DStore: A Fast, Tailless, and Quiescent-Free Object Store for PMEM

Shashank Gugnani
The Ohio State University
gugnani.2@osu.edu

Xiaoyi Lu
University of California, Merced
xiaoyi.lu@ucmerced.edu

ABSTRACT

The advent of fast, byte-addressable persistent memory (PMEM) has fueled a renaissance in re-evaluating storage system design. Unfortunately, prior work has been unable to provide both consistent and fast performance because they rely on traditional cached or uncached approaches to system design, compromising at least one of the requirements. This paper presents DStore, a fast, tailless, and quiescent-free object store for non-volatile memory. To fulfill all three requirements, we propose a novel two-level approach, called DIPPER, which fully decouples the volatile frontend and persistent backend by leveraging the byte addressability and performance of PMEM. The novelty of our approach is in allowing the frontend and backend to operate independently and in parallel without affecting crash consistency. This not only avoids the need to quiesce the system but also allows for increased concurrency in the frontend through the use of observational equivalency. Using this approach, DStore achieves optimal scalability and low latency without compromising on crash consistency. Evaluation on Intel's Optane DC Persistent Memory Module (DCPMM) demonstrates that DStore can simultaneously provide fast performance, uninterrupted service, and low tail latency. Moreover, DStore can deliver up to 6x lower tail latency service level objectives (SLO) and up to 5x higher throughput SLO compared to state-of-the-art PMEM optimized systems.

CCS CONCEPTS

• **Information systems** → **Storage class memory**; **Cloud based storage**.

KEYWORDS

Decoupled Persistence, PMEM, Object Store

ACM Reference Format:
Shashank Gugnani and Xiaoyi Lu. 2021. DStore: A Fast, Tailless, and Quiescent-Free Object Store for PMEM. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21), June 22–25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3431379.3460649

1 INTRODUCTION

The emerging persistent memory (PMEM) technology offers unprecedented high performance while supporting data persistence, and has fueled a renaissance in re-evaluating the design of persistent storage systems [8, 12, 15–17, 20, 23, 24, 51, 62, 64]. Considering the foreseeable trend of deploying PMEM in data centers, this is high time to rethink the crucial features and design space of PMEM-aware storage engines in modern I/O intensive systems and applications, particularly for the applications in a cloud setting. Hence, this paper takes up the challenge of designing a storage system which is simultaneously fast, tailless, and quiescent-free.

Providing these three features is essential for predictable and consistent performance, which is important in satisfying latency and throughput service level objectives [38, 58]. Inconsistent user experience has been shown to directly result in loss of revenue [29]. Several enterprise storage workloads have been shown to be read-heavy [3, 9, 32]. Despite the small percentage of writes, write operations significantly impact the overall system tail latency and throughput. Our intention, in this paper, is to lower the impact of write operations by hiding their persistence overhead, particularly for ready-intensive HPC and cloud workloads.

Most file and database systems cache important data and employ a journal or write-ahead log (WAL) to support fault-tolerance. Numerous studies [1, 6, 7, 19, 23, 36, 44, 49, 60, 67, 71] have focused on providing PMEM-aware data structures and logging schemes to reduce latency and guarantee failure recovery. These designs can indeed provide much better performance than SSD- or disk-based schemes, but they are unable to provide a performant quiescent-free storage system, which means that users' requests in the frontend may have to wait for completion of data persistence activities in the backend, particularly during checkpoints. The reason for this is that the cache must be write-protected during checkpoints to ensure that the persistent backend is updated in a consistent manner. This limitation results in significant delay for requests arriving during checkpoints. As a result, the system must either quiesce for a short time or accept high tail latency.

On the other hand, the storage and memory like nature of PMEM has spawned a new class of storage systems that place and access data in-place [7, 12, 15, 16, 19, 20, 30, 51, 60–62, 64, 71]. Such systems unburden themselves from the limitations of checkpoints since data is always persistent. However, the challenge in designing such systems is that updates to PMEM must be done in a crash-consistent manner. This is further complicated by the fact that data in CPU caches is not persistent and cache lines can be evicted implicitly. Ensuring consistency requires explicit cache flushes and store fences, while ensuring atomicity requires the use of transactions or journaling. Any practical solution on PMEM must deal with both atomicity and consistency issues. This requires expensive, and often complex, protocols to be used which significantly lower end-to-end performance [22, 47]. The high cost of atomic data persistence compromises performance.

In this paper, we propose a new approach for storage system design, called **D**ecoupled, **I**n-memory, and **P**arallel **PER**sistence (DIPPER) that exploits the byte addressability of PMEM to efficiently achieve the decoupling of system and checkpoint spaces. The system space to store data structures is entirely in DRAM while the checkpoint space (including an operation log) is entirely in PMEM. The novelty of DIPPER is in its fully decoupled architecture. The key idea is to let the frontend operate independently on DRAM while recording logical operations in the log and applying these updates to the backend asynchronously on an identical backend in PMEM. Operations need not wait for the updates to be applied to the backend to be considered durable. DIPPER ensures that the log replay is deterministic, so the checkpoint space can be updated in the background without involvement of the system space. The frontend and backend are kept consistent by applying the concept of observational equivalence [54]. The goal is to hide the latency of persistence by using the volatile system space for all requests and taking checkpoints in the background. This approach solves the limitations of prior work – By keeping the frontend in fast DRAM, and only recording operations in a log, the cost of persistence is significantly lowered. In addition, the decoupled persistence process prevents the need to quiesce the system and can deliver low tail latency.

DIPPER can be used to design fast and crash-consistent storage systems with PMEM. Our approach uses PMEM to store identical persistent shadow copies of DRAM structures. We use shadow updates for backend atomicity to avoid costly transactions and cache flushes. This process is seamlessly handled by our PMEM allocator. In this manner, the same code can be used to perform operations on both structures and the need to serialize data is avoided. This also simplifies the backend design and prevents the need to hand-modify code to add cache flushes and transactions. Our design not only increases productivity, but also makes the process of keeping the volatile and persistent copies consistent much easier. PMEM bandwidth (~30GB/s for read and ~10GB/s for write on our cluster) is comparable to DRAM, which implies that the checkpoint space can keep up with the system space.

We design a generic storage sub-system, called **DStore**, short for Decoupled Store which uses DIPPER to implement its control plane. We deploy DStore on a server with Intel Optane DCPMM. With the aforementioned techniques, DStore can reduce software overhead to ~10%. Experimental results demonstrate that DStore can deliver up to 6x lower tail latency service level objectives (SLO) and up to 5x higher throughput SLO compared to state-of-the-art PMEM optimized systems like PMEM-RocksDB [52, 63], MongoDB-PMSE [28], and NOVA [64, 65].

To summarize, we make the following contributions:

- DIPPER, a novel decoupled approach which utilizes the advantages of PMEM to provide low-overhead persistence and quiescent-free checkpoints (§3)
- DStore, a high-performance userspace storage sub-system designed using DIPPER and its quiescent-free checkpoint architecture (§4)
- Extensive evaluation clearly demonstrating that DStore outperforms state-of-the-art PMEM optimized systems on several metrics (§5)

The rest of this paper is organized as follows. §2 describes background and motivation. §3 presents the architecture and design of DIPPER. §4 discusses the design and implementation of DStore. §5 presents a comprehensive experimental analysis of our design, §6 discusses related work and finally, §7 concludes the paper.

2 BACKGROUND AND MOTIVATION

PMEM allows applications to use standard *load* and *store* instructions to access data that is persistent across power cycles. PMEM has three orders of magnitude better latency and an order of magnitude better bandwidth as compared to SSD [45, 66]. It is evident that PMEM is a practical option for compact logical logs. Given that PMEM is an emerging technology and its cost is much higher than SSD [35], we do not expect PMEM to replace SSD as primary storage media in the foreseeable future. Instead, we expect that PMEM available in next generation systems will be utilized for special purposes, such as latency critical logging and metadata storage.

Despite its performance benefits, PMEM is hard to program with. Accessing persistent data in-place means that any update must be done in a crash-consistent manner. This is further complicated by the fact that data in CPU caches is not persistent, atomicity of writes is only at 8B granularity, and cache lines can be evicted implicitly. Ensuring consistency requires explicit cache flushes and store fences, while ensuring atomicity requires the use of transactions or journaling. Both atomicity and consistency issues must be dealt with for any practical solution, which requires non-trivial effort.

2.1 Limitations of Existing Approaches

In this section, we discuss why existing systems are unable to provide low tail latency, fast performance, and quiescent freedom simultaneously. Storage systems can be broadly classified into three types – cached, uncached, and decoupled.

