File Systems are not Enough: Rethinking the Storage API for Microsecond-Scale Cloud Applications

Ashlie Martinez Katie Lim Pratyush Patel Irene Zhang Dan Ports
Jacob Nelson Thomas Anderson

Abstract

Cloud applications struggle to efficiently and correctly persist state required for fault tolerance. This paper argues that traditional file systems, which many microsecond-scale cloud applications directly build upon, do not provide the right abstractions to meet application needs of fast persistence and fast recovery. We propose raising the semantic level of the storage interface exposed to cloud applications from a file-centric one to one centered around a coordinated persistent log and transactional key-value store. In doing so, we simultaneously address the crash consistency problems applications face and hide internal storage library details, enabling portability across diverse storage interfaces. We show that a modified version of Redis using our new abstraction is able to reduce latency up to 30% and improve throughput by 42% when using SPDK.

1 Introduction

A growing class of cloud applications are designed to accept client requests, modify internal state, and return a response to the client. These applications must provide both fault tolerance, ensuring that data is preserved when a machine crashes or the application stops running, and high availability, that recovery is rapid after a crash. To accomplish these goals, applications write state modifications to persistent storage and checkpoint internal state periodically. For performance, applications are often designed to serve reads directly from in-memory data, persisting state updates only for recovery purposes. We call these persistent in-memory applications (PIMAs).

While slow storage operations have traditionally led PIMAs to batch storage updates, recent advances in storage technology make fine-grained persistence more feasible. New 3D XPoint drives are capable of performing operations in as little as 7µs and recently released PCIe 4.0 SSDs are capable of 7 GB/s sequential reads and writes [31, 38]. Other storage technologies like non-volatile memory (NVM) also exist but are not yet widely available. In this paper, we restrict ourselves to block-based storage devices.

A storage backend, such as a POSIX-based file system, often sits between cloud applications and storage devices. Originally designed as a human-compatible interface to the storage layer, file systems are a poor fit for modern cloud applications as they have high overhead relative to storage devices [33]; provide only a generic, low-level API that does not match application needs [34]; do not provide the crash consistency guarantees applications need; and have an API incompatible with newer and faster storage backends such as SPDK [8] and io_uring [18]. For PIMAs, the result is often poor performance coupled with complex and potentially buggy persistence code.

Despite the drawbacks, cloud applications still overwhelmingly use POSIX-based file systems to persist their data. One reason is that there is no single, widely used, "next generation" storage backend that can serve as the universal standard going forward. New storage backends like SPDK and io_uring have different APIs from each other, and their APIs are even lower level than POSIX, requiring even more complex reasoning by application developers in order to adapt to them. If a different storage backend becomes popular, applications would require further changes to support it.

Instead, we propose a replacement for POSIX that provides low overhead, crash consistent persistence along with support for multiple storage backends. We accomplish this by raising the semantic level of the API from a file-centric one to one that has an application-accessible log and a transactional key-value store (kv-store). A log and kv-store are appropriate for PIMAs because the log can quickly absorb application updates on the critical path and the kv-store provides a consistent snapshot mechanism to speed recovery after a failure. Additionally, a log allows PIMAs to easily persist information even when log entries do not have a one-to-one correspondence with kv-store updates. This is common in distributed applications that use the log for both application data and consensus state.

PIMAs care about their data being persisted in a crash consistent way, not the exact format used for persistent data. Based on this observation, we move the semantic level of the API up to something that better suits application needs and is independent of underlying storage backends. In doing so, we hide implementation details of the storage stack from applications, giving us leeway to

implement optimizations and interface with new storage backends. Finally, by making crash consistency of the log and kv-store part of the API guarantee, we free applications of the need to implement complex crash consistency logic of their own.

FASL (Fast Application Storage Layer), our prototype implementation of this new API, includes a crash consistent application log and a copy-on-write transactional kv-store. FASL currently supports the POSIX, SPDK, and io_uring storage backends.

We show how applications can use FASL to quickly log data on the critical path while moving data from the log to the kv-store in the background. Our benchmarks show that even highly optimized applications like Redis gain latency improvements of up to 30% and throughput improvements of 42% when using FASL with newer storage backends.

In summary, this paper:

- defines a new persistence API for PIMAs
- presents FASL, an implementation of the API that abstracts over POSIX, SPDK, and io_uring
- shows FASL can provide performance benefits to even heavily optimized applications like Redis.

2 Background/Motivation

The persistence problems that PIMAs face are not confined to any single layer of the storage stack or any single design choice. At a high level, PIMAs require a fast, crash consistent persistence mechanism on the critical path and a low-overhead way to recover after a crash. Such requirements necessitate the use of multiple data structures for persistence: a log and a snapshot. However, in choosing to split the persisted application state, PIMAs create serious data consistency issues for themselves which are further exacerbated by the weak crash consistency guarantees provided by POSIX-based file systems.

2.1 Application Overview

To illustrate these issues, we use Redis [7], a widely-used kv-store, as an example. Like many PIMAs, Redis uses a POSIX-based file system as the storage backend. We discuss, at a high level, what Redis does when serving client requests and how Redis initializes its state at startup. Section 2.3 takes a closer look at how Redis and other applications persist data for crash consistency.

Serving requests. Figure 1 shows the different operations that occur during a single iteration of the event loop in Redis. In a single iteration, all requests that have been received are processed, and responses are returned to clients for requests that require no further processing. Read requests for PIMAs do not require any storage I/O, so we omit them from this discussion.

For every request that is pending in the event loop, Redis updates the in-memory state of the database (②)

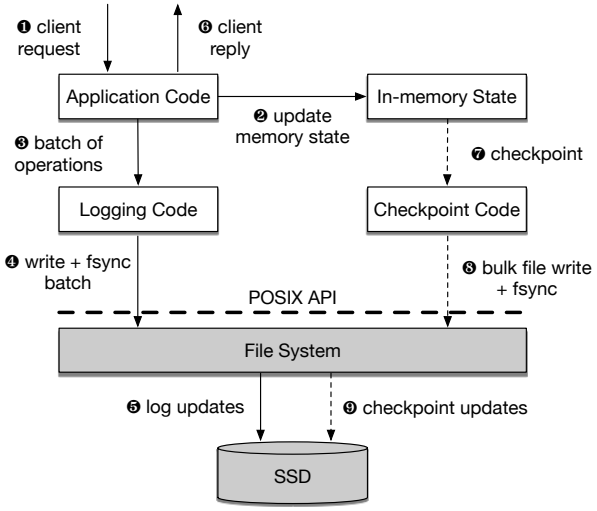


Figure 1: The different steps a PIMA must perform for write operations. Solid arrows are executed for every operation the application receives while dashed arrows represent work that is only periodically executed. Shaded boxes show components outside the application's control.

and appends the request body to an in-memory buffer for the log (③). Once all requests for that iteration have been processed, Redis appends the log buffer containing all the processed requests to the persistent log file on disk (④). After the log has been synced to disk in ⑤, Redis sends replies to clients (⑥).

