Karlsruher Institut für Technologie

Low-Latency Synchronous I/O For OpenZFS Using Persistent Memory

Masterarbeit
von

Christian Schwarz

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Lukas Werling, M.Sc.

Bearbeitungszeit: TODO – TODO

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 30. Mai 2021

Abstract

We propose to explore persistent memory (PMEM) as the storage medium for the ZFS Intent Log (ZIL). Specifically, we propose a new type of ZIL called ZIL-PMEM that bypasses existing block-device oriented abstractions to take advantage of the low latency provided by PMEM. ZIL-PMEM will maintain the same consistency guarantees to applications as the current ZIL implementation, maintain data integrity of log contents and correctly handle data corruption.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special contents, but the length of words should match the language.

Contents

Chapter 1

Introduction

The task of a filesystem is to provide non-volatile storage to applications in the form of the *file* abstraction. Applications modify files through system calls such as `write()` which generally does not provide any durability guarantees. Instead, the system call modifies a buffer in DRAM such as a page in the Linux page cache and returns to userspace. Synchronization of the dirty in-DRAM data to persistent storage is thus deferred to a — generally implementation-defined — point in the future.

However, many applications have more specific durability requirements. For example, an accounting system that processes a purchase needs to ensure that the updated account balance is persisted to non-volatile storage before clearing the transaction. Otherwise, a system crash and recovery after clearing could result in the pre-purchase balance being restored, enabling double-spending by the account holder. These **synchronous I/O** semantics must be requested through APIs auch as `fsync()` which "assure that after a system crash [...] all data up to the time of the `fsync()` call is recorded on the disk." [**posix_fsync_opengroup**].

The **Zettabyte File System (ZFS)** is a combined volume manager and filesystem. It pools many block devices into a single storage pool (*zpool*) which can hold thousands of sparsely allocated filesystems. The ZFS on-disk format is a merkle tree that is rooted in the *uberblock* which is ZFS's equivalent of a superblock. ZFS on-disk state is always consistent and moves forward in so-called *transaction groups* (txg), using copy-on-write to apply updates. Whenever a new version of the on-disk state needs to be synced to disk, ZFS traverses its logical structure bottom up and builds a new merkle tree. The updated parts of the tree are stored in newly allocated disk blocks while unmodified subtrees re-use the existing blocks written in prior txgs. Once all updates have been written out, the

new uberblock is written, thereby atomically moving the on-disk format to its new state. This procedure is called *txg sync* and is triggered periodically (default: every 5 s) or if the amount of dirty data exceeds a configurable threshold.

Synchronous I/O semantics cannot be reasonably implemented through txg sync due to the write amplification and CPU overhead inherent to the txg sync procedure. Instead ZFS maintains the **ZFS Intent Log (ZIL)** which is a per-filesystem logical write-ahead log. The ZIL's *records* describe the logical changes that need to applied to the filesystem in order to achieve the state that was reported committed to userspace. On disk, the log records are written into a chain of *log-write blocks* (LWBs), each containing many records. The LWB chain is rooted in the filesystem's *ZIL header* and is extended independently of txg sync.

By default, LWBs are allocated from the zpool's main storage devices. Consequently, the lower bound for synchronous I/O latency in ZFS is the time required to write the LWBs that contains the synchronous I/O operation's ZIL records. To address the case where this latency is insufficient, ZFS provides the ability to add a *separate log device* (SLOG) to the zpool. The SLOG is typically a single (or mirrored) block device that provides lower latency than the main storage devices. A typical configuration today is to add a fast NVMe drive to an HDD-based pool. Adding a fast SLOG accelerates LWB writes because LWBs are preferentially allocated from the SLOG. Note that SLOGs only need very limited capacity since LWBs are generally obsolete after three txgs.

Persistent Memory (PMEM) is an emerging storage technology that provides low-latency memory-mapped byte-addressable persistent storage. The Linux kernel can expose PMEM as a pseudo block device whose sectors map directly to the PMEM space. Thereby existing block devices consumers can benefit from PMEM' low latency without modification. Block device consumers that wish to bypass block device emulation use the kernel-internal *DAX* API which translates sector numbers to kernel virtual addresses, giving software **d**irect **ac**cess to PMEM.

The motivation for this thesis is to accelerate synchronous I/O in ZFS by using PMEM as a ZFS SLOG device. A single DIMM of the current Intel Optane DC PMEM product line can sustain 550k random 4k write IOPS from a single CPU core which corresponds to a write latency of 1.81 us. However, when configuring the Linux PMEM block device as a SLOG in ZFS, a single thread only achieves 10k IOPS (100 us/IOP), scaling up to 100k IOPS at 7 threads (70 us/IOP per thread). This discrepancy prompted us to perform a more detailed analysis of ZFS's severe latency overhead in this benchmark. We find that approximately 80% of wall clock time for the average IOP is spent in the ZIL code. We identify severe

overheads caused by both the ZIL's physical data structure (LWBs) as well as the mechanism to persist it (ZIO pipeline).

Based on our observations, we propose **ZIL-PMEM**, a new ZIL design that exclusively targets PMEM. It coexists with the existing ZIL which we refer to as *ZIL-LWB* in the remainder of this document. ZIL-PMEM achieves 128k random 4k sync write IOPS with a single thread and scales up to 400k IOPS as early as four threads before it becomes CPU-bound. Our implementation is extensively unit-tested and passes the ZFS test suite's SLOG integration tests.

The remainder of this thesis is structured as follows: in Chapter 2 we review prior work in the field of persistent memory related to filesystems and provide background knowledge on the integration of PMEM in the Linux kernel as well as a detailed introduction to the components of ZFS that are relevant for this thesis. Chapter 3 describes our analysis of ZIL-LWB's sub-par performance on a PMEM SLOG device, motivating a new PMEM-specific ZIL implementation. Our solution, ZIL-PMEM, is then presented Chapters 4, 5, and 6. Chapter 4 defines the project goals and provides a high-level overview of the overall design. Our main contribution, the *PRB/HDL* the data structure, is then presented in Chapter 5. Chapter 6 then describes how we integrate PRB/HDL into ZFS. We evaluate our implementation in Chapter 7 and conclude with a summary of our work in Chapter 8.

Chapter 2

Literature Review & Background

In this chapter we present prior work in the field of persistent memory and its application in various storage systems developed in research and industry. The last two sections on PMEM in Linux and ZFS provide the technical background knowledge that is necessary to understand the performance analysis of ZIL-LWB and the design of ZIL-PMEM in the subsequent chapters.

2.1 Literature Review

We have surveyed publications in the area of persistent memory storage systems, filesystem guarantees & crash-consistency models, PMEM-specific crash-consistency checkers and general methods to determine filesystem robustness in the presence of hardware failures.

2.1.1 PMEM Filesystems

In this subsection we present research filesystems that were explicitly designed for persistent memory. ZIL-PMEM integrates into ZFS, a production filesystem that was not designed for persistent memory. Hence we focus on techniques for crash consistency and data integrity that might be applicable to our work.

In-Kernel PMEM Filesystems

The initial wave of publications around the use of PMEM in filesystems produced a set of systems that were implemented completely in the kernel.

BPFS [Con+09] is one of the earliest filesystems expressly designed for PMEM. The filesystem layout in PMEM is inspired by WAFL ([HLM94]) and resembles a

we don't yet compare these systems with ZIL-PMEM. should we? Maybe in a compact section after we presented the design?

tree of multi-level indirect pages that eventually point to data pages. BPFS's key contribution is the use of fine-grained atomic updates in lieu of journaling for crash consistency. For example, updates to small metadata such as *mtime* can be made using atomic operations. For larger modifications, the authors introduce *short-circuit shadow-paging*, a technique where updates are prepared in a copy of the page. The updated page is then made visible through an update of the pointer in its parent indirect page. The difference to regular copy-on-write is that, as soon as the update to an indirect page can be done through an atomic in-place operation, the atomic operation is used. Updates thereby do not necessarily propagatate up to the root of the tree.

PMFS [Dul+14] is another research filesystem that targets persistent memory. The authors make frequent comparisons to BPFS. The main differentiator from BPFS with regards to consistency is PMFS's use of undo-logging for metadata updates and copy-on-write for data consistency in addition to hardware-provided atomic in-place updates. The evaluation shows that their approach for metadata has between 24x and 38x lower overhead compared to BPFS (unit: number of bytes copied). PMFS also introduces an efficient protection mechanism against accidental *scribbles*. Scribbles are bugs in the system that accidentally overwrite PMEM, e.g., due to incorrect address calculation or out-of-bounds access in the kernel. By default, PMFS maps all PMEM read-only, thereby preventing accidental corruption of PMEM from code outside of PMFS. When PMFS needs to modify PMEM, it temporarily disables interrupts and clears the processor's `CR0.WP`, thereby allowing writes to read-only pages in kernel mode. [Dul+14; SDM413]

NOVA [XS16] is the most mature research PMEM filesystem. NOVA uses per-inode logs for operations scoped to a single inode (e.g. write syscalls) and per-CPU journals for operations that affect multiple inodes. The intended result is high scalability with regard to core count. The per-inode log data structure is a linked list with a head and tail pointer in the inode. NOVA leverages 8-byte atomic operations to update these pointers after it has written log entries. While not explicitly called so by the authors it is our impression that the log is a logical redo log except for writes, which — judging from the text — are always logged at page granularity. The authors explain the recovery procedure and measure its performance but do not address correctness in the evaluation.

NOVA-Fortis [Xu+17] is a version of NOVA that introduces snapshots and hardening against data corruption. Whereas snapshots are not relevant for this thesis because ZFS already provides this feature, the data corruption countermeasures are representative of the state of the art:

- Handling of machine check exceptions (MCE) in case the hardware detects bit errors. This is done by using `memcpy_mcsafe()` for all PMEM access.

ref mcsafe explainer

- Detection of metadata corruption through CRC32 checksums.
- Redundancy through metadata replication. For metadata recovery, NOVA-Fortis compares checksums of the primary and replica and restores the variant with the matching checksum.
- Protection against localized unrecoverable data loss through RAID4-like parity. This feature only works while the file is not DAX-mmapped.
- Protection against Scribbles using `CR0.WP` as described in the paragraph on PMFS.

The authors use a custom fault injection tool to corrupt data structures in a targeted manner and test NOVA Fortis's recovery capabilities.

Hybrid PMEM filesystems

A recurring pattern in PMEM filesystem design is to split responsibilities between kernel and userspace component in order to eliminate system call overhead.

Aerie [Vol+14] is a user-space filesystem based on the premise that "SCM [Storage Class Memory] no longer requires the OS kernel [...]. Applications link to a file-system library that provides local access to data and communicates with a [user-space] service for coordination. The OS kernel provides only coarse grained allocation and protection, and most functionality is distributed to client programs. For read-only workloads, applications can access data and metadata through memory with calls to the file-system service only for coarse-grained synchronization. When writing data, applications can modify data directly but must contact the file-system service to update metadata." The system uses a redo log maintained in each client program which is shipped to the filesystem service periodically or when exclusive access to a file system object is relinquished. It is our understanding that only log entries shipped to and validated by the filesystem service will be replayed. The authors state that log entries can be lost if a client crashes before the log entries are shipped. The evaluation does not address crash consistency or recovery at all.

Strata [Kwo+17] is a cross-media filesystem with both kernel and user-space components. Since its distinguishing feature is the intelligent migration of data between different storage media we discuss it in the Section 2.1.2 on cross-media storage systems.

SplitFS [Kad+19] is a research filesystem that proposes a "split of responsibilities between a user-space library file system and an existing kernel PM file system. The user-space library file system handles data operations by intercepting POSIX calls, memory-mapping the underlying file, and serving the read and

overwrites using processor loads and stores. Metadata operations are handled by the kernel PM file system (ext4 DAX)". SplitFS uses a redo log with idempotent entries that is written from userspace. As a performance optimization the authors use checksums and aligned start addresses to find valid log entries instead of an explicit linked list with persistent pointers. The SplitFS evaluation of correctness is limited to a comparison of user-observable filesystem state between SplitFS and ext4 in DAX mode. Recovery is evaluated only through the lens of recovery time (performance), not correctness.

EvFS [YCH19] is a "user-level POSIX file system that directly manages NVM in user applications. EvFS minimizes the latency by building a user-level storage stack and introducing asynchronous processing of complex file I/O with page cache and direct I/O. [...] EvFS leads to a 700-ns latency for 64-byte non-blocking file writes and reduces the latency for 4-Kbyte blocking file I/O by 20 us compared to a kernel file system [EXT4] with journaling disabled." In contrast to Aerie, EvFS does not require a coordinating user-space service. Crash consistency and recovery is not addressed: "EvFS is not a production-ready file system because it neither provides all the POSIX APIs or crash-safe properties".

2.1.2 Cross-Media Systems

Cross-media storage systems combine the advantages of multiple storage devices from different levels of the storage hierarchy. Historically, these kinds of systems strive to exploit hard disk for high capacity at low cost and more expensive flash storage for low latency random access IO. With persistent memory, a new class of storage has become available whose role in cross-media systems is still to be determined. In the context of this thesis we find it most useful to compare the overall system architecture.