In a cached system, a volatile cache is used to improve performance by caching a part of the persistent data. This cache is tightly coupled with the persistent backend, i.e., the cache is needed to update the backend. A logical or physical log is used to ensure atomicity and durability of operations. Traditional systems used physical logging which suffered from the large size of log records. To improve logging performance, recent systems [24, 34, 42, 51, 55, 64, 65] have moved to logical or operation logging. Usually, both the cache and the log have limited space. Therefore, checkpoints are necessary to cleanup space in the log, or cache, or both. Herein lies the major problem with cached systems. Data from the cache is used to update the persistent backend and since the backend must be updated atomically, the cached pages cannot be modified until they are made persistent. As a result, during checkpoints the system either becomes temporarily unavailable or clients experience intolerable delay. This compromises tail latency or quiescent freedom, and sometimes both.

System	Persistence Technique	Type	Low Tail Latency	Fast Performance	Quiescent Freedom
MongoDB-PM [46], sqlite [51]	Periodic Async Checkpoint	Cached	✗	✗	✓
PMEM-RocksDB [52, 63]	Continuous Async Checkpoint	Cached	✗	✗	✗
NOVA [65], Pronto [44]	Copy on Write (CoW)	Cached	✗	✓	✓
MongoDB-PMSE [28]	Inline Persistence	Uncached	✓	✗	✓
DudeTM [39], NV-HTM [4]	Decoupled Durability + HTM	Decoupled	✓	✗	✓
DStore (proposed)	Parallel Decoupled Checkpoint	Decoupled	✓	✓	✓

Table 1: Comparison of related work

To demonstrate the weaknesses of cached systems, we conduct an experiment to compare the performance of two popular PMEM-optimized NoSQL storage engines, PMEM-RocksDB [52, 63] and MongoDB-PM (WiredTiger) [46] with and without checkpoints. PMEM-RocksDB is an optimized version of the log-structured merge (LSM) tree-based vanilla RocksDB [18] and uses a PMEM resident log to improve performance. MongoDB-PM uses an optimized WiredTiger engine with the index and journal placed in a DAX filesystem formatted on PMEM. We also implement the copy-on-write (CoW) checkpoint scheme used in NOVA [65] and Pronto [44] in DStore and perform the same experiment. We compare the tail latency of writes for a full-subscription (28 core) 50% read, 50% write workload. Figure 1 shows the result of this analysis[1]. We observe that by disabling checkpoints, all systems show lower tail latency, particularly for p999 and p9999. PMEM-RocksDB uses an LSM tree with level 0 placed in DRAM. Therefore, during checkpoints, the level 0 files must be locked until they have been compacted and merged into the next level. Similarly, MongoDB-PM uses a btree with a DRAM-backed page cache. On checkpoint, the page cache is locked until all pages are made durable. The need to lock the frontend results in significant delay for requests arriving during checkpoints and consequently high tail latency. The CoW checkpoint scheme has similar drawbacks. When cache space needs to be cleared, volatile pages are marked as read only. When a client tries to modify a read-only page, a page fault is triggered and a handler copies the page to PMEM. Clients must wait until the page is copied before making any modification which increases tail latency.

In an uncached system, all data is immediately persistent and nothing is cached. To ensure atomic updates, transactions are used. Since data is immediately persistent, checkpoints are not required. While uncached systems were considered impractical with disks and SSDs because of their slow write performance, they are gaining popularity in recent times due to the advent of PMEM. The fast performance of PMEM makes uncached systems practical in production scenarios. Unfortunately, the overhead of transactions to atomically update data in PMEM is too high, resulting in performance being compromised. The main reason for this, as mentioned earlier, is that the CPU caches are not persistent and cache lines can be implicitly evicted. To ensure correct ordering of updates, explicit cache flushes and store fences are needed. By avoiding the need for checkpoints, uncached systems can attain low tail latency and quiescent freedom but at the cost of performance. The overhead of transactions and cache flushes have been well studied [22, 47].

Table 1 presents a comparison of related work with DStore. Note that none of the existing PMEM systems are able to satisfy all desirable characteristics. This paper proposes a new decoupled approach to storage system design, which can provide all desired features. By keeping the frontend in DRAM, our design achieves fast performance. In addition, by effectively decoupling the frontend and backend and allowing them to operate in parallel, our design provides tailless and quiescent-free performance. The next section discusses the proposed design in more detail.

Like DStore, DudeTM [39] and NV-HTM [4] are two systems for PMEM that have explored a decoupled approach. These systems rely on expensive physical logging, as opposed to the more efficient logical logging used in DStore. Further, they require special hardware support for consistency – hardware transactional memory (HTM). Therefore, we believe that these systems are neither performant nor widely applicable. Like DStore, Bullet [24] separates volatile and persistent domains and proposes a cross-referencing technique to keep the two consistent. However, Bullet requires both the key and value to be added to the log. In contrast, DStore's design allows values to be omitted from the log and be placed only in persistent storage. Furthermore, DStore's DIPPER is more generic than Bullet's cross-referencing logs in that it can be applied to any storage system and not just key-value stores. This is because Bullet uses per-thread logging and adds cross references between the logs to track operation dependencies. This is practical for key-value based data structures but becomes non-trivial for other data structures, where complex dependencies may exist between operations, making the process of cross referencing impractical and slow. Finally, Bullet requires complex transactions to update its backend but DStore avoids this by using shadow updates which are seamlessly handled by its memory allocator[2]. Persimmon [69] is another system which proposes a decoupled design, however it does not provide any concurrency.

3 DIPPER DESIGN

Motivated by the above observations, this paper proposes a new approach, called **D**ecoupled, **I**n-memory, and **P**arallel **PER**sistence (DIPPER) that *fully decouples* the frontend and backend. We compare DIPPER with existing approaches in Table 1.

[1]Note that DStore with DIPPER does not suffer from the tail latency overhead of checkpoints

[2]We were unable to compare with Bullet since its source code is unavailable

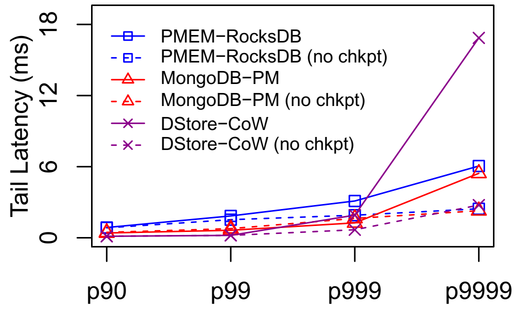


Figure 1: Tail latency overhead of checkpoints

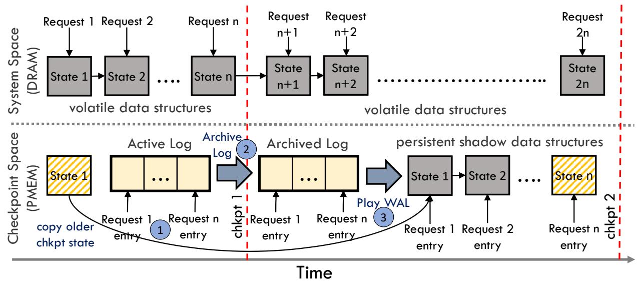


Figure 2: DIPPER and its atomic quiescent-free checkpoint architecture. Pattern filled states represent a checkpoint.

3.1 Key Ideas

We present DIPPER as an approach on PMEM to make a set of DRAM data structures persistent by only logging information necessary to perform an operation on this set. DIPPER generates a linear order for all operations, preserved in the log, and operates deterministically on the data structures, according to this linear order. In case of recovery from failure, this linear order, as preserved in the log, can be easily recovered and the same scheme can be used to reconstruct the state of DRAM data structures by treating all operations in the log as new requests. To be applicable in practical scenarios, DIPPER must solve two challenges. First, deterministic operation on data structures should not pose as a scalability bottleneck. Second, DIPPER must provide an atomic quiescent-free checkpoint solution to ensure uninterrupted service to end-users.

To solve these challenges, DIPPER fully decouples volatile system space and persistent checkpoint space both logically and physically. The system space to store data structures is entirely in DRAM while the checkpoint space (including the log) is entirely in PMEM. When a checkpoint is triggered, the checkpoint space is updated using the operations recorded in the log. Since the log replay is deterministic, the checkpoint space can be updated in the background without affecting the system space. The main idea is to hide the latency of persistence by using the volatile system space for all requests and taking checkpoints in the background. In this manner, requests operate directly on the volatile version and experience DRAM-like latency, though write operations will experience a minor overhead for operation logging. The expectation is that the rate of write requests is low enough that the persistent shadow copy can be updated quickly and kept consistent with the volatile version. This is generally true since most requests arriving at storage systems are read requests [3, 9, 32]. Furthermore, write rates vary hugely, with some periods of low activity and some periods of bursty traffic [32, 33]. The volatile frontend can absorb bursty traffic easily and the persistent backend can then be updated during the periods of low activity in the background.