When the log file grows large enough, Redis forks off a background process to dump the current in-memory database to disk as a snapshot in a new file (⑦- ⑨). While the new snapshot is being created, incoming requests are logged to the existing log file and the background process is notified of the requests. In this period, in-memory updates for requests use copy-on-write to ensure the snapshot thread has a consistent view of the in-memory database.

When the snapshot is completed, all updates the background process was notified of are added to a new log file. The background process then exits, and the main process appends any remaining requests to the new log file and replaces the old snapshot and log with the new ones.

Crash recovery. Initializing Redis after a crash requires reading the snapshot and log to rebuild the in-memory application state. The snapshot is first read back into memory to initialize the base application state. Next, the log is read sequentially, and each operation is applied to the current in-memory state. If an invalid operation is encountered, Redis exits with an error by default.

In general, an application can handle an invalid log entry if it can ensure that no response was returned for that entry or any entries that follow it in the log. In those instances, the remainder of the log is truncated, and the application begins serving client requests with the state

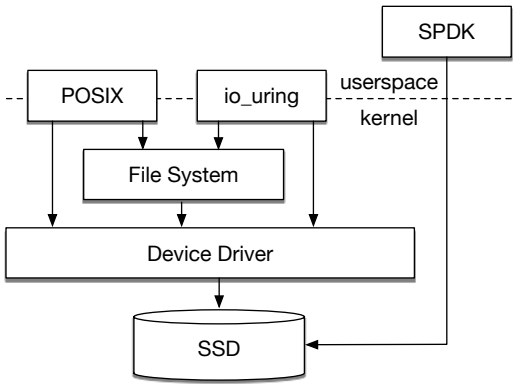


Figure 2: Storage backends present an uneven layer of APIs for applications to program against. Some storage backends, like SPDK, also do not include common features like access to the page cache or file system, requiring developers to write more code to use them.

generated from the snapshot and prefix of the log.

Distributed PIMAs. Distributed applications like ZooKeeper and etcd [1, 2] use a log and snapshot similar to Redis. The only difference is that the log contains consensus operations instead of client requests. Because these consensus operations include information required to execute client operations, the application is still able to reconstruct its state from the log and snapshot. However, recovery is more complex because multiple log entries may be required for each client request to determine if the operation was committed or aborted by the consensus protocol. An important consequence is that log operations may not directly correspond to updates in the snapshot.

Storage backends. Figure 2 illustrates the different APIs presented by storage backends. For example, SPDK, unlike POSIX, provides kernel-bypass access to storage devices. However, it does not include a file system nor accesses the kernel page cache. Therefore, applications that use SPDK must design and implement their own on-disk data structures and manage I/O when multiple reads or writes are issued to the same block on disk.[1]

io_uring uses shared ring buffers to provide asynchronous, zero-copy communication between the kernel and userspace for file operations. Although it uses the page cache and file system, it requires crash consistent applications to implement their own storage I/O manager because completions are not ordered.

2.2 Why a Log and Snapshot?

Even though using a log and a snapshot complicates application logic, it allows PIMAs to obtain both fast persistence on the critical path and fast recovery times. For fault tolerance, operations that update the application state

[1]The NVMe spec [10] allows storage devices to execute operations in the submission queue out of order, so users must ensure that multiple operations targeted at the same block are not present in the submission queue at the same time.

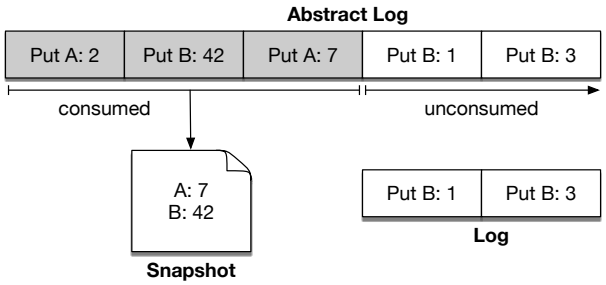


Figure 3: Relationship between the abstract log of all operations ever executed by a PIMA and the concrete snapshot and log that encode them. Shaded entries in the abstract log represent data present in the snapshot but not the concrete log. Operations that modify the same part of application state may be duplicated in the concrete log, but not in the snapshot. If operations are not idempotent, the concrete log and the snapshot must contain disjoint subsets of the abstract log.

must be persisted on the critical path before responding to clients. This makes a write-optimized structure like an append-only log attractive. However, during recovery, the entire application state must be restored. Because the recovery time and storage space of a log scale linearly with the number of operations executed by the PIMA, applications use a snapshot to represent state more compactly and decrease recovery time.

Figure 3 shows how the snapshot and log work together to compactly represent the application state, shown as the abstract log of all operations the application has executed. Importantly, the snapshot only needs a single data point to represent a series of updates to a piece of state (key A in the figure), allowing it to reduce the amount of data that must be read, deserialized, and applied to the in-memory state during recovery.

The catch is that the application must ensure the log and snapshot are consistent *with each other* [13]. If the snapshot represents a compacted prefix of the abstract log, then the persisted log must contain all operations in the abstract log spanning from the one immediately following the snapshot's prefix to the last operation the application executed. If the log contains operations that occurred before the final operation represented in the snapshot and operations are not idempotent, then that portion of the log must be skipped during recovery. If operations are idempotent, then it is safe for the application to replay the portion of the persisted log that overlaps with the snapshot's data.

2.3 File Systems are Not Enough

When it comes to implementing a coordinated, crash consistent log and snapshot, POSIX-based file systems present real problems for PIMAs in terms of crash consistency and portability. Together, these shortcomings often result in application developers creating their own storage stack, which is often tightly coupled with application logic.

Section 2.1 already addressed the problem of portability across storage backends, so here we restrict ourselves to crash consistency.

PIMAs struggle with two different high-level operations related to crash consistency: updating information in the state snapshot atomically and maintaining consistency between the snapshot and the log. These operations translate into atomic updates to parts of a single file and atomic updates to multiple files, neither of which POSIX-based file systems support.

Single file consistency. As an API, POSIX defines some crash consistency operations, but relies on underlying file systems to provide crash consistency guarantees. Unfortunately file systems offer only weak guarantees and, in practice, those guarantees differ across various POSIX-based file systems [11,34]. Therefore, the only crash consistency guarantees developers may assume are that appends to a file are atomic (atomic append)[2] and renaming a file over an existing file in the same directory is atomic (atomic replace-via-rename).

While atomic appends can be used to grow the application's log files in a crash consistent manner, POSIX provides no way to do incremental updates to the application snapshot. As a result, many applications dump the entire in-memory state to a new file when making a new snapshot instead of just applying the changes specified in the log file to the existing snapshot. This transforms the crash consistency problem into a performance and bandwidth problem and leads to application-level write amplification. A few applications, like etcd, perform incremental updates to persistent application state, but they rely on a transactional storage library that runs on top of POSIX to do so. For example, etcd uses a fork of boltdb [3], a transactional kv-store. However, these applications still require a custom crash consistent log implementation.