ZFS: Allocation Classes []

cut this

Whereas ZFS was initially designed for a large pool of hard disks, it has gained several cross-media features over its lifetime. When configuring a zpool, the administrator assigns the block device to an *allocation class*. The following allocation classes exist: *normal* (main pool), *log* (SLOG device), *aux* (L2ARC, a victim cache for ZFS's ARC), *special* (small blocks), and *dedup* (deduplication table data). When a function in ZFS allocates a block in the pool, it must specify the desired allocation class. If the allocation succeeds, the allocated block is guaranteed to be located on device(s) within that class. The use case for allocation classes is to combine the advantages of different storage media in a single pool. In many setups devices in the *normal* class have high capacity and are grouped in storage-

check that we have explained ARC already

efficient redundancy configurations such as *raidz* or *draid*. A *log* or can be added to a pool in order to accelerate latency-sensitive I/O such as ZIL writes. And *aux* devices can be used as a victim cache for ZFS's primary DRAM cache. Many administrators configure lower redundancy for these devices — either to gain performance or to reduce costs — which, depending on business requirements, may be justified due to the short-lived nature of ZIL writes. In comparison to the other systems presented in this section, allocation classes are very inflexible: once an allocation is made and the data is written, the data stays in that place until it is freed or the device is removed from the pool. In particular, there is no automatic tiering that takes usage patterns into account. Further, *aux* devices reduce space efficiency because the cached data still occupies space in the main pool. Similarly, *log* devices are somewhat wasteful since they are exclusively used for ZIL logging and never read except during recovery after a crash. Systems such as Strata derive more value from its PMEM-based log. Finally, it should be noted that while Linux's /dev/pmem block device can be used with allocation classes, ZFS's ZIO pipeline is unable to exploit its low latency, as we will show in Section 3.

ref backwards

ZFS: ZIL Performance Improvements For Fast Media [Ope20] At the OpenZFS 2020 Developer summit, Saji Nair of storage vendor Nutanix presented a ZIL prototype that handles fast block devices more efficiently. The prototype avoids unnecessary sequential ordering of I/O operations when writing ZIL LWBs. It also avoids using the ZIO pipeline due to context switching overheads, issuing block I/O directly from the application thread instead. The evaluation is limited to a single benchmark: a fio 8k () sync write workload, spread across four zpools in identical configuration, achieves 75k IOPS at four threads, which is a 4x speedup over the upstream implementation. With 16 threads, the system achieves its peak IOPS at approx. 125k IOPS, which is a speedup of 2.5x. The source code for the prototype has not been published and the design is incomplete with regards to replay.

check that LWBs have been introduced by now

Strata [Kwo+17] is a cross-media research filesystem. "Closest to the application, Strata's user library synchronously logs process-private updates in NVM while reading from shared, read-optimized, kernel-maintained data and metadata. [...] Client code uses the POSIX API, but Strata's synchronous updates obviate the need for any sync-related system calls". We classify Strata as a hybrid filesystem in Section 2.1.1 because it consists of both a userspace library and an in-kernel component. The log, written from userspace, is an idempotent logical redo log. The kernel component then *digests* the logs asynchronously and performs aggregation of the logged operations during digestion. Aggregation permits the kernel component to issue "sequential, aligned writes" to the lower-

tier storage devices such as SSDs or HDDs. ZFS with and without ZIL-PMEM compares to Strata in the following ways:

- Both systems use a logical redo operation log instead of a block-level journaling mechanism.
- Both systems perform asynchronous write-back and thereby reap similar benefits from it (parallel batch processing, optimized allocation).
- Strata's kernel component digests the logs written from user-space in order to write them back to other tiers. ZFS accumulates the write-back state in DRAM and never reads the log except for recovery. (It is unclear to us how often Strata digests the logs. ZFS performs write-back of dirty state after at most 5 seconds.)
- Strata seems to tune allocations for SSDs, e.g. allocating blocks in erasure block size to prevent write amplification. ZFS supports TRIM and supports variable block sizes up to 16 MiB, but there are no automatic optimizations that specifically target write amplification in SSDs.
- ZFS is in-kernel and requires no modifications to applications whereas Strata requires linking to or LD_PRELOADing a user-space library, which, tangentially, makes it incompatible with statically linked binaries.

Ziggurat [ZHS19] "Ziggurat exploits the benefits of NVMM through intelligent data placement during file writes and data migration. Ziggurat includes two placement predictors that analyze the file write sequences and predict whether the incoming writes are both large and stable, and whether updates to the file are likely to be synchronous. Ziggurat then steers the incoming writes to the most suitable tier based on the prediction: writes to synchronously-updated files go to the NVMM tier to minimize the synchronization overhead. Small, random writes also go to the NVMM tier to fully avoid random writes to disk. The remaining large sequential writes to asynchronously-updated files go to disk". The authors compare Ziggurat to Strata as follows: ZFS with and without ZIL-PMEM compares to Ziggurat as follows:

- Ziggurat actively migrates data into PMEM based on access pattern. ZFS has no provisions for data migration within the pool after block allocation. The ZFS architecture such a feature is unlikely to be developed in the future (keyword: blockpointer rewrite).
- Ziggurat sends writes directly to the suitable tier based on prediction of future access patterns. If the access pattern is anticipated to be synchronous, Ziggurat choses the PMEM tier. The ZIL, and ZIL-PMEM specifically, only serves as a stop-gap between txg sync points of the main pool. Data is

always written twice — once to the log and once to the main pool —, and never read from the log except during recovery.
- Both systems are fully in-kernel and require no modifications to user-space applications.
- Ziggurat builds on NOVA-Fortis and thus inherits the PMEM-specific data integrity and redundancy mechanisms provided for the PMEM storage tier. ZIL-PMEM data integrity measures are currently limited to data and metadata checksumming.
- The Ziggurat paper does not mention data integrity measures or redundancy mechanisms for the block device layers beneath PMEM. Possibly, existing Linux features such as Device Mapper could be used to compensate. In contrast, ZIL-PMEM benefits from ZFS's strong data integrity and redundancy mechanisms once the logged data is written to the main pool by txg sync.
- Ziggurat was not evaluated on actual persistent memory. The authors used memory on another NUMA node to simulate lower latencies. We evaluate ZIL-PMEM on commercially available PMEM hardware.

dm-writecache [] is a new (2018) Linux Device Mapper target that implements a write-back caching layer for block devices. Given an *origin* device and a *cache* device, it exposes a new virtual block device with the same capacity as the origin device. Write-back is asynchronous and controlled by relative thresholds on the cache device space utilization (*low* and *high watermark*, default to 45% and 50%). It starts when the cache fill ratio reaches the *high watermark* and stops once it reaches the *low watermark*. Reads are not handled by dm-writecache and instead served from the Linux page cache. []
Dm-writecache is relevant to ZIL-PMEM because it is the closest functional analogue to the ZIL in the Linux Device Mapper stack. It has been designed with explicit support for persistent memory and its authors have presented it as a means to accelerate database workloads [Tad+19]. However, both the data flow and data integrity guarantees differ substantially between the two systems. First, ZIL-PMEM pools the PMEM space for all filesystems and ZVOLs in a zpool whereas dm-writecache is limited to a single block device and thus would require partitioning. Second, ZIL-PMEM persists writes as log entries to an append-only log structure and does not perform coalescing or or delta-encoding. In contrast, dm-writecache is a block-level cache where an overwrite of the virtual block results in an over-write in the cache. Further, ZIL-PMEM protects data integrity through checksums whereas dm-writecache fully relies on hardware error correction and detection, reported via `memcpy_mcsafe()`. After ZFS has synced out modifications to the main pool, they benefit from ZFS's strong data redundancy mechanisms such as *raidz*. In contrast, with dm-writecache, the origin block

device driver must handle data redundancy if desired, e.g., though another Device Mapper target such as *dm-raid*. We compare the performance of the two implementations in Section 7.3.3 of our evaluation.

2.1.3 Journaling & Write-Ahead Logs Adapted To PMEM

The following publications use persistent memory to accelerate file system journals and write-ahead logs.

Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory [LBN13] is an academic paper that presents a PMEM-aware buffer cache design which "subsumes the functionality of caching and [block-level] journaling". From the buffer cache's perspective, the on-disk blocks that make up a filesystem journal are indistinguishable from non-journal filesystem blocks. Thus, for a journal block A' that logs an update to a block A, both blocks A' and A will sit in the buffer cache. This waste of space can be reduced with PMEM by the author's proposal. Instead of journaling on top of the block layer, one journals in the buffer cache itself by a) placing the buffer cache in PMEM and b) introducing a new buffer cache entry state "frozen" so that an entry can be "clean", "dirty" or "frozen". When modifying a block A, the filesystem no longer makes a journal entry but instead modifies the buffer cache entry directly. If the entry was "clean", it is now "dirty". If the entry was "frozen", a copy is made and the modification goes to the copy, making it "dirty" as well. Committing a block to the journal is a simple state transition from "dirty" to "frozen". On a crash + restart, "dirty" buffer cache entries are discarded but "frozen" entries remain. We find the approach exceptionally interesting and innovative and can imagine an application of the idea to the ZFS ARC. However, the system's real-world performance is unclear (evaluation on DRAM). Also, we see non-trivial software engineering and maintenance problems with the approach. Finally, the paper does not address hardware error handling or data corruption concerns at all.

ext4 fast commits [] is a feature released in Linux 5.10. "The fast-commit journal [...] contains changes at the file level, resulting in a more compact format. Information that can be recreated is left out, as described in the patch posting: For example, if a new extent is added to an inode, then corresponding updates to the inode table, the block bitmap, the group descriptor and the superblock can be derived based on just the extent information and the corresponding inode information. [...] Fast commits are an addition to — not a replacement of — the standard commit path; the two work together. If fast commits cannot handle an operation, the filesystem falls back to the standard commit path." The cited article mentions ongoing work to use persistent memory for ext4 fast commits.

The approach is inspired by per-inode journaling as proposed by Park and Shin in [PS17].

Disk-oriented database management systems often use write-ahead logs (WALs) to allow a transaction to commit before the modified pages are written back to stable storage. However, appending to a shared WAL file has historically been a latency and scalability bottleneck which lead to amortization techniques such as *commit groups* (aka *group commit*) and *pre-committed transactions* that batch the WAL entries of multiple committing transactions into a single physical WAL record.[1] With persistent memory, the latency bottleneck no longer lies in the raw I/O but rather the coordination overhead between multiple CPU cores and sockets, prompting new distributed logging designs that avoid a central point of contention. [Fan+11; Pel+13; Joh+10] ZIL-LWB employs *group commit* as well although the terminology is different. It batches log records issued by independent threads for the same filesystem into the currently *open log-write block* (LWB). Once the open LWB is full or a timeout has passed, the open LWB is written (*issued*) to disk (see Section 2.3.4 for details). In contrast, ZIL-PMEM commits log records directly to PMEM and allows multiple cores to do so in parallel with minimal global coordination. Our evaluation shows that the design scales well on single-socket systems. NUMA and multi-socket systems, which are addressed in the cited publications from the database community, have been explicitly out of scope for this thesis (Section 4.1.1).

ensure this term has been established by now

minimal global coordination = the commit slots + semaphore

2.1.4 Testing Filesystem Crash Consistency

In this section we survey techniques to determine and verify crash consistency guarantees of file systems. A formal and automatically verifiable model would have been been extremely helpful to ensure that ZIL-PMEM maintains the same crash consistency guarantees as ZIL-LWB.

ref

All File Systems Are Not Created Equal [Pil+14] contributes a survey of the atomicity and ordering guarantees of several popular Linux filesystems and provides a tool called ALICE to validate or derive the guarantees required by applications. ZFS with and without ZIL-PMEM could benefit from this survey as well. The survey could be used to characterize ZFS's guarantees. ALICE could be used to determine whether ZFS's guarantees are sufficient for applications and/or whether ZFS's guarantees exceed the requirements of the majority of applications. However, since ZIL-PMEM shall maintain the same guarantees as

[1] According to [DeW+84] "the notion of gronp commits appears to be past of the unwritten database folklore. The System-R implementors claim to have implemented it."

ZIL-PMEM (see Section 4.1.1), such findings would only be relevant for future work.

Specifying and Checking File System Crash-Consistency Models [Bor+16] The authors "present a formal framework for developing crash-consistency models, and a toolkit, called FERRITE, for validating those models against real file system implementations." The system provides means to express expected filesystem behavior as a *litmus test*. A litmus test encodes expected behavior of a filesystem though a series of events (e.g. write to a file) and a final predicate expressed as a satisfiability problem. FERRITE can execute the litmus test against an axiomatic formal model of the filesystem to ensure that, iff the actual filesystem adheres to the formal model, the litmus test's expectations hold. Notably the litmus test is executed symbolically and the validation predicates are checked for satisfiability by an SMT solver; this is exhaustive and not comparable to a unit or regression test. FERRITE can also execute litmus tests against the actual filesystem to test whether it adheres to the formal model. This test is non-exhaustive. It is based on executing the litmus test "many times" where for each execution all disk commands emitted during execution are recorded. All permutations and prefixes of these traces that are allowed under the semantics of the disk protocol are used to produce test disk images which are fed back to the filesystem under test for recovery. After recovery the litmus test's predicate must hold against the concrete filesystem state after recovery. If it does not hold the given permutation of the trace is proof that the filesystem does not match its model (assuming the litmus test passes execution against the model), or that the filesystem's assumptions about the disk do not match the FERRITE *disk model*. FERRITE appears to be a useful tool to build a model of ZFS's undocumented crash-consistency guarantees (ZFS has not been evaluated by the authors). Such a model would be helpful to validate ZIL-PMEM's goal to maintain the same semantics as ZIL-PMEM. However, this would require a FERRITE disk model for persistent memory.

