MemSnap µCheckpoints: A Data Single Level Store for Fearless Persistence

Emil Tsalapatis*
emil.tsalapatis@uwaterloo.ca
RCS Lab, University of Waterloo
Waterloo, Canada

Ryan Hancock*
krhancoc@uwaterloo.ca
RCS Lab, University of Waterloo
Waterloo, Canada

Rakeeb Hossain
rakeeb.hossain@uwaterloo.ca
RCS Lab, University of Waterloo
Waterloo, Canada

Ali José Mashtizadeh
ali@rcs.uwaterloo.ca
RCS Lab, University of Waterloo
Waterloo, Canada

Abstract

Single level stores (SLSes) have recently resurfaced as a system for persisting application data. SLSes like EROS, Aurora, and TreeSLS use application checkpointing to replace file-based APIs. These systems checkpoint at a coarse granularity and must be combined with file persistence mechanisms like WALs, undermining the benefits of their SLS design.

We present MemSnap, a new system that completes the single level store vision by eliminating the need for the WAL API. MemSnap introduces µCheckpoints that persist updates to memory for individual write transactions concurrently with other threads. We introduce a novel per-thread dirty set tracking mechanism in the kernel and use it to transparently persist application data. We use virtual memory techniques to prevent modifications to in-flight µCheckpoints, without blocking the application.

MemSnap-based persistence has 4.5×−30× lower latency than file-based random IO and is within 2× of direct disk IO latency. We integrate MemSnap with the SQLite, RocksDB, and PostgreSQL databases, gaining performance while retaining ACID. MemSnap increases the throughput of SQLite by 5× over file APIs and achieves a 4× throughput improvement for RocksDB compared to Aurora.

CCS Concepts: • **Computer systems organization** → **Reliability**; • **Software and its engineering** → **Operating systems**.

---

*Denotes equal contribution.

Keywords: single level store, persistence, virtual memory

1 Introduction

Single level stores (SLSes) like EROS [41], Aurora [44, 45], and TreeSLS [48] promise to simplify the development of application persistence by offering an alternative to file IO. Application writers build applications completely in memory as if they never fail, and SLSes provide transparent or nearly transparent persistence by checkpointing entire applications. Applications in an SLS forego file-based storage logic and its semantics that have led to data loss even for mature systems like PostgreSQL [3].

The problem with current SLS systems is that they provide persistence with coarse granularity that leads to them supplementing their APIs with traditional journaling mechanisms. EROS persisted applications every 30 s on a spinning disk, Aurora every 10 ms using PCIe SSDs, and TreeSLS achieves persistence every 1 ms using non-volatile memory (NVM). Aurora and EROS provide a write-ahead log API to bridge the performance gap between the coarse grained checkpoints and individual application operations that require durability.

SLS WAL APIs undermine the core premise of the single level store by retaining much of the complexity of the file API. Developers must still implement WAL and checkpoint systems to achieve good performance in SLSes, same as the file API. Thus, applications retain most of the code required for crash consistency, which is prone to bugs [37, 39, 50]. Even worse, using the WAL usually leads to infrequent SLS checkpoints causing larger and significant pause time to atomically track memory. The pause time adds tail latency to the application.

SLSes will only qualitatively change how applications are written if they truly deliver on memory semantics. We need an API that has low latency and works at a fine granularity to completely eliminate the checkpoint and WAL paradigm.

To support high performance applications with the full promise of a simplified developer model, we need a system that persists atomic checkpoints as small as a single memory page in microseconds. This would fully replace the file API abstraction with a simpler design that ensures atomicity across memory and storage in the face of system crashes. Developers would not need WALs or other approaches for correctness in their storage systems. Furthermore, it can lead to potential performance gains by eliminating the need to write data both to the WAL and to the primary store.

This paper reexamines the core mechanism behind the single level store: atomic checkpointing across virtual memory and storage. We focus on incremental checkpointing of memory regions similar to Aurora's region API. Applications work only with memory-mapped data that they modify in place and then update on disk with a single call. Our approach revisits the core idea of SLSes by providing a per-thread memory tracking and checkpointing API that outperforms file APIs and existing SLSes.

We introduce MemSnap, a new novel single level store that provides per-thread checkpoints as small as a 4 KiB page with low overhead. MemSnap is the first single level store to track individual per-thread transactions at the kernel level as opposed to the entire process without application intervention. MemSnap provides atomic checkpoints fast enough to subsume the need for the WAL by introducing new page table handling primitives in the virtual memory system. MemSnap uses the hardware TLB and fast page protection kernel code paths both to track the dirty set and prevent application threads from modifying in-flight data.

MemSnap makes the following contributions:

- Low overhead dirty set tracking for individual threads. MemSnap tracks the pages dirtied by each thread without help from the application.
- A novel per-thread checkpointing mechanism that persists dirty data into μCheckpoints with low pause times. MemSnap supports the existing SLS semantics of checkpointing all threads' data.
- A data persistence API that does not depend on a WAL. MemSnap's atomic μCheckpoints across memory and storage are fast enough to replace storage paradigms such as WAL and checkpoint and LSM Trees.
- Three case studies that demonstrate MemSnap's performance gains and simplify the code of production grade databases: SQLite, RocksDB, and PostgreSQL.

MemSnap is faster than existing file APIs and past SLS-based approaches. MemSnap μCheckpoints have 2×-1.17× overhead for writes of 4 KiB-64 Kib over direct disk IO. For comparison, file based random write IO incurs a 9×-43× overhead.

In our case studies we show that MemSnap is faster in large applications that use the file API or traditional SLSes. MemSnap increases the performance of SQLite by 5× over file APIs. For RocksDB, MemSnap achieves a 4× improvement over Aurora's checkpointing.

2 Motivation

File APIs lack atomicity in three important ways: write-tearing, multi-file writes and multi threaded isolation. First, write-tearing happens when a crash occurs during an IO that spans multiple sectors, resulting in the partial writing of the IO. The calls used by applications to flush changes to disk, i.e., fsync and msync, are not atomic on disk. Disks provide atomicity at the level of individual sectors.

Second, file systems lack the ability to atomically update multiple files. The fsync and msync calls only operate on a file or a contiguous file region.

Third, flushing changes is not atomic with respect to other threads. Modifications made by other threads to the same file will be flushed by any call to fsync or msync. There is no way to differentiate the write-sets of individual threads.

The traditional solution to solving the first two problems is through the use of a write-ahead log (WAL). Applications first persist updates to the WAL, acknowledge the original request, and then update the primary data structure in place (e.g., B+ Tree, LSM Tree). If a crash were to occur during the WAL write, the user is never acknowledged. If a crash occurs while updating the primary data structure, the WAL is used to complete the operation.

The third problem is solved by combining a variety of approaches including software level tracking of updates, checkpointing of the main data and multi version concurrency control (MVCC). This problem often leads to applications making trade-offs between design complexity and performance. For example, MVCC allows unfinished operations to reach the disk, but introduces costly garbage collection.

Flushing the WAL to the primary data structure is expensive and necessary. Applications must flush the WAL to reclaim space at some point. The flush operation depends on how the system represents data, for example the SQLite database creates a checkpoint of the database file, while the RocksDB key-value store adds a new node to its LSM tree. SQLite checkpoints generate many random IOs that increase latency, while writes to RocksDB's LSM trees are sequential but require additional IO because of background compaction operations (i.e., garbage collection).

Systems that build on fsync and msync share the same limitations as the original calls and still need WALs to function. Atomic msync [35] extends msync to be atomic in the presence of crashes, so an interrupted msync call does not corrupt data. AdvFS has the syncv [46] call that persists multiple files atomically using copy-on-write (COW) at the file system level. Both systems are slow, taking milliseconds to

Task	% Time	Task	% Time
Userspace		**Kernel**	
Tx Memory	18.3%	Buffer Cache	5.1%
Log	8.0%	File System	3.1%
Tx Disk	8.5%	VFS	6.4%
IO Generation	4.3%	Locking	6.1%
Serialization	1.1%	Rangelock	2.1%
Other Userspace	16.2%	Syscall	4.4%

Table 1. Breakdown of RocksDB's total execution time between userspace and the kernel. Only 18.3% of CPU time is spent on the in-memory transaction - the rest is spent on persistence across userspace and the kernel.

flush data to disk, require an internal WAL for consistency across multiple files, and do not address per-thread isolation.

The Cost of Persistence File APIs are not only lacking in atomicity, but expensive due to a tight integration with bulky subsystems such as the virtual file system (VFS). To illustrate the cost of file based persistence used by databases, we examine RocksDB [6] running the Facebook MixGraph workload [14]. RocksDB is a popular key-value store that is representative of other systems. Table 1 shows the CPU breakdown between components of RocksDB and the OS kernel for the workload.

RocksDB spends the majority of its CPU time on persistence instead of servicing requests. Approximately 40% of RocksDB's total time is spent writing the WAL, serializing records and issuing IO. RocksDB splits its time between userspace (56%) and the kernel (44%). Only 18.3% of userspace cycles are updating the in-memory representation, and almost half of its userspace cycles are spent on IO-related work like logging to the WAL, creating LSM tree nodes, and garbage collecting stale data. Almost all of RocksDB's time in the kernel is IO related: taking file rangelocks, running file system-specific code, and updating the buffer cache with new file data.