Cross-file consistency. POSIX provides no mechanism for applications to atomically update multiple files at the same time, forcing applications to develop their own solutions to ensure the log and snapshot are consistent with each other. Redis works around this by storing the log and snapshot in the same file, ensuring that if one is present, so is the other. ZooKeeper stores logs and snapshots separately and ensures that only log files with operations before the snapshot was taken are garbage collected. etcd uses a transactional kv-store to store the location of the most recently executed log operation before garbage collecting the operations in the log applied to the snapshot.

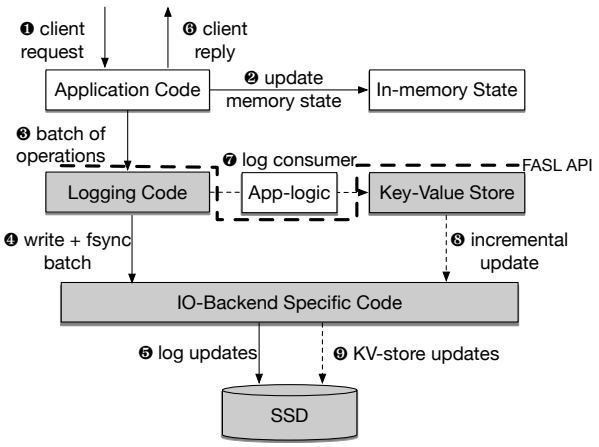


Figure 4: PIMA architecture when using the FASL API. The persistent log acts like a persistent producer-consumer queue connecting application logic to the incrementally updated state snapshot. Dashed arrows show code executed in the background and shaded boxes denote elements outside the application's control.

3 A Unified, Persistent Log Abstraction

We design the FASL API to meet the following goals: (1) provide microsecond-scale persistence, (2) simplify the persistence mechanism applications use, and (3) allow the use of different storage backends without further application modifications.

Non-goals for the FASL API are providing a thread-safe log and kv-store and creating a persistence abstraction that understands application data. Thread-safety is not addressed in this project because POSIX's lack of concurrent access guarantees already forces applications to coordinate access to storage. We do not propose an abstraction that understands application data because that would limit the generality of the API, either making it unsuitable for some PIMAs or requiring extra specialized logic in our implementation.

System and Data Models. The FASL API targets a specific set of applications: those that cache all their state in memory for performance and rely on block-based persistent storage for fault tolerance. It does not place any restrictions on the type of data that applications can store, but the application must be able to interpret the data to move it from the log to the kv-store. The FASL API does not change the requirement that applications must be able to construct the current state of the system by replaying log data on top of some (possibly empty) checkpoint; this means the application must deterministically update state based on information in the log.

[2]Appends may be atomic, but not all file systems guarantee *prefix appends*, further complicating matters. Bornholt et al. use litmus tests to find and test which basic file system guarantees developers can expect [11].

3.1 Abstraction Interface

The FASL API is made up of two parts: a log and a kv-store. Both have relatively simple interfaces that allow applications to interact with them to read and write data. Like existing custom persistence solutions, the API takes advantage of the natural split in applications that separates data quickly persisted on the critical path from data that is stored en masse as an application checkpoint. However, as shown in Figure 4, the log also acts as a producer-consumer queue that connects the portion of the application that communicates with other machines over the network to the local persistent data store. The FASL API is built on the idea of making constant, incremental updates to the application checkpoint instead of occasionally dumping the entire application state to disk as a snapshot. The full API is show in Table 1.

3.1.1 Log Interface

The log acts like a persistent FIFO queue, allowing applications to quickly persist data before moving the data to a long-term store. We design the log interface such that it meets applications' persistence needs while providing a flexible interface and taking advantage of the way applications normally use a log for recovery.

The log provides applications with the following guarantees for data placed in it: (1) appended entries are guaranteed to be recoverable once `LogWrite()` returns; (2) if an entry is read from the log, then it is guaranteed to be durable and complete; and (3) the log will always return a (possibly complete) prefix of the entries currently in it.

By returning only persisted log entries, the log ensures the application can recover any changes made before a crash that were dependent on log data. Entries read from the log are also guaranteed to be complete, meaning no data in the log has been lost due to a crash. Finally, the log will stop returning entries if it finds a corrupted entry (e.g., if it was being written at the time of the crash). This makes it easy for an application to reason about what information will be available after a crash by reasoning about the order that data is appended to the log.

We provide a flexible interface that allows applications to choose when they flush log entries to disk by providing separate append and sync functions. This allows applications to decide the level of batching they want when persisting log data.

We take advantage of existing application behavior by limiting the concurrency the log supports and requiring the log to be read during initialization. Our log does not support concurrent readers or concurrent writers. Instead, we support one reader and one writer operating on the log concurrently. Note that most PIMAs using POSIX already fit these restrictions. POSIX does not provide any guarantees about data when there are concurrent writers, and since log reading only happens during recovery and copy-back, PIMAs typically do not use concurrent readers to consume their log.

Our interface also does not stipulate the log's write pointer must be persisted. Instead, we provide functions for log bootstrap. As applications already need to read the log when they start up to reconstruct the current application state, this allows us to overlap application work with finding the end of the log. By not persisting the log write pointer, we avoid contention on in-memory data structures and avoid sending additional writes to the storage device.

In order to allow applications to restart as quickly as possible, we provide the `LogRewind()` function, allowing them to read the log into the in-memory state during recovery, rewind the log, and then replay it again into the kv-store. Doing so avoids the need to wait for the kv-store to catch up with the log before allowing the application to start serving requests again.

3.1.2 Key-Value Store Interface

The kv-store provides a place for the application to store state long-term. It uses transactions to group operations into collections that are executed atomically. The kv-store API provides applications with functions to update individual parts of the state and iterate through kv-pairs.

Transactions provide atomicity and durability to applications, but not isolation. We omit isolation guarantees because we expect applications to either use a single thread to access the kv-store or coordinate access to the store themselves.

Based on our investigations of existing PIMAs, we expect different kv-store functions to be called based on the phase of execution the application is in. While the application is initializing, we expect the application to use the cursor to sequentially access keys to move data from the kv-store to memory. During normal operation, we expect PIMAs to mostly use put and delete operations to execute log operations on the kv-store in a background thread. Get operations may be used if the application needs to do read-update-writes, e.g., on a partial key update.

3.1.3 Cross-Structure Consistency

Since the APIs of both the log and the kv-store are constructed to provide strong crash consistency guarantees alone, the only remaining challenge is ensuring the log and kv-store are consistent with each other. We solve this by having the kv-store update the log's read pointer inside transactions that update the store. This ensures no operation in the log is skipped or repeated due to a crash and the log read pointer always matches the most recently changed data in the kv-store. Log entries are only eligible for garbage collection after the transaction updating the read pointer has been persisted.

