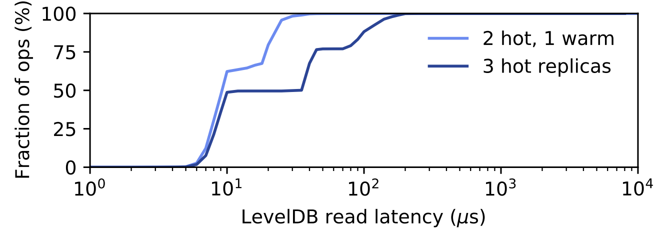


Figure 7: LevelDB random read latencies with warm replica.

worst case scenario. In the absence of sharing, processes can quickly fill up their allocated log space. Figure 5 shows the normalized write throughput at different log sizes. Throughput increases with log size, but the performance impact is small. Throughput increases by only 22% when using a 2GB log size versus a 16MB log size, a 128× increase in log size. For workloads that share data, we expect this gap to be smaller, as logs are evicted upon lease handoff. With 6TB of NVM per machine, Assise can scale to thousands of processes even with 2GB update logs. At 16MB, 100,000s of processes can be supported.

5.3 Application Benchmarks

We evaluate the performance of a number of common cloud applications, such as the LevelDB key-value store [33], the Fileserver and Varmail profiles of the Filebench [75] benchmarking suite, emulating file and mail servers, and the MinuteSort benchmark. We use 3 machines for LevelDB and Filebench and 5 machines for MinuteSort.

LevelDB. We run a number of single-threaded LevelDB latency benchmarks using LevelDB's `db_bench`, including sequential and random IO, skewed random reads with 1% of highly accessed keys, and sequential synchronous writes (`fsync` after each write). All benchmarks use a key size of 16B and a value size of 1KB with a working set of 1M KV pairs. Figure 6 presents the average measured operation latency, as reported by the benchmark.

Assise, Ceph, and NFS perform similarly for reads, where caching allows them to operate close to hardware speeds. For non-synchronous writes, NFS is up to 26% faster than Assise, as these go to its client kernel buffer cache in large batches (LevelDB has its own write buffer), while Assise is 69% faster than NFS for synchronous writes that cannot be buffered. Random IO and synchronous writes incur increasing LevelDB indexing overhead for all systems. Ceph performs worse than NFS for writes because it replicates (as does Assise) and Assise performs 22× better. Octopus bypasses the cache and thus performs worst for reads and better only than Ceph for writes, as it does not replicate.

Warm replica read latency. Warm replicas reduce read latency for warm data by allowing these reads to be served from remote NVM, rather than cold storage. For this benchmark, we configure Assise to limit the aggregate (LibFS and SharedFS) cache to 2GB and use the local SSD for cold storage. We then run the LevelDB random read experiment with a 3GB dataset. We repeat the experiment with two setups: (1) with 3 hot replicas and (2) with 2 hot and 1 warm replica. Figure 7 shows a CDF of read latencies. The benchmark accesses data uniformly at random, causing 33% of the reads to be warm. Consequently, at the 50th percentile, read latencies are similar for both configurations (served from cache). At the 66th percentile, reads in the first setup are served from SSD and have 2.2× higher latency than warm replica reads in the second setup. At the 90th percentile, the latency gap extends to 6×.

Filebench. Varmail and Fileserver operate on a working set of 10,000 files of 16KB and 128KB average size, respectively. Files grow via 16KB appends in both benchmarks (mail delivery in Varmail). Varmail reads entire files (mailbox reads) and Fileserver copies files. Varmail and Fileserver have write to read ratios of 1:1 and 2:1, respectively. Varmail leverages a write-ahead log with strict persistence semantics (`fsync` after log and mailbox writes), while Filebench consistency is relaxed (no `fsync`). Figure 8 shows average measured throughput of both benchmarks. Assise outperforms Octopus (the best alternative) by 6.7× for Fileserver and 5.1× for Varmail. Ceph performs worse than NFS for Varmail due to stricter persistence requiring it to replicate frequently and due to MDS contention, as Varmail is metadata intensive.