The userspace IO handling code also adds to RocksDB's complexity. Approximately 40% of RocksDB's code is persistence related. This contributes to the complexity of the system and to bugs. Worst of all, bugs in the persistence code can lead to data loss and corruption [1].

Together these overheads present an opportunity for performance gains and improving the reliability of storage systems. Storage systems can benefit from consolidating the software stack through a better API.

Single Level Stores Single level stores (SLSes) [41, 45, 48] unify virtual memory and persistent storage to simplify application persistence. SLSes eliminate the dichotomy between memory and storage by saving incremental application checkpoints, which include the entire memory, OS and CPU state of a running application. SLSes recover from a crash by restoring the application from the on-disk image.

Operation	Aurora
Stopping Threads	26.7 μs
Shadow Creation	79.8 μs
Write IO	27.9 μs
Shadow Collapse	91.7 μs
Total	208.1 μs

Table 2. Latency breakdown for synchronous Aurora region checkpointing during RocksDB's dbbench benchmark. Most of the overhead comes from the shadowing mechanism that applies to entire mappings, even though the amount of dirty data is 64 KiB.

Checkpointing application state is complex and too slow for individual database transactions. For example, EROS checkpoints every 30 s while Aurora checkpoints every 10 ms on PCIe flash. TreeSLS [48] breaks POSIX compatibility and requires byte-addressable persistent memory DIMMs for low-latency persistence every 1 ms. All of these SLSes are still too slow and require serialization across all threads. To remedy the overhead, Aurora and EROS complement checkpointing with a WAL API, however, this undermines the single level store goal of eliminating application storage logic.

Aurora offers an intermediate API that persists single address mappings (regions), to reduce the amount of data checkpointed and avoid the overhead of OS state. However, region checkpointing still uses Aurora's underlying system shadowing mechanism, which requires the stopping of all threads resulting in a serialization point that induces high overheads. This API is still too slow for single transactions and thus requires a WAL.

Table 2 breaks down the performance overhead of region checkpointing for a single IO done by RocksDB for the MixGraph workload. Aurora's checkpointing is based on the "system shadowing" mechanism, which is responsible for 180 μs of latency out of 208 μs. System shadowing stops all threads, applies COW by creating a "shadow object", and resumes the threads in parallel with issuing IOs. "Shadow objects" are created to retain an atomic snapshot of memory while allowing threads to continue executing during the IO write. After the IOs complete, Aurora "collapses" the shadow object back into the base object. The operation merges the page lists of the shadow with those of the base and costs 90 μs because its latency is proportional to the mapping's size. The frequency of executing region checkpointing is limited by the total time including collapsing.

MemSnap: Our Approach MemSnap is a novel approach to the single level store, which runs on commodity PCIe SSDs and subsumes the need for a WAL by addressing the three aforementioned problems of the file API with microsecond level performance.

MemSnap provides an API for persisting in-memory data to the disk without file operations or a WAL. MemSnap

System	Subset	Atomic	Per-Thread	<1 ms
fsync	No	No	No	Yes
msync	Contig.	No	No	Yes
atomic msync	Contig.	Yes	No	No
Aurora	Contig.	Yes	No	No
memsnap	Yes	Yes	Yes	Yes

Table 3. Comparison between different persistence mechanisms. `fsync` and `msync` flush an entire file or region's dirty set without atomicity. Atomic `msync` requires contiguity and does not track the per-thread dirty set. Aurora is not microsecond-scale and does not track the dirty set per thread. MemSnap combines strong guarantees, performance and per-thread set tracking.

tracks the per-thread dirty set of the application at a page granularity, rather than a global view of the dirty set that requires serializing across all threads. The result is an API that fully delivers on the single level store vision of writing applications that solely modify memory and do not require additional logic for atomicity and persistence, and addresses the limitations of existing systems (Table 3).

3 Design

MemSnap replaces file I/O based persistence with a fast and intuitive API shown in Table 4. Users create/open memory regions that are used to hold the main dataset they wish to persist atomically. Users modify data in these regions then persist it with a single call. Applications do not track their dirty pages/blocks explicitly as MemSnap does this transparently.

Applications use `msnap_open` to open or create new memory regions. The call maps the region into the process address space and returns a descriptor `md`. Similar to POSIX shared memory descriptors, these are opaque descriptors, not files. Each MemSnap mapping has a unique address where it is mapped every time. This ensures that pointers in the persistent data set are valid across reboots. We use the high end of the address space for MemSnap mappings, for a total address space of 32 PiB for machines with 5-level page tables.

Applications use `msnap_persist` to atomically persist modifications, then continue executing. `msnap_persist` is synchronous by default, but is callable with an `MS_ASYNC` flag that makes the call return after initiating the IO. The thread can later wait for the IO to complete using `msnap_wait`.

Flags to `msnap_persist` also specify which part of the dirty set it should persist. Passing a region descriptor to the call persists only the pages belonging to the corresponding region. Passing a descriptor of −1 persists all modifications across all regions. `msnap_persist` by default persists modifications made by the calling thread unless passed the `MS_GLOBAL` flag to persist modifications made by all threads.

On a `msnap_persist` call MemSnap creates a µCheckpoint to persist the dirty pages required by the call. To track dirty data MemSnap uses a page-granularity copy-on-write (COW) storage mechanism and the hardware TLB. MemSnap stores the data on its COW object store. Using MemSnap, application threads can safely modify memory even if it is currently being flushed. Applications use page-granularity locking or equivalent mechanisms only for concurrency, while MemSnap handles the atomicity of persistence-related IO. MemSnap's asynchronous mode lets a thread unlock the data in memory after `msnap_persist` to unblock other transactions.

Our design introduces three novel mechanisms that span the virtual memory and storage subsystems. First, the usage of a custom page handling technique that uses the hardware TLB and allows for the transparent tracking of dirty data for each thread. Second, our unified COW mechanism, which bridges the virtual memory and storage systems by transparently performing page table modifications in response to concurrent accesses to flushing data. Third, a fast COW storage system that solely provides direct IO and namespace requirements for our persisted objects.

Hardware Assisted Dirty Set Tracking Our dirty page tracking mechanism tracks the dirty set of each application thread individually across all memory mappings. We use the TLB hardware to track page writes by read protecting pages and trapping writes using a custom page fault handler.

MemSnap uses minor write faults to track dirty pages. All pages in a MemSnap memory region are initially read-only. MemSnap marks the region as writable (but not COW) and the PTE (page table entry) of each page as readable. This mapping configuration is unique to MemSnap and does not interfere with existing code paths in the VM fault handler or with swapping. When an application writes to a page, it takes a minor page fault. The page fault handler appends the dirty page to a thread-local list if the page is not already being tracked. MemSnap tracks the working set at the physical page level to avoid the overheads of tracking the write set of each mapping separately, e.g. using shadowing (Section 2).

The minor write fault has a lower cost than a regular COW fault because no page copy is necessary. The overall cost of a write fault is a small part of the total cost of persistence. Every page faulted is eventually committed to the disk, which is a much larger cost than the write fault itself.

MemSnap constructs the µCheckpoint using the threads' page list. The `msnap_persist` call goes through the list to create and issue a scatter/gather IO (i.e., vectored IO).

MemSnap reapplies read protection to each flushed page after writing, so subsequent writes to a page will attach it to a new µCheckpoint. `msnap_persist` reapplies read protection efficiently by going through the thread-local page list of the caller and marking the PTEs corresponding to the page as read-only. To handle multiprocess applications, MemSnap uses the page's physical-to-virtual mappings to find all page tables that contain it and modify their PTE. For

Command	Description
`int msnap_open(char *name, void **addr, size_t len, int flags)`	Create or open a region for snapshotting and return id
`epoch_t msnap_persist(int md, int flags)`	Persist the snapshot synchronously or asynchronously
`int msnap_wait(int md, epoch_t epoch)`	Wait for the snapshot to persist

Table 4. The MemSnap API. Users either explicitly register page-aligned regions of memory into the snapshot or implicitly track them throughout execution. MemSnap atomically persists these pages. Threads that should block until the data persists do so using `msnap_persist` with the MS_SYNC flag or `msnap_wait`.
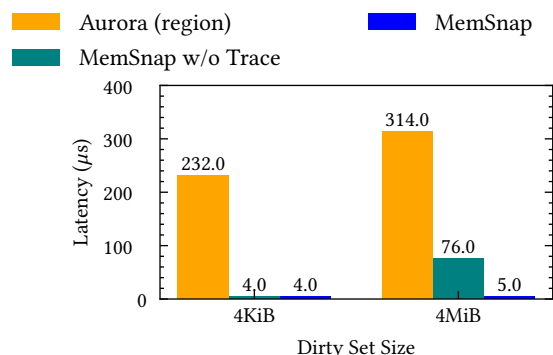

Figure 1. Comparison of the three techniques for marking pages as read only. Traversing the mapping's page tables is expensive even for small dirty sets, and traversing the page table for each page adds overhead for large dirty sets. MemSnap's trace buffer is both fast and scalable.

small checkpoints MemSnap then issues a TLB shootdown for the dirty pages, or invalidates the entire TLB for larger working sets.