Our approach reduces the size of each log record, since only high-level operations and their parameters need to be logged. This reduces the log fill up rate. Further, the high bandwidth of PMEM lowers checkpoint cost and improves the rate of log clean up. DIPPER works with a PMEM backend and not SSD or disk backend, because only then is the rate of log fill up lower than the rate of the log cleanup (checkpoint). Thus, the checkpoint process can be completely overlapped with system operation.

The background checkpoint process does not compromise crash consistency for two reasons. First, the system logs all updates and log records are not discarded until a checkpoint is complete. Second, the checkpoint process is designed to be completely atomic. In this manner, we achieve quiescent-free checkpoints while maintaining fault-tolerance.

3.2 Architecture and Abstraction

Figure 2 shows the architecture of DIPPER. Three simple steps are involved in DIPPER: ① log all logical operations in a persistent log, ② archive the log when it is full, and ③ update backend by applying operations in the archived log. Note that frontend operation can proceed in parallel to ③. DIPPER relies on the concept of observational equivalence to allow concurrency while guaranteeing determinism and correctness (see §3.7). DIPPER treats the set of DRAM data structures as a black box, logging only logical operations performed on this box and the input required from outside the box. To achieve fault-tolerance, DIPPER proposes a novel abstraction of shadow persistent data structures (or simply shadow copies) in PMEM, that represent a shadow copy of the volatile system space located in the persistent checkpoint space. We call these shadow copies because their state lags behind their volatile counterparts. These represent a snapshot of the system at a particular point in time and are only used for recovery in case of failure. During normal operation, the system operates purely on DRAM data structures while logging operations in the log. Each logical operation translates to a set of functions to be performed on each data structure. This mapping needs to be statically defined as it will be used by the recovery logic to update the shadow copies. During background checkpoints, taken in parallel to frontend operation, the operations in the log are applied to the persistent backend.

DIPPER is a derivative of logical logging. However, DIPPER differs from prior algorithms utilizing logical logging in one critical aspect. DIPPER leverages its determinism property along with byte addressable PMEM to decouple normal system operation from checkpointing. This allows DIPPER to provide fault-tolerance while allowing requests to only operate on DRAM data structures. This is in stark contrast to other systems [23, 44, 46, 52] that only cache pages in memory that are eventually persisted.

3.3 Memory Management

Memory management is an important component of our design. We delegate several important functions to the memory allocator

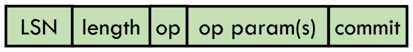


Figure 3: DIPPER log record: *LSN* is log sequence number, *length* is length of log record, *op* is operation type, and *commit* is committed flag. The generic nature of the record allows any arbitrary operation to be captured in the log.

to make it easy for DIPPER to be applied to any data structure. Our backend design uses shadow updates for atomicity, so the PMEM allocator need not be crash consistent itself. This means that DIPPER can work with any off-the-shelf DRAM allocator. The same allocator can be used for both DRAM and PMEM management. Keeping both allocator designs the same makes it easier to reconstruct the volatile space from the persistent space in the event of a crash. We expect the allocator to implement two additional functions – one to iterate over all allocated memory regions and flush them to PMEM and the other to create a copy of the allocator state. The first function is used to ensure durability at the end of a checkpoint. The second function is used to for two purposes. The first is to avoid persistent memory leaks. During a checkpoint, a copy of the PMEM allocator is created along with copies of other data structures for recovery in the event of a crash. The second is to recover volatile space from the persistent space after a crash. In addition, to allow the data structures to be seamlessly copied and work in spite of PMEM address space relocation, we use relative pointers and pointer swizzling [11, 56] for both DRAM and PMEM structures. This means that we store offsets to the base address instead of pointers. On each pointer de-reference, the base address is added to the offset to obtain the actual pointer to data.

3.4 Logging on PMEM

DIPPER relies on a PMEM resident log to record all processed operations. We exploit the byte addressability of PMEM to access log records in-place. Figure 3 shows the structure of a DIPPER log record. We capture each *operation* and its parameters within the log record. The log sequence number (LSN) is used to verify the validity of log records. Consequentially, the LSN must be persisted atomically. With PMEM, only single word writes (usually 8B) are atomic. Furthermore, spurious cache line evictions can change the order in which portions of a log record are made persistent. To ensure that the LSN is persisted atomically, we flush cache lines containing each log record in the reverse order. We *write* and *flush* the LSN only after all other cache lines in the log record have been persisted. LSN is the first field in the log record. Thus, the log record will only be considered valid once the first cache line containing the LSN is flushed as the last step of the log write. Cache lines are flushed by calling `clflushopt` or `clwb`, followed by a store fence. Note that this is just one possible implementation for the log. DIPPER can work with any log implementation as long as arbitrary operations can be added to the log and records can be written atomically to PMEM.

3.5 Atomic Quiescent-Free Checkpoint

To prevent the log from growing without end, checkpoints are triggered once the free space in the log fall below a pre-defined threshold. As shown in Figure 2, the checkpoint process begins by swapping the active and archived logs (this is fast and only involves a pointer swap), and moving any uncommitted log records to the new active log. Once this process is complete, frontend operations can proceed on DRAM data structures and the checkpoint is processed asynchronously. The checkpoint procedure involves playing *committed* records from the archived log on the shadow copies. A dedicated checkpoint thread pool is responsible for operating on the persistent backend in parallel with the frontend.

We leverage the byte addressability of PMEM to reuse the functions defined for each DRAM data structure operation. In this manner, the shadow copies iterate through the same states that the volatile copies went through. We guarantee durability by flushing all modified cache lines upon completion of the checkpoint process. This is done by iterating over all allocated pages in the PMEM allocator (including the allocator state) and flushing each cache line in the page to PMEM. Once all operations in the log have been processed, the shadow copies will have the same state as the DRAM structures had at the start of a checkpoint. Thus, the state of the shadow copies at the end of a checkpoint represents the checkpoint image. Correctness is guaranteed by leveraging the determinism property of DIPPER. To guarantee idempotency during a checkpoint, we always create a new copy of the shadow copies. In case of a crash during a checkpoint, the recovery logic uses the old persistent versions for recovery. A root object, placed in a well known offset in PMEM contains pointers to current and old copies of the shadow copies as well as the current state of the checkpoint process. Our PMEM allocator is responsible for flushing cache lines to ensure durability and creating copies of backend structures to ensure atomicity.

Finally, to achieve atomicity, we update the locations of shadow copies in the root object atomically and *only* upon successful completion of the checkpoint process. As is evident, checkpoints are processed in the background without significant impact on system throughput. Our evaluation (see §5.3) verifies this claim.

This approach for implementing the backend has several benefits. First, since the representations of the DRAM and PMEM data structures are the same, the same code can be used for both. Further, by using shadow updates to maintain backend atomicity, the cost of consistency is significantly lowered. There is no need to ensure that data structures are updated in a consistent and durable manner for each operation. Therefore, complicated techniques, such as transactions, which are often employed to attain atomicity are not required. This simplifies the implementation of the backend and allows code written for volatile structures to be used as is for the persistent structures.

3.6 Idempotent System Recovery

DStore provides complete crash consistency because it records all operations in a persistent log and it can recover volatile state by replaying these operations from the log. Our recovery protocol guarantees crash safety for both possible failure scenarios – failure during a checkpoint and failure outside a checkpoint. For recovery, the root object provides pointers to the data structures and the state of the checkpoint process. If we detect that the system crashed during a checkpoint, we first need to reconstruct the latest versions of the shadow copies (this step is skipped if the crash was not during a checkpoint). This is done by playing records from the archived log on the old copies of the shadow copies. Essentially, we redo the checkpoint procedure ongoing at the time of crash. This ensures that we operate on a consistent checkpoint image in the next step. Next, we recover the volatile state. This involves replicating the PMEM allocator state in the DRAM allocator and copying pages from PMEM to DRAM. Finally, we replay log records in the active log on the data structures to restore system state to what it was before the crash.

After recovery is complete, the system state is restored and new requests can be accepted. During recovery, we only play *committed* log records on the shadow copies. So, there is no need to log undo operations since the shadow copies represent a consistent checkpoint image (or in database terms, a transaction consistent checkpoint [53]). Hence, DIPPER can be thought of as a *redo-only* logging algorithm. The state of the system is defined exclusively by volatile structures which means that the recovery process is guaranteed to be idempotent.