Optimistic crash consistency. We repeat this benchmark for Assise in optimistic mode (Assise-Opt) and change Varmail to use synchronous writes for the mailbox, but non-synchronous writes for the log. Prefix semantics allow Assise to buffer and

System	Processes	Partition [s]	Sort [s]	**Total [s]**	**GB/s**
Assise	160	20.3	43.0	**63.3**	**5.1**
	320	52.1	43.0	**95.1**	**6.7**
NFS	160	60.9	79.3	**140.2**	**2.3**
	320	104.1	84.2	**188.3**	**3.4**
DAX	320	–	44.1	–	–

Table 4: Average Tencent Sort duration breakdown.

coalesce the temporary log write without losing consistency. Assise-Opt achieves 2.1× higher throughput than Assise. Fileserver has few redundant writes and Assise-Opt is only 7% faster.

MinuteSort. We implement and evaluate Tencent Sort [46], the current winner of the MinuteSort external sorting competition [7]. Tencent Sort sorts a partitioned input dataset, stored on a number of cluster nodes, to a partitioned output dataset on the same nodes. It conducts a distributed sort consisting of 1) a range partition and 2) a mergesort (cf. MapReduce [32]). Step 1 presorts unsorted input files into ranges, stored in partitioned temporary files on destination machines. Step 2 reads these files, sorts their contents, and writes the output partitions. Each step uses one process per partition; the parallelism equals the number of partitions. A distributed file system stores the input, output, and temporary files, implicitly taking care of all network operations.

We benchmark the MinuteSort Indy category, which requires sorting a synthetic dataset of 100B records with 10B keys, distributed uniformly at random. Creating a 2GB input partition per process, we run 160 or 320 processes in parallel, uniformly distributed over 4 machines. MinuteSort does not require replication, so we turn it off. It calls `fsync` only once for each output partition, after the partition is written. We compare a version running a single Assise file system with one leveraging per-machine NFS mounts. For Assise, we configure the temporary and output directories to be colocated with the mergesort processes. We do the same for NFS, by exporting corresponding directories from each mergesort node. We conduct three runs of each configuration and report the average. We use the official competition tools [7] to generate and verify the input and output datasets. We use equal dataset sizes to compare Assise and NFS, rather than equal time. Table 4 shows that Assise sorts up to 2.2× faster than NFS. Running twice the number of processes only marginally improves performance, as Assise is bottlenecked by network bandwidth.

To show that Assise's POSIX implementation does not reduce performance, we modify the sort step to map all files into memory using EXT4-DAX and use processor loads and non-temporal stores to sort directly in NVM, rather than using file IO. We can see that the sort phase is 3% slower with DAX. libc buffers IO in DRAM to write 4KB at a time to NVM, performing better than direct, interleaved appends of 100B records.

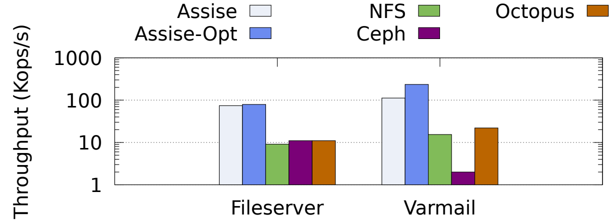


Figure 8: Avg. Varmail and Fileserver throughput. Log scale.

5.4 Availability

Ceph and Assise are fault tolerant. We evaluate how quickly these file systems return an application back to full performance after the fail-over and recovery situations of §3.4. To do so, we run LevelDB on the same dataset (§5.3) with a 1:1 read-write ratio and measure operation latency before, during, and after fail-over and recovery. We report average results over 5 benchmark runs.

Process fail-over. For this benchmark, we simply kill LevelDB. In this case, the failure is immediately detected by the local OS and LevelDB is restarted. Ceph can reuse the shared kernel buffer cache in DRAM, resulting in LevelDB restoring its database after 1.63s and returning to full performance after an additional 2.15s, for an aggregate 3.78s fail-over duration. With Assise, the DB is restored in 0.71s, including recovery of the log of the failed process and reacquisition of all leases. Full-performance operations occur after an additional 0.16s, for an aggregate 0.87s fail-over time. Assise recovers this case 4.34× faster than Ceph, showing that process-local caches do not impede fast recovery.