Table 1: **Log and key-value store APIs.**

Function	Description
`added = LogAppend(pair<char *, size> entries...)`	Append to the in-memory log
`LogWrite()`	Synchronously write log data
`data = LogRead(bool isBootstrap)`	Return next log entry if exists
`LogRewind()`	Rewind read pointer to start
`tx = TxBegin(isWrite)`	Start new transaction
`TxCommit(numCompletedEntries)`	Commit transaction and persist a new log read pointer
`TxRollback()`	Rollback transaction
`TxGet(key, val)`	Retrieve `val` if `key` in store
`TxPut(key, val)`	Add or update `key`
`TxDelete(key)`	Delete `key`
`c = TxGetCursor()`	Get cursor object
`CursorFirst(key, val)`	Move to and return first key
`CursorLast(key, val)`	Move to and return last key
`CursorNext(key, val)`	Move to and return next key
`CursorPrev(key, val)`	Move to and return previous key
`CursorCurrent(key, val)`	Return the current key-value pair
`CursorSeek(key, val)`	Move to and return the given key or the key immediately after
`CursorDelete()`	Delete current key

This also supports non-idempotent operations. Traditionally, these operations have been challenging for log-based systems because the storage system was not designed to atomically update the log's read pointer when an operation was applied to the state checkpoint.

3.2 Contrasting with Write-Optimized Key-Value Stores

Since we focus on quickly persisting writes in the log prior to moving them to the kv-store, it may seem we are suggesting PIMAs use a write-optimized storage library, such as a log-structured merge (LSM) tree. However, the design we propose differs from LSM trees in two fundamental ways: the log is visible to the application and the kv-store does not know how to interpret log data. Even though our kv-store can view the logged data as a logical log of the work it has yet to do, it relies on the application's own logic to translate that data into operations it understands. Combined with the fact that the log is visible to the application, the FASL API is especially beneficial to applications that log data which does not directly specify kv-store operations. Distributed systems are an excellent example of this as they log prepare and commit messages.

This is in contrast to LSM trees where the log is an internal element in the kv-store architecture. The kv-store completely controls the log, down to the format of the data in it, and executes operations in the log on the kv-store state without any application input. This leaves the application two choices: either implement its own crash consistent log and coordinate with the LSM tree or use the LSM tree to persist the log. Implementing, maintaining, and coordinating the log with the LSM tree has all the problems discussed in Section 2. Placing the log in the LSM tree can lead to high write amplification due to the design of LSM trees [35].

4 Implementation

We implement the log and kv-store in C++ code and interface with SPDK, io_uring, and POSIX. For portability, we create an interface between the log and kv-store and the code that performs I/O, allowing us to change storage backends without modifying the log or B+tree.

4.1 Key-Value Store

We implement the kv-store as a transactional, copy-on-write (CoW) B+tree. Our choice of a B+tree is guided by a few different observations: (1) the kv-store does not have to be write-optimized because the log will absorb writes on the critical path, (2) for flexibility, the kv-store should allow multi-key transactions, and (3) the kv-store should be easy to implement. Other kv-store structures like LSM trees and B^ϵ-trees might suffice but write-optimized trees tend to be harder to implement and increase write amplification. On the other hand, persistent hash-maps do not provide a way to atomically update multiple keys.

Overview. Our design is loosely based on BoltDB [3] and prioritizes simplicity over optimizations or features, so it does not have zero-copy for writes and assumes it will be accessed by only a single thread. Therefore, transactions are used to group operations into atomic units of work instead of providing data isolation for concurrent threads. We do not find this a major drawback since existing applications already have logic to coordinate access to storage as POSIX also lacks an isolation mechanism for file data.

The store provides a number of guarantees about data during a transaction. Buffers associated with `TxPut()` operations can be freed once the operation completes as

the store keeps an internal copy of the data. Similarly, all buffers returned from a `Get()` operation are guaranteed to be valid until the end of the current transaction, even if the key-value pair is deleted during the transaction.

Internals. The kv-store persists three different structures: a free list, tree nodes, and superblocks. The free list contains all blocks (each of fixed size) that do not currently contain valid data.

Tree nodes are either leaf nodes that contain keys and values or internal nodes that contain keys and block numbers for the block that contains the child node for that key. Data is rebalanced among the tree nodes when transactions are committed. This amortizes the rebalancing cost across multiple operations but can lead to higher lookup costs within a transaction if the transient in-memory version of a node has been the target of many insertions.

The kv-store maintains two superblocks, each of which contains bootstrap information for a version of the kv-store as well as the log read pointer. One of the superblocks is written as the last step of committing a transaction, and the superblock that is written alternates for each transaction. This guarantees that at least one superblock points to a consistent version of the kv-store.

As noted in Section 3.1.1, we persisted only the log's read pointer as the write pointer can be discovered during application initialization with no extra cost. Section 4.2 details internal log state changes required to support this.

4.2 Log

Our prototype log implementation is a simple, fixed size, statically allocated log. The log focuses on providing low-overhead persistence for applications to use on the critical path. To ensure fast appends, garbage collection, log wrapping, and recovery, the log adds a small amount of metadata to each log entry.

Metadata. The log adds metadata to each entry detailing the size of the application data in the entry, the log epoch of the entry (how many times the log has wrapped around), and a checksum of the other two metadata fields and the application data in the entry.

Wrapping. When the write pointer reaches the end of the log, it wraps back around to the beginning, and the log epoch is incremented for the first log entry that is completely after the wrap. Log entries that span the log wrap are automatically split when they are persisted and reassembled when they are read.

Garbage collection. Garbage collection only requires changing the persisted read pointer of the log. Updating the read pointer makes previous log entries inaccessible to future reads as they only operate on data greater than or equal to the pointer. Bits in the garbage collected space are not modified. Instead, the log relies on the persisted read pointer and log epoch in entries to distinguish old entries from new ones.

Ensuring complete, valid entries. The checksum and log epoch metadata entries are used to determine if an entry is complete and valid. The log epoch recorded in each entry is used to separate old entries from new ones. Old entries will have a log epoch that is lower than the previous entry's log epoch while new entries will have a log epoch that is greater than (if the log wrapped) or equal to the previous entry's log epoch.

Initialization. As the log's write pointer is not persisted, work is required during initialization to determine where the next append to the log should go. The crux of the problem is that the log needs to be read to determine what the last complete entry is (which will also be the location of the next append). Since the application must also read the log to reconstruct its state, we can overlap reading the log for application recovery with reading the log to find the next spot to append. Internally, reading the log during recovery adjusts log state variables and checks the log epoch in entries.

5 Evaluation

We aim to answer the following with our evaluation:

- how does FASL compare to other kv-stores?
- how do storage backends affect performance?
- how does available log space affect performance?
- do single-node applications benefit from FASL?
- do distributed applications benefit from FASL?