3.7 Observational Equivalence & Concurrency

Supporting concurrent operations is a must for any practical solution. DIPPER supports concurrency through the notion of *observational equivalence* [54]. Using this notion, we can state that two data structure states are observationally equivalent if they both give the same answer to any observation from a prespecified set. Two operations on a data structure are said to be commutative or non-conflicting if reordering the operations results in observationally equivalent states. Commutative operations are permitted to operate concurrently on a data structure. The use of commutativity for concurrency is not new and has been widely studied [10, 57]. However, applying it to logical logging has been problematic. This is because if the volatile and persistent domains are not fully decoupled, then different portions of the persistent backend must be updated atomically since the log only contains operations and not actual data. Concurrent modification of data structures could lead to an inconsistent backend. Prior work [41] has shown how a write graph can be used to delay the persistence of data objects and increase concurrency. Nevertheless, this solution requires costly maintenance of the write graph state. Further, concurrency is limited when data are being made persistent. DIPPER overcomes these challenges by fully decoupling volatile and persistent domains. The volatile structures do not need to be involved to update the persistent backend. Therefore, commutativity can be fully exploited to increase concurrency. For instance, operations on distinct keys in a hashtable are non-conflicting and can be executed in parallel. The hashtable must, of course, support concurrency and avoid concurrent modification of a single bucket. In DIPPER, the implication is that log records are not required to be in serialized order but only conflicting order. So, not only can a single data structure be updated concurrently, by non-conflicting requests but different data structures can also be operated on in parallel. In case of recovery after a crash the exact representations of the data structures will not be the same as those before crash, but the observable state of the data structures will be the same. This guarantee is sufficient to ensure correctness of the system [54].

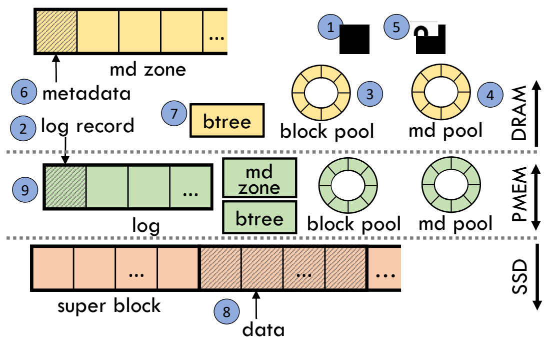


Figure 4: DStore layout and steps followed by a write request

4 DSTORE

In this section, we present the design and implementation of DStore and its integration with DIPPER.

4.1 API and Semantics

To demonstrate the effectiveness and efficiency of DIPPER, we propose a new fast and durable object storage sub-system, called DStore. DStore is designed with the goal of being a generic fault-tolerant embedded storage sub-system.

We do not opt for POSIX compliance so as to avoid the shortcomings of POSIX IO [2, 68]. The growing popularity of simpler cloud services which offer access to objects instead of files needs to be recognized. As a consequence, we propose a new set of APIs to provide scalable access to objects. Table 2 lists the main API for accessing DStore. The API and semantics of DStore have been specifically constructed to handle a wide variety of use cases and requirements. We provide both key-value and filesystem style APIs while storing the data as *objects*. Unlike object stores such as OpenStack Swift [50], DStore treats *objects* as modifiable entities. The primary difference between the key-value and filesystem API is statefulness. The key-value API does not require an *object* handle, releasing DStore of the arduous task of tracking open handles. This allows applications to achieve much higher scalability.

The *oopen*, *oclose*, *oread*, and *owrite* primitives are semantically related to their filesystem counterparts. The *oget* and *oput* primitives are derivatives of the standard kev-value functions, *get* and *set*. Each thread submitting IO needs to initialize a context for submitting requests using *ds_init*. For concurrency control, *olock* and *ounlock* primitives are provided to specify complex dependencies between objects. Concurrency control for a single object is implicitly handled by DStore.

4.2 Data Layout

DStore distributes data and internal structures between DRAM, PMEM, and SSD (see Figure 4). Data are stored purely on SSD because of its capacity and non-volatility. SSD pages are grouped into *blocks* which are the unit of data allocation in DStore. The first block is reserved for the superblock, which contains relevant recovery information about the system. Most importantly, it contains the PMEM root object which is required for recovery in case of failure. A block pool is used to manage the SSD *blocks*. To store metadata pages, a metadata zone is used along with a metadata

API Function	Type	Description
ds_ctx_t ds_init()*	environment	Initialize *context* for application thread
void ds_finalize(ds_ctx_t ctx)*	environment	Terminate *context*
OBJECT oopen(ds_ctx_t* ctx, char* name, size_t size, op_t op)*	filesystem	Open an *object* for reading, writing, or both
void oclose(OBJECT object)*	filesystem	Close an *object*
ssize_t oread(OBJECT object, void* buf, size_t size, off_t offset)*	filesystem	Partial reads on *object*
ssize_t owrite(OBJECT object, void* buf, size_t size, off_t offset)*	filesystem	Partial writes on *object*
ssize_t oget(ds_ctx_t ctx, char* key, void* value)*	key-value	Get *value* for *key*
ssize_t oput(ds_ctx_t ctx, char* key, void* value, size_t size)*	key-value	Set *value* for *key*
int odelete(ds_ctx_t ctx, char* name)*	key-value	Delete *object* or *key*
int olock(ds_ctx_t ctx, char* name)*	concurrency control	Acquire *lock* on *object*
int ounlock(ds_ctx_t ctx, char* name)*	concurrency control	Release *lock* on *object*

Table 2: DStore Interface Overview

pool to allocate free entries in the metadata zone. The metadata and block pools are circular buffers containing free blocks and metadata pages. A PMEM-based log is used for recovery in case of failure. For maintaining an index of objects in the system, we utilize a btree. All data structures including the metadata and btree are stored in DRAM with their shadow copies stored in PMEM. Essentially, DStore implements its control plane using DIPPER, with the frontend in DRAM and backend in PMEM, while the data plane is placed on a high-capacity SSD.

We use a DAX filesystem formatted on PMEM for space management. We directly map a part of PMEM into the address space of the system by using `mmap` on a file located on the filesystem. To allocate memory for shadow copies, we use a simple slab-based memory allocator. This allocator uses the `mmap`ed memory and creates slabs in different size classes that are a power of two. For DRAM management, we use a similar slab-based allocator but with slabs allocated from the volatile heap.

DStore does not utilize a write cache, but writes directly to the internal DRAM-based write cache in SSDs, providing significant data transfer time savings. We find that SSDs with internal DRAM have enhanced power-loss data protection [27]. In the event of power failure, device capacitors will assist in flushing write cache data to non-volatile storage. DStore transparently leverages device capacitance to reduce the overhead of crash consistency.

4.3 Exploiting DIPPER in DStore

DStore uses DIPPER to make all DRAM structures shown in Figure 4 persistent. Data are updated *in-place* and are thus omitted from the log. We write log records for *oopen*, *owrite*, *oput*, and *odelete* operations. Log records for *oopen* and *owrite* are only written if they modify any metadata. The input parameters (excluding data) for all operations are stored in the log record. In our design, the size of each log record is just 32B plus the object name. In practice, we expect most log records to fit within a single cache line.

In DIPPER, a write operation works as follows (see Figure 4): ① Lock the block and metadata pool, ② allocate and write the log record, ③ allocate blocks from block pool, ④ allocate pages from metadata pool, ⑤ unlock the block and metadata pool, ⑥ write metadata with allocated blocks in metadata zone, ⑦ write btree record to memory, ⑧ write data to SSD, and ⑨ commit and flush log record to PMEM. Steps ③, ④, ⑥, and ⑦ are responsible for constructing a metadata and btree entry. Exactly the same set of steps is used to reconstruct metadata and btree state upon recovery from failure.

As we can infer from the description above, the btree and metadata zone are updated in parallel outside the synchronous region. This parallelism is achieved by using the observational equivalency property of DIPPER. Our concurrency control algorithm (see §4.4) ensures that log records are added in conflicting order to maintain correctness.

4.4 Concurrency Control

We propose a concurrency control (CC) algorithm, which forwards information readily available in the PMEM log to determine concurrently executing operations. Our goal while designing this algorithm is to minimize additional memory usage and keep the latency for detecting conflicting requests minimal. Most systems use per-object locks to provide concurrency control which increase linearly with the number of *live* objects in the system. In contrast, our CC algorithm embeds a lock flag within the log records for each request. The number of locks is therefore limited by the size of the log. Conflicting requests do not use a hold and wait approach, but rather spin on dedicated flags corresponding to their conflict.