OS fail-over. NVM's performance allows for instant local recovery of an OS failure, rather than requiring a backup replica. To demonstrate, we run the primary in a virtual machine (VM). We kill the primary VM, then immediately start a new VM from a snapshot stored in NVM. The snapshot starts in 1.66s. We restart SharedFS within the new VM, which recovers the file system within 0.23s. Finally, as in the process fail-over experiment, LevelDB is restarted and resumes database operations after another 0.68s. The aggregate fail-over time is 2.57s, 40× faster than Ceph's fail-over to a backup replica (evaluated next).

Fail-over to hot backup. All following experiments use 2 machines (primary and backup). The LevelDB client processes poll the file system's cluster manager for membership state, using a standard primary-backup ZooKeeper design pattern for node fail-over [47]. LevelDB initially runs on the primary, where we inject failures. Failures are detected by LevelDB clients using a 1s heartbeat timeout via the cluster manager. Once a node failure is detected, LevelDB immediately restarts on the backup.

A time series of measured LevelDB operation latencies during one experiment run is shown in Figure 9. Pre-failure, we see bursts of low latency in between stretches of higher latency. This is LevelDB's steady-state. Bursts show LevelDB

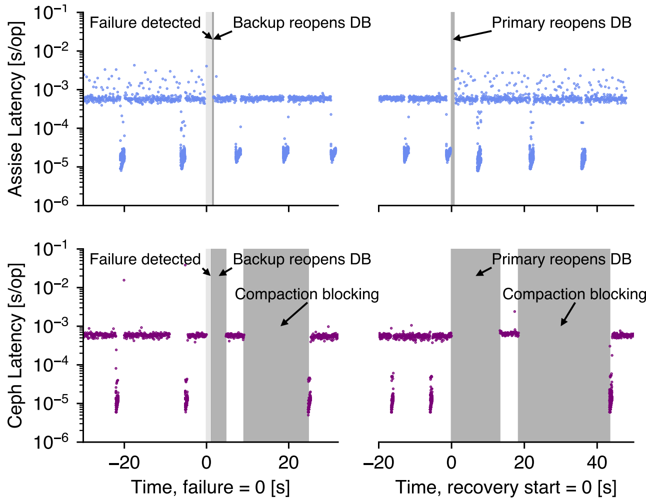


Figure 9: LevelDB operation latency time series during fail-over and recovery. Log scale.

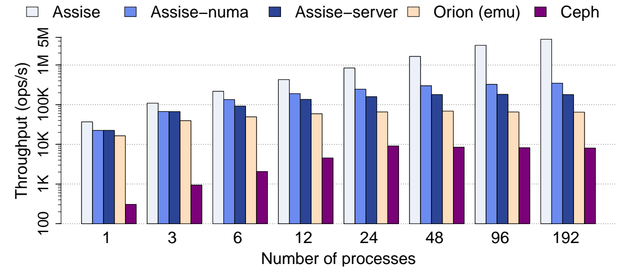


Figure 10: Scalability of atomic 4KB file operations. Log scale.

writes to its own DRAM log. These are periodically merged with files when the DRAM log is full, causing writes that are higher latency (and sometimes blocking with Ceph), as the writes wait on the log to become available.

We inject a primary failure by killing the primary's file system daemon (SharedFS for Assise and OSD for Ceph) and LevelDB. During primary failure, no operations are executed. It takes 1s to detect the failure and restart LevelDB on the backup (light shaded box). Due to unclean shutdown, LevelDB first checks its dataset for integrity before executing further operations (dark shaded box). For failover, Assise need only evict the per-process log (up to 1GB) on the backup hot replica, making fail-over near-instantaneous. LevelDB returns to full performance in both latency and throughput 230ms after failure detection. Ceph takes 3.7s after failure detection to return to full performance. However, LevelDB stalls soon thereafter upon compaction (further dark shaded box), which involves access to further files, resulting in an additional 15.6s delay, before reaching steady-state. Ceph's long aggregate fail-over time of 23.7s is due to Ceph losing its DRAM cache, which it rebuilds during LevelDB restart. Assise reaches full performance after failure detection 103× faster than Ceph. LevelDB performs better on the backup, as neither file system has to replicate.