MemSnap's design minimizes the time necessary for reapplying read protections. MemSnap does not stop other threads to read protect the dirty set because it does not modify the address mappings themselves, unlike existing fork-based read-protection routines that directly manipulate the address space. The dirty per-thread page list also means that MemSnap only reapplies protection to the pages added to the checkpoint instead of the entire address space. This is done instead of the default routine which scans the entire page table region for the mapping to find dirty data.

To reduce the costs of traversing the page table, MemSnap records the physical address of the PTE during the page fault and stores it in a thread-private trace buffer. The OS is guaranteed not to move the PTE entry so the mapping stays stable. When reapplying page protections after a μCheckpoint we avoid traversing the page tables from the root by locking them, traversing the trace buffer and directly modifying the stored PTEs. This optimization eliminates the need for multiple page table traversals to reapply COW due to dirty pages being sparsely spread throughout the table.

We show the performance gains of our design in Figure 1. The baseline system traverses the page tables of a 1 GiB memory mapping to find and protect the dirty pages, and

has large overheads even when protecting a single 4 KiB data page. Moving to a page-based approach to page protection reduces the cost of protecting a single page but its cost scales for larger dirty sets like 4 MiB. MemSnap's trace buffer reduces the cost of page protection to almost nothing.

MemSnap manages contention for hot pages through a per-page COW mechanism. Avoiding locking the pages in the μCheckpoint is important to prevent contention with userspace threads, e.g., for the root of a tree data structure.

MemSnap sets a new "checkpoint in progress" flag in the physical page structure (`vm_page`) to signify the page as busy. Once complete, MemSnap returns to the calling thread. All threads can continue to read and write pages as they are being flushed to the disk. Writes to pages that have the checkpoint in progress flag trigger a COW path that duplicates the original page and updates the mapping and page tables to point to the new copy. This allows MemSnap to create an atomic checkpoint, while avoiding the cost of copy-on-write operations on all pages. Pages that do not have the checkpoint in progress flag are handled by the dirty set tracking fault code path.

Persisting MemSnap Regions MemSnap uses a COW object store to persist μCheckpoints. The API of the store is similar to the internal APIs of popular COW file systems [4, 9, 12, 22], and these file systems are potential backends for the MemSnap mechanism if their API is extended to expose lightweight object checkpoints. Our object store is a simple implementation of the minimum viable design required to support MemSnap.

The store uses COW radix trees instead of the more complex, read optimized, COW B-Trees used by many storage systems [40]. The object store optimizes for random access patterns and small object sizes and heavily prioritizes writes over reads. Radix trees work well for the write patterns generated by the workload because they are block based and do not suffer from the extent fragmentation problems that B-Trees have if snapshotted frequently.

The object store has a key-value interface and is not burdened by the traditional file system API or POSIX standards. The object store does not integrate with the buffer cache and instead does direct IO, deferring the management of memory pressure to the VM page cache. The store only allows memory mapping the data and does not support file IO.

Operation	Overhead
Resetting Tracking	5.1 μs
Initiating Writes	6.5 μs
Waiting on IO	39.7 μs
Total	51.4 μs

Table 5. Breakdown of an `msnap_persist` call for 64 KiB worth of dirty pages. The call protects the pages to reset tracking in 5.1 μs, and spends 39.7 μs waiting for the IO to complete. Minimizing COW application time quickly frees up the modified pages to be accessed and modified by other threads and thus avoids contention.

Each object maintains its own logical history. An on disk object contains a monotonic epoch number that increments after each μCheckpoint that the object is a part of. This allows for restores independent of the state of the object store. Further, it allows for μCheckpoints to occur concurrently with each other as objects are not tied to some global epoch.

MemSnap's OS/workload co-design reduces the cost of persistence close to the latency of the disk without any optimizations from the user at all. Table 5 shows the cost of persistence for a single 64 KiB transaction done by the RocksDB workload we presented in Section 2. An `msnap_persist` call from RocksDB takes 51.4 μs, only 7 μs more than the latency of 44 μs for a direct disk IO of the same size. Most of the additional latency is in resetting tracking to the pages by modifying their PTEs, a fundamental cost for dirty page tracking. The other 45.2 μs are spent on the actual IO, same latency as a direct IO.

4 MemSnap-based Crash Consistency

We now present how to adapt an application to the MemSnap API. First, we describe how MemSnap replaces the file interface while leaving its ACID guarantees intact. We then enumerate the key properties that the in-memory data structures must adhere to. Finally, we show how these properties combine with the data structures' existing locking mechanisms to guarantee on-disk consistency.

MemSnap modifies the lowest level of the database's software stack and leaves most components unmodified. Databases split their logic into a storage engine that manages the data across memory and files, and an upper layer that implements abstractions like tables and transactions. The upper layers manage high level data modifications and transactional locking and do not directly interact with on-disk data, instead using the storage engine for serialization and disk access. Enabling MemSnap only changes the storage engine's persistence calls and leaves the upper-layer logic as-is.

We modify the storage engine to use MemSnap regions instead of files. The unmodified storage engine stores the data in files for persistence, and also caches part of it in an in-memory data structure, e.g., a B-tree, for fast access. We
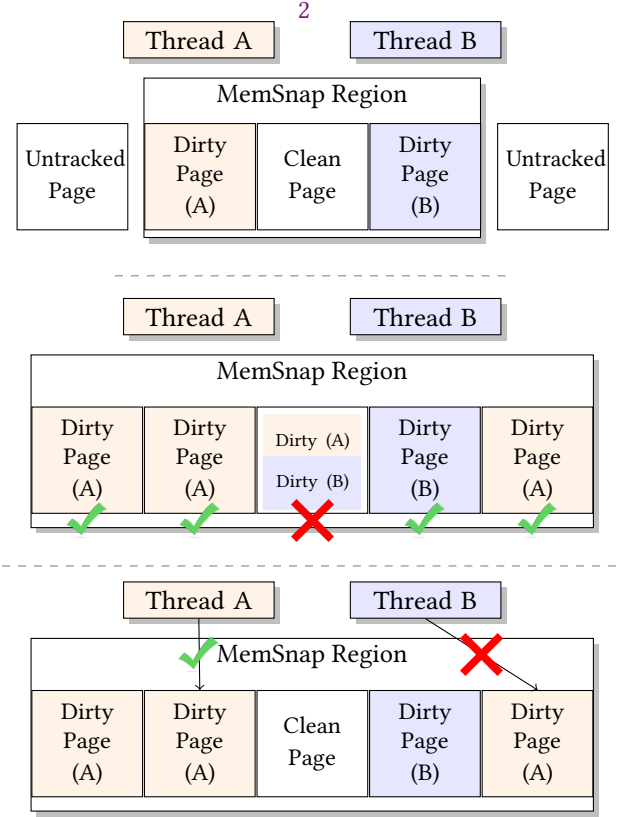

Figure 2. The three conditions required for MemSnap-based persistence. ① MemSnap persists only dirty pages that are in a MemSnap region. ② Two threads cannot simultaneously dirty different parts of the same page. ③ A page dirtied by a thread cannot be dirtied by another until it is flushed out.

expand the in-memory data structure to hold all data and place it in MemSnap persistent regions, removing the files.

MemSnap flushes write transactions by tracking the changes it makes to the data structure's memory, and writing the changes as a μCheckpoint to disk. The unmodified storage engine handles writes by first logging the data in a WAL, then modifying the in-memory copy. We replace WAL writes with MemSnap's dirty page tracking, and replace WAL `fsync` with `msnap_persist` calls.

We do not modify the storage engine's concurrency control. We call `msnap_persist` where `fsync` was previously called. The calls provide the same persistence guarantees, so the surrounding code's assumptions remain valid.

Persisting the In-Memory Representation MemSnap uses its COW-based dirty page tracking to determine the contents of each transaction's corresponding μCheckpoint. For a μCheckpoint to correctly update the on-disk copy, it must include all the data structure pages dirtied by the transaction that created it. The μCheckpoint must not include any

pages dirtied by other concurrently executing transactions to avoid persisting uncommitted data through a crash.

An example is a B-tree updated by a write transaction with new keys. The corresponding µCheckpoint must persist the new key-value pairs, and any B-tree nodes dirtied by the operation. The µCheckpoint must also not persist any changes made to the B-tree by other transactions.

The in-memory data structures must adhere to three properties to ensure MemSnap's correctness (Figure 2):

① All data falls within MemSnap persistent regions for compatibility with our API. We enforce this property by allocating the in-memory data structure from MemSnap regions.

② Two key-value pairs or data structure nodes cannot be on the same OS page. By nodes we refer to the individual components of the data structure, e.g., a B-tree node or a skip list node. We ensure this property by making each node or key value pair of a data structure page-aligned, if this is not already the case. Two transactions with non-overlapping write sets will then write to non-overlapping pages.

③ A node cannot be simultaneously modified by multiple active transactions. When a transaction modifies a key, it must lock it to prevent others from writing to it until it is persisted. This is already done by some storage engines, e.g., SQLite. However, some data structures like RocksDB's skip lists require additional fine-grained locking (Section 7.2). This extra locking does not change data structure semantics.