Write-Write Conflicts. The log contains records of all operations currently in execution. When a new request arrives, we scan the log to check if any operations in execution are operating on the same object. If so, we spin on the committed flag of the conflicting record until the operation finishes. This ensures that we do not have two concurrently executing requests on the same object. However, scanning the complete log to check for conflicts is inefficient and unnecessary. Scanning from the first uncommitted record until the end of the log enables us to detect conflicting operations *without* adding significant overhead.

Read-Write Conflicts. Since read requests are not added to the log, read-write request conflicts can still occur. For resolving read-write concurrency, we introduce a new in-memory hash table that maps object names to their current read count. The read count is updated using the atomic `fetch-and-add` instruction to ensure consistent count values during parallel operation by threads. By looking at the read count at the start of a write request, we can be sure that no request is reading that object at that time. In case the read count is non-zero, we simply poll on it until it is zero.

4.5 Additional Design Considerations

In this subsection, we discuss some of the additional considerations that were made while designing DStore.

Inter-Object Dependencies. Having the ability to specify complex inter-object dependencies is invaluable for many applications. For example, in a filesystem, dependencies between a file and its directory are captured by locking the directory before modifying the file. Such cases can be handled by using the *olock* and *ounlock* primitives to lock objects. To implement these primitives we introduce a novel `NOOP` log operation. This operation represents a `NOOP` on the DRAM data structures and is ignored by DIPPER recovery logic. The *olock* primitive places a `NOOP` record in the log and *ounlock* marks this record as committed. Thus, a log scan can correctly identify locked objects as conflicts.

Durability and Consistency. DStore writes data directly to storage device-level RAM (if available, otherwise to non-volatile media). In the event of an unexpected crash, device capacitors will safely flush data to non-volatile media. By eschewing buffering, DStore provides strong guarantees of data durability. Further, DStore only marks log records as committed once data is made durable. This implies that metadata will always be consistent, i.e., objects can never contain garbage data.

CoW Design. To enable fair comparison of DIPPER with CoW checkpoints used in related work, we implement CoW in DStore. This works as follows. When a checkpoint is triggered, all volatile pages in the frontend are marked as read only. As soon this is done, the frontend can process write operations again. When a client tries to modify a read-only page, a page fault is triggered and a handler copies the page to PMEM. Clients can assist in this copying process, but must wait until the page is copied before making any modification to it.

5 EXPERIMENTAL ANALYSIS

In this section, we present the evaluation of DStore. Our goal was to find answers to the following questions:

- Is DStore fast? If so, why? (§5.2)
- Is DStore quiescent-free? If so, why? (§5.3)
- Is DStore tailless? If so, why? (§5.4)
- How long does DStore take to recover? (§5.5)
- What is the storage footprint of DStore? (§5.6)

We first describe our experimental testbed and methodology before considering each of the questions listed above.

5.1 Experimental Testbed

Our experimental testbed consists of a Linux (3.10) server equipped with two Cascade lake CPUs (8280L@2.70GHz), 384GB DRAM (2 x 6 x 32GB DDR4 DIMMs), and 6TB PMEM (2 x 6 x 512GB DCPMMs) configured in App Direct mode, and a 750GB Intel P4800X NVMe drive. Each CPU has 28 cores (with hyperthreading disabled) and 38.5MB of L3 cache (LLC). All available PMEM is formatted as two `xfs`-DAX filesystems, each utilizing the memory of a single CPU socket as a single interleaved namespace.

All code was compiled using gcc 9.2.0. PMDK [26] 1.8 was used across all evaluations to keep comparisons fair. Hardware counters were obtained using a combination of Intel VTune Profiler [25] and Linux iostat utility. DCPMM hardware counters were collected using the `ipmwatch` utility, a part of the VTune Profiler.

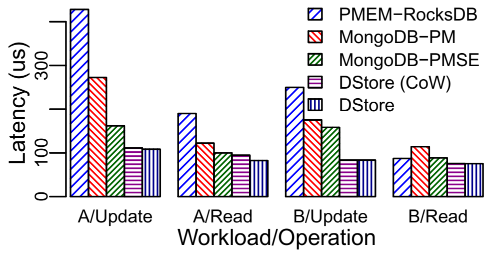


Figure 5: YCSB Operation Latency

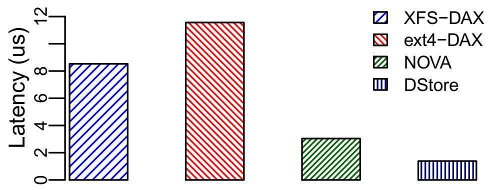


Figure 6: Metadata Overhead

We compare performance with at least one system from each category in Table 1. We directly compare DStore with three popular PMEM-optimized NoSQL storage systems, PMEM-RocksDB [52, 63], MongoDB-PM [46], and MongoDB-PMSE [28]. PMEM-RocksDB is an optimized version of the log-structured merge (LSM) tree-based vanilla RocksDB [18] and uses a PMEM resident log to improve performance. MongoDB-PM uses an optimized btree-based WiredTiger engine with the btree index and journal placed in a DAX filesystem formatted on PMEM to improve performance. MongoDB-PMSE uses PMEM optimized data structures to store data in-place and uses PMDK's pmemobj-cpp library for crash consistency. We also implement the copy-on-write checkpoint scheme used in NOVA [65] and Pronto [44] in DStore for comparison purposes. We also compare metadata overhead with the Linux direct-access (DAX) filesystems `xfs` [59], `ext4` [43], and NOVA [64]. Finally, we indirectly compare with DudeTM [39] and NV-HTM [4] by incorporating the physical logging design used in them in DStore. We were unable to compare with Bullet since its source code is unavailable.

We primarily use 4KB sized operations for experiments. This is to conform with the SSD hardware block size. Smaller sized operations will result in write amplification and their throughput will match that of 4KB operations. Larger operations will be limited by SSD bandwidth, so we do not focus on them in our evaluation.

5.2 Is DStore Fast?

To evaluate DStore performance, we measure the average latency of 4KB read and update operations at full-subscription (28 cores) with YCSB [13] workloads A (50% read, 50% write) and B (95% read, 5% write). We compare DStore latency with that of other PMEM-optimized systems. From Figure 5, we can observe that DStore provides the best latency in all cases, up to 4x lower than other systems. The reason for this is that metadata requests experience ultra-low latency since the frontend is entirely in DRAM. In contrast, other systems must access persistent storage for metadata updates, which increases latency. For instance, MongoDB-PMSE must update both data and metadata in PMEM for each operation. This is also

Size	Time	NVMe Write	BTree	Metadata	Log Flush	Total
4KB	**Time (cycles)**	24029	789	807	1663	27288
	Time (ns)	8899.63	298.89	292.22	615.93	10106.67
	% of Total	**88.06**	2.96	2.89	**6.09**	100
16KB	**Time (cycles)**	108844	1284	789	1945	112862
	Time (ns)	40312.60	475.56	292.22	720.38	41800.74
	% of Total	**96.44**	1.14	0.70	**1.72**	100

Table 3: Time breakdown of write requests

the reason that we see higher improvement for update than read operations. In general, update latency is lower for workload B compared to A for all systems because the high read:write ratio implies that the cost of persistence can be more easily overlapped with updates to the volatile cache. Finally, we also observe that DStore with copy-on-write checkpoints provides nearly the same latency as DStore. This is because checkpoint design only impacts tail latency and not average latency.