Primary recovery. After 30s, we restart the file system daemons on the primary, emulating the time for a machine reboot from NVM. During this time, many file system operations occur on the backup that need to be replayed on the primary. As soon as the primary is back online, we cleanly close the database on the backup and restart on the primary. Both Assise and Ceph allow applications to operate during primary recovery, but performance is affected. Assise detects outdated files via epochs and reads their contents from the remote hot replica upon access. Once read, the local copy is updated, causing future reads to be local. LevelDB returns to full performance 938ms after restarting it on the recovering primary. Ceph also rebuilds the local OSD, but eagerly. Ceph takes 13.2s before LevelDB serves its first operation due to contention with OSD recovery and suffers another delay of 24.9s on first compaction, reaching full performance 43.4s after recovery start. Assise recovers to full performance 46× faster than Ceph.

Fail-over to cold backup. We measure cascaded LevelDB fail-over time to an Assise replica with a cold cache. LevelDB serves its first request on the cold backup 303ms after failure detection, but with higher latency due to SSD reads. LevelDB returns to full performance after another 2.5s. At this point, the entire dataset has migrated back to cache.

5.5 Scalability

We evaluate Assise's scalability via 1) sharded file operations under increasing load and increasingly localized lease management, and 2) parallel email delivery in Postfix [79].

5.5.1 Sharded File Operations

Processes in parallel create, write, and rename 4KB files with random data in private directories. This benchmark uses 3 machines (6 sockets) and can scale throughput linearly with the number of processes. To eliminate network bottlenecks to scalability, we turn replication off.[1] Figure 10 presents average throughput over 5 runs of an increasing number of processes, each operating on 480K files, balanced over processor sockets. Ceph uses 3 sharded MDSes (1 per machine). However, MDS sharding has negligible impact on Ceph's performance.

Ceph's remote MDSes have high overhead for atomic operations, as each client has to communicate with remote MDSes. This design prevents scalability beyond 8Kops/s. We emulate Orion by restricting CC-NVM to use a single SharedFS lease manager. In this case, data is stored on local NVM, but atomic operations still use a remote lease manager. Orion has RDMA mechanisms that simplify communicating with its MDS, but these mechanisms cannot be used for operations that affect multiple inodes (e.g., renames). Orion and Assise both use RDMA RPCs. While Orion operates in the kernel, our emulation uses user-level RDMA, which is light-weight, and Orion (emu) outperforms Ceph by 8×.

[1] Due to the small size of our cluster, primary nodes would replicate to each other and scalability would be limited by per-node link bandwidth. A larger cluster would replicate to independent machines for each primary.

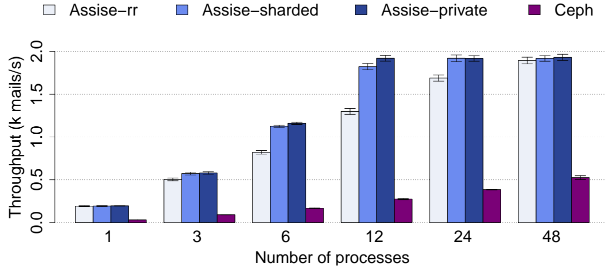


Figure 11: Postfix mail delivery throughput scalability.

To break down the benefit of local lease management in Assise, we progressively shard it, first by server (Assise-server), then by socket (Assise-numa), and finally by process (Assise). Assise-server outperforms Orion (emu) by 2.77× and Assise-numa improves throughput by another 1.93×. Assise scales linearly with the number of processes until it hits NVM write bandwidth, improving throughput by another 12.86×. Assise outperforms Orion by 69× and Ceph by 554× at scale.