These properties combined ensure that we update the on-disk dataset in a *serializable* fashion, i.e., our persistence operations form a series of transactions applied to the image one by one. Property ② ensures that MemSnap's page tracking adds to a thread's working set all the data it has dirtied, and never includes unrelated dirty data. Property ③ ensures that pages added to the dirty set were initially clean and cannot be written until the transaction commits to the disk, so the µCheckpoint does not include other transactions' changes. MemSnap's `msnap_persist` calls thus write out all data structure modifications done by a single transaction.

MemSnap applications restore after a crash by mapping persisted data back into memory. The new instance first recreates components e.g., transaction managers, allocators, lost during the crash. The instance then retrieves a list of all MemSnap regions in an application, along with their address in memory. It recreates these mappings and uses memory accesses to page in the data. All pointers in the mappings that point to persistent data are still valid after the crash because regions are always mapped at the same address. The application also updates stale pointers to lost volatile state.

MemSnap Region Crash Consistency All MemSnap regions are crash consistent due to our COW object store. Each MemSnap region is an object in the store and its data is stored in a radix tree. The object store uses COW to update data without overwriting it. Every µCheckpoint first writes the data to newly allocated space. Because of COW the object store then creates a copy of the leaf node that points to the

new data, then a copy of that node's parent that points to the new node, and so on until it writes a new tree root. Changes commit after the new root persists.

Region data is consistent after a crash. Interrupted transactions and µCheckpoints do not persist and the application recovers from the last completed µCheckpoint.

5 Scope and Limitations

MemSnap handles persistence and does not address memory overcommit. The file API's semantics for memory-mapped data is problematic both for persistence, and for tiering larger-than-memory datasets between disk and memory [17]. MemSnap leaves the question of a memory-mapped overcommit API out of scope, much like the other SLSes, because it is orthogonal to its core goal of efficient persistence. MemSnap does not limit the design of new overcommit APIs. The lack of an overcommit API however, limits our MemSnap case studies to datasets that fit in memory.

MemSnap is designed with page-aligned data structures in mind and causes write amplification with data structures that use small key-value pairs. MemSnap flushes at a 4 KiB page granularity regardless of how much data in the page is actually dirty, incurring disk write amplification. MemSnap also requires transactions to lock the dataset at a page granularity. Buffer caches already do this, but for systems like key-value stores, changes are required to either use a single per-page write lock for all key-value pairs in the same page, potentially losing parallelism, or placing each key-value pair in its own page, causing space amplification.

6 Evaluation

Implementation MemSnap takes up 2.5 KSLOC of kernel code for tracking pages, initiating the IOs and enforcing COW on the pages being written out. We implement these mechanisms on top of FreeBSD 12.3. We built MemSnap's storage system is 3 KSLOC with 1 KSLOC for radix trees while the rest is for the object store's logic.

We wrote 500 SLOC for SQLite, 400 SLOC for RocksDB, and 1 KSLOC for PostgreSQL, porting each to MemSnap. MemSnap makes redundant 7 KSLOC in SQLite, 40 KSLOC in RocksDB, and 10 KSLOC in PostgreSQL.

In this section we analyze MemSnap's performance and show why it is faster than file-based persistence mechanisms and Aurora's checkpointing. First we characterize the `msnap_persist` call and compare its performance against `fsync` and Aurora's region checkpointing. We then show that MemSnap has major benefits for three real world case studies: SQLite, RocksDB, and PostgreSQL. We present the databases and the changes necessary to use MemSnap, and evaluate their benefits in terms of performance and code complexity.

For all benchmarks we use a dual Intel Xeon Silver 4116 CPUs (Skylake-SP) running at 2.1 GHz with 96 GiB of memory. For storage, we use two Intel 900P PCIe SSDs striped together in 64 KiB blocks.

Comparing Persistence APIs MemSnap persists data at a lower latency than the file API and region checkpointing. We compare `msnap_persist` against `fsync` and Aurora by using them to persist writes with access patterns that we see in our case studies. We test `fsync` under both the FFS and ZFS file systems to show that `fsync`'s overheads are consistent across file systems. FFS [27] uses soft updates [20] and journaling for consistency, while ZFS uses copy-on-write. We show that MemSnap has lower latency than Aurora's region checkpointing even though both checkpoint the same amount of data.

Table 6 shows the latency of different persistence APIs for sequential and random writes. The microbenchmark flushes dirty data using `msnap_persist`, `fsync`, or direct disk IO. Random write patterns are seen during database checkpointing. Sequential write patterns are prevalent during write ahead logging and LSM-Tree writes. We also measure direct IO to the striped disks with one outstanding IO.

We evaluate MemSnap with the random IO workload and measure the synchronous and asynchronous persistence latency. Asynchronous latency is the CPU time spent on reapplying page protections to each dirty page. Synchronous latency is the total end-to-end time for persisting a transaction to the disk. We evaluate `msnap_persist` only with random writes as it persists modifications to primary data made in-place.

Table 6 shows that MemSnap outperforms both FFS and ZFS in sequential and random workloads. This is due to MemSnap's optimized dirty page tracking and COW object store, which translates random object updates into sequential writes on disk. MemSnap's random IO latency (sequential on disk) outperforms our direct disk IO measurement for large writes because having more outstanding IOs results in better saturation of the disk's throughput.

MemSnap vs. Aurora Figure 3 shows that MemSnap is up to 60× faster than Aurora's application checkpoints and 7× faster than its region checkpoints. We show this by synchronously persisting a randomly distributed dirty set using MemSnap and Aurora's application and region checkpointing then measure the total latency of the call. We keep all data in a single mapped region because region checkpoints cannot handle dirty sets spread across multiple mappings.

Region checkpointing provides data persistence like MemSnap does, but uses object shadowing and collapsing to manage tracking the memory region. Aurora's application checkpointing overhead is an order of magnitude larger than region checkpointing because it must protect the entire address space and has larger post-checkpoint cleanup operations that

Size (KB)	Disk (µs)	fsync (µs) Sequential	fsync (µs) Random	memsnap Random (µs)
		FFS|ZFS	FFS|ZFS	Sync.|Async.
4	17	70|64	156|232	34|6
8	18	79|71	252|371	36|6
16	22	89|80	464|706	41|6
32	31	111|134	828|1.4K	48|6
64	44	134|137	1.9K|2.9K	50|6
128	N/A	164|204	4.3K|7.8K	70|9
256	N/A	218|347	8.8K|11.7K	112|13
512	N/A	338|672	12.6K|15.6K	168|23
1024	N/A	581|937	17.9K|18.2K	297|36
2048	N/A	1.1K|1.7K	23.5K|20.2K	552|57
4096	N/A	1.9K|3.0K	33.7K|30.9K	1.0K|108

Table 6. The latency of persistence for `fsync`, `msnap_persist`, and Aurora. We use `fsync` after writing out a random or sequential series of 4 KiB pages and compare its latency with that of `msnap_persist` for a random IO pattern of the same size. MemSnap outperforms disk writes for large write sizes because disk writes always have one outstanding IO.
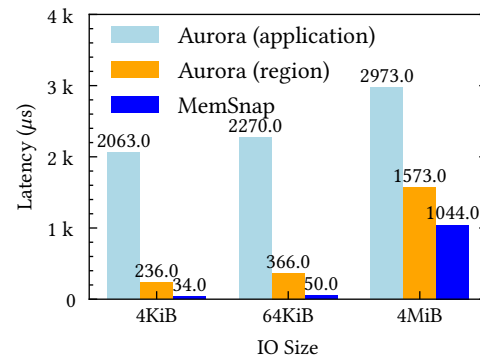

Figure 3. Comparison of MemSnap with Aurora's region and application checkpointing. We measure the latency of a synchronous persistence operation for randomly distributed dirty sets of different sizes. MemSnap outperforms Aurora's region checkpointing by 7× for small IOs because it processes individual pages.

add latency. MemSnap has 70% the latency of region checkpointing as it avoids the overhead of system shadowing.

7 Case Studies

To show how MemSnap is a general purpose tool for persistence we apply our API across three databases: SQLite, RocksDB, and PostgreSQL. Databases are unique applications as they stress many edge cases of a persistence API while demanding performance. These databases are chosen as they represent different database designs. SQLite is a read optimized single writer SQL database. RocksDB is a write

optimized key/value store that uses LSM-Trees. PostgreSQL is a multi process system that uses multi version concurrency control (MVCC) for highly concurrent transactions.

7.1 Case Study: SQLite

The MemSnap integration with SQLite shows how the MemSnap mechanism speeds up database persistence without changing the design or requiring major modifications to the database. SQLite is a popular [7] database that is embedded into larger applications to provide data persistence. We present SQLite before and after adding MemSnap, show the equivalence of the two versions, and compare performance.

SQLite's architecture follows the general layered designed outlined in Section 4. The database models each table as a B-tree with a key-value pair for each row, and the database's upper layer translates SQL statements down to API calls on the B-tree's keys. The API includes locking and persistence. For example, upper layers isolate transactions by locking keys to prevent uncommitted values from being visible.