Using Model Checking to Find Serious File System Errors [Yan+06] The authors present an "implementation-level model checker" that removes the need to define a formal model for the filesystem. Instead, the model is inferred by running the OS with the filesystem and recording both syscalls emitted by the application and disk operations emitted by the filesystem. These traces are subsequently fed to a process that produces disk images created from reorderings of disk operations. The disk images are fed to the filesystem's fsck tool. Disk images that can be 'repaired' by fsck are then fed to the "recovery checker" which examines filesystem state and compares it to the expected state which (if we Understand section 4.2 of the paper correctly) is derived from the recorded system

calls. (The role of the "volatile file system" in this process is still unclear to us). The authors mention several shortcomings of their system:

- Lack of multi-threading support (this applies to FERRITE as well).
- The recovery checker produces a projection of the filesystem state (e.g. only names and content but no atime). Thus, the system can only check guarantees at the projection level.
- Restrictive assumptions such as "Events should also have temporal independence in that creating new files and directories should not harm old files and directories".

The idea of using the filesystem's recovery tools (fsck) to infer its guarantees seems useful to avoid the requirement of a formally specified model. However, it is our understanding that the resulting model is rather a larger regression test than a truly derived exhaustive model. Such regression tests would be useful as a starting point for a formal model, e.g., as initial litmus tests for use with FERRITE. We do not believe that we can apply the presented approach to ZFS with and without ZIL-PMEM with reasonable effort.

2.1.5 PMEM-specific Crash-Consistency Checkers

A growing body of work introduces tools that check whether code that manipulates persistent memory actually issues the architectually required instructions for persistence. However, most systems only target userspace code and are thus inapplicable to ZIL-PMEM which is implemented in the ZFS kernel module. Whereas ZIL-PMEM can be compiled for user-space as part of the *libzpool* library, the large amount of conditional compilation involved in the libzpool build process diminishes the significance of user space tests. We surveyed the following userspace-only tools:

- **pmemcheck [intoductionToPmemcheckPart]**, a tool in Intel's Persistent Memory Development Kit (PMDK). It integrates with Valgrind ([]) and requires annotation of all PMEM accesses. Notably, the PMDK libraries include these annotations.
- pmeminsp — todo
- Agamotto — todo

The remaining tools in this subsection support kernel code and are applicable to ZIL-PMEM in principle.

Yat: A Validation Framework For Persistent Memory Software [Lan+14] "Yat is a hypervisor-based framework that supports testing of applications that use Persistent Memory [...] By simulating the characteristics of PM, and inte-

grating an application-specific checker in the framework, Yat enables validation, correctness testing, and debugging of PM software in the presence of power failures and crashes." The authors used Yat to validate PMFS (see Section 2.1.1). Yat's hypervisor-based approach makes it the ideal tool for evaluating ZIL-PMEM. To our great dissatisfaction, Yat has never been published and remains an Intel-internal project.

we introduced it above, context still there?

PMTest: A Fast And Flexible Testing Framework For Persistent Memory Programs [Liu+19] PMTest is a validation tool that claims to be significantly faster than Intel's *pmemcheck*. The implementation is based on traces of PMEM operations which are generated by (manually or automatically) instrumented application code. PMTest ships with two built-in checkers that assert correct instruction ordering (e.g.: missing store barriers) and durability (e.g.: missing cache flushes). Higher-level checks must be implemented by the programmer. PMTest works within kernel modules but the implementation is limited to a single thread. The processing of the operation trace happens in userspace. PMTest could be used to check ZIL-PMEM kernel code: for a single ZPL filesystem and a single thread that performs synchronous I/O, the pool's PMEM is only written from that thread.

ref design sections

2.1.6 Fault Injection

Error handling code paths are notoriously difficult to test but often critical for correctness. Fault injection is a common technique to simulate failures and thereby exercise these code paths.

Model-based Failure Analysis Of Journaling File Systems [PAA05] The authors present an analysis of the failure modes caused by incorrect handling of disk write failures in the journaling code in ext4, ReiserFS and IBM JFS. These filesystems each implement one or more of the following journaling modes: data journaling, ordered journaling, and writeback journaling. Any block write performed in one of these modes falls in one of the following categories: "J represent journal writes, D represent data writes, C represent journal commit writes, S represent journal super block writes, K represent checkpoint data writes [...]". For each of the three journaling modes, the authors present a state machine that describes all permitted sequences of block writes. The state machine includes transitions to an error state if a block write fails. The system works as follows. A kernel module tracks the filesystems' state as modelled by the state machine. It intercepts block device writes from the filesystem code and injects write failures. If the filesystem subsequently performs another block write that is not permitted by the state machine, an implementation error has been found. The authors distinguish several classes of failures with varying degrees of data loss.

The methodology is very filesystem specific which already shows in the adjustments required for IBM JFS. The ZIL's structure and its interaction with txg sync is substantially different from the journaling modes presented in the paper (See Sections 2.3.4 and **??**). The system is thus not applicable to ZIL-PMEM. review ref

ndctl-inject-error [] is a subcommand of the `ndctl` administrative tool that "can be used to ask the platform to simulate media errors in the NVDIMM address space to aid debugging and development of features related to error handling." The kernel driver forwards injection requests to the NVDIMM firmware via ACPI. [] The errors then surface as *machine check exceptions* (MCE), which is the same mechanism used to indicate detected but uncorrectable ECC errors for DRAM and PMEM. The functionality is useful for testing correct error handling in the filesystem when *reading* PMEM. ZIL-PMEM uses `memcpy_mcsafe()` to buffer all PMEM reads in DRAM and thus is able to handle MCEs gracefully. The handling is the same as for corrupted or missing log entries, for which we have good unit test coverage (ref. Sections 5.5, 5.6 and 7.2.1).

ZFS Fault Injection [] ZFS supports runtime IO error injection through APIs usable by the *ztest* stress test and through the *zinject* tool used by the *ZFS Test Suite* (ref. Section 7.2.2). The ZFS Test Suite makes use of the command for high-level features such as automatic hot spares. The mechanism is implemented in the ZIO pipeline and supports scoping of errors either to an entire device or to specific logical data objects in ZFS. Most of the semantics are tied to ZFS's `zbookmark` and `blkptr` structures which we do not use in ZIL-PMEM. Since ZIL-PMEM's existing unit tests already cover many scenarios for lost log entries at a higher level, we do not see an immediate for ZIO Fault Injection in this thesis. However, a fault injection tool or "editor" utility with an understanding of ZIL-PMEM's persistent data structures could be useful for end-to-end testing in the future, in particular in comparison to *ndctl-inject-error*.

2.2 Persistent Memory in Linux

2.3 OpenZFS Background

In this section, we provide an overview of ZFS's architecture and a more detailed description of the current ZIL implementation. We encourage readers to look into [Bon+03], [Zha+10], [MNW14], and [Ope16] for more comprehensive descriptions of ZFS.

2.3.1 Basic Concepts

ZFS combines volume management and filesystem services in a monolithic storage system that breaks with the traditional hard layering between block devices and filesystems.

When an administrator creates a ZFS storage pool (*zpool*), they supply ZFS with a set of *VDEVs* that represent the pool's volume topology. VDEVs can be either concrete (*disk*, *file* VDEV) or virtual (*mirror*, *raidz{1,2,3}*, or *draid*). Concrete VDEVs supply physical storage space whereas the virtual types compose multiple concrete VDEVs to achieve redundancy. The space supplied by virtual VDEVs is thus less than the sum of its children.

The *Storage Pool Allocator* (SPA) consumes a zpool's VDEVs and makes their aggregate space available through an interface that is reminiscent of *malloc* and *free*, albeit limited to allocation sizes that are multiples of the disk block size. The handle to an allocation is the *data virtual address* (DVA). DVAs are an indirection layer introduced by the SPA to decouple the upper layers of ZFS from the storage hardware configuration, enabling features such as online expansion or replacement of faulty devices. The consequence is that all reads from and writes to the allocated space must always go through the SPA.

Neither the SPA nor VDEVs provide any form of consistency. That is the job of the *Data Management Unit* (DMU). It uses the SPA to provide the abstraction of *object sets* and *objects*. An object set can contain up to 2^{64} objects, each identified by a 64-bit number. Each object in turn is a linear storage space with a theoretical size of up to 2^{64} bytes. The operations that the DMU exposes for objects are reminiscent of files, e.g., creation, read, write, destruction. However, in contrast to files, consumers must make all modifications to objects from within so-called *DMU transactions*. DMU transactions are always part of a *transaction group* (txg). The DMU accmuluates the changes to all object sets and objects on a per-*txg* basis in DRAM, as so-called *dirty state*. Once a txg's dirty data exceeds a threshold or a periodic timer triggers, the DMU atomically updates the on-disk state (more on this later).

The DMU is used by the *ZFS Posix Layer* (ZPL) to implement a POSIX-compliant filesystem, and by the *ZVOL* layer to implements virtual block devices, e.g. for storage virtualization via iSCSI. ZVOLs register as a block device driver in the kernel and map block-level read, write, and discard operations (`struct bio`) to the corresponding DMU operations of a single DMU object. In contrast, a ZPL filesystem (just "filesystem" from now on) uses multiple DMU objects to implement filesystem abstractions such as directories and files. Both the data struc-

tures that represent the filesystem metadata and the actual file data are stored in DMU objects.

A single zpool can hold thousands of filesystems and ZVOLs, each of which keeps its state (=object(s)) in its own private object set. This necessitates another software module to create, delete, and manage all of these object sets. Its name is the *dataset layer* (DSL) and it provides the following services:

Creation & Destruction Of Object Sets The DSL provides APIs to create and destroy the object sets for filesystems and ZVOLs.

Naming & Hierachical Organization Of Object Sets The DSL assigns names to object sets and organizes them in a tree structure. For example, a zpool named `tank` has a root ZPL filesystem object set `tank` and possibly three child filesystems `tank/a` `tank/b` and `tank/b/c`, each with their own objset set.

Properties & Inheritance Per-object-set configuration options, e.g., on-the-fly compression, can be inherited along the tree hierarchy to aid administration of large pools.

Snapshots & Clones The DSL provides facilities to create near-zero-cost snapshots and copy-on-write clones of filesystems and ZVOLs. (This functionality depends on some integration with the DMU on which we will not elaborate here.)

The DSL stores all of the metadata that is required to realize these features in a special object set, the *meta object set* (MOS). The ZFS term for the metadata about an individual object set that contains user data (filesyste, ZVOL, snapshot, clone) is "dataset".

2.3.2 On-Disk State

ZFS's on-disk structure is a tree of SPA-allocated blocks that link to their children reference? through *block pointers*. Block pointers are 128 byte sized structures that, among other metadata, contain the size, checksum and DVA of the pointee. The root of the tree structure is located in the *uberblock*. The uberblock points, with a block pointer, to an `objset_phys_t` structure. `objset_phys_t` is the root of an object set's persistent representation that itself is a tree of SPA-allocated blocks linked by block pointers. However, in the interest of brevity, we will view `objset_phys_t` as an opaque structure for now. Regarding the uberblock, the specific `objset_phys_t` structure that it points to is that of the MOS whose object's contains the DSL's metadata about all datasets. The most important piece of per-dataset metdata is a block pointer (`dsl_dataset_phys_t::ds_bp`) that points to a block that contains the root of the dataset's object set, i.e., an `objset_phys_t` structure. The interpretation

of that `objset_phys_t`'s content depends on the dataset type, i.e., whether it is a filesystem or ZVOL.

The DMU moves the on-disk state forward atomically at the granularity of transaction groups. To "sync out" a txg, the DMU's "syncing thread" (referred to as *txg sync* from here on) constructs a new version of the on-disk tree structure. The new tree re-uses unmodified subtrees of the existing on-disk state whenever possible and allocates new blocks from the SPA for updated state. Since newly allocated blocks have by definition a different block pointer, changes at any level of the tree propagate up to the location in the uberblock where the block pointer that points to the MOS's `objset_phys_t` is stored. This root block pointer is then replaced atomically, thereby moving the pool's on disk state forward by one txg. If the system crashes at any point in time before this final update, the pool's state after recovery is still that of the previous txg.

The uberblock is not part of the tree structure described above but stored in the *VDEV labels* instead. VDEV labels contain the necessary metadata to import the pool and the set of the most recently used uberblock pointers. Each physical storage device has four VDEV label locations at well-known offsets. When a pool is imported during boot, ZFS searches all VDEV labels for the uberblock with the highest txg and interprets it as the root block pointer of the tree structure. The atomic replacement of the uberblock that *txg sync* depends on is implemented through two-phase in-place overwrite scheme that ensures that there are always at least two old or new valid copies of the VDEV label on each storage device.

DMU consumers should be able to make progress while a transaction group is being written to disk. Therefore, the DMU always maintains three unwritten transaction groups, each in a different state. The *syncing* txg is the oldest unwritten txg. It is the txg that is currently being written out to disk by the procedure outlined above. The *open* txg is the youngest unwritten txg. It is the txg that is used for any newly started DMU transactions. Between syncing and open txg sits the *quiescing* txg. It is not used for any new DMU transactions, but some of the DMU transactions assigned to it may not yet have completed. However, once that has happened, it is guaranteeed that the accumulated dirty state for the quiescing txg is not going not change and is ready to be written out by *txg sync*. Thus, as soon as the syncing txg has been fully written, the quiescing txg becomes the new syncing txg, the open txg becomes the new syncing txg, and a new open txg is born.

may not have yet?

It is of crucial importance that the completion of a DMU transaction, indicated by a call to the `dmu_tx_commit` function, does not guarantee persistence. DMU transactions merely scope modifications on the DMU level to a txg; more specifically, the txg that was the *open txg* at the time the transaction was started. If the sys-

tem crashes before that txg has finished *syncing*, the in-flight txgs' modifications are lost.