5.5.2 Postfix

We use the unmodified Postfix mail server to measure the performance of parallel mail delivery. A load balancer machine forwards incoming email from as many client machines as necessary to maximize throughput to Postfix queue daemons running on 3 testbed machines, configured as replicas. On each Postfix machine, a pool of delivery processes pull email from the machine-local incoming mail queue and deliver it to user Maildir directories on a cluster-shared distributed file system. To ensure atomic mail delivery, a Postfix delivery process writes each incoming email to a new file in a process-private directory and then renames this file to the recipient's Maildir.

We send 80K emails from the Enron dataset [51], with each email reaching an average of 4.5 recipients. This results in a total of 360K email deliveries. Each email has an average size of 200KB (including attachments) and the dataset occupies 70GB. We repeat each experiment 3 times and report average mail delivery throughput and standard deviation (error bars) in Figure 11 over an increasing number of delivery processes, balanced over machines. We compare various Assise configurations and Ceph with 2 MDSes (1 and 3 MDSes performed similarly).

Round-robin. In the first configuration (Assise-rr) the load balancer uses a round-robin policy to send emails to mail queues. Due to a lack of locality, mails delivered to the same Maildir often require synchronization across machines, causing CC-NVM to frequently delegate leases remotely, which increases delivery latencies. Despite this, Assise-rr is able to outperform Ceph by up to 5.6× at scale. Ceph cannot improve throughput much further—even with 300 delivery processes, its throughput improves by 8% versus 48 processes.

Sharded. We shard Maildirs by Enron suborganization over machines [26]. The load balancer is configured to prefer the recipient's shard. For mail messages with multiple recipients, it picks the shard with the most receivers. In case of mail queue overload, the load balancer sends mail to a random unloaded shard. Sharding users in this manner provides up to 20% better performance (Assise-sharded) due to the fact that repeated deliveries to users of the same clique are likely to occur on the same server, allowing CC-NVM to synchronize delivery locally. At 15 processes, we are network-bound due to replication. Sharding did not improve Ceph's performance.

Private directories. We shard Maildirs by delivery process, using process IDs for Maildir subdirectories, thereby eliminating the need for synchronization (Assise-private). This change is not backward compatible with existing mail readers, but it is the logical limit for sharding-based optimization. Assise-private scales linearly until it is bottlenecked by network bandwidth, but performance is similar to Assise-sharded. This shows that local synchronization in Assise has minimal overhead. Ceph performance continues to be gated by the MDS.

Summary. Our results show that, with careful sharding of the workload, Assise's hierarchical coherence allows LibFS processes to synchronize deliveries locally, providing almost the same performance and scalability as private directories.

6 Conclusion

Assise is a distributed file system that provides low tail latency, high throughput, scalability, and high availability with a strong consistency model. To take advantage of low-latency NVM, Assise demonstrates that filesystem metadata and data should be colocated with applications. Colocation not only enables high performance, but also fast recovery. Assise proposes a novel, crash-consistent cache coherence protocol that can leverage the performance of NVM, while providing linearizability. Assise uses hot replicas in NVM to minimize application recovery time and ensure data availability, while leveraging a crash-consistent file system cache-coherence layer (CC-NVM) to provide scalability. In comparing with several state-of-the-art file systems, our results show that Assise improves write latency up to 22×, throughput up to 56×, fail-over time up to 103×, and scalability up to 6× versus Ceph, while providing stronger consistency semantics.

Assise is available at `https://github.com/ut-osa/assise`.

Acknowledgments. Waleed Reda was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC), funded by the European Commission (EACEA) (FPA 2012-0030). This work is supported in part by ERC grant 770889, NSF grant CNS-1900457, and the Texas Systems Research Consortium. This work is also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2020R1C1C1014940). We thank Intel for access to the evaluation testbed. We thank the anonymous reviewers and our shepherd, Kim Keeton, for their comments and feedback.

References

[1] Amazon Elastic Block Store (EBS). https://aws.amazon.com/ebs/.

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

[3] Amazon S3. https://aws.amazon.com/s3/.

[4] Apache Crail. http://crail.apache.org/.