The SQLite storage engine implements the B-tree API and moves data between the WAL and the database (DB) file. The storage engine mmap()s both the DB file and the WAL into memory. The WAL doubles as a cache for the DB file and services reads and writes. The engine also includes a lock manager for the WAL and DB file's contents.

Baseline SQLite writes B-tree nodes dirtied during a transaction to the WAL, and calls fsync on it when the transaction ends. When the WAL becomes large from all the logged data, SQLite checkpoints by copying its contents into the DB file. SQLite fsyncs both files, then truncates the WAL.

To integrate MemSnap with SQLite, we developed an SQLite plugin that replaces file IO with MemSnap equivalents. The plugin is loaded at runtime, requiring no modification or recompilation of SQLite. The plugin is 347 SLOC and replaces the standard 4.8 KSLOC Unix file module.

MemSnap SQLite persists changes directly in the DB file and skips persisting the WAL with fsync. In our system, the storage engine backs the WAL in volatile memory and the database with a MemSnap persistent region. During a commit we move the transaction's dirty set from the WAL to the persistent region, and flush with msnap_persist.

Our change retains the storage engine and lock manager's logic. We still use the "WAL" as a cache, and when we flush the transaction data to the persistent region we reuse the calls that baseline SQLite uses to flush the WAL to the DB file during a checkpoint. To the upper layers of SQLite, the MemSnap plugin semantically is identical to a checkpoint occurring after every transaction. However, checkpointing frequency only impacts performance and does not degrade transaction semantics or crash consistency.

Our system satisfies all properties in Section 4:

① We have moved the main database from a DB file to a persistent region. The main database is already accessed
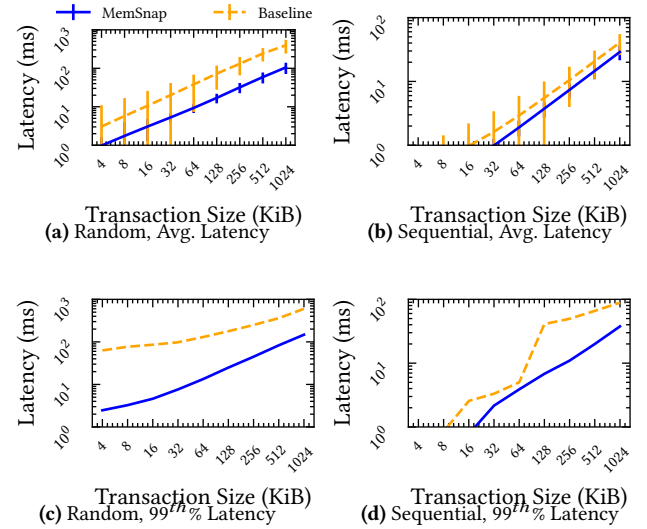

Figure 4. Performance of MemSnap vs the WAL+Checkpoint baseline for transaction sizes from 4 KiB to 1 MiB. The actual IO done is larger than the transaction size because of write amplification. The latency for both configurations scales linearly with transaction size, but MemSnap is faster both for average and tail latency across all transaction sizes. MemSnap has a fixed per-4 KiB page latency across all transaction sizes, while the baseline's latency rises for smaller sizes.

through memory-mapping, so this change does not affect the rest of the code.

② SQLite already stores data in database page-aligned chunks, and locks them at the same granularity. SQLite thus already prevents two transactions from dirtying the same page. We configure the size of each database page to be 4 KiB, to make SQLite lock data at the same granularity with which MemSnap tracks the dirty set.

③ SQLite already uses the lock manager to uphold its ACID guarantees and crash consistency. Our changes do not affect the lock manager, so MemSnap's μCheckpoints do not include uncommitted modifications by other threads.

Evaluation We first compare MemSnap to the baseline by measuring the number and latency of persistence related system calls made during the dbbench workload. dbbench generates up to 1M keys with 128 byte values. Key/value pairs are batched sequentially or randomly into write transactions ranging from 4 KiB to 1 MiB in size until 2 million total key value pair writes have been performed. Checkpointing is configured to occur every 4 MiB worth of dirty data as is the default [10].

Table 7 shows the latency and number of system calls for MemSnap versus the file API baseline. We show the average call latency and total calls made for each system call over several transaction sizes. SQLite makes one fsync call on

Transaction Size	memsnap		fsync		write		read	
	Latency	Total Ops	Latency	Total Ops	Latency	Total Ops	Latency	Total Ops
Random IO								
4 KiB	152.2 µs	63.1 K	1137.3 µs	67.4 K	6.7 µs	7584.1 K	2.9 µs	2846.8 K
64 KiB	1621.0 µs	3.9 K	10024.4 µs	6.2 K	6.7 µs	6703.2 K	2.9 µs	2729.7 K
1024 KiB	17521.8 µs	0.2 K	41521.7 µs	0.9 K	6.7 µs	4127.1 K	2.9 µs	2046.8 K
Sequential IO								
4 KiB	51.0 µs	59.4 K	155.6 µs	63.5 K	6.7 µs	1174.7 K	3.1 µs	94.6 K
64 KiB	97.7 µs	3.5 K	471.4 µs	3.8 K	6.7 µs	247.3 K	2.9 µs	90.2 K
1024 KiB	564.8 µs	0.2 K	3429.1 µs	0.3 K	6.7 µs	176.1 K	2.8 µs	88.0 K

Table 7. The number and cost of persistence-related system calls for both MemSnap and the baseline. MemSnap only requires the `msnap_persist` call that is both less frequent and cheaper than `fsync`. The baseline in contrast uses WAL-and-checkpoint that generates a significant amount of file IO.

Baseline	%CPU	MemSnap	%CPU
Random IO			
userspace	1.58%	userspace	10.51%
fsync	29.15%	memsnap	9.31%
write	30.34%	memsnap flush	8.57%
read	3.92%	page faults	30.90%
Wall clock time	**175s**		**35.4s**
Sequential IO			
userspace	24.63%	userspace	60.34%
fsync	14.33%	memsnap	3.15%
write	26.62%	memsnap flush	2.74%
read	2.51%	page faults	10.38%
Wall clock time	**12.5s**		**7.2s**

Table 8. CPU usage and total `dbbench` execution time for MemSnap and the baseline. MemSnap runs 2× to 5× faster than the baseline while doing a similar amount of work in userspace.
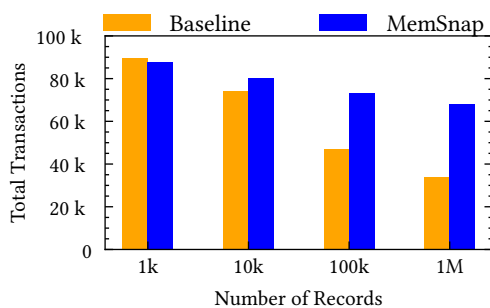

Figure 5. TATP benchmark. The baseline slightly outperforms MemSnap for small database sizes by because of its WAL index, but heavily degrades in performance for larger databases.

every transaction that ranges from 4 KiB to 1 MiB in size, and an fsync call on every checkpoint after 4 MiBs of writes.

The table shows that MemSnap does fewer and faster calls than the file API, lowering the cost of persistence. The number of MemSnap calls is equal to the number of transactions, while the number of `fsync` calls is equal to the number of transactions and checkpoints made during the benchmark.

The number of checkpoints made by the baseline is larger due to write amplification of the WAL. SQLite implements the WAL such that when any block is dirtied through a write, the block is appended to the WAL. This means every 128-byte value written is amplified to a 1 KiB write into the WAL resulting in more checkpoints. Every write to a block causes a write to the WAL even during the same transaction.

We show the average call latency for different transaction sizes in Table 7. Write amplification to the WAL causes the total IO to exceed the transaction size and further increases the performance benefit of MemSnap. MemSnap's cost scales linearly with the transaction size, with 64 KiB transactions having 16× the cost of a 4 KiB transaction. `fsync` calls are on average an order of magnitude slower compared to MemSnap for 4 KiB transactions, because the average `fsync` latency is skewed by checkpoint `fsync`s.

Both systems use `memcpy` to update the in-memory data and file IO is only ever done for the baseline. The `write` and `read` calls have low latency as they are serviced from the OS buffer cache, but are numerous and so add significant latency. MemSnap does not do `write`/`read` system calls.

Figure 4 shows average and 99th percentile latency. MemSnap has 4 times lower latency with low variance, while the baseline's variance increases as smaller transactions incur less latency while checkpointing latency remains constant and skews the average. The performance gap is even larger for random transactions. MemSnap does sequential IO even for random in-memory writes, while the baseline's IO patterns are affected by the writes' sequentiality. All configurations are subject to write amplification as values are 128 bytes long, but pages and FS blocks are both 4 KiB.

CPU Usage MemSnap-based SQLite increases throughput because it reduces kernel time. Table 8 shows the CPU usage breakdown of the dbbench workload. The table shows that MemSnap spends 10% of its CPU time in userspace for random IO, and 60% for sequential. This higher ratio is because aggregating the dirty page set into a μCheckpoint takes less than 1% of total CPU time (i.e., memsnap).