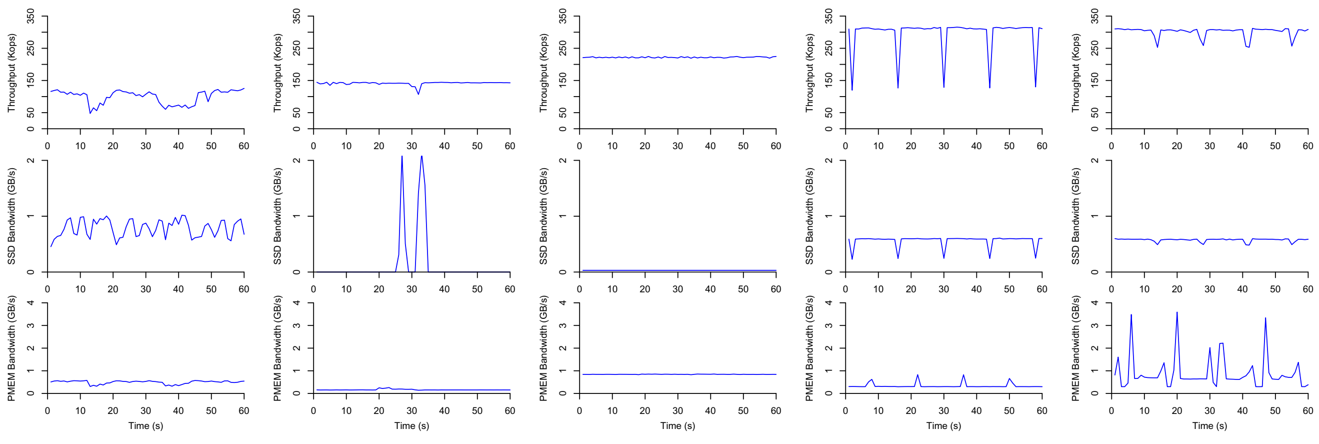
We also compare with PMEM-optimized DAX filesystems (`xfs`-DAX, `etx4`-DAX, and NOVA) to evaluate the filesystem interface of DStore. Since these filesystems place data in PMEM while DStore does so in SSD, we were unable to make a direct comparison. Instead, we measure the metadata overhead of 4KB writes to a file for each system. Figure 6 shows this comparison. Just like the previous experiment, we find that DStore is the fastest in terms of updates to metadata. This is because updating metadata only requires making changes to in-memory data structures and recording the operation in the log. In contrast, other systems need to update changes to PMEM because their volatile and persistent domains are not decoupled. For instance, NOVA must update the file's inode as well as add the operation to the inode's log, both of which must be made in PMEM for durability. `xfs`-DAX and `ext4`-DAX suffer from similar limitations. Overall, we find that by keeping metadata in DRAM and using compact logical logging, DStore significantly reduces operation latency compared to other systems. Through experimental analysis, we also discover that our userspace run-to-completion pipeline is successful in avoiding context switches in the critical path, which also contributes to latency reduction.

Why is DStore Fast? To better understand why DStore is fast, we analyze its write pipeline in more detail. Table 3 shows the latency breakdown of 4KB and 16KB writes. The time spent in each component is given in cycles, nanoseconds, and as a percentage of total time. The right-most column shows the total time for the write request. Metadata indicates the time required to allocate blocks and update the corresponding metadata. Log flush represents time spent in flushing the log record to PMEM. What stands out the most is the percentage of time spent doing NVMe writes. This indicates that we are successfully able to reduce software overhead to ~10% of total time. We also observe that a log flush takes less than 2000 cycles (or 740 ns), minimizing the impact of logging on write performance. This also indicates that 3D-XPoint technology provides extremely low latency for a single cache line flush. Finally, we find that the metadata and log flush overheads are similar for both IO sizes. This is because the use of logical logging in DIPPER leads to request-size-agnostic software overhead. Therefore, we conclude that having a DRAM frontend with the entire metadata in the reason for DStore's fast performance.

5.3 Is DStore Quiescent-Free?

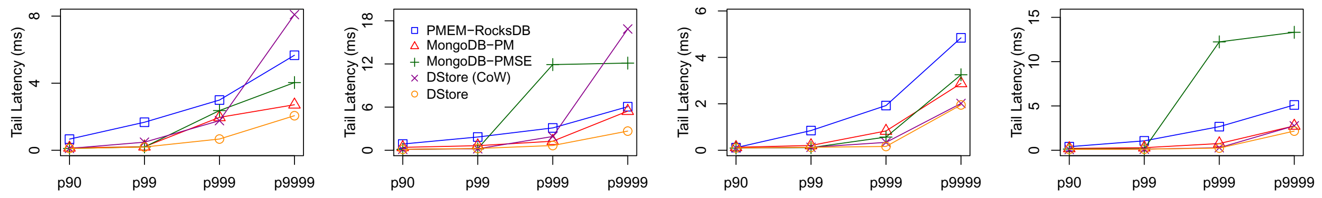
To measure the effects of checkpoints on system throughput, we conduct an experiment with a full-subscription (28 cores) 50% read, 50% write workload. We measure the aggregate throughput over a 1 minute window. We chose a 1 minute window because it was long enough for each system to trigger a checkpoint at least once. The first row of graphs in Figure 7 shows the result of this analysis. We can clearly observe that DStore is able to sustain higher throughput than other systems for the entire time window. The troughs in the graph represent periods of checkpoint. DStore throughput drops slightly during a checkpoint because of the impact of background threads applying operations to the backend. Nevertheless, even the lowest throughput achieved is greater than the highest of any other system, i.e. DStore is able to satisfy a high throughput SLO. Importantly, the system never fully quiesces for both read and write operations. Other systems are unable to achieve high throughput because of the reasons discussed earlier (see §5.2). Apart from MongoDB-PMSE, all systems experience throughput drops during checkpoints. MongoDB-PMSE uses inline persistence, so no checkpoints are required and the throughput is consistent over time. Despite this, the overheads of cache flushes and transactions prevent it from achieving good performance even though it places data on PMEM. DStore delivers 15% higher throughput SLO compared to MongoDB-PMSE and is more cost effective because it places data on SSD. Another observation we make is that the copy-on-write design significantly lowers throughput during checkpoints. This is because client threads need to block and wait until the pages they want to modify are made durable. Finally, we find that the continuous background compaction in RocksDB prevents the frontend from achieving consistent throughput. In fact, for a short duration, it was unable to serve any update requests, violating quiescent freedom.

Why is DStore Quiescent-Free? To understand our observations further, we measured the SSD and PMEM bandwidth during the experiment. The SSD bandwidth for PMEM-RocksDB and MongoDB-PM clearly show the activity of the asynchronous checkpoints, which explains the dips in throughput during them. MongoDB-PMSE does not use the SSD at all as it places all data in PMEM. Interestingly, for DStore (including CoW case), the SSD bandwidth curve mirrors the throughput curve. This is because data is directly updated in the SSD for each request and metadata is only placed in DRAM/PMEM. If we look at the PMEM bandwidth, we notice that except DStore, other systems utilize only a small percentage of available bandwidth. The reason for this is that their throughput is limited by other factors, such as checkpoints or transactions. These systems are unable to effectively utilize the byte-addressability and performance of PMEM. Even CoW does not utilize PMEM effectively because pages are flushed individually when page faults are triggered and not in batches. DStore is able to better exploit its performance by using shadow updates for the backend. In this manner, the backend throughput is not limited since transactions are not needed. We also observe that the PMEM backend is not always active which indicates that DStore can handle workloads with an even higher write:read ratio without quiescing. Our analysis shows that only workloads having more than 70% writes will lead to backlogging. This is sufficient, as most enterprise workloads have been



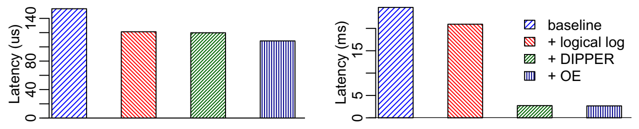
(a) PMEM-RocksDB (b) MongoDB-PM (c) MongoDB-PMSE (d) DStore (w/ CoW chkpt) (e) DStore

Figure 7: System throughput and storage bandwidth over a 1 minute window for a full-subscription (28 cores) 50% read, 50% write workload



(a) Workload A Read Tail Latency (b) Workload A Update Tail Latency (c) Workload B Read Tail Latency (d) Workload B Update Tail Latency

Figure 8: Tail latency curves at full-subscription (28 cores) for YCSB A (50% read, 50% write) and B (95% read, 5% write)



(a) average (b) p9999

Figure 9: Effect of optimizations on write latency

shown to have less than 50% writes [3, 9, 32]. We conclude that logical and physical decoupling of normal operation and checkpoints allows them to be completely overlapped. As a result, DStore can provide uninterrupted service to end users.

Is DStore Scalable? DStore uses atomic operations for generatic LSNs and locks for allocating blocks and metadata pages. However, we do not find their use to be a scalability bottleneck. We compare performance at full-subscription (28 cores) with other systems and outperform all of them (see Figure 7). In addition, from Figure 3, we can see that work done while holding a lock (for metadata) takes less than 300 ns and is not a bottleneck.

5.4 Is DStore Tailless?

We measure the tail latency of 4KB read and update operations at full-subscription (28 cores) with YCSB workloads A (50% read, 50% write) and B (95% read, 5% write). Figure 8 shows the tail latency curves for both operations. Overall, DStore has flatter tail latency curves and also the lowest latency values for all cases, up to 6x lower than other systems. The reason for this trend is the decoupled design, which is successful in hiding the latency of persistence. In contrast, other systems have longer tails because of the effects of checkpoints on client requests. Requests arriving during checkpoints must wait for data to be persisted, resulting in this trend. We already verified and discussed this reason in Figure 1. Some other interesting observations we make are as follows. CoW shows high p9999 latency for the update-heavy workload A but close to DStore latency for the read heavy workload B. This is due to a reduction in the frequency of checkpoints for workload B as compared to A. We find that read tail latency is also worse for other systems as compared to DStore. This implies that checkpoints impact both read and write requests. Finally, we observe that MongoDB-PMSE has high p999 and p9999 latency despite using an uncached design. We believe this trend is because of the high tail latency of PMEM itself and not the software design. The high tail latency of Optane DCPMM has been verified in [66].