2.3.3 ZFS Intent Log (ZIL)

The ZIL bridges the gap between the time at which a DMU transaction T_i completes via `dmu_tx_commit` and the time at which the changes made by T_i actually reach the main pool through *txg sync*. For that purpose, the DMU consumer that performs the DMU transaction encodes the logical change C_i that was made in T_i as an *intent log transaction* (ITX) and *assigns* it to the ZIL. For synchronous semantics, the DMU consumer must invoke the ZIL's `zil_commit` API after completing its DMU transaction(s) but before reporting the logical change as committed to any upper layer. `zil_commit` persists assigned ITXs to a sequential log on disk, independent of the DMU's *txg sync* procedure. This persistent log is used to recover lost changes in the event of a crash.

As an example, let us consider the a `write` system call to a file in a ZPL filesystem that was opened with the `O_SYNC` flag. The ZPL's VFS operation for this system call performs the following steps:

1. Start a DMU transaction,
2. modify the DMU object that represents the file,
3. create an ITX that describes the file's object number, the offset, payload, and modification timestamp (`zil_itx_create`),
4. assign the ITX to the object set's ZIL (`zil_itx_assign`),
5. complete the DMU transaction with `dmu_tx_commit`, and
6. call `zil_commit(file_object_number)` before returning to userspace.

Instead of `O_SYNC`, many applications use `fsync` or `fdatasync`. Their VFS operations in the ZPL map directly to the `zil_commit` call in Step 6. The `sync` system call, which requests flushing of the state of all files in a filesystem, uses the special parameter zero (`zil_commit(0)`).

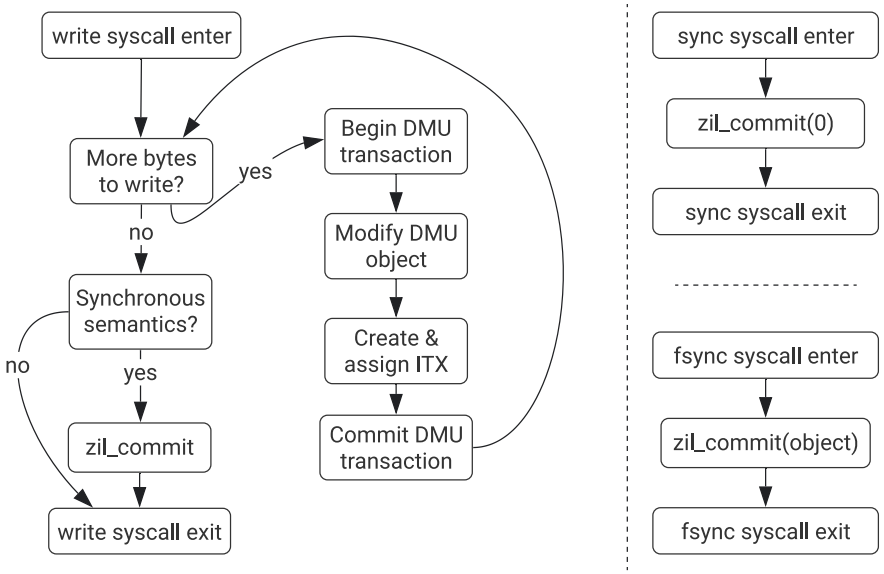


Figure 2.1: ZIL usage on the write path in by example of the `write()`, `fsync()`, and `sync()` system calls.

The ZIL tracks assigned ITXs in data structures called `itxg`. There exists one `itxg` for each of the three unsynced DMU transaction groups. Each `itxg` structure is split into a *sync list* and the *async tree*. The *sync list* is a simple list of ITXs whereas the *async tree* is a search-tree that maps from object ID to a list of ITXs. Most ITXs are appended to the *sync list* when they are assigned to the ZIL. However, ITXs that only affect a particular file are placed into the *async tree* instead. For example, an ITX that logs the creation of a file is added to the *sync list* whereas an ITX that logs a write within an existing file is added to the *async tree*. `zil_commit` uses the `itxgs` to determine the list of ITXs that need to be appended to the on-disk log so that successful recovery is possible. The following pseudo-code illustrates how `zil_commit` constructs this so-called ***commit list***, and how the records on the commit list are encoded as so-called ***log records*** and appended to the log.

```
zil_commit(object_id)
    commit_list := []
    for each unsynced transaction group 'txg':
        itxg <- the itxg for txg
        if object_id == 0:
            move all itxs in itxg.async to itxg.sync
        else:
            move only the itxs in
                itxg.async[object_id] to itxg.sync
        append itxg.sync to commit_list

    for each itx in commit_list:
```

```
        log_record := encode itx as log record
        append log record to a persistent log structure

    return
```

After a system crash, the zpool is in the state of the last synced transaction group which we call $precrash\_txg$. All datasets (i.e, filesystems, ZVOLs) are at their $precrash\_txg$ state and any changes made in younger DMU transactions are missing. The ZPL and ZVOL code uses the ZIL's `zil_replay` API when the filesystem is mounted (or the ZVOL opened) to recover the lost changes. The API contract is that the caller provides a callback that is invoked for exactly those log records that were lost, i.e., whose ITX's txg was $> precrash\_txg$. The callback is invoked in the order of the sequential on-disk log. Its job is to interpret the log record and to re-apply the change to the dataset in a new DMU transaction. Figure 2.2 exemplifies the rather unintuitive behavior of `zil_commit` and `zil_replay`.

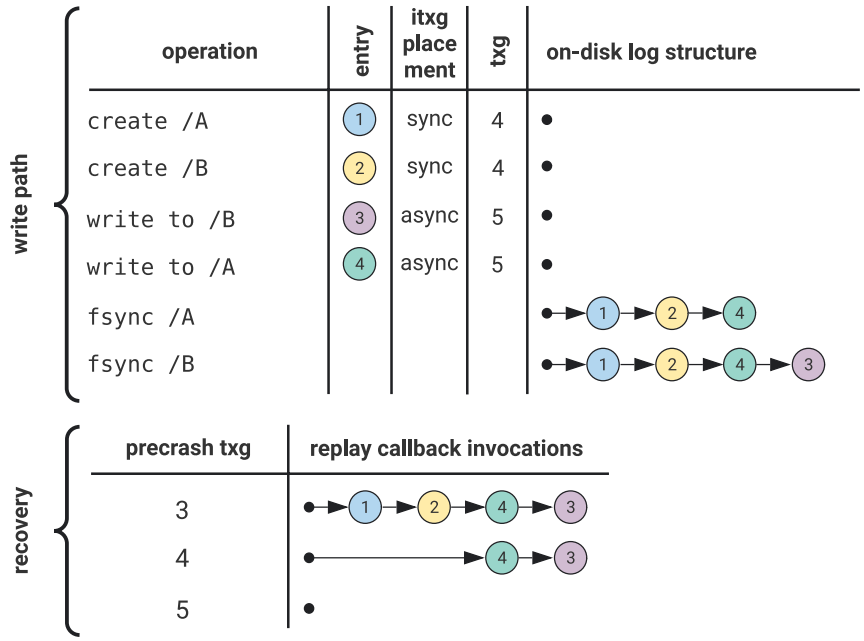


Figure 2.2: Example for the log structure that results from the VFS operations in the upper left column, and for the replay callback invocations for different values of $precrash\_txg$. Note that `fsync /A` not only logs `create /A` and `write to /A` but also `create /B` since file creation is always placed on the itxg's *sync list*. Second, note how records 1 and 2 are skipped during replay if $precrash\_txg = 4$. If they were not skipped, the replay callback would fail to create file /A when replaying entry #1 because /A already exists in the on-disk state of txg 4.

Note that replay only re-enacts the logical changes to the dataset but does not necessarily restore the exact physical state of the dataset that would have been synced in the absence of a crash. For example, changes that would have been spread across transaction groups 23, 24, and 25 at write time may land in transaction groups 42 and 43 during replay. Further, the system could crash again during replay. For example, a power outage could cause txg 42 to be the last-synced txg whereas 43 was still in *syncing* state. Since the replay callbacks for the individual log records are not idempotent, the ZIL must ensure that, after a crash during replay, the changes that were already replayed in transaction group 42 are not replayed again. For example, the second replay attempt could apply the remaining set of changes in transaction group 83. (But it may also happen that they are spread over several txgs, e.g., 83 and 84). The consequence for any ZIL implementation is that it needs to persist the following data:

Precrash-txg The *precrash-txg* for open logs at the first time during pool import so that it can filter log records by that criterion, and

Replay progress Information that tracks which log records have already been replayed so that replay can be resumed at the correct log record after a crash.

2.3.4 ZIL(-LWB) Persistence

The upstream ZIL implementation uses a singly-linked list of so-called *log-write blocks* (LWBs) to represent the on-disk log structure. The block pointer to the first LWB is stored in the *ZIL header* structure of the filesystem's or ZVOL's object set. Each LWB contains a contiguous sequence of the variable-length log records.

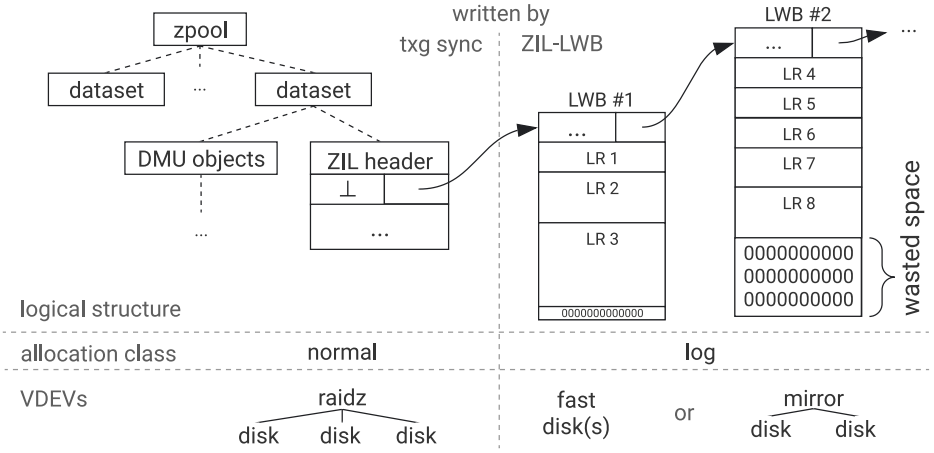


Figure 2.3: On-disk structure of ZIL-LWB as described in the previous paragraph.

After `zil_commit` has produced the commit list, it iterates over it and appends the log record representation of each ITXs to an in-DRAM buffer that will become the new tail LWB in the chain. Once this buffer is full, it is issued to disk in the background and the procedure continues with a new tail LWB. The packing of N log records on the commit list into M fewer but larger LWBs has been advantageous in the past:

Latency Amortization Under the assumption that disk latency (lat_{disk}) dominates overall synchronous I/O latency, writing log records one-by-one would result in a total latency of $N * lat_{disk}$. In contrast, grouping records into $M << N$ LWBs reduces the latency to $M * lat_{disk}$

LWB Timeout / Group Commit ZIL-LWB uses a *timeout* mechanism to extend the amortizing effect of LWB packing. As explained above, when a thread A packs log records into LWBs $L_1 \dots L_n$, it issues the IO operations for all but the last LWB L_n. A releases its lock on the LWB chain and goes to sleep, waiting for another thread B to continue to fill L_n. If such a thread B `zil_commit`s to the same dataset within that time window, it might fill L_n completely and issue it to disk, making L_{n+1} the new tail LWB. The IO completion callback for L_n then wakes up A and which is now allowed to return from `zil_commit` because the last entry on its commit list is now fully persisted. If no thread other threads picks up L_n, a timeout wakes up A so that it can issue the IO operation itself.
Without this mechanism, thread B would need to wait for the last LWB of A to be persisted, *and* for its own LWBs to be persisted because B's first LWB is pointed to by A's last LWB. By sharing the last LWB of A and B, up to $1 * lat_{disk}$ of waiting time can be avoided. Note that whereas ZFS refers to this mechanism as *LWB timeout*, the technique is well-established in disk-oriented databases under the name *group commit* or *commit group*.

review this

Space Efficiency Log records are stored as a contiguous sequence within the LWB. This avoids fragmentation if the commit list consists of small log records because many such log records can be packed into the smallest LWB (4 KiB). However, the on-disk format does not allow for splitting of individual log records. If the log record cannot be split in software (`WR_NEED_COPY`), the open LWB is issued with wasted space and a new LWB is allocated for the log record. Conversely, if the current LWB is large but only a few small entries need to be committed, a mostly empty LWB will be issued.

LWBs present a special case for ZFS's on-disk format because the LWB chain is *rooted* in the tree structure that is written by txg sync but *extended* independently by the ZIL. The solution is to pre-allocate LWBs such that the first LWB's block pointer is stored in the ZIL so that the first LWB of the chain can be found during

recovery. The first LWB's content and any subsequent LWBs in the chain are written and allocated by the ZIL. However, this represents a special case for the on-disk structure because, unlike all blocks written by *txg sync*, an LWB's content and checksum is not known at the time that the block pointer is written to the ZIL header or the ancestor LWB in the chain. Thus, intead of storing the checksum in the block pointer, it is stored in the LWB itself — LWBs are *self-checksumming*. The full procedure for adding an LWB to the chain thus consists of the following steps:

1. Wait until the LWB is filled or the timeout triggers the LWB to be issued.
2. Predict the best size for the next LWB based on a simple heuristic.
3. Allocate the next LWB from the SPA, prerrably from a SLOG (next section).
4. Store the resulting block pointer in the current LWB so that it points to the next LWB.
5. Compute the checksum of the current LWB.
6. Repurpose the current LWB's block pointer field to store the computed checksum.
7. Write out the current LWB.