MemSnap's speedups over the baseline are due to cheaper persistence-related calls in SQLite's critical path. The baseline spends 25% of its CPU time on writes to the WAL, and 15-30% of its CPU time on the fsync calls required for persistence. MemSnap's cost in contrast mainly stems from the page faults required to track the dirty set, about 30% of CPU time for random access and 10% for sequential due to locality.

TATP Benchmark We demonstrate MemSnap throughput benefits using the TATP database workload [8]. TATP is used by SQLite's authors to evaluate its performance for real-world database workloads [19]. These workloads have a 80% read, 20% write transaction mix and span across multiple tables. All write transactions commit synchronously to the disk. We run the benchmark for 60 seconds using different database sizes (1 K to 1 M records as are typical [19]), and measure the total transactions done.

MemSnap consistently outperforms the baseline with the difference increasing for larger database sizes. MemSnap sees a 23% reduction in throughput as it increases the database size from 1 k to 1 M records, while the baseline system sees a 63% reduction. The baseline system's fsync costs increases with the resident size of the mapping and not just the dirty set. This behavior is consistent across OSes including Linux and FreeBSD. MemSnap's overhead is independent of the resident size of the mapping and so its throughput stays more consistent. Both systems lose throughput because larger databases include userspace overheads, e.g, SQLite's internal B-Trees become slower because their height increases.

7.2 Case Study: RocksDB

The main challenge of integrating RocksDB with MemSnap is its in-memory skip list data structure, which was not designed to be written to the disk. This is in contrast to our other workloads where MemSnap persists the data under the same on-disk format as the original database. We optimize the skip list for MemSnap and ensure crash consistency.

MemSnap replaces RocksDB's WAL-and-checkpoint mechanism. RocksDB distributes data between a WAL file, in-memory skip lists called MemTables, and *SSTable* files. Put calls in RocksDB write new KV pairs to the WAL, persist them with fsync, then add them to the skip list. When the WAL gets too large, RocksDB checkpoints the skip list by serializing it into a new file. Our goal is to remove WAL logging and serialization and use MemSnap-based persistence.

We remove the WAL and use MemSnap to directly persist the skip list, merging both their functionality into a single data structure. Skip lists are ordered singly linked lists where each node holds one KV pair. Nodes have extra *skip pointers* that skip ranges of keys, speeding up random searches.

To satisfy Property ①, we move the skip list from volatile memory to a persistent MemSnap region. We also add an `msnap_persist` call at the end of the skip list's Insert call made by RocksDB's Put. Our changes retain the persistence semantics of the Put call as Insert happens after the WAL logging and within the same critical section.

Changes to the skip list retain RocksDB's transaction semantics. The skip list's own internal locking only preserves its own integrity, not the database's ACID guarantees. RocksDB implements range queries and transactions on top of RocksDB's snapshot versioning and lock management mechanisms respectively, and we do not modify either.

We adjust the skip list's internal locking to prevent threads from being able to overwrite unpersisted modifications, satisfying Property ③. The skip list's Insert operation updates linked list pointers in existing nodes using compare-and-set (CAS). To prevent an uncommitted node from being modified before the `msnap_persist` call we replace the CAS operation with a per-node spinlock. This introduces minimal performance overhead, in the order of a few dozen cycles.

As an optimization, we do not persist skip pointers because we only need to guarantee the crash consistency of the underlying linked list. For each write we then persist only the new node and its previous in the list. Skip pointers speed up searches, and act as an index on top of the linked list structure. We can recreate this index after a crash by traversing the restored linked list.

Finally, we adjust the size of each node to be compatible with MemSnap's page tracking and satisfy property ②. We adjust the node size to 4 KiB to align them with MemSnap's page tracking mechanism. This makes it possible to track each KV pair separately using our dirty page tracking at the cost of additional write amplification.

Crash Recovery Interrupted μCheckpoints are automatically cleaned up after restoring. MemSnap-RocksDB recovers after a crash by restoring the MemSnap region from the on-disk state. The object store updates its data using COW, so interrupted μCheckpoints cannot overwrite persisted data.

RocksDB creates a single μCheckpoint for each transaction commit, so the on-disk data includes no uncommitted writes. We use RocksDB's WriteCommitted configuration that writes data to the MemTable only when committing a transaction, using a single MultiPut to write all changes.

To restore, the RocksDB instance retrieves the on-disk regions in the object store and maps these regions to their specified in-memory location to ensure that the linked list pointers are valid. It then traverses the linked list nodes to recompute skip pointers. The restored RocksDB skip list is consistent and includes all data committed before the crash.

We run a test to ensure that in-memory transactions remain properly atomic, isolated, and crash consistent. The test starts with an initialized database containing 100 k random

key-value pairs and running 20 threads. Each thread creates a transaction that randomly selects 100 keys and increments each of their values, and repeats 100 k times. We then verify the consistency of the dataset by summing all values.

We re-run the above test to verify that the database is crash consistent, by crashing with a kernel panic during the test. We load the on-disk data to a new instance and verify that the values sum up to the correct amount, by referencing acknowledged transactions that occurred before the crash.

Effects of Removing the LSM Trees Our design removes the on-disk *LSM tree* that RocksDB uses to tier data on the disk, by placing all data in a single MemTable (i.e skip list). This choice was made because the LSM tree's main advantage is automatic tiering for larger-than-memory workloads that we do not consider in this work. Removing the on-disk LSM tree has the advantages that it reduces application complexity and removes the need for compaction.

MemSnap integration does not fundamentally require using a single MemTable or removing the LSM tree. Alternative designs can periodically swap out MemTables to generate multiple smaller on-disk regions and tier them into an LSM tree in the same way that the baseline creates an LSM tree out of SSTable files.

Evaluation We compare a MemSnap-RocksDB system with a system that uses Aurora's region checkpointing. The Aurora system stores all MemTable data in a single mapping and issues a `checkpoint` after each write. On a checkpoint call Aurora stops all application threads and atomically creates an incremental checkpoint of the region, then resumes all threads except for the caller and synchronously flushes the data. We modify Aurora to synchronously persist region checkpoints so that region checkpointing provides identical guarantees to MemSnap and baseline RocksDB.

We evaluate MemSnap with RocksDB using the Meta's MixGraph workload. MixGraph replicates usage patterns seen in large-scale deployments and is commonly used to evaluate RocksDB [14]. The workload is composed of 84% `Get`, 14% `Put`, and 3% `Seek` requests and represent queries and updates to Facebook's social media graph. For this workload RocksDB persists writes synchronously. We fill the database with 20 M 48-byte keys - 100-byte value pairs (3 GiB). Keys are chosen uniformly, while writes are chosen using a generalized Pareto distribution. We run with 12 threads, and measure total throughput, average and tail latency.

We benchmark our system using three different configurations. The first uses MemSnap for persistence after every write to the MemTable. The second uses the unmodified RocksDB code that uses a WAL for single write persistence and creates a new SSTable file every 64 MiB worth of writes. The third configuration uses Aurora's region checkpointing which checkpoints after every write to the database.

Table 9 compares the overall performance of the three systems and shows how MemSnap's low latency translates to

Configuration	Metric		
	Kops	Avg(μs)	99th(μs)
`memsnap`	420.7	138.9788	239.62
Baseline+WAL	388.0	162.7709	248.44
`Aurora`	91.8	751.9326	4.2K

System Call	Metrics	
	Latency (μs)	Total Count
`memsnap`	51.4	208.1K
`fsync`	63.1	190.4K
`write`	19.4	190.6K
`checkpoint`	204	88.6K

Table 9. RocksDB latency comparison between MemSnap RocksDB, a WAL-based baseline, and Aurora's region checkpointing. MemSnap provides 3× lower latency than Aurora as it persists data with a single call. MemSnap's costs scale down with the dirty set, unlike Aurora's region checkpoints.

better total throughput. MemSnap has significantly lower latency than the baseline system, both in the average and 99th percentile. The lower latency directly translates to higher overall throughput for the workload because persistence-related overheads are the main bottleneck. The table also demonstrates that Aurora's region-based persistence has unacceptable overheads for multi threaded applications. Checkpointing an address space region with Aurora reduces overall throughput by 75% as region checkpointing includes a fixed-cost overhead that does not scale down with the dirty set.

Table 9 explains the performance differences between the three systems by measuring the number and latency of persistence-related calls. Persistence-related calls make up for the majority of end-to-end operation latency, while the total number of operations done heavily correlates with the overall throughput of the benchmark. MemSnap uses a single `msnap_persist` call to persist the modified data, while the baseline uses a `write` call immediately followed by an `fsync`, and Aurora uses a `checkpoint` call. The `msnap_persist` call has an average latency of 51 μs compared to the cumulative latency of `write` and `fsync` of 82.5 μs. The `fsync` call is costlier than `msnap_persist` because of its more complex code path that includes traversing the buffer cache and initiating the write in the file system.