Why is DStore Tailless? For improving write latency, we proposed effective optimizations, including PMEM-based logical logging, decoupled persistence (DIPPER), and observational equivalency (OE). To determine the impact of these optimizations on system performance, we first evaluate the baseline design, adding optimizations one-by-one and measuring performance again. The naïve baseline uses ARIES-style physical logging [21], used in NV-HTM and DudeTM, with CoW checkpoints. Figure 9 gives us an idea of the impact of optimizations on both average and p9999 latency at

System	Shutdown Type	Metadata	Replay	Total Time
PMEM-RocksDB	clean	0	156	156
MongoDB-PM	clean	474	139	613
MongoDB-PMSE	clean	985	0	985
DStore	clean	846	841	1687
PMEM-RocksDB	crash	5013	150	5163
MongoDB-PM	crash	7525	12786	20311
MongoDB-PMSE	crash	1708	0	1708
DStore	crash	11580	861	12441

Table 4: System recovery time (in ms): metadata is time to recover metadata and replay is time to replay log records.

full-subscription. The naïve design has the worst performance. This is due to the high log write latency and overhead of CoW checkpoints. Moving from physical to compact logical logging improves average latency by 21% and tail latency by 15%. Incorporating DIPPER (+DIPPER) on top of this improves tail latency significantly (~7.6x). DIPPER impacts tail and not average latency because it only improves response times for requests during a checkpoint. Finally, adopting OE completely removes any synchronization overhead, further improving average and p9999 latencies by 9% and 2%, respectively. OE particularly helps at high concurrency by allowing parallel operation on the btree and metadata zone. We conclude that logical logging is the most beneficial for average latency while DIPPER is the most beneficial for tail latency.

5.5 Recovery Performance

To test recovery performance, we evaluate two cases: one with a normal shutdown and the other with an unexpected crash just before the checkpoint process is complete (the worst possible failure point). We simulate system failure by forcing a hard shutdown (`SIGKILL`) and restarting the process. For this experiment, we load two million 4KB objects into each system. Table 4 shows average system recovery times for both cases mentioned earlier. When recovering from a clean shutdown, DStore must reconstruct its volatile space from the persistent space. Other systems do not have this overhead because they only bring data into the cache on-demand. For this reason, recovery from a clean shutdown takes longer for DStore. In case of failure during a checkpoint, all systems take longer to recover. This is because any operations in-flight need to be re-executed and lost data must be reconstructed using the log. For DStore, the ongoing checkpoint process during a crash must be redone upon recovery in addition to the normal recovery process. MongoDB-PM and PMEM-RocksDB must recover any lost volatile data by replaying records from the log. All three have a similar recovery procedure and therefore show similar performance. MongoDB-PMSE only needs to re-execute in-flight operations using transaction data and recovers the fastest. In general, the two-level design prevents instant recovery for DStore. Since recovery is not the common case, we believe that the current design provides adequate recovery times but leaves room for improvement in the future.

5.6 Storage Footprint

DStore maintains two copies of metadata (in DRAM and PMEM) and uses shadow updates for the PMEM copy. We consider it important

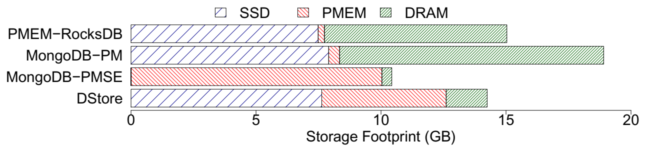


Figure 10: Storage footprint with 2M 4KB objects

System	Throughput	p9999 Lat.	Recovery Lat.	Space Ampl.
MongoDB-PM	106704 IOPS	5439 us	20311 ms	2.47
MongoDB-PMSE	219221 IOPS	12119 us	**1708 ms**	**1.36**
PMEM-RocksDB	47532 IOPS	6055 us	5163 ms	1.97
DStore (CoW)	119685 IOPS	16863 us	12441 ms	1.86
DStore	**252832 IOPS**	**2665 us**	12441 ms	1.86

Table 5: Summary of achievable service level objectives

to measure the storage overhead and compare it with other systems. To evaluate the physical storage footprint, we load two million objects into the system and then measure the total space (DRAM, PMEM, and SSD) consumed by each system. Figure 10 shows the result of this analysis. Default settings are used for all systems. Interestingly, we find that all systems have similar storage footprint. Overall, DStore only consumes more space than MongoDB-PMSE. MongoDB-PMSE consumes the least space because it does not require a volatile cache. While the actual data storage footprints for all systems are virtually the same, the metadata overheads differ significantly. Both PMEM-RocksDB and MongoDB-PM reserve a large chunk of DRAM as their cache space but only actually utilize a small portion of it. For this reason, both have a higher storage footprint than DStore. In the worst case, DStore may need space for three copies of metadata. However, since the space is allocated adhoc, this overhead is kept to a minimum. Further, the performance benefits of the decoupled approach far outweigh the drawbacks of additional PMEM usage.

5.7 Evaluation Summary

To get a complete picture of how different systems might perform in a cloud setting, we analyzed their achievable SLO. Table 5 provides a summary of these for throughput, p9999 and recovery latency, and space amplification[3]. These represent the worst case values we obtained in our experiments. For each metric, the best values have been highlighted. DStore achieves the best throughput and p9999 SLO. This is because the use of DIPPER prevents sudden drops in throughput and keeps tail latency low, resulting in consistent and predictable performance. For recovery and space amplification, MongoDB-PMSE is able to deliver the best SLO. This is expected because it does not utilize a cache and data is persisted inline. Therefore, there is no storage overhead of the cache and recovery can be near instantaneous. DStore (CoW) has the same recovery and storage overhead as DStore because it uses the same recovery and memory allocation design. However, the CoW design is unable to deliver the same performance characteristics as DIPPER.

[3]We define space amplification as the ratio of size of application data to the size of space utilized by the storage system across DRAM, PMEM, and SSD.

Key Takeaways. Our results indicate that using a decoupled design is better for throughput and latency SLO while an uncached design is better for recovery and space SLO. Cached approaches can provide fast performance, but because of inconsistent performance during checkpoints, they are ineffective in providing reasonable SLO. Depending on the required tradeoffs, a decoupled or uncached design should be chosen for storage systems. Overall, we conclude that DStore provides the best performance SLO, good space SLO, and adequate recovery SLO and satisfies the three requirements we set out to achieve.

6 RELATED WORK

Over the last decade, a significant body of work has attempted to design transactional abstractions, persistent data structures, and storage systems for PMEM. All of these systems can be classified as either cached, uncached, or decoupled.

Cached Systems. Despite the byte-addressability and performance benefits of PMEM, cached systems remain a popular class of storage systems. Several cached systems [8, 14, 17, 23, 44, 46, 52, 63] have been recently proposed to leverage PMEM. Despite differences in the design and implementation of these systems, all of them suffer from the same flaw. Clearing cache or log space is a costly operation and typically requires the cache to be write-protected for the duration of this operation. Ultimately, this design is unable to provide consistent performance that is desired by cloud providers.

Uncached Systems. Several systems [7, 12, 15, 16, 19, 20, 31, 51, 60–62, 64, 71] have been proposed to take advantage of the storage and memory characteristics of PMEM to modify and access data in-place. Unfortunately, ensuring crash consistency for updates is non-trivial, requiring complicated transactional or journaling algorithms to be used. It has been shown that durable transactions have high overhead and significantly lower end-to-end performance [22, 47]. iDO [40] is a recent compiler based technique to reduce the overhead of persistence and crash consistency by logging updates only at the granularity of idempotent code regions. Although it outperforms several other uncached approaches, it still relies on expensive physical logging.

Decoupled Systems. Some recent systems like DudeTM [39], NV-HTM [4], and Bullet [24] have proposed partially or fully decoupling operation and persistence phases to lower the cost of persistence and improve performance. However, these systems still rely on expensive physical logging which is particularly bad when the working set of transactions is large. DudeTM and NV-HTM both use HTM for atomicity. Unfortunately, HTM requires hardware support which limits their wide-scale applicability. Further, DudeTM requires hardware changes to support its timestamp design, making it incompatible with commodity HTM. In contrast, DStore does not require special hardware support and uses more efficient logical logging. DStore is similar in concept to Bullet, but as discussed in Section 2, Bullet's approach is not as generic and performant as DStore.