5.1 Experimental Setup

We run our benchmarks on up to 3 machines, each of which has a Xeon Gold 6138 CPU with 20 cores. We disable HyperThreading for all experiments. Machines that run FASL use an Intel Optane 905P 380GB NVMe PCIe 3.0 x4 SSD. The machines run Ubuntu 20.04.3, and unless stated otherwise, use Linux kernel version 5.9.16 with TurboBoost disabled. All benchmarks that require a file system use xfs as it supports io_uring in polling mode. io_uring in polling mode requires files to be opened with the O_DIRECT flag, bypassing the page cache and reducing performance compared to regular xfs for benchmarks that have a large working set. When FASL with a log is used in the benchmark, we report the size of the log used for each configuration. We use an additional machine to run clients. The client machine has a Xeon E5-2680 v3, 64GB of RAM, and runs Ubuntu 18.04.1.

5.2 Applications

As a microbenchmark, we connect DBBench directly into FASL's log and kv-store APIs as a driver program where it makes requests without any other application logic. We also port two applications to FASL: Redis as an example single-node application and a viewstamp-replicated optimistic concurrency control kv-store (VR+OCC) from the

public TAPIR repo [9] as an example distributed application.

Redis is a fast in-memory key-value store, with the optional ability to persist data. As we described in Section 2.1, Redis logs batches of operations to disk. When the log size is a configured percentage larger than the previous log size, Redis forks off a background thread to make a new snapshot. We replace the Redis persistence layer with FASL.

VR+OCC is a replicated in-memory kv-store. Originally, it lacked a persistence layer. OCC transaction operations are replicated on top of VR. Each node has an in-memory log that tracks the status of operations at the VR layer and an in-memory kv-store for application state. When a VR commit is logged, an upcall is made to the OCC layer to modify the in-memory kv-store as appropriate. Provided no more than f out of $2f+1$ nodes fail, a failed replica can recover its VR state by updating its log from other nodes. To recover from more than f failures, both log and application data must be persisted to disk so that the outcome of consensus operations can be reconstructed. A recovering replica first reads its local persisted data before requesting log entries made since the last seen local VR operation. Finally, the replica enters the same view as the VR leader and finishes recovery.

5.3 Ease-of-Use

To estimate whether our new storage system is easier to use than creating a custom storage stack for each application, we count the lines of code required to convert Redis and VR+OCC to FASL.

Redis changes. Porting Redis to FASL required modifying about 1,700 lines of code. Of those, approximately 1,600 add logic while the remaining lines define structs or instantiate extra state. Our changes focus on logging data, consuming the log, and reloading data from FASL. Calls to the FASL log replace existing code for logging data. We add code to run another thread to consume and garbage collect the log. Logic for parsing and executing entries in the log already exists in Redis, we just modify it to operate on either the kv-store or the in-memory data, so it can be used for consuming the log and for recovery. For recovery, we also add code to iterate through all keys in the kv-store but use existing code to deserialize kv-pairs.

VR+OCC changes. VR+OCC required adding about 1,250 lines of code. About 1,000 of those lines add logic that appends or consumes the log and restores the in-memory state after failures. The remaining define structs or instantiate additional state.

To add FASL as a persistence layer, all operations written to the in-memory log are also written to the FASL log. The in-memory log is modified in-place, but since the FASL log is append-only, new entries are created for every operation on the in-memory log. As we consume FASL's log, we track the status of operations both at the VR layer and the OCC layer. Logic for tracking these operations is based on pre-existing logic for maintaining the in-memory data structures. When the log shows that an OCC transaction has committed, we modify FASL's kv-store. We also log VR metadata, namely the last prepared and last committed operation seen during log consumption, for use on recovery. We only garbage collect entries in the persistent log such that either all or no VR operations for an OCC transaction will be in the log to ensure that all operations can be reconstructed in the event of a crash.

For recovery, we first recover entries directly from FASL's kv-store and log to recreate application in-memory state by committing up to the persisted last committed operation. To bring the replica to the current leader's state, we reuse the log-shipping logic from in-memory VR and obtain the subset of the log since the last seen local VR operation. If remote replicas are unreachable, the local replica remains at the state read from FASL's persistence layer.

Table 2: Average operation latency for dbbench writes and reads. FASL reduces write latency by 45%. FASL-no-log has 25% worse latency than other kv-stores. FASL has worse random read performance due to its simplistic caching algorithm.

	Average latency per op (μs)			
Benchmark	**FASL**	**FASL-no-log**	**leveldb**	**lmdb**
fillrand	31.9	72.6	61.3	61.7
fillseq	32.6	71.7	57.1	61.0
readrand	-	7.06	1.97	2.81
readseq	-	0.302	0.147	2.71

5.4 DBbench

We use dbbench [6] to get a better understanding of FASL's performance in comparison to other common kv-stores. We also compare the performance of different storage backends in FASL to show the benefits of newer storage backends like SPDK and io_uring.

Configuration. We use dbbench to benchmark FASL, leveldb [4], and lmdb [5]. Dbbench is configured to use 16B keys and 100B values. Unless stated otherwise, each benchmark consists of 10M operations and FASL uses a 1GB log. We run only the sync version of fill benchmarks to ensure lmdb and leveldb persist data at the end of every transaction but drop the "sync" specifier in tables for space. Fill benchmarks start with an empty database and perform only writes while read benchmarks use a database that has the entire range of keys pre-populated and perform only reads. All benchmarks in this subsection except those using io_uring were run on kernel version 5.6.19. io_uring benchmarks use kernel 5.9.16.

Comparing with existing systems. We run the sequential and random fill workloads on leveldb, lmdb, and FASL

Table 3: Average operation latency for dbbench writes and reads. Storage backends that do not use a file system, like SPDK, show clear advantages over file-system-based storage backends when the kv-store is accessed directly for writes. Due to caching effects, the xfs performs much better than others for reads.

	Benchmark	Average latency per op (μs) SPDK	io_uring/blkdev	io_uring/xfs	xfs
Log	fillrand	29.5	29.1	31.6	31.9
	fillseq	32.3	29.0	32.0	32.6
No Log	fillrand	46.6	56.1	83.0	72.6
	fillseq	36.2	43.4	62.1	71.7
	readrand	16.5	24.3	24.9	7.06
	readseq	0.623	0.755	0.768	0.302

with and without the log. Table 2 shows that when FASL has access to log space, it reduces per-operation latency by almost 45% compared to other, more mature kv-stores. In the extreme case that only the FASL kv-store is used (FASL-no-log), it has about 25% worse latency than other kv-stores.

The table also shows per-operation latency for reads for FASL-no-log, lmdb, and leveldb. We do not provide read benchmarks for FASL because it does not provide indexing on log entries. FASL has comparable performance to leveldb and lmdb for sequential reads but has 3.5x worse latency when doing random reads. This latency increase is likely due to the simplistic LRU cache that FASL uses. Increasing the cache size and improving the caching algorithm would likely reduce per-operation latency by a few microseconds, thus closing the gap. However, we expect applications to use reads mainly during recovery, where most reads are sequential as the application transfers state from the kv-store to main memory. Applications may occasionally use random reads to find specific kv-pairs during recovery or to do read-modify-write operations when transferring data from the log.