[5] Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[6] Octopus - github repository. https://github.com/thustorage/octopus.

[7] Sort benchmark home page. http://sortbenchmark.org/.

[8] syscall_intercept. https://github.com/pmem/syscall_intercept.

[9] Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, Sept. 2014.

[10] Apache ZooKeeper. https://zookeeper.apache.org, Aug. 2017.

[11] Persistent memory programming, Aug. 2017. http://pmem.io/.

[12] The Sprite Operating System. https://www2.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html, Aug. 2017.

[13] Accelio, Aug. 2018. https://github.com/accelio/accelio.

[14] Intel Optane DC persistent memory, Mar. 2019. http://www.intel.com/optanedcpersistentmemory.

[15] Intel SSD DC P4610 1.6TB, Apr. 2019. Google Shopping search. Lowest non-discount price.

[16] NFS - Xfstests, 2019. http://wiki.linux-nfs.org/wiki/index.php?title=Xfstests&oldid=5652.

[17] NVM Express over Fabrics 1.1, 2019. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf.

[18] Xfstests, 2019. https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/.

[19] DDR4-3200 DRAM ECC Registered 128GB, Oct. 2020. Google Shopping search. Lowest non-discount price.

[20] Intel Optane DC Persistent Memory Module 128GB, Oct. 2020. Google Shopping search. Lowest non-discount price.

[21] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis. File systems unfit as distributed storage backends: Lessons from 10 years of Ceph evolution. In *27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[22] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, 2008.

[23] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 109–126, 1995.

[24] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, 2009.

[25] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. Engbersen. Failure analysis of virtual and physical machines: Patterns, causes and characteristics. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 1–12, 2014.

[26] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. In *Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 178–179, 1981.

[27] M. Burrows. *Efficient data sharing*. PhD thesis, University of Cambridge, UK, 1988.

[28] Ceph Documentation. Differences from POSIX. http://docs.ceph.com/docs/master/cephfs/posix/.

[29] Y. Chen, Y. Lu, and J. Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *14th EuroSys Conference 2019*, pages 1–14, 2019.

[30] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, 2013.

[31] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *10th USENIX Conference on File and Storage Technologies*, FAST'12, 2012.

[32] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design and Implementation*, OSDI'04, 2004.

[33] J. Dean and S. Ghemawat. LevelDB: A Fast Persistent Key-Value Store. https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html, 2011.

[34] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen. Performance and protection in the ZoFS user-space NVM file system. In *27th ACM Symposium on Operating Systems Principles*, pages 478–493, 2019.

[35] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[36] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 61–74, 2010.

[37] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, 2003.

[38] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210, 1989.

[39] T. Haynes and D. Noveck. Network file system (NFS) version 4 protocol, Mar. 2015. https://tools.ietf.org/html/rfc7530.

[40] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th edition, 2017.

[41] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter 1994 Technical Conference*, WTEC'94, 1994.

[42] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. West. Scale and performance in a distributed file system. *SIGOPS Oper. Syst. Rev.*, 21(5):1–2, Nov. 1987.

[43] InsideHPC. Intel Optane DC persistent memory comes to Oracle Exadata X8M, Sept. 2019. https://insidehpc.com/2019/09/intel-optane-dc-persistent-memory-comes-to-oracle-exadata-x8m/.

[44] N. S. Islam, M. Wasi-ur Rahman, X. Lu, and D. K. Panda. High performance design for HDFS with byte-addressability of NVM and RDMA. In *2016 International Conference on Supercomputing*, ICS '16, pages 8:1–8:14, 2016.

[45] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the Intel Optane DC Persistent Memory Module, Apr. 2019. https://arxiv.org/abs/1903.05714v2.

[46] J. Jiang, L. Zheng, J. Pu, X. Cheng, C. Zhao, M. R. Nutter, and J. D. Schaub. Tencent sort. Technical report, Tencent Corporation, 2016. http://sortbenchmark.org/TencentSort2016.pdf.

[47] F. Junqueira and B. Reed. *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, Inc., 1st edition, 2013.

[48] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, 2019.

[49] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 44(4):295–306, 2015.