Logical Logging. Some recent works [42, 51] have successfully applied logical logging to database systems, while WAFL [34] and NOVA [64, 65] have done the same to filesystems. However, unlike DStore, these works do not provide a truly atomic quiescent-free checkpoint solution.

Hand-Crafted Indexes. Recent work [5, 37, 48, 70] has proposed highly-optimized hand-crafted indexes on PMEM which achieve crash consistency by carefully and cleverly ordering and persisting updates to PMEM rather than use logging. Although these approaches offer good performance, the hand-crafted designs are specific to a particular data structure, and thus, are not easily applicable to other data structures.

7 CONCLUSION

In this paper, we presented the design and implementation of DStore, a high-performance fault-tolerant userspace storage subsystem. DStore provides fast performance, low tail latency, and quiescent freedom simultaneously through the proposed DIPPER approach which decouples system and checkpoint space. We showed that by using PMEM to store the checkpoint space, we can leverage its performance to effectively overlap the operation of the two spaces. The novelty of our approach was in the design of the backend, which allowed us to use the same code for both spaces and provide low overhead persistence. Evaluation with Intel Optane DCPMM demonstrates the clear superiority of DStore over other state-of-the-art storage systems. In the future, we plan to extend our designs to build a disaggregated storage system.

ACKNOWLEDGMENTS

We thank our anonymous reviewers and shepherd for their valuable feedback. We would also like to thank Tuong Nguyen, Abel Gebrezgi, Jonathan Burton, and Timothy Witham from Intel for their support and help in access to the test platform. This work was supported in part by NSF research grant CCF #1822987.

REFERENCES

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. 2015. Let's Talk About Storage and Recovery Methods for Non-Volatile Memory Database Systems. In *SIGMOD'15*. 707–722.

[2] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 2016. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *EuroSys'16*. 19.

[3] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *FAST'20*. 209–223.

[4] Daniel Castro, Paolo Romano, and Joao Barreto. 2019. Hardware Transactional Memory meets Memory Persistency. *J. Parallel and Distrib. Comput.* 130 (2019), 63–79.

[5] Hokeun Cha, Moohyeon Nam, Kibeom Jin, Jiwon Seo, and Beomseok Nam. 2020. B3-Tree: Byte-Addressable Binary B-Tree for Persistent Memory. *ACM Transactions on Storage (TOS)* 16, 3 (2020), 1–27.

[6] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *OOPSLA'14*. 433–452.

[7] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. In *PVLDB'15*, Vol. 8. 497–508.

[8] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS'20*. 1077–1091.

[9] Yanpei Chen, Kiran Srinivasan, Garth Goodson, and Randy Katz. 2011. Design Implications for Enterprise Storage systems via Multi-Dimensional Trace Analysis. In *SOSP'11*. 43–56.

[10] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. 2013. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *SOSP'13*. 1–17.

[11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *ASPLOS'11*. 105–118.

[12] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-Addressable, Persistent Memory. In *SOSP'09*. 133–146.

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC'10*. 143–154.

[14] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *USENIX ATC'18*. 373–386.

[15] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *SOSP'19*. 478–493.

[16] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *EuroSys'14*. 15.

[17] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *EuroSys'18*. 42.

[18] Facebook. 2012. RocksDB. https://rocksdb.org/ (accessed Aug. 2020).

[19] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *PPoPP'18*. 28–40.

[20] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. 2016. Using Storage Class Memory Efficiently for an In-Memory Database. In *SYSTOR'16*. 21.

[21] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques*. Elsevier.

[22] Swapnil Haria, Mark D Hill, and Michael M Swift. 2020. MOD: Minimally Ordered Durable Data Structures for Persistent Memory. In *ASPLOS'20*. 775–788.

[23] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. In *PVLDB'14*, Vol. 8. 389–400.

[24] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *USENIX ATC'18*. 967–979.

[25] Intel. 2014. Intel VTune Profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html (accessed Aug. 2020).

[26] Intel. 2014. PMDK. https://github.com/pmem/pmdk (accessed Aug. 2020).

[27] Intel. 2016. Enhanced Power-Loss Data Protection in the Intel Solid-State Drive 320 Series. Available at: https://newsroom.intel.com/wp-content/uploads/sites/11/2016/01/Intel_SSD_320_Series_Enhance_Power_Loss_Technology_Brief.pdf (accessed Aug. 2020).

[28] Intel. 2016. Persistent Memory Storage Engine for MongoDB. https://github.com/pmem/pmse (accessed Aug. 2020).

[29] Jake Brutlag. 2009. Speed Matters. https://ai.googleblog.com/2009/06/speed-matters.html (accessed Aug. 2020).

[30] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *SOSP'19*. 494–508.

[31] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *FAST*. 191–205.

[32] Anil Kashyap. 2018. Workload Characterization for Enterprise Disk Drives. *ACM Transactions on Storage (TOS)* 14, 2 (2018), 1–15.

[33] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. 2008. Characterization of Storage Workload Traces from Production Windows Servers. In *IISWC'08*. 119–128.

[34] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. 2017. Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL. In *FAST'17*. 1–14.

[35] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence Chiu. 2014. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. In *FAST'14*. 33–45.

[36] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *ASPLOS'16*. 385–398.

[37] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. 2018. clfB-tree: Cacheline Friendly Persistent B-Tree for NVRAM. *ACM Transactions on Storage (TOS)* 14, 1 (2018), 1–17.

[38] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling Low Tail Latency on Multicore Key-Value Stores. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1091–1104.

[39] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *ASPLOS'17*. 329–343.

[40] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *MICRO*. 258–270.

[41] David Lomet and Mark Tuttle. 1999. Logical Logging to Extend Recovery to New Domains. In *SIGMOD'99*. 73–84.

[42] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking Main Memory OLTP Recovery. In *ICDE'14*. 604–615.

[43] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 Filesystem: Current Status and Future Plans. In *Proceedings of the Linux Symposium*, Vol. 2. 21–33.

[44] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *ASPLOS'20*. 789–806.

[45] Sparsh Mittal and Jeffrey S Vetter. 2016. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1537–1550.

[46] MongoDB Inc. 2009. MongoDB. https://www.mongodb.com/ (accessed Aug. 2020).

[47] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory use with WHISPER. In *ASPLOS'17*. 135–148.

[48] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *FAST*. 31–44.

[49] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. 2015. SQLite Optimization with Phase Change Memory for Mobile Applications. In *PVLDB'15*, Vol. 8. 1454–1465.

[50] OpenStack. 2010. Swift. http://swift.openstack.org/ (accessed Aug. 2020).

[51] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. 2017. SQL Statement Logging for Making SQLite Truly Lite. In *PVLDB'17*, Vol. 11. 513–525.

[52] Peifeng Si. 2019. Persistent Memory Storage Engine for RocksDB. https://github.com/pmem/pmem-rocksdb (accessed July 2020).

[53] Slawomir Pilarski and Tiko Kameda. 1992. Checkpointing for Distributed Databases: Starting from the Basics. *IEEE Transactions on Parallel and Distributed Systems* 3, 5 (1992), 602–610.

[54] Donald Sannella and Andrzej Tarlecki. 1985. On Observational Equivalence and Algebraic Specification. In *Colloquium on Trees in Algebra and Programming*. Springer, 308–322.

[55] Mohit Saxena, Mehul A Shah, Stavros Harizopoulos, Michael M Swift, and Arif Merchant. 2012. Hathi: Durable Transactions for Memory using Flash. In *DaMoN'12*. 33–38.

[56] Steve Stargall. 2020. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature.

[57] Guy L Steele Jr. 1989. Making Asynchronous Parallelism Safe for the World. In *POPL'89*. 218–231.

[58] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *NSDI'15*. 513–527.

[59] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System. In *USENIX ATC'96*. 1–14.

[60] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST'11*, Vol. 11. 61–75.

[61] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS'11*. 91–104.

[62] Xiaojian Wu and AL Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *SC'11*. 39.

[63] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *ASPLOS'19*. 427–439.

[64] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST'16*. 323–338.

[65] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *SOSP'17*. 478–496.

[66] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST'20*. 169–182.

[67] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *FAST'15*, Vol. 15. 167–181.

[68] Erez Zadok, Dean Hildebrand, Geoff Kuenning, and Keith A Smith. 2017. POSIX is Dead! Long Live... errr... What Exactly?. In *HotStorage'17*. 12–12.

[69] Wen Zhang, Scott Shenker, and Irene Zhang. 2020. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *OSDI*. 1029–1046.

[70] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *OSDI*. 461–476.

[71] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. In *OOPSLA'19*. 1–26.