Figure 10 shows MemSnap is 4× faster than Aurora's `checkpoint`. This is because the cost of region checkpointing scales with the size of the memory region and the cleanup routines must run after the IO completes. These routines are associated with Aurora's "system shadowing" COW mechanism and are unavoidable. Aurora also only supports one outstanding checkpoint at a time for each region resulting in serializing calls from multiple threads even if their working sets are independent. RocksDB avoids contention in Aurora

	Time (µs)	
Operation	**MemSnap**	**Aurora**
Waiting for Calls	N/A	26.7
Applying COW	5.1	79.8
Flush IO	46.3	27.9
Removing COW	N/A	91.7
Total	**51.4**	**208.1**

Table 10. Breakdown of MemSnap vs Aurora's persistence cost. Aurora's region COW tracking causes 80% of its total latency. MemSnap's page tracking avoids these overheads. MemSnap also avoids contention between concurrent persistence operations compared to Aurora.
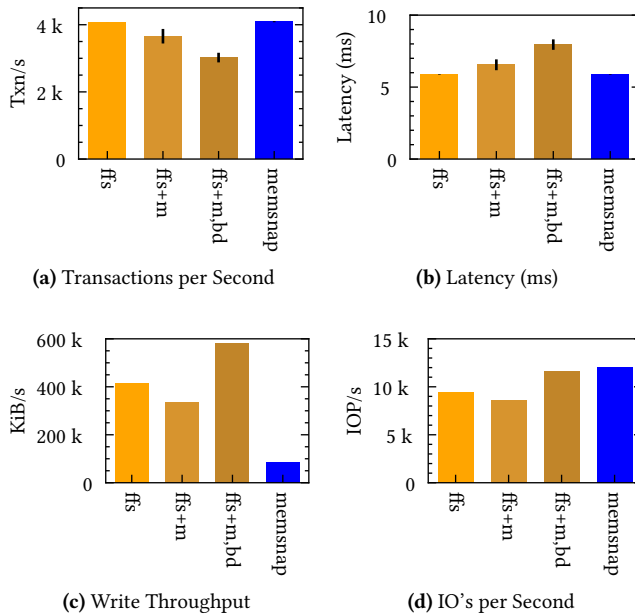

(a) Transactions per Second

(b) Latency (ms)

(c) Write Throughput

(d) IO's per Second

Figure 6. Performance of four variations of PostgreSQL running a TPC-C benchmark: Baseline FFS, FFS with `mmap`'d files, FFS with `mmap` and also directly writing data instead of using the buffer cache (i.e., bufdirect, labeled "bd"), and MemSnap.

by also taking advantage of flat-combining but still experiences an average of 26.7 µs in stall time per checkpoint.

7.3 Case Study: PostgreSQL

We evaluate the effectiveness of MemSnap with PostgreSQL, a widely used database with over 25 years of development. PostgreSQL is highly configurable and optimized and is unique among our case studies as it is a multiprocess application with a complex buffer cache.

The buffer cache is a shared memory region, which consists of 8 KiB blocks that are associated with a file and offset. Each buffer is described by a buffer descriptor, which holds metadata and a logical pointer to the file block. The buffer

cache manager persists buffers through an OS agnostic file interface that uses OS-specific persistence calls (e.g., `fsync`).

For MemSnap integration we must convert the underlying file interface to MemSnap regions, and modify the buffer cache to point directly to these regions. This removes the need for PostgreSQL to first write to the WAL and stage writes in its volatile buffer cache region.

We now show the changes required for the storage engine to satisfy MemSnap's three properties, and how these changes retain the ACID guarantees of PostgreSQL.

①To uphold the first property we replace files with Mem-Snap regions. We modify PostgreSQL's file module and file data structure to include a MemSnap region. We further modify read and write to perform memory copies to the region versus using file related system calls. We then change Post-greSQL's buffer cache to directly point to these new memory regions instead of using its volatile shared memory.

②Page isolation between writes is satisfied within Post-greSQL due to locking semantics and PostgreSQL's use of MVCC semantics. MVCC means PostgreSQL never directly modifies values and instead appends changes to the buffer. Changes that are a part of an uncommitted transaction can be flushed without risking data corruption, allowing a transaction to commit other transactions' writes safely. This means MemSnap can persist buffer pages even if another transaction thread is currently still working on them.

③PostgreSQL upholds the key modification property due to the same MVCC mechanism that upholds property ②. Key data is never directly modified by a transaction and always appended, so a transactions' uncommitted writes cannot be overwritten by subsequent transactions.

We disable the appending of data to the WAL, by disabling the "full page writes" configuration option within PostgreSQL. As described in Section 3 our on-disk COW mechanism allows us to safely write directly to MemSnap regions, which represent table data, to ensure crash consistency in the event of a power failure. In case of a crash we use the last persisted copy of the region in the store.

Recovery PostgreSQL starts by fetching files required to initialize the database. Our changes modify the file module to instead use MemSnap related calls. MemSnap persisted regions are named after the same path used by PostgreSQL.

After restoring, MemSnap-PostgreSQL maps all MemSnap regions to their original fixed memory address. PostgreSQL then runs its existing recovery logic, since the difference between regions and files is transparent to the upper layers.

Evaluation We evaluate PostgreSQL by using the sysbench TPC-C benchmark, a 50% write, online transaction processing (OLTP) benchmark. The benchmark is scaled to 150 warehouses (around 30 GiB). The TPC-C benchmark was run for 2 minutes, run with 24 connections, and all processes were pinned to the first CPU socket to reduce NUMA effects. The sysbench threads were pinned to the other socket.

We configured PostgreSQL using a performance tuning guide [5] and used a 48 GiB buffer cache. Huge pages are disabled, as this resulted in a performance decrease, likely due to the random write workload of TPCC [2].

To better show MemSnap's performance improvements, we demonstrate that modifying PostgreSQL's data in place using existing OS APIs for persistence incurs significant performance penalties. We do this with two additional systems: the first ("ffs-mmap"), which directly maps table data and second ("ffs-mmap-bufdirect"), which uses directly mapped table data and allows for the direct modification of the data over using buffer pages. Through these intermediate variations we show data that corroborates the historical observation that directly mapping data incurs performance penalties [17], in contrast to a buffer cache.

Our last system replaces all mmap persistence API calls with MemSnap calls ("ffs-mmap-bufdirect"). We see in Figure 6 that rather then incurring a 25% decrease in transactions per second, we see a 1.5% increase. This can be attributed to an overall reduced write throughput and the lower latency of MemSnap.

As seen in Figure 6, the overall disk throughput as reported by disk statistics during the running of the benchmark was reduced by 80% when compared to the baseline for MemSnap. This reduction is due to two reasons: First, the block size by default in PostgreSQL is 8 KiB, resulting in larger amounts of write amplification for smaller writes to the WAL and when checkpointing. Second, checkpointing flushes of buffers in the buffer cache are no longer required as MemSnap modifies data in place further reducing write amplification. For example, a 4 KiB dirty page within standard PostgreSQL can result in 16 KiBs of writes. The 16 KiBs comes from the block being written to the WAL and then to the file directly.

There is an increase in overall IO per second by 26% for MemSnap. This is because MemSnap requires creating an IO for every table object modified during every transaction. The baseline aggregates writes into one IO to the WAL. Buffer flushes occur more rarely.

A major advantage MemSnap brings to PostgreSQL is a simplification of the storage stack. For example, PostgreSQL's buffer/file subsystems account for over 10 KSLOC not including the hundreds of various call sites, and logic within the other subsystems. Furthermore, data is no longer managed in four separate forms: the userspace buffer cache, the kernel buffer cache, the WAL, and all file objects. Rather, the storage stack can be collapsed into in memory versions of tables with serialization points at transaction commits, greatly reducing code and likelihood of bugs.

Furthermore, the MemSnap system is the bottom end of what is achievable with better kernel supplied virtual memory abstractions. Further optimizations would require significant engineering effort as PostgreSQL is a highly complex system, with a storage backend of 600 KSLOC.

An example is the transition states required by PostgreSQL to notify users when a buffer is currently being read in. These states and surrounding policy is no longer required as paging itself by the OS is able to perform this task transparently.

8 Related Work

Checkpointing systems like Remus [18] use continuous differential checkpointing for transparent fault tolerance of VMs. Project PBerry [13] eliminates the overheads of page-level dirty set tracking and can benefit many checkpointing systems but requires specialized hardware.

File system extensions [22–24, 35, 38, 42, 46, 49] extend file semantics or extend buffer cache usability, however are designed solely within the storage subsystems. Other systems focus solely on improving virtual memory systems or paging mechanisms [11, 29, 32, 33, 43].

Numerous works [21, 28, 34, 36, 47] attempt to eliminate persistence bugs by reducing the complexity associated with transactional systems and databases. These works demonstrate the difficulty of building persistence as an application developer, and motivate our work on MemSnap.

Some systems remove the overheads of persistence by delaying the externalization of side effects until commit time [15, 30, 31]. Rethink the sync [31] uses this to speed up operations like file moves that require multiple commit points. These techniques are used in SLSes to gain throughput at the cost of extra operation latency.

Recent key-value stores have novel architectures aimed towards high throughput, random access-friendly SSDs. The Kvell KV store [25] uses a shared-nothing architecture and avoids sorting data on disk to minimize CPU usage and maximize throughput. Kvell+ [26] expands this system to optimize long-lived operations like scans. SplinterDB [16] minimizes the CPU and IO write amplification costs of compaction for small keys using STB$^\varepsilon$ trees to reduce writes during compaction. MemSnap is compatible with these systems as it only changes applications' persistence-related logic and leaves operations like range scans and compactions unmodified.