The checksum ensures crash-consistency of the append operation since a partially written block is assumed to have an invalid checksum. And even in the case where no crash occurs, ZIL-LWB relies on an invalid checksum to detect the end of the LWB chain during recovery. (It is assumed unlikely for an unwritten but allocated LWB to be mistaken for an already written LWB because of additional metadata repeated in each LWB that must match for all LWBs in a chain.) For OpenZFS native encryption, the LWBs are not checksummed but encrypted and authenticated using an AEAD algorithm such as `aes-256-gcm`.

LWB's that only contain obsolete entries (their maximum txg $\leq$ last synced txg) are garbage-collected by *txg sync*. Every time *txg sync* visits the object set to construct the new version of the on-disk structure, it unlinks LWBs from the head of the chain whose maximum txg is $\leq syncing\_txg$ until it reaches an LWB that contains at least one log record for a txg $> syncing\_txg$. The unlinking updates the chain's head block pointer in the ZIL header but, due to the atomic nature of txg sync, the update only becomes visible on disk if $syncing\_txg$ successfully finishes syncing. In that case, $syncing\_txg$ becomes the new $precrash\_txg$ txg and thus all entries in the freed LWBs are by definition obsolete.

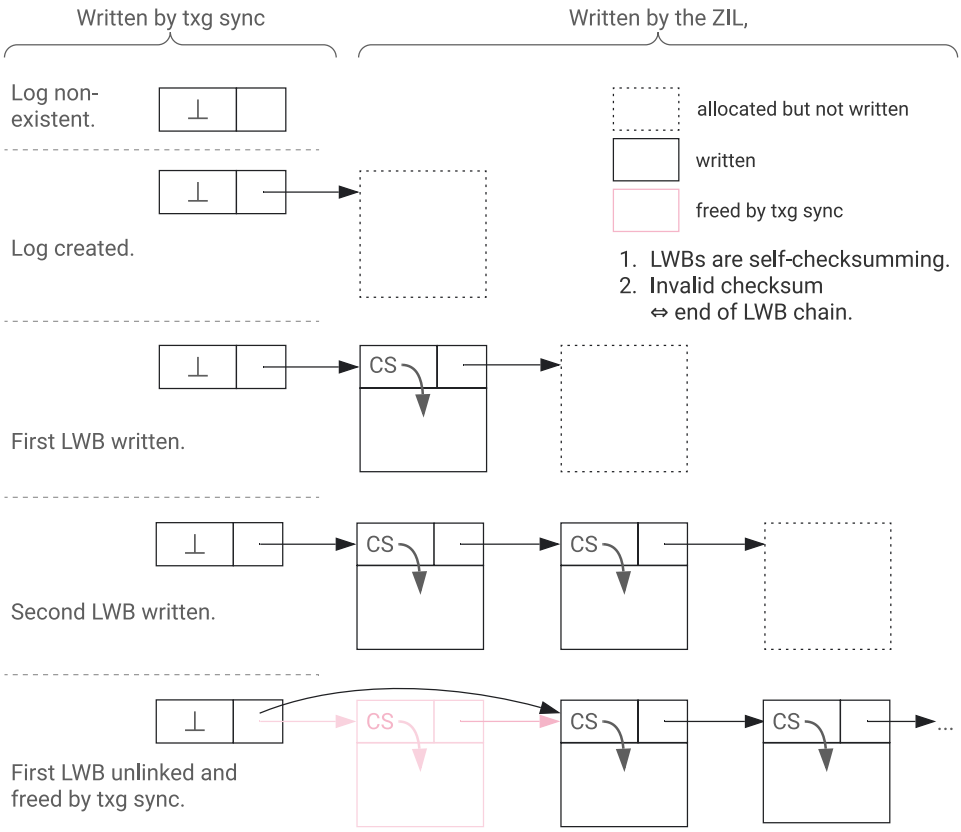


Figure 2.4: Visualization of how the LWB chain is extended by the ZIL and garbage-collected by *txg sync*.

2.3.5 Separate Log Devices (SLOGs) & Allocation Classes

By default, the SPA uses the same VDEVs for *txg sync*'s tree blocks and the ZIL's LWB allocations. However, it is possible to mark some of the pool's VDEVs as *separate log devices* (SLOGs). If a SLOG is present, the SPA uses it exclusively and preferentially for LWB allocations. As hinted in Figure 2.3, a typical configuration is the use high-capacity HDDs in a space-efficient *raidz* configuration for the main pool and a costlier but faster NVMe drive (*disk* VDEV) as a SLOG. Thereby, the advantages of both storage media are combined. Note that SLOGs only need very limited capacity (tens of GiBs at most) since any LWB is guaranteed to be obsolete after three txgs.

SLOGs were the earliest cross-media capability added to ZFS. They have recently evolved into a more general feature called *allocation classes* []. When VDEVs are

added to a zpool, they are assigned to an allocation class.[2] Functions that allocate space from the SPA must also specify the desired allocation class. If an allocation succeeds, the space is guaranteed to be located on a VDEV within the specified class. The following classes exist in upstream ZFS:

aux A second-level victim cache for ZFS's primary in-DRAM cache, and/or hot spare devices.
special A recently added class for small allocations (still $\geq$ block size).
dedup Deduplication table data.
log The allocation class used for LWBs.
normal The default allocation class. Often used as a fallback if the preferred class has no available space.

Allocation classes are relevant for this thesis because we introduce a new allocation class called *exempt* in Section 6.3.2.

ZIO Pipeline

The unified infrastructure for all IO performed by ZFS is the *ZIO pipeline*. As suggested by its name, ZIO implements a pipeline execution model to execute IO operations. To issue IO operations, consumers of ZIO allocate a `zio_t` object and annotate it with the pipeline stages that it needs to pass through. Among others, there are pipeline stages that abstract DVA allocation, DVA resolution, checksum computation, compression, block-level deduplication, encryption, and VDEV IO. The pipeline execution is heavily parallelized using `taskq` which are comparable to the Linux kernel's work queues or a dynamically scaling thread pool in user space.

The ZIO pipeline is relevant for the ZIL because it uses ZIO to write its LWBs. Integrating LWB I/O into the ZIO pipeline is also beneficial for the case where the pool does not have a SLOG configured because the latency-sensitive LWB operations can be prioritized over *txg sync* which is purely throughput-oriented. However, as we will show in the next chapter, ZIO comes with significant latency overhead, which results in sub-par performance of ZFS on low-latency storage devices such as PMEM.

[2] Actually, allocation classes are implemented at the level of *metaslab groups*, which are coarse chunks of VDEV space. However, from the user's perspective, allocation group assignment happens at the VDEV level.

Chapter 3

ZIL-LWB on PMEM

Our motivation for this thesis is the significant overhead of the current ZIL implementation (ZIL-LWB) in 4k random synchronous write workloads compared to the performance of the raw PMEM hardware in the same workload. In this chapter, we describe how we measured this overhead (Section 3.1) and present the resulting data (Section 3.2). In Section 3, we proceed with an analysis of the distribution of wall clock time among the different ZFS components under this workload. Our findings, presented in Section 3.3.1, show that approximately 80% of the overall latency is spent on persisting LWBs, most of which is software overhead. We conclude that ZIL-LWB's structure and the ZIO pipeline as its persistence mechanism are unfit to take advantage of PMEM-level performance, motivating the development of ZIL-PMEM.

3.1 Benchmark Setup

Our main evaluation system has the following hardware configuration:

CPU 2 x Intel(R) Xeon(R) Silver 4215 CPU 2.50GHz
Mainboard Supermicro X11DPi-N(T)/X11DPi-NT, BIOS 3.1a 10/16/2019
DRAM 16 x Micron 8GiB DDR4 2933MT/s (18ASF2G72PDZ-2G9E1), evenly distributed across sockets.
NVMe 3 x Micron PRO 960GB NVMe, 512 byte namespace format (MTFDHBA960TDF)
PMEM 4 x Intel Optane DC Persistent Memory, 128 GB, (NMA1XXD128GPS), two per socket.

We use the following software stack:

Kernel Linux 5.9, Debian buster (5.9.0-0.bpo.5-amd64)
Userland Debian GNU/Linux (buster)
fio fio-3.23-28-g7064
bpftrace bpftrace v0.12.0
bcc v0.16.0-11-ga74413b0
OpenZFS Our tree of OpenZFS with support for *ZIL kinds* (Section 6.1). For this experiment, we use the ZIL-LWB ZIL kind. Note that using our tree rather than upstream OpenZFS is (slightly) favorable to ZIL-LWB because our tree contains performance optimizations in the ITX code that are shared among all ZIL kinds.

We use the following system configuration:

- We leave SMT enabled, resulting in 16 hardware threads per socket.
- We disable the entire second CPU in software using the `isolcpus=8-15,24-31` kernel command line parameter.
- We configure all Optane DIMMs in *AppDirectNotInterleaved* mode.
- We create a 40 GiB-sized *fsdax* namespace on the region of the first DIMM on the first socket.
- We create a 40 GiB-sized *devdax* namespace on the region of the first DIMM on the first socket.
- We partition each of the 3 NVMe drives into 10 equal-sized partitions each.
- We create a zpool called `dut` with the 30 partitions as top-level vdevs, and the *fsdax* namespace's /dev/pmem device as a SLOG. The reason for the large number of partitions is that we observed slightly improved *txg sync* performance, presumably due to a higher degree oif parallelism towards the NVMe drives.
- We create 8 datasets in the zpool, named `dut/ds$i` and mounted at `/dut/ds$i`.
- For all datasets, we configure `recordsize=4k` to match the fio workload and set `compression=off` to avoid CPU overhead in the ZIO pipeline during *txg sync*.

check defined

ref

We use *fio* to generate a workload of random 4KiB writes (`blocksize=4k`, `rw=randwrite`). Each of the one to eight `numjobs` threads performs random synchronous write system calls to a separate file per thread (`ioengine=sync`, `sync=1`, `direct=0`, `fsync=0`). Each file has a size of `size=100MiB` which means that the written data volume grows with `numjobs`, but the amount of dirty data (max. 800 MiB) remains well below the NVMe drive's bandwidth limits. To avoid scalability-bottlenecks in the ITX code, we place each thread's file onto a separate dataset `/dut/ds$jobnum/fio_file` using the `filename_format` option. For each value of `numjobs`, we measure for one minute (`time_based=1`, `runtime=60`), with a ramp-up time of 2 seconds (`ramp_time=2`),

and set `end_fsync=1`. Before we run the benchmark, we prepare the benchmark files with a separate fio invocation with the additional flag `create_only=1`.

We run this fio workload on the following storage configurations:

zil-lwb The zpool with ZIL-LWB ZIL kind as described above.

async The same configuration as above, but with `sync=disabled` set on all datasets. This setting effectively disables the ZIL (thereby violating the synchronous semantics) and serves as an upper bound for the performance that can be achieved by any ZIL implementation. The remaining work done by *async* is only CPU- and/or DRAM-bound, i.e., DMU object modification and ITX allocation (ref. Figure 2.1).

fsdax The fio configuration above, applied directly to the /dev/pmem0 block device (`direct=1` instead of `direct=0`, and `filename=/dev/pmem0` instead of `filename_format=...`). This configuration demonstrates the performance of the Linux block device emulation around the PMEM hardware. It includes the syscall overhead.

devdax The fio configuration above, with `ioengine=dev-dax`, applied directly to the `/dev/dax` namespace (`filename=/dev/dax0.1` instead of `filename_format=...`). In this configuration, fio mmaps the PMEM namespace directly and thereby bypasses the kernel completely.

3.2 Results

The following graphs show the achieved IOPS and per-thread latency by `numjobs`.

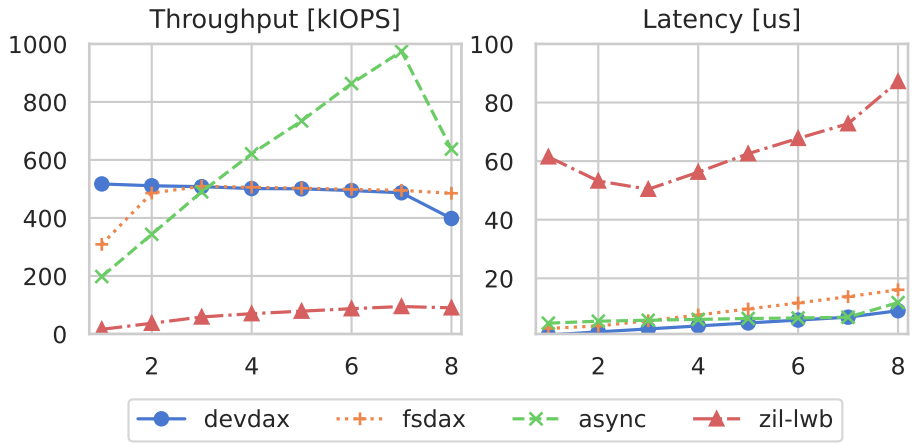


Figure 3.1: IOPS and latency ZIL-LWB compared to ZFS in async mode as well as the raw fsdax and devdax device.

ZIL-LWB only achieves approximately 10k IOPS at one thread and peaks at approximately 100k IOPS at seven threads. In contrast, ZFS in *async* mode starts with 200k IOPS with one thread and achieves over 900k IOPS at seven threads. This exceeds the performance of the PMEM hardware which mostly stays at 500k IOPS and demonstrates that the ZIL is clearly the bottleneck in the *zil-lwb* configuration. A look at the per-IOP average latency emphasizes the vast overhead that ZIL-LWB adds compared to what is possible with the raw PMEM hardware. Whereas ZFS in async mode only requires 5 us per IOP with `numjobs=1`, and raw writes to fsdax require approximately 3 us, ZIL-LWB with the same PMEM hardware as SLOG takes more than 60 us per write. The minimum latency achieved by ZIL-LWB is at `numjobs=4` at approximately 50 us before it starts increasing to up to 85 us at `numjobs=8`. Note that we do not use the results from the *dev-dax* as a baseline for PMEM hardware because the IOPS and latencies reported by fio do not match. For example, fio reports less than 1 us of latency but only reports 550k IOPS for *dev-dax* at `numjobs=1` whereas $\frac{1}{1\ us/IOP} = 10^6$ IOPS would be anticipated at this latency and thread count. Figure 3.2 shows the same latency data as Figure 3.1 above, albeit zoomed to a scale that allows us to distinguish *dev-dax*, *fsdax*, and *async* latencies.