Effects of storage backends. Next, we compare the performance of the SPDK, io_uring, and POSIX storage backends in FASL both with and without the log by repeating the above benchmarks. Table 3 shows that newer storage backends like SPDK and io_uring on a raw block device (io_uring/blkdev) have better performance than configurations that use xfs when doing writes. Although performance is similar for all storage backends when using the log, SPDK and io_uring/blkdev have much better performance when storing data directly in the kv-store.

The performance for reads shown in Table 3 is a little counter-intuitive as FASL with xfs outperforms all other configurations, with the largest gains seen for random reads. Additional experiments reveal that xfs is able to take advantage of the page cache, allowing it to effectively increase the amount of cached tree data. Neither io_uring nor SPDK can use the page cache, the former because polling mode requires the O_DIRECT flag and the latter because it does not use the kernel for storage I/O.

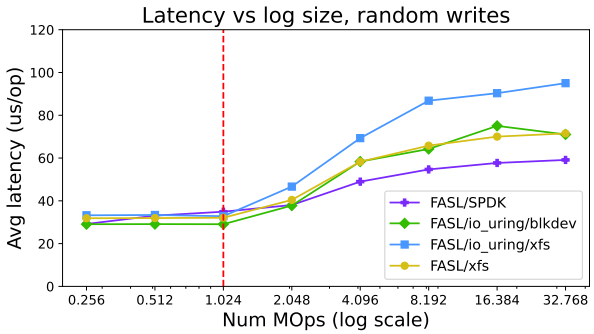


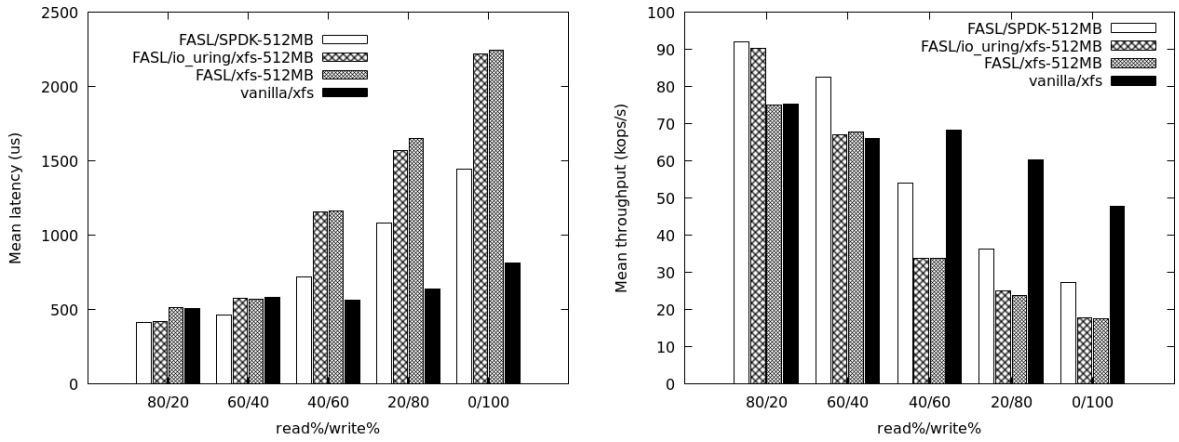
Figure 5: Mean operation latency for random write operations as the number of operations is increased while maintaining a fixed size log and keyspace. The vertical line denotes when the dataset size equal the log size. When log space is available, latency per operation remains low. As the dataset size increases, the average time per operation approaches the time required to commit a transaction in the kv-store.

Log size tuning. Next, we characterize the performance impact of the size of the log relative to the dataset size. We fix the log to be 135MB and run workloads with a modified version of dbbench that uses a fixed keyspace of 2M keys and increasing numbers of operations. Doing so allows us to increase the amount of data logged in the workload without causing the B+tree to grow without bound. Figure 5 shows that when the dataset size is less than the log size (left of the vertical line), FASL provides latencies on par with those found in the fill random benchmark. Once the dataset size surpasses the size of the log, the latency per operation begins to rise, eventually evening out at a little more than the latency of operations on the kv-store alone. The slightly higher latency for points on the right of the line compared to Table 3 may be due to SSD contention between the log and the kv-store.

5.5 Redis

Configuration. We benchmark a single Redis node configured to persist data after every operation and Redis with FASL configured with different storage backends and log sizes. For both vanilla Redis and Redis with FASL, only put operations are logged. Vanilla Redis is configured to rewrite the log file when it reaches 1.25x the size the file was to start with. For all configurations, Redis is prepopulated with 250M kv-pairs and has an empty log. Keys are 32B and values are 128B. Both get and put operations use a uniform random distribution to pick the key to operate on, and all operations use keys that already exist in the Redis. We run closed loop clients on another machine and vary the number of clients to scale load.

Mixed workloads. Figure 6a shows the mean latency of 15M operations sent from 40 clients for different read-write mix workloads. The greater the percentage of reads, the more slack FASL has to copy updates from the log in the background. Redis with FASL uses a 512MB



(a) Mean latency per operation in µs

(b) Mean throughput in op/s

Figure 6: Mean operation latency and throughput of Redis in various configurations with different read/write workloads. Redis with FASL/SPDK is able to provide lower latency and higher throughput than vanilla Redis for workloads with 60% or fewer writes.

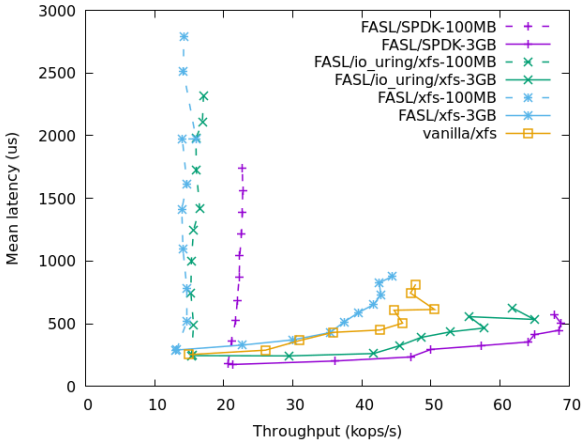


Figure 7: Mean operation latency vs. mean throughput for various Redis configurations for a 100% write workload. Redis with FASL/SPDK or FASL/io_uring is able to outperform vanilla Redis as long as there is log space.

log (roughly a quarter of the total data operated on during the benchmark). When compared to vanilla Redis, FASL/io_uring/xfs and FASL/SPDK reduce mean latency by 17% and 18% for workloads with 20% writes respectively. For workloads with 40% writes, FASL/io_uring/xfs has no performance gain while FASL/SPDK decreases latency by 20% compared to vanilla Redis. As the percentage of writes in the workload increases, the fixed size log in FASL becomes consistently full, leading to higher latency for all storage backends. With 100% writes, FASL shows latencies 1.75-2.75x that of vanilla Redis.

When the dataset size is larger than the log (>20% writes), the performance of FASL/xfs and FASL/io_uring/xfs is about equal. This implies that when the log is full, file system overheads, not kernel crossings dominate the cost of persistence.