9 Conclusion

In this paper we presented MemSnap, a system for efficient persistence using an architecture inspired by single level stores. MemSnap provides clear persistence semantics, allowing users to directly modify their main dataset in memory, reducing code complexity while outperforming both WAL-and-checkpoint and application checkpointing techniques. We introduce our unified COW mechanism, which is necessary to perform atomic µCheckpoints at sub-millisecond latency. Our experiments show that MemSnap has a 14% overhead over direct disk IO, compared to file APIs that are 43× slower. We also show speed ups in databases like RocksDB of 8.5% and PostgreSQL of 1.5%, while reducing code complexity by up to 40%.

A Artifact Appendix

A.1 Abstract

The artifact contains the MemSnap code and the benchmarks presented in this paper. The artifact includes modified versions of RocksDB and PostgreSQL, and a SQLite module to be used with an unmodified SQLite codebase. The MemSnap code runs on any hardware that can run a FreeBSD 12.3 instance, and the benchmarks assume low-latency SSD storage. The artifact contains the MemSnap kernel module, configuration scripts to set up and run the benchmarks, and graphing scripts to reproduce the paper tables and figures.

A.2 Artifact Checklist (Meta-Information)

- **Compilation:** LLVM C and C++ compilers
- **Run-time environment:** FreeBSD 12.3
- **Hardware:** Dual Intel Xeon Silver 4116 CPUs (Skylake-SP) 2.1 GHz, 96 GiB of memory. Two Intel 900P PCIe SSDs.
- **Output:** Stored in data directories in each database. Output is PNG files or LaTeX tables printed to the command line.
- **Experiments:** File system microbenchmarks, SQLite dbbench and TATP, RocksDB dbbench, Postgres TPC-C.
- **How much disk space required (approximately)?:** 50GiB for root, 96 GiB for data
- **How much time is needed to prepare workflow (approximately)?:** Setting up the FreeBSD 12.3 instance and compiling the benchmarks: 2 hours
- **How much time is needed to complete experiments (approximately)?:** Microbenchmarks (10 minutes), SQLite (2 hours), RocksDB (30 minutes), Postgres (5 hours)
- **Publicly available?:** On Zenodo. A development version is available on https://www.github.com/etsal/memsnap-artifact.
- **Code licenses (if publicly available)?:** BSD 3-Clause

A.3 Description

A.3.1 How to access. The artifact can be accessed on Zenodo (https://doi.org/10.5281/zenodo.10864510).

A.3.2 Software dependencies. FreeBSD 12.3, SQLite 3.41.

A.4 Installation, Experiment Worklow

Download the repository from Github and follow the instructions outlined in the README. The README provides a walkthrough on how to set up the system, generate the numbers, and reproduce the graphs.

A.5 Evaluation and expected results

The supplied scripts within the provided repository can be used to fully regenerate all graphs and figures in the paper.

References

[1] PostgreSQL's fsync() suprise. https://lwn.net/Articles/752063/, April 2018.

[2] PostgreSQL and Huge Pages. https://wiki.postgresql.org/images/7/7d/PostgreSQL_and_Huge_pages_-_PGConf.2019.pdf, 2019.

[3] PostgreSQL's fsync() surprise. https://lwn.net/Articles/752063, January 2021.

[4] bcachefs: The COW filesystem for Linux that won't eat your data. https://www.sqlite.org/wal.html, August 2023.

[5] Chapter 14. Performance Tips. https://www.postgresql.org/docs/current/performance-tips.html, August 2023.

[6] RocksDB | A persistent key-value store. https://www.rocksdb.org, April 2023.

[7] SQLite: Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html, August 2023.

[8] TATP Benchmark. https://tatpbenchmark.sourceforge.net/TATP_Description.pdf, August 2023.

[9] Welcome to BTRFS Documentation. https://btrfs.readthedocs.io, August 2023.

[10] Write-Ahead Logging. https://www.sqlite.org/wal.html, August 2023.

[11] Chloe Alverti, Vasileios Karakostas, Nikhita Kunati, Georgios Goumas, and Michael Swift. Daxvm: Stressing the limits of memory as a file interface. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 369–387, 2022. https://doi.org/10.1109/MICRO56248.2022.00037.

[12] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. 215, 2003.

[13] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. Project pberry: Fpga acceleration for remote memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 127–135, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3317550.3321424.

[14] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association. https://dlnext.acm.org/doi/abs/10.5555/3386691.3386712.

[15] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 228–243, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2517349.2522726.

[16] 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. USENIX Association, July 2020.

[17] Andrew Crotty, Viktor Leis, and Andrew Pavlo. Are you sure you want to use mmap in your database management system. In *CIDR 2022, Conference on Innovative Data Systems Research*, 2022.

[18] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 161–174, USA, 2008. USENIX Association. https://dl.acm.org/doi/10.5555/1387589.1387601.

[19] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. Sqlite: Past, present, and future. *Proc. VLDB Endow.*, 15(12):3535–3547, aug 2022.

[20] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2):127–153, may 2000.

https://doi.org/10.1145/350853.350863.

[21] 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. https://doi.org/10.1145/3318159.

[22] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: Write-optimization in a kernel file system. *ACM Trans. Storage*, 11(4), nov 2015. https://doi.org/10.1145/2798729.

[23] Dong Hyun Kang, Changwoo Min, Sang-Won Lee, and Young Ik Eom. Making application-level crash consistency practical on flash storage. *IEEE Transactions on Parallel and Distributed Systems*, 31(5):1009–1020, 2020. https://doi.org/10.1109/TPDS.2019.2959305.

[24] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-memory assisted buffer management. *Proc. ACM Manag. Data*, 1(1), may 2023. https://doi.org/10.1145/3588687.

[25] 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, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell+: Snapshot isolation without snapshots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 425–441. USENIX Association, November 2020.

[27] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, aug 1984. https://doi.org/10.1145/989.990.

[28] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 221–234, USA, 2015. USENIX Association. https://dl.acm.org/doi/10.5555/2813767.2813784.

[29] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, dec 2003. https://doi.org/10.1145/844128.844138.

[30] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 308–318, New York, NY, USA, 2008. Association for Computing Machinery. https://doi.org/10.1145/1346281.1346321.

[31] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Trans. Comput. Syst.*, 26(3), sep 2008. https://doi.org/10.1145/1394441.1394442.

[32] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 490–502, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3267809.3267824.

[33] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped i/o for fast storage devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association. https://dl.acm.org/doi/abs/10.5555/3489146.3489202.

[34] Daejun Park and Dongkun Shin. Stackable transactional file system using kernel-level wal. *IEEE Access*, 10:110088–110099, 2022. https://doi.org/10.1109/ACCESS.2022.3214521.

[35] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 225–238, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2465351.2465374.

[36] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application crash consistency and performance with ccfs. *ACM Trans. Storage*, 13(3), sep 2017. https://doi.org/10.1145/3119897.

[37] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 433–448, USA, 2014. USENIX Association. https://dl.acm.org/doi/10.5555/2685048.2685082.

[38] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 161–176, New York, NY, USA, 2009. Association for Computing Machinery. https://doi.org/10.1145/1629575.1629591.

[39] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can applications recover from fsync failures? *ACM Trans. Storage*, 17(2), jun 2021. https://doi.org/10.1145/3450338.

[40] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3), aug 2013.

[41] Jonathan S. Shapiro and Jonathan Adams. Design evolution of the eros single-level store. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, page 59–72, USA, 2002. USENIX Association. https://dl.acm.org/doi/10.5555/647057.713855.

[42] Kai Shen, Stan Park, and Meng Zhu. Journaling of journal is (almost) free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, page 287–293, USA, 2014. USENIX Association. https://dl.acm.org/doi/10.5555/2591305.2591333.

[43] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1093–1108, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3373376.3378493.

[44] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora operating system: Revisiting the single level store. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 136–143, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3458336.3465285.

[45] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora single level store operating system. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 788–803, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3477132.3483563.

[46] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Mannarswamy, Terence Kelly, and Charles B. Morrey. Failure-atomic updates of application data in a linux file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 203–211, USA, 2015. USENIX Association. https://dl.acm.org/doi/10.5555/2750482.2750498.

[47] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled io stack for flash storage. In *Proceedings of the 16th USENIX Conference on File*

and Storage Technologies, FAST'18, page 211–226, USA, 2018. USENIX Association. https://dl.acm.org/doi/10.5555/3189759.3189779.

[48] Fangnuo Wu, Mingkai Dong, Gequan Mo, and Haibo Chen. Treesls: A whole-system persistent microkernel with tree-structured state checkpoint on nvm. In 29th ACM Symposium on Operating Systems Principles (SOSP 23). USENIX Association, 2023.

[49] Yang Zhan, Alex Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. Copy-on-abundant-write

for nimble file system clones. ACM Trans. Storage, 17(1), jan 2021. https://doi.org/10.1145/3423495.

[50] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, page 449–464, USA, 2014. USENIX Association. https://dl.acm.org/doi/10.5555/2685048.2685083.