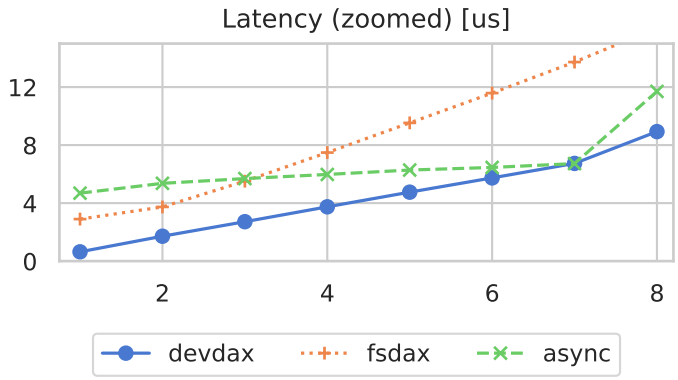


Figure 3.2: Zoomed section of the latency plot in Figure 3.1.

3.3 Analysis

By comparing the numbers for *async* and *zil-lwb*, it is safe to assume that regardless of the value for `numjobs`, ZIL-LWB adds at least 40 us of latency. We want to determine where this time is spent.

3.3.1 Analysis

ref

We use dynamic instrumentation (eBPF via bpftrace) to sum up the wall clock

time spent in ZFS functions that are executed by the fio threads during a synchronous write operation. We then use the model in Figure 3.3 to compute the time spent in the asynchronous part of the write operation, the ITX layer, and ZIL-LWB specific code. We visualize the results as stacked bar charts in Figure 3.4.

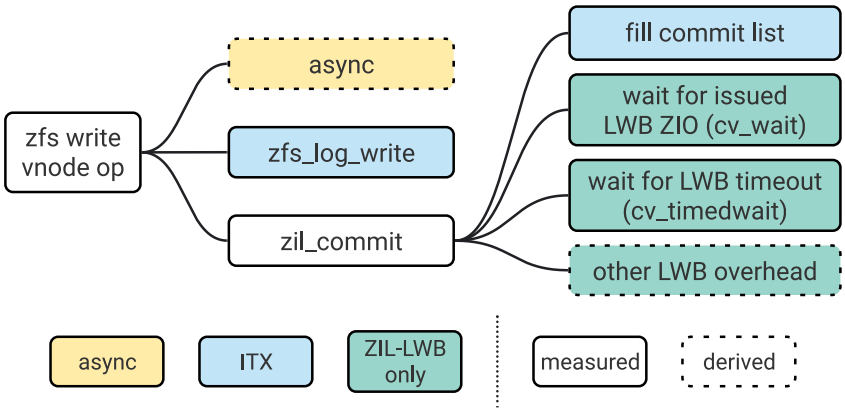


Figure 3.3: Our model of the time spent in ZFS by a fio thread that performs a write system call. The different colums represent the different levels of the dynamic call graph The nodes in each column describe all activity that happens at this level of the call graph. Activities with black text color are instrumented using eBPF. Gray text color indicates that the value was computed in post-processing by subtracting the sum of the columns instrumented time from the instrumented time of the parent in the column to the left. For example, to compute *async*, we compute `zfs_write - (zil_commit + zfs_log_write)`. The border style of the node visualizes the subsystem to which we attribute the time spent in the activity represented by the node.

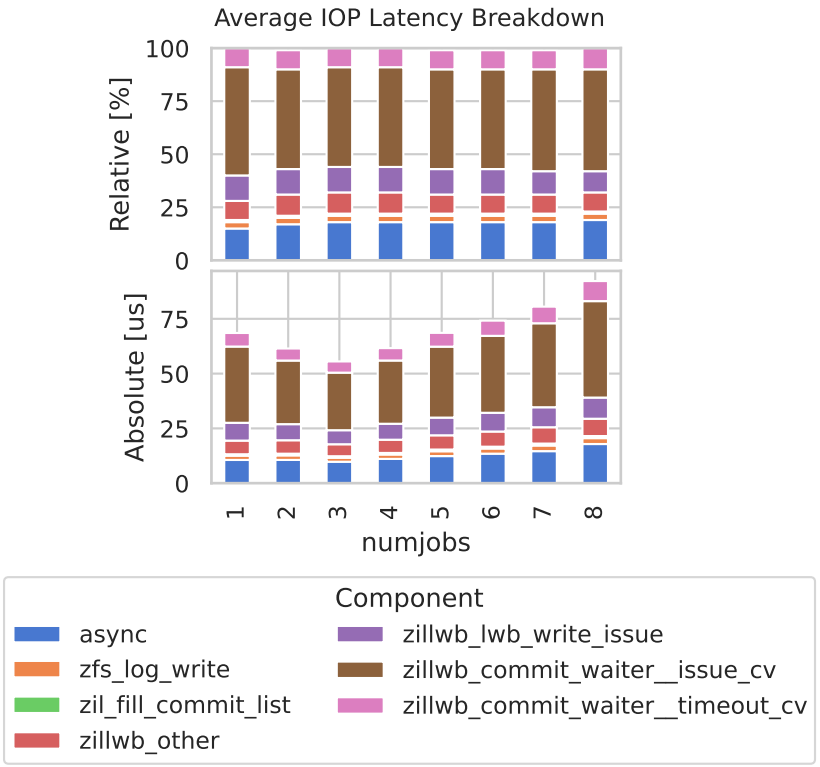


Figure 3.4: The relative and absolute breakdown of latency by activity. The absolute breakdown is normalized by the number of write operations. We compared the latencies in the breakdown with the latencies observed by fio to ensure that our accounting is correct under the given model.

Our observations are as follows:

- The instrumentation overhead is 5–10 us per IOP. (We compared the latencies that fio reports for the instrumented and uninstrumented run.)
- The relative distribution of latency remains mostly unchanged for the different values of `numjobs`.
- The *async* part of the write path and the ITX layer together only amount to approximately 20% of overall latency. The remaining 80% are spent on LWB-specific activities.
- The LWB timeout mechanism amounts to 4% of overall latency.
- At least 20% of overall latency are spent on filling LWBs, issuing their corresponding ZIOs, and other LWB-related activities.
- 45–50% of overall latency is spent waiting for the ZIO pipeline to persist the LWBs. In absolute numbers, the value ranges from 25–40 us. We have separately confirmed that ZIL-LWB uses 12 KiB LWBs which is the smallest

possible LWB size allowed by the implementation. In our benchmarking setup (per-thread datasts), each LWB only holds a single 4 KiB write log record with 192 B of additional metadata. This amounts to write amplification of 3x. However, even with this increased data volume per IOP, the results from the previous experiment suggest that with a conservative estimate of 3 us per 4k write to the raw PMEM hardware, we should expect less than 9 us of PMEM write time per LWB. Thus, a major fraction of the 25–40 us that are spent waiting for ZIO is actually pure software overhead.

Our observations lead us to the following conclusions:

First, we have reason to believe that the high overhead of ZIO is unlikely to be reduced to a degree that allows for exploitation of PMEM-level latency, regardless of whether the hardware is actual PMEM or simply very fast NVMe drives. The reason is that there is an inherent conflict of goals for ZIO: whereas the ZIL is strictly latency-oriented, all other conumers (*txg sync*, *scrubbing*, *zfs send/recv*) are throughput-oriented. The pipeline-oriented, parallelized architecture of ZIO is important for throughput and provides great flexibility, but increases latency through context switches. This problem is well known in the ZFS community, see [Ope20].

more quotes, find gh issues, etc

Second, the persistent representation of the ZIL as a chain of LWBs poses a severe and unnecessary overhead on PMEM. For one, the latency amortization provided by LWBs and the timeout mechanism is unnecessary at PMEM-level latencies where it is cheaper to persist the log records on an individual basis than batching them in LWBs and coordinating with other threads on the matter. And for another, since PMEM is byte-addressable, the persistent representation is no longer constrainted by disk block sizes, opening up the possibility of better space efficiency.

Chapter 4

Design Overview

Given the insights described in the previous chapter, we propose an alternative ZIL implementation called **ZIL-PMEM** that exclusively targets PMEM SLOG devices. In this chapter we define the project goals for ZIL-PMEM and provide a high-level overview of its design. The subsequent two chapters then introduce the core data structure and our approach to integrating it into ZFS.

4.1 Project Goals & Scope

4.1.1 Requirements

Coexistence ZIL-PMEM must coexist with ZIL-LWB in code and at runtime due to limited availability of PMEM hardware and the limitations of the ZIL-PMEM design.

Same Guarantees ZIL-PMEM must maintain the same crash consistency guarantees towards user-space as ZIL-LWB for both ZPL and ZVOL.

Simple Administration & Pooled Storage Pooling of storage resources and simple administration are central to ZFS [Bon+03]. ZFS should automatically detect that a SLOG device is PMEM and, if so, use ZIL-PMEM for all of the pool's datasets. No further administrative action should be required to fully benefit from ZIL-PMEM.

Correctness In the absence of PMEM media errors and data corruption, ZIL-PMEM must be able to replay all data that it reported as committed. The result must be the same as if ZIL-LWB would have been used in lieu. Specifically:

- Replay must respect the logical dependencies between log records.
- Logging must be crash-consistent, i.e., the in-PMEM state must always be such that replay is correct.
- Replay must be crash-consistent, i.e., if the system crashes or loses power during replay, it must be possible to resume replay after the crash. Resumed replay must continue to respect logical dependencies of log records.

Data Integrity Data integrity is a core feature of ZFS [Bon+03]. ZIL-PMEM must detect corrupted log records using an error-detecting code. Detected corruption must be handled *correctly* (as outlined in the previous paragraph) and *gracefully* with the following behavior as the baseline: "Assume a sequence of log records $1 \dots N$ where log record 1 does not depend on a log record and each record $i > 1$ depends on its predecessor $i - 1$. Data corruption in record $i \in 1 \dots N$ must not prevent replay of records $1 \dots i - 1$".

Low Latency The latency overhead of ZIL-PMEM compared to raw PMEM device latency for the same data volume should be minimal for single-threaded workloads. Multi-threaded workloads are addressed below.

Multi-Core Scalability Since PMEM is added as a pool-wide resource used by all of the pool's datasets, ZIL-PMEM should scale well to multiple cores. Barring PMEM throughput limitations, the speedup in throughput (IOPS) achieved by parallelizing synchronous I/O to multiple cores on a ZIL-PMEM system should be as follows:

1 private dataset per thread Always near-linear speedup.

1 shared dataset

ZPL filesystem No speedup.

ZVOL No speedup in standard mode, potentially sub-linear speedup in bypass mode (Section 6.3.5).

Maximum Performance On Intel Optane DC Persistent Memory We develop and evaluate ZIL-PMEM exclusively for/on Intel Optane DC Persistent Memory since it is the only broadly available non-volatile main memory product on the market. Whereas supercapacitor-backed persistent memory modules (NVDIMM-N) should be usable with ZIL-PMEM, our goal is to design a system that makes optimal use of the Optane hardware.

CPU-Efficient Handling Of PMEM Bandwidth Limits If the maximum write bandwidth to any kind of storage device is exceeded, the I/O stack must somehow apply back-pressure to avoid losing in-flight data. With PMEM, the I/O

stack is the CPU microarchitecture and the back-pressure manifests as stalling instructions. However, from the OS thread scheduler's perspective, threads whose instructions stall because they wait for PMEM are indistinguishable from actually busy threads. Yang et al. have shown that a single Optane DIMM's write bandwidth can be exhausted by one CPU core at 2 GB/s and that write bandwidth decreases to 1 GB/s at ten or more CPU cores. Since ZIL-PMEM shares PMEM among all datasets in a zpool, we expect bandwidth exhaustion to be a phenomenon that wil happen in practice. ZIL-PMEM should thus provide a mechanism to shift excessive PMEM I/O wait time off the CPU.

Testability ZIL-PMEM must be architected for testability. The core algorithms must be covered by unit tests. Further, ZIL-PMEM should be integrated into the ztest user-space stress test as well as the SLOG tests of the ZFS Test Suite.

4.1.2 Out Of Scope For The Thesis

The following features were omitted to constrain the scope of the thesis. We believe that our design can accomodate them without major changes.

Support For OpenZFS Native Encryption The ZIL-PMEM design presented in this section does not address OpenZFS native encryption. Intel Optane DC Persistent Memory supports transparent hardware encryption per DIMM at zero overhead. In contrast, OpenZFS native encryption is per dataset and software-based. Given these significant differences in data and threat model, ZIL-PMEM cannot rely on Optane hardware encryption. Instead, ZIL-PMEM would need to invoke OpenZFS native encryption and decryption routines when writing or replaying log entries.

cite spec

Protection Against Scribbles Scribbles are bugs in the system that accidentally overwrite PMEM, e.g., due to incorrect address calculation or out-of-bounds access in the kernel. PMEM-specific filesystems such as PMFS and NOVA-Fortis have already introduced mechanisms to protect againt scribbles [Dul+14; Xu+17]. We believe that these mechanism can be applied to our design as well.

4.1.3 Limitations

The following features are deliberately not addressed by our design. More experimentation and experience with ZIL-PMEM will benecessary to determine which features are useful in practice, how they can be realized, and how they interact with the existing requirements.