Figure 6b shows the mean throughput for the same benchmark. FASL supports up to 25% higher throughput compared to vanilla Redis when writes make up 40% or less of the workload. With more than 40% writes in the workload, the FASL log is full most of the time, so throughput drops as the system spends more time waiting for log garbage collection.

Table 4: Redis and VR+OCC recovery times. VR+OCC with FASL outperforms over-the-network recovery by 54x when recovering 1.2GB of data. Redis with FASL is up to 1.65x slower than vanilla Redis when recovering 40GB of data, largely due to a lack of prefetching for FASL kv-store tree nodes. Recovery times for VR+OCC with io_uring are excluded due to stability issues.

	Mean Recovery Time (s)			
App	**vanilla**	**FASL/xfs**	**FASL/io_uring**	**FASL/SPDK**
Redis	327	538	531	478
VR+OCC	65.1	1.41	-	1.21

Worst-case latency vs. throughput. To really stress the system, we run another set of benchmarks performing 15M puts and vary the number of clients from 4 to 40 in increments of 4. Figure 7 shows the latency vs. throughput curve for different Redis configurations. Due to the bimodal performance of FASL, we use two different log configurations: one very small log of 100MB which is full for the majority of the benchmark, and another log of 3GB which is roughly the size of the dataset.

Similar to the read-write mix benchmarks, FASL with a large log is able to provide higher throughput at a lower latency than vanilla Redis. FASL/SPDK with log space continues to dominate, with 42% higher thoughput and 30% lower latencies than vanilla Redis with 40 clients. When FASL is configured with a small log, the throughput drops to that of the FASL kv-store alone while the latency continues to increase as more clients are added.

Recovery. The first row of Table 4 shows the worst case time Redis recovers 250M kv-pairs of data under each configuration. All experiments enable TurboBoost, and

Table 5: Retwis workload operation breakdown.

Transaction Type	# gets	# puts	workload %
Add User	1	3	5%
Follow/Unfollow	2	2	15%
Post	3	5	30%
Load Timeline	rand(1,10)	0	50%

ones that use FASL have a modified database image containing a snapshot of all data and a full 512MB log. Vanilla Redis recovers from an append-only file containing both a snapshot and log. Redis normally compresses data in the snapshot, so the append-only file is 16GB. Redis configurations that use FASL show recovery latencies ranging from 1.46x to 1.65x that of vanilla Redis. Optimizing sequential iterations through the FASL kv-store by prefetching the next tree node would help reduce this latency gap, especially since Redis with FASL spends no more than 1.9% of the total recovery time processing the log across all configurations.

5.6 Consensus Application

Normal operation. Adding persistence allows us to provide better fault tolerance via faster and more capable recovery as we show later. However, persistence also has a runtime penalty, so we start by comparing the performance of the in-memory VR+OCC system with our modified persistent VR+OCC assuming no failures.

Workload & Configuration. We use three replicas and run Retwis [28], an open-source Twitter clone, to generate a synthetic workload. Retwis has several transaction types which perform different mixes of puts and gets on a kv-store. The transaction mix and number of gets and puts per transaction is shown in Table 5. We use 64B keys and values, and a FASL log of 1GB. We populate the kv-store with 1M keys and run 100,000 transactions. Keys are selected using a Zipf distribution with a coefficient of 0.75. We then vary the number of clients in increments of 10 from 10 to 40 for the persistent versions and 10 to 60 for the non-persistent version to find the knee of the curve. For each data point, we do five runs and average the measurements.

Results. Figure 8 shows that at low loads, the difference in performance between all versions is small; with all systems within 20% of the throughput of the non-persistent version. However, as load increases, the impact of persistence on performance becomes clearer with the non-persistent version able to provide about 1.6x the maximum throughput of SPDK.

Recovery. We run two recovery experiments with our 3-replica cluster. First, we compare the best-case recovery latency of a single replica failure across the persistent and non-persistent versions of VR+OCC. To do so, we fill the system with 5M single-op write transactions, which

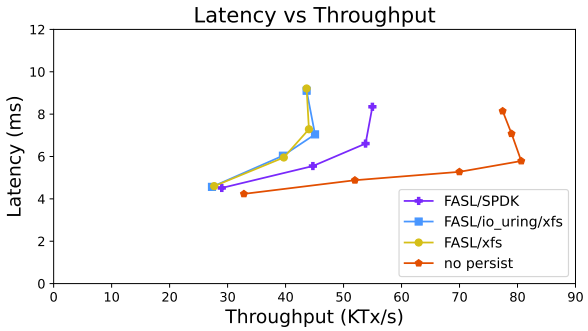


Figure 8: Average transaction latency vs. throughput for VR+OCC running Retwis transactions on different backends. The SPDK configuration has 65% the peak throughput of the no persistence configuration but outperforms other configurations with persistence by 25%.

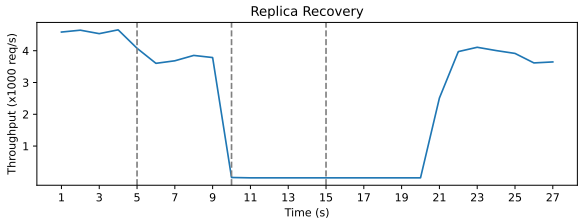


Figure 9: Average system throughput over time as VR+OCC replicas crash and are brought back up. Vertical dashed lines indicate crash and recovery events. At t=5s, one of the three replica crashes. Next, at t=10s, the remaining replicas crash. Finally, at t=15s, all replicas are restarted in recovery mode.

corresponds to about 1.2GB of in-memory log data and wait for replicas to consume the entire FASL log. Then, we crash one of the replicas and restart it, measuring the time it takes to recover. Table 4 shows our results. The non-persistent (vanilla) VR+OCC takes over a minute to ship the entire log from a quorum of replicas over the network. In contrast, FASL-based recovery finishes in under 2s for both POSIX and SPDK as the entire state is read locally, and no remote log shipping takes place. While not empirically evaluated, we further note that by avoiding or reducing the amount of log data to be shipped, FASL-based VR+OCC uses less network bandwidth for recovery than the in-memory version.

Second, we demonstrate that the persistence offered by FASL allows us to enhance the fault tolerance guarantees provided by VR+OCC. Specifically, persistence enables recovery of application data even when all replicas fail. This was previously not possible as at least a quorum of replicas had to be alive to ship in-memory logs for recovery [9]. Figure 9 demonstrates this by plotting the system throughput as replicas crash and recover over time. The workload is generated by 10 clients performing single-op write transactions using FASL/SPDK. At t=5s, one of the replicas crashes, causing a slight drop in throughput. Next, at t=10s, both remaining replicas crash, causing the throughput to drop to zero. Finally, at t=15s, we restart all the replicas in recovery mode. This causes two of the

replicas to come back up, and ship the log-delta to the replica that crashed first in order to finish recovery. After all replicas recover at around t=20s, the system is able to serve requests about 4,000 transactions per second again.