[50] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: Group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 297–312, 2018.

[51] B. Klimt and Y. Yang. The Enron corpus: A new dataset for email classification research. In *European Conference on Machine Learning*, pages 217–226. Springer, 2004.

[52] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A cross media file system. In *26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, 2017.

[53] T. Le, J. Stern, and S. Briscoe. Fast memcpy with SPDK and Intel I/OAT DMA engine, Apr. 2017. https://software.intel.com/en-us/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.

[54] S. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. RECIPE: Reusing concurrent in-memory indexes for persistent memory. In *27th ACM Symposium on Operating Systems Principles*, 2019.

[55] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, 2014.

[56] H. Li, Y. Zhang, D. Li, Z. Zhang, S. Liu, P. Huang, Z. Qin, K. Chen, and Y. Xiong. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 15:1–15:17, 2019.

[57] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[58] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: An RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 773–785, 2017.

[59] P. MacArthur and R. D. Russell. A performance study to guide RDMA programming decisions. In *IEEE 14th International Conference on High Performance Computing and Communication and IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 778–785, 2012.

[60] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Linux Symposium*, volume 2, June 2007.

[61] Mellanox. Mellanox Introduces Revolutionary DPU based SmartNICs for Making Secure Cloud Possible, 2019. https://blog.mellanox.com/2019/08/mellanox-introduces-revolutionary-smartnics-for-making-secure-cloud-possible/.

[62] C. Metz. The epic story of Dropbox's exodus from the Amazon cloud empire, Mar. 2016. https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/.

[63] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pages 257–273, 2014.

[64] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[65] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram. Crashmonkey and ACE: Systematically testing file-system crash consistency. *ACM Trans. Storage*, 15(2), Apr. 2019.

[66] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[67] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 433–448, 2014.

[68] H. Qiu, X. Wang, T. Jin, Z. Qian, B. Ye, B. Tang, W. Li, and S. Lu. Toward effective and fair RDMA resource sharing. In *2nd Asia-Pacific Workshop on Networking*, pages 8–14, 2018.

[69] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *26th ACM Symposium on Operating Systems Principles*, SOSP '17, 2017.

[70] A. Rosenbaum and A. Margolin. Dynamically-Connected Transport, 2018. Talk. 14th Annual Open Fabrics Alliance Workshop.

[71] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 69–87, 2018.

[72] SNIA. *NVM Programming Model (NPM) Version 1.2*, June 2017.

[73] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, July 1981.

[74] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and atomic RDMA-based replication. In *2018 USENIX Annual Technical Conference*, USENIX ATC '18, pages 851–863, 2018.

[75] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX ;login:*, 41(1), 2016.

[76] S.-Y. Tsai, Y. Shan, and Y. Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference*, USENIX ATC '20, pages 33–48, 2020.

[77] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *6th Symposium on Operating Systems Design and Implementation*, OSDI'04, 2004.

[78] B. K. R. Vangoor, V. Tarasov, and E. Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies*, FAST'17, pages 59–72, 2017.

[79] W. Venema. Postfix project. http://www.postfix.org/.

[80] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus scalable block store. In *10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 357–370, 2013.

[81] S. Watanabe. *Solaris 10 ZFS Essentials*. Prentice Hall Press, USA, 1st edition, 2010.

[82] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, 2006.

[83] J. Xu, J. Kim, A. Memaripour, and S. Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 427–439, 2019.

[84] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies*, FAST '16, pages 323–338, 2016.

[85] J. Yang, J. Izraelevitz, and S. Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *17th USENIX Conference on File and Storage Technologies*, FAST '19, pages 221–234, 2019.

[86] ZDNet. Google cloud taps new Intel memory module for SAP HANA workloads, July 2018. https://www.zdnet.com/article/google-cloud-taps-new-intel-memory-module-for-sap-hana-workloads/.

[87] ZDNet. Baidu swaps DRAM for Optane to power in-memory database, Aug. 2019. https://www.zdnet.com/article/baidu-swaps-dram-for-optane-to-power-in-memory-database/.

[88] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 461–476, 2018.