No NUMA Awareness Yang et al. recommend to "avoid mixed or multi-threaded accesses to remote NUMA nodes. [...] For writes, remote Optane's latency is

2.53x (ntstore) and 1.68x higher compared to local" [Yan+20]. We do not account for this behavior in the design and do not evaluate ZIL-PMEM in a NUMA configuration.

future work

No Data Redundancy ZIL-PMEM provides data integrity protections but does not provide a mechanism for data redundancy.

future work

Only Works With SLOGs Our approach to integrate ZIL-PMEM into ZFS is only applicable to PMEM SLOGs and does not work for a zpool that uses PMEM as main pool vdevs. Such pools continue to use ZIL-LWB.

No Software Striping Our design only supports a single PMEM SLOG device. Users may wish to use multiple PMEM DIMMs to increase log write bandwidth. With Intel Optane DC Persistent Memory, multiple PMEM DIMMs can be interleaved in hardware with near-linear speedup [Yan+20]. Whereas software striping would be the natural approach to ZFS, it will be non-trivial to achieve the same speedup as hardware-based interleaving.

No Support For `WR_INDIRECT` ZIL-LWB writes the data portion of large write log records directly to the main pool devices. The ZIL record then only contains metadata such as `mtime` and a block pointer to the location in the main pool. This technique avoids double-writes which is particularly advantageous if the pool does not have a SLOG, which in turn is a use case that ZIL-PMEM does not address (see above). Further, if a SLOG is available, `WR_INDIRECT` log record write latency is likely to be dominated by the main pool's IO latency if it consists of regular block devices. If the main pool's IO latency were acceptable, a fast NVMe-based ZIL-LWB SLOG or no SLOG at all would likely be sufficient for the setup in question.

Space Efficiency ZIL-PMEM is allowed to trade PMEM space for time and simplicity when presented with the option. Our justification is twofold. First, PMEM capacities are significantly higher than DRAM. For example, the smallest Intel Optane DC Persistent Memory DIMM offered by Intel is 128 GiB. [**optanepricing_missing**] Second, the maximum amount of log entry space required from any ZIL implementation is a function of the maximum amount of dirty data allowed in the zpool. For ZIL-LWB, small SLOG devices of 16 to 32 GiB are sufficient in practice. Thus, there is sufficient headroom for PMEM space usage in ZIL-PMEM.

ref background section? this sentence should remain, though

4.2 Design Overview

ref ix systems truenas M

We introduce the concept of *ZIL kinds* to ZFS. The ZIL kind is a pool-scoped variable that determines the pool's strategy for persisting ZIL entries. A zpool's

ZIL kind is determined by the following rule: if the pool has exactly one SLOG and that SLOG is PMEM, the ZIL kind is ZIL-PMEM. Oterwise, it is ZIL-LWB. As explained in Section 2.3.4, ZIL-LWB uses ZFS's metaslab allocator to allocate log-write blocks (LWBs) from the storage pool with a bias towards SLOG devices. In contrast, ZIL-PMEM disables metaslab allocation for the PMEM SLOG device and uses the PMEM space directly.

We partition the PMEM SLOG's space into fixed-size contiguous segments called *chunks*. We develop a data structure called *PRB* that consumes theses chunks and exposes the abstraction of an unordered persistent storage layer for *log entries*. A log entry is the unit of data that can be written to and read from PRB. It consists of the the ZIL's log *record* and PRB-specific metadata. PRB scales to many concurrent writers and features a mechanism to avoid excessive on-CPU waiting for PMEM I/O. Log entries stored in PRB are automatically garbage-collected when they become obsolete because their transaction group has been synced.

PRB provides a pool-wide storage substrate for log entries but does not define any structure. This is the role of the *HDL* abstraction which implements a mostly sequential log structure on top of PRB. Each dataset in the zpool has a separate HDL to which it writes log entries. The HDL adds metadata to the entry that attributes it to the HDL and encodes the log's structure. After a system crash, the HDLs scan PRB for entries that need to be replayed and *claim* them to hold them back from garbage collection. For replay, the HDL API provides a callback-based interface that allows its consumer to apply the changes that are encoded in the log entries in sequential, deterministic order. Loss of entries, e.g., due to data corruption, is handled as gracefully as possible under the constraints of the logical dependencies encoded in the log structure.

Pools of the ZIL-PMEM ZIL kind use PRB/HDL to persist ZIL log records. For this purpose, we refactor the existing *zil.c* code module. Before the introduction of ZIL kinds, the struct `zilog_t` implemented all ZIL-related functionality. With ZIL kinds, `zilog_t` acts as an abstract base class that encapsulates shared ZIL functionality. This includes the definitions of the ZIL log record format and the *itxg* data structure that tracks which log entries need to be persisted when synchronous semantics are requested by a syscall. The code that is responsible for persisting log entries resides in ZIL-kind-specific subclasses. The pool's ZIL kind determines which subclass is instantiated at runtime. The subclass for ZIL-LWB is `zilog_lwb_t` which contains the original LWB code that we move from *zil.c* into the new *zil_lwb.c* module. The subclass for ZIL-PMEM is `zilog_pmem_t`. It is a thin wrapper around HDL's methods for writting and replaying log entries. `zilog_pmem_t` is implemented in the *zil_pmem.c* code module. This module also

language

contains the code that sets up PRB/HDL on top of the PMEM SLOG vdev, and the code that synchronizes the lifecylces of HDLs their corresponding datasets.

check

The next two chapters describe our design in detail. Chapter 5 describes our main contribution — the PRB/HDL data structure. The integration of PRB/HDL into ZFS is then presented in Chapter 6.

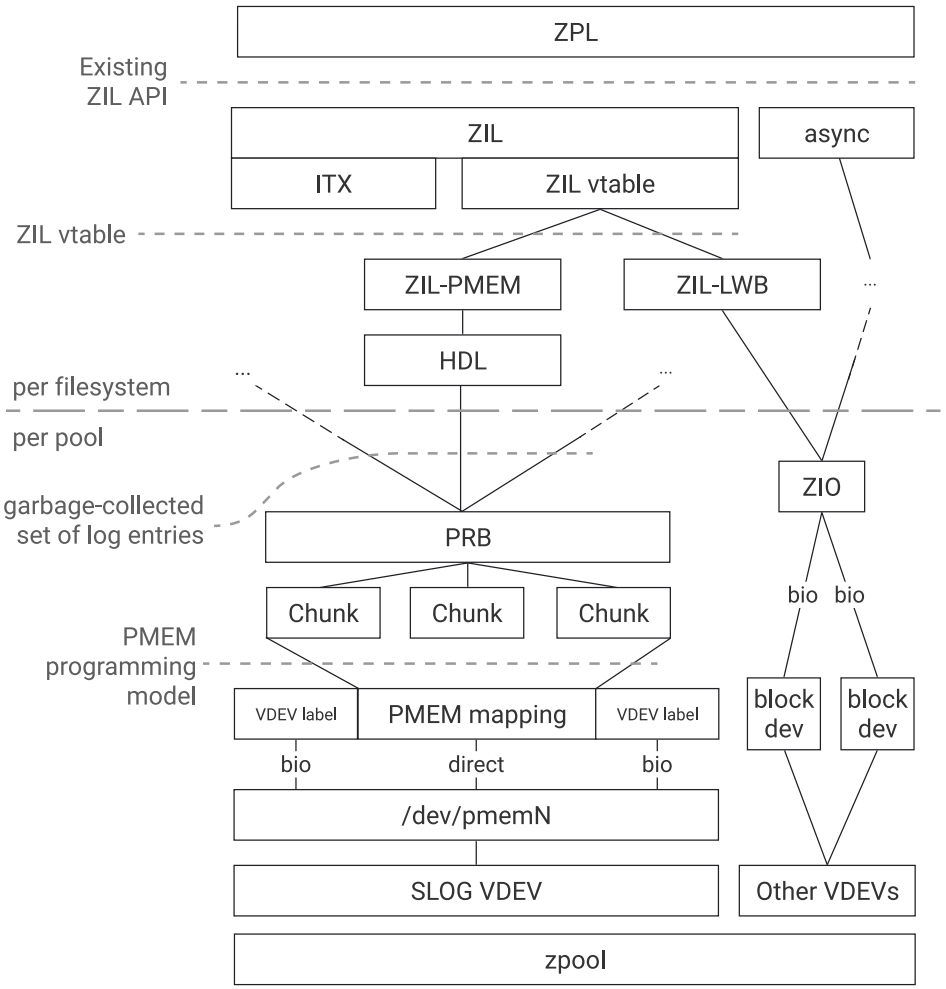


Figure 4.1: Overview of the system architecture as described in this section.

actual figure

Chapter 5

The PRB/HDL Data Structure

In this chapter we describe the PRB/HDL data structure which implements the bulk of ZIL-PMEM's functionality. PRB/HDL abstracts the allocatable space of a PMEM SLOG vdev as virtual persistent logs for each dataset in the pool. The *zil_pmem.c* module, which we present in Chapter 6 uses this abstraction to implement the ZIL-PMEM ZIL kind. We present the design and implementation of PRB in a top-down manner. In Section 5.1 we isolate the requirements that ZIL-PMEM puts on PRB by recapitulating how the ZIL fits into ZFS's overall architecture. Afterwards, in Section 5.2, we give a high-level overview of our design. Section 5.3 presents the virtual log abstraction that is exposed by HDL and Section 5.4 describes the high-level approach for replay. Sections 5.5, 5.6, and 5.7 then progressively refine our understanding of replay and explain how data corruption and crash-consistency is addressed by HDL. Subsequently, we describe PRB which is the storage substrate that the HDLs use for persistence: Sections 5.8 and 5.9 present the data structures that we use for PMEM space management and log entry storage. In Section 5.10 we describe the algorithm that traverses the in-PMEM data structure during log recovery. Secton 5.11 then explains how garbage collection removes entries from it. The low-latency and CPU efficient design of the write path is then presented in Section 5.12. Finally, Section 5.13 provides an overview of the PRB/HDL API that is consumed by *zil_pmem.c*.

language, help

help, don't want to repeat 'in-PMEM structure'

5.1 Context & Requirements

Remember from Section 2.3 that all ZPL filesystem state is represented as DMU objects. When a filesystem call modifies DMU objects, it must do so from within a DMU transaction T_i that belongs to a transaction group $T_{i_{txg}}$. After the system call handler has completed the DMU transaction by calling `dmu_tx_commit(T_i)`,

the logical change C_i made in T_i is not yet persisted to stable storage. Instead, the DMU accumulates the changes from many DMU transactions in DRAM as so-called *dirty state*, grouped by the transaction's transaction group (txg). This dirty state is eventually persisted to ZFS's on disk state in a procedure that, through copy-on-write techniques, is atomic at the granularity of txgs.

The ZIL bridges the gap between txg syncs so that synchronous VFS operations can safely correctly return to userspace, even though their DMU transaction's txg has not yet been synced to disk. For that purpose, a VFS operation summarizes eacht logical change C_i that it makes in each DMU transaction T_i as an *intent log transaction* (ITX). Before completing the DMU transaction through `dmu_tx_commit`, it *assigns* the ITX to the ZIL. Before a synchronous operation reports completion upstack, it calls `zil_commit` to append the ITXs that are relevant to the operation to a sequential persistent log.

In upstream OpenZFS, the only available implementation of this persistent log is the chain of *log write blocks* (LWBs) which we described in Section 2.3.4. With the introduction of ZIL kinds (details in 6.1), the persistence layer becomes pluggable: whereas the shared *itxg* data structure continues to determine the *commit list* that describes which ITXs need to be persisted in what order, the ZIL kind is free to represent the log structure however it sees fit.

However, regardless of the persistence strategy, all ZIL kinds must implement the same interface to recover committed state (`zil_replay`) . The ZPL invokes `zil_replay` during the mounting procedure of a filesystem. It provides a callback that the ZIL kind must invoke for precisely those log records (= encoded ITXs) whose changes C_i were lost in the crash. (Their txg $T_{i_{txg}} > precrash\_txg$ where $precrash\_txg$ is the last synced transaction group before the system crashed.) The callback re-applies the change encoded in the log record, thereby recovering the committed state.

The reader should remember that the set and order of log records for which the callback is invoked is not necessarily intuitive because a) ITXs may be reordered by the ITXG structure and b) the log records must be filtered by $precrash\_txg$. We refer to Figure 2.2 in Section 2.3.3 to an example and more details.

We derive the following abstract view of **what** needs to be stored **per log**:

- The log records themselves.
- The transaction group of the DMU transaction that the log record encodes.
- Structural information that defines replay order and/or logical dependencies between log records that replay must respect.
- For replay, the precrash txg to discern replayable from obsolete log records (see previous section).
- For replay, some representation of *replay progress* to enable resumption of replay if the system crashes during replay. (The log record format and replay callbacks are not idempotent.)

For ZIL-PMEM, we put the following requirements on the PRB/HDL:

- On the write path, the overhead added to the raw log record write time should be minimal.
- It must scale well on a multicore system since many datasets write their log records in parallel. This scalability requirement includes efficient use of CPU time in case the PMEM write bandwidth is exceeded.
- It must garbage-collect log records after they are obsoleted by *txg sync* and/or completed replay.
- It must provide the replay interface described above.
- It must detect data corruption using checksums. (Repair and redundancy are out of scope for this thesis, see Section 4.1.1.)

5.2 Approach

We introduce the pool-wide *PRB* object which abstracts the PMEM SLOG vdev's space as a persistent, unordered set of *log entries*. A log entry encodes a logical change to a particular dataset that was applied in a single DMU transaction. PRB provides facilities for adding log entries to the set and for iterating over its contents. It automatically garbage-collects entries after they become obsolete because their transaction group has synced.