6 Related Work

Existing PIMAs. Redis [7], ZooKeeper [1], and etcd [2] each implement their own log and snapshot system tailored to their application logic. As noted in Section 2, each application solves the coordination problem between the log and the snapshot in a slightly different way. However, none of these applications create a more general persistence API that other applications can use nor do they allow other storage backends to be used without application changes. We extend these works by solidifying the persistence API and providing a prototype implementation that runs across POSIX, SPDK, and io_uring.

NVM frameworks. NVM frameworks that either store all application state in NVM [12, 22, 26, 42] or allow programmers to specify what data to store in NVM [16, 21, 23, 37, 39, 41] offer opportunities for low-latency fault tolerance. Persimmon is closest in spirit in that it proposes a framework for fault tolerance in state-machine-based applications. Using a log of operations and shadow execution, it maintains application state in both RAM and NVM, allowing fast execution and fast recovery. Our evaluation does not compare to any of these systems because the different in persistence latency between NVMe devices and NVM is so large.

Our work differs from previous work on NVM because we assume only a block device interface to storage devices, and we propose a generic persistence API that mirrors already existing persistence structures in PIMAs. By proposing an interface similar to what developers already implement, we ease the transition from custom code to library code because developers do not have to reason about new persistence and crash consistency models. Furthermore, many implementations of our API can exist, allowing future work to implement an NVM version of the FASL API.

Application crash consistency. Researchers have long known that crash consistency was difficult both for distributed applications [13] and applications in general. This is due to both the complexity of crash consistency mechanisms in applications as well as the subtle differences in the guarantees that file systems provide [11, 34]. Though Chandra et al. present their crash consistency problems in the context of Paxos, they apply more broadly to applications providing both fault tolerance and crash recovery with minimal latency. We build upon these works by first detailing the interplay between maintaining a log and snapshot and file system crash consistency guarantees. We then provide a new storage API that avoids these crash consistency problems.

Transactional file systems. Transactional file systems [19, 24, 36] and some dependency-based file systems [20] solve the cross-file and single-file crash consistency problems by extending the POSIX API. Transactions and write dependencies provide a generic, powerful interface compared to what the FASL API offers, but forces application developers to create and performance-tune their own customized storage library.

KV-stores. The release of NVMe and NVM storage devices has led to a renewed interest in kv-store research, where each system has its own performance tradeoffs [14, 15, 17, 25, 27, 29, 30, 32]. While our work includes a kv-store implementation, it is used to show the FASL API can provide material benefits to PIMAs. Therefore, previous works on kv-store implementations are orthogonal to the FASL API and can be added as part of a concrete implementation of the FASL API as long as the kv-store allows multi-key transactions. Although this paper focuses on block-based storage devices, NVM implementations of our API are possible. However, the precise kv-store implementation used should be carefully chosen to balance the cost of recovery with how quickly data can be written. Furthermore, the kv-store implementation should avoid maintaining its own log for crash consistency as that would lead to double journaling overheads.

Cross I/O-backend portability. Demikernel [40] proposes a set of library OSes providing a single interface to microsecond-latency applications. While Demikernel mostly focuses on networking, it does include a simple queue-based storage interface. Similar to Demikernel, we choose a single, high-level interface for applications and rely on the library we provide to fill in any missing pieces of functionality storage backends may lack. However, we tailor the FASL API to the needs of PIMAs, especially when it comes to crash consistency and moving data between the log and snapshot.

7 Conclusion

As microsecond-scale cloud applications grow increasingly popular, and storage devices continue to improve, we observe that the traditional POSIX API used for storing application data is inefficient, difficult to port to upcoming fast storage backends, and complicates reasoning about crash consistency. After studying how existing microsecond-scale applications are structured, this paper proposes raising the semantic level of the storage interface from a file-centric API to one centered around a coordinated persistent log and transactional key-value store. We demonstrate the practicality of our API by implementing it in FASL, a portable, crash-consistent storage library that can be used to construct persistent in-memory applica-

tions. Our evaluation shows that applications can improve throughput by up to 42% while reducing latency by 30% when using FASL. We also show that FASL can help distributed applications recover faster and enable them to recover after failure of all replicas, which is impossible to support with only in-memory data structures.

References

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

[2] etcd. https://etcd.io.

[3] etcd-io/bbolt. https://github.com/etcd-io/bbolt.

[4] google/leveldb. https://github.com/google/leveldb.

[5] LMDB – Lightening Memory-Mapped Database Manager. http://www.lmdb.tech/doc/.

[6] LMDB/dbbench. https://github.com/LMDB/dbbench.

[7] Redis. https://redis.io.

[8] Storage Performance Development Kit. https://spdk.io.

[9] UWSysLab/tapir. https://github.com/UWSysLab/tapir.

[10] What is the NVM Express Base Specification? https://nvmexpress.org/developers/nvme-specification/.

[11] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 83–98, 2016.

[12] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 433–452, 2014.

[13] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 398–407, 2007.

[14] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: An Embedded Concurrent Key-Value Store for State Management. *Proc. VLDB Endow.*, 11(12):1930–1933, Aug 2018.

[15] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies*, FAST '21, pages 17–32, February 2021.

[16] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, page 105–118, 2011.

[17] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 49–63, July 2020.

[18] Rubini J. Corbet. Ringing in a New Asynchronous I/O API. https://lwn.net/Articles/776703/.

[19] Microsoft Windows Developer Documentation. Transactional NTFS (TxF). https://docs.microsoft.com/en-us/windows/win32/fileio/about-transactional-ntfs.

[20] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. 41(6).

[21] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. go-pmem: Native Support for Programming Persistent Memory in Go. In *2020 USENIX Annual Technical Conference*, USENIX ATC '20, pages 859–872, July 2020.

[22] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-Threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 468–482, 2017.

[23] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-Structured Non-Volatile Main Memory. In *2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 703–717, July 2017.

[24] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. *ACM Trans. Storage*, 15(2), May 2019.

[25] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 187–200, February 2018.

[26] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 427–442, 2016.

[27] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 424–439, 2021.

[28] Costin Leau. Spring Data Redis - Retwis-J, 2013. http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[29] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies*, FAST '17, pages 257–270, February 2017.

[30] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, 2019.

[31] Paul Lilly. Bandwidth Busting PCIe 4.0 SSDs Rated Over 7GB/s are Launching by the Day. https://www.pcgamer.com/bandwidth-busting-pcie-40-ssds-rated-over-7gbs-are-launching-by-the-day/.

[32] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies*, FAST '16, pages 133–148, February 2016.

[33] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14*, pages 1–16, October 2014.

[34] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 433–448, October 2014.

[35] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 497–514, 2017.

[36] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *7th USENIX Conference on File and Storage Technologies*, FAST '09, February 2009.

[37] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, page 91–104, 2011.

[38] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '19, July 2019.

[39] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing Java For More Non-Volatility with Non-Volatile Memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 70–83, 2018.

[40] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, 2021.

[41] Lu Zhang and Steven Swanson. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 897–912, July 2019.

[42] Wen Zhang, Scott Shenker, and Irene Zhang. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 1029–1046, November 2020.