Log entries are not written directly to PRB but instead to the *HDL* object of dataset that they affect. A HDL is a virtual log built on top of PRB that organizes individual entries for a single dataset in a mostly sequential structure. After a system crash, the HDL provides a replay facility that recovers the replayable entries, orders them according to their logical dependencies, and handles missing entries.

At any time, there exists one HDL for each head dataset in the pool. They are set up early during pool import and torn down late during export. If a dataset is created or destroyed, the corresponding HDL is set up or torn down as well. HDL is stateful. Its internal states represent the different phases that a dataset goes through with regards to the ZIL.

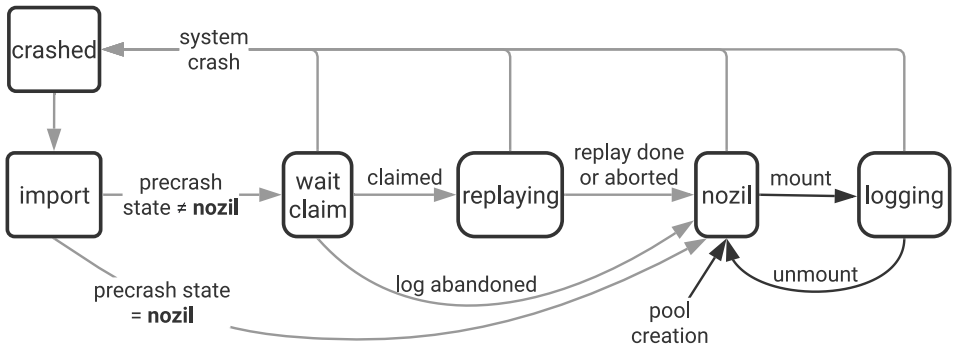


Figure 5.1: The runtime states of a HDL.

When the HDL is created, it does not have a log and is in state *nozil*. When the dataset is mounted, the HDL allocates a *log GUID* and persists it to the ZIL header. The log GUID uniquely identifies the log's entries in the PRB. The HDL is now in state *logging* and threads can write entries to it. If the dataset is unmounted, the log GUID is discarded and the log transitions back to state *nozil*. Otherwise, the HDL remains in state *logging* until the system crashes.

When a zpool is imported, the import procedure examines the ZIL header of each dataset to recover the HDL's runtime state from before the crash. If the HDL state was *nozil*, there is nothing to do and the HDL transitions to that state. If the HDL state was *logging* or *replaying*, it must be claimed *txg sync* starts. For HDLs in state *logging*, the claiming procedure saves the zpool's *last synced* txg as the *precrash-txg* and transitions the HDL and ZIL header to state *replaying* in the initial replay position. If the HDL was already in state *replaying* at import time, the precrash-txg and replay position are recovered from the ZIL header. At this point, all HDLs are either in state *nozil* or *replaying* but claiming is not yet complete. For every HDL in state *replaying*, the claiming procedure scans the PRB for log entries that need to be held back for replay. Entries that are held back by at least one HDL are exempt from PRB's garbage collection. After this scanning step is complete, the claiming phase is done and the txg sync thread starts. HDLs are not replayed until their dataset is mounted by the user. After replay is complete, the HDL discards the log GUID and transitions to state *nozil*.

At this point, the mount procedure behaves as if log replay had not happened and starts a new log with a new log GUID, thereby closing the circle. Note that the HDL (and PRB) are able to tolerate a system crash in any of the HDL or ZIL header states. We address this issue in Section 5.7.

5.3 HDL: Log Structure

The structure of the virtual log that each HDL represents is defined by metadata stored with each entry that is written to PRB.

We **attribute** entries to a given HDL's log through the *log GUID*:

Log GUID A 128 bit random identifier stored in the HDL's ZIL header and repeated in every entry written through that HDL.

The following pieces of metadata define the structure of the log.

Transaction Group (txg) The transaction group in which the change encoded in the entry was or would have been synced out by *txg sync*.

Generation Number (gen) The log is a sequence of generations, each of which contains many log entries. The generations encode logical dependencies between entries. Entries within the same generation do not depend on each other. Entries from newer generations unconditionally depend on all entries in all previous generations. We represent generations as unsigned 64-bit non-zero integers.

Generation-Scoped ID (gsid) Within a generation, we identify entries by another by the $gsid$, another unique unsigned 64-bit non-zero integer. As the name suggests, the $gsid$ only needs to be unique within a generation.

Note that the tuple $(gen, gsid)$ unqiuely identifies an entry within a log.

We visualize the structure of the log in a grid. The rows represent the transaction group (txg) and the columns represent the generation (gen). For readability, we represent entries not by $(gen, gsid)$ but by a single unique letter. The projection of entries onto the horizontal axis shows the dependency relationship encoded by the generations. The projection of entries onto the vertical axis shows the sets in which entries are garbage-collected.

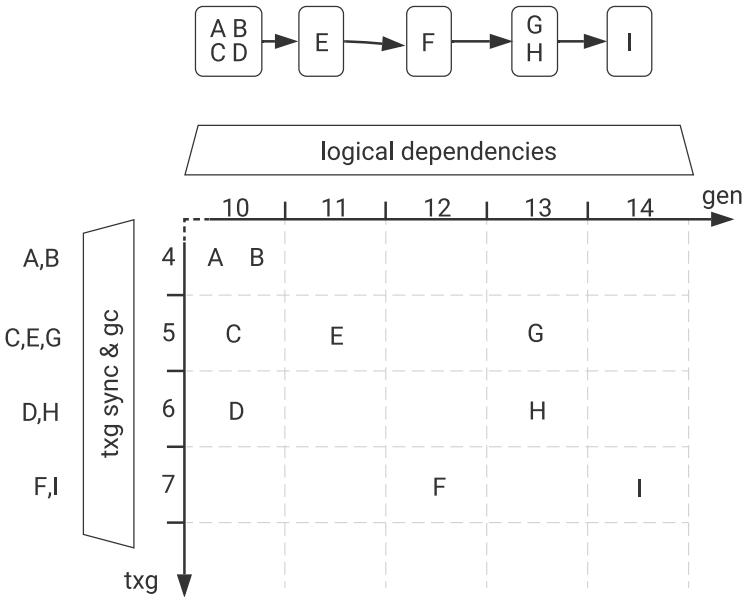


Figure 5.2: Structure of a single HDL's log as described in the previous paragraph.

5.4 HDL Replay: Basic Approach

Replay must apply the changes that were successfully logged to the HDL but whose DMU transaction did not sync before the system crashed. To accomplish this task, it scans the PRB for entries that belong to the HDL and determines a *replay sequence* S.

$$\begin{aligned} &\text{Entry } E_i \in S \\ \Leftrightarrow &E_i.log\_guid = HDL.log\_guid \wedge E_i.txg > precrash\_txg \end{aligned}$$

S is sorted in *replay order*, which is the order the changes encoded in the entries need to be applied. Given two entries E_a, $E_b \in S$ for the same HDL, the replay order is a total order defined by

$$\begin{aligned} &E_a <_{replay} E_b \\ \Leftrightarrow &(E_a.gen, E_a.gsid) <_{lexicographical} (E_b.gen, E_b.gsid) \\ \Leftrightarrow &E_a.gen < E_b.gen \vee (E_a.gen = E_b.gen \wedge E_a.gsid < E_b.gsid) \end{aligned}$$

For our example in Figure 5.2, this results in the following replay sequences, depending on the value of $precrash\_txg$.

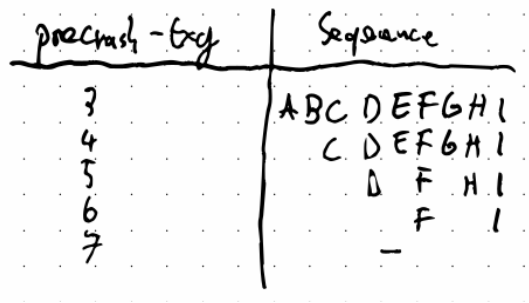


Figure 5.3: Replay sequences for the log depicted in Figure 5.2, by $precrash\_txg$.

We use a total order so that we can precisely encode replay progress by storing the last-replayed $(E.gen, E.gsid)$. This is necessary for crash consistency because log entry replay is not idempotent. We revisit this topic in Section 5.7.

Note that the definition of the replay order does not account for overflows of gen or $gsid$ — entries written after the overflow would be incorrectly ordered as smaller than entries written before the overflow. Overflows could be handled in software, e.g., by temporarily destroying the log of the dataset and creating a new one with a fresh log GUID. However, our implementation has no such provisions because even with the (absurdly) conservative of 1 ns write time per log entry, the first overflow event would only occur in 584 years if gen starts at 1.

5.5 HDL Replay: Dependency Tracking

Replay is complicated by the fact that the entries that were stored in the PRB can be lost. For example, bitflips in PMEM might corrupt the entry's log GUID or PRB-internal metadata. If an entry E_m with $E_m.txg > precrash\_txg$ is missing, any entry E_d that logically depends on E_m ($E_m < E_d$) and is for an unsynced txg ($E_d.txg > precrash\_txg$) must no longer be replayed. However, all entries E_p that do not depend on E_m and need replay should still be replayed (their $E_p \leq E_m \wedge E_p.txg > precrash\_txg$).

We detect missing entries through a set of counters that we store in the metadata of each entry. For an entry E, the counter $E.C_{ctxg_i}$

$$E.C_{ctxg_i} := \#\{D : D.gen < E.gen \wedge D.txg = ctxg_i\}$$

stores the number of entries that were written to the log since its inception, for a DMU transction with txg $ctxg_i$, until generation $E.gen$. During replay, we first construct the replay sequence (example in the previous section, Figure 5.3). Then, we re-compute and compare the counters for each entry in the replay

sequence. If an entry E_m has been lost, the counters of any dependent entry E_d (their $E_d.gen > E_m.gen$) will not match, causing replay to stop with E_d as a *witness*. Missing entries in the last generation (the "tail" of the log) cannot be detected with this scheme.

The per-txg scoping of the counters is critical to accomodate garbage collection. Suppose we only used a single sequence counter for all log entries of a HDL. After a txg t_{synced} has been synced, PRB garbage-collects all entries $\mathcal{S}_{gc}$ that are obsolete:

$$\mathcal{S}_{gc} := \{E : E.txg \leq t_{synced}\}$$

These entries no longer show up when the claiming or replay procedures scan the PRB for entries with the HDL's log GUID. If we used a single sequence counter to check for missing entries, we would be unable to discern garbage-collected entries from missing entries.

It is sufficient include only those counters $E.C_{txg_i}$ in the entry metadata whose transaction groups txg_i had not yet been synced at the time that $E.gen$ started. The reason is that a) the replay sequence only contains entries in unsynced txgs and b) E only depends on entries D with $D.gen < E.gen$. Since there are only three possible unsynced txgs at any time (*open*, *quiescing*, *syncing*), we only need to store three (txg_i, C_{txg_i}) tuples per log entry. In fact, since txgs never skip a number, we only store the value of txg_{open} and recompute

$$txg_{quiescing} = txg_{open} - 1$$
$$txg_{sycing} = txg_{open} - 2$$

compact encoding is just an idea, not in impl yet

We use the same algorithm to compute the counters on both the write and recovery path. This works because the write path must use monotonically increasing generation numbers, and the entries in the replay sequence are sorted in that order. The counters are stored in a table called *live table*. It has four rows $R_i := (txg, ctr), i \in \{0, 1, 2, 3\}$. When an entry E is written or visited, it modifies the counter in the row with index $I_T := T \bmod 4$ where $T := E.txg$. We distinguish the following conditions:

- If $R_{I_T}.txg = T$, we simply increment $R_{I_T}.ctr$ and are done.
- If $T > R_{I_T}.txg$, we can infer that txg $R_{I_T}.txg$ must have been synced out because there are only three unsynced txgs at any given time but four array entries that are reused in a circular manner, curtesy of indexing by $T \bmod 4$. In that case, we can discard the state in the row and reuse it for T by setting $R_{I_T} \leftarrow (T, 1)$.

- Conversely, if $T < R_{I_T}$, we can infer that the entry we are writing is for an already synced txg and hence obsolete. In that case, we do not change the *live table* and turn the entry write operation into a no-op.

Note that the use of 4-ary arrays allows the use of *bitwise-and* for indexing, which is a common ZFS idiom (`table[txg&3]`).

To detect the start of a new generation, and support crash-consistent replay (Section 5.7), we maintain a separate variable $E_{last} := (gen, gsid)$. When an entry E is written or visisted, we compare it to E_{last} using the following rules:

$E < E_{last}$ This case is prohibited because the API contract for log writers is that generation numbers must be monotonically increasing. We consider the log corrupted if such an entry E is found.

$E.gen = E_{last}.gen$ The writer did not start a new generation and we leave the *last table* unmodified.

$E.gen > E_{last}.gen$ We create a copy of the *live table*. We refer to this snapshot as *last table*.

Regardless of whether a new generation was started or not, we always update $E_{last} \leftarrow E$ and increment the counter in the *live table* as described in the previous paragraph after evaluating the rules above.

The counters $E.C_{txg_i}$ have the same value for all entries in generation $E.gen$ because, by defintion, they only count entries written up to but not including $E.gen$. Hence we compute them from the *last table* once and cache the results until the next generation is started:

1. Determine row index $i_{max} := \max_{i \in \{0,1,2,3\}} R_i.txg$ where $R_i \in$ last table.
2. Invariant: $R_{i_{max}}.txg$ is the highest potentially unsynced txg in generations $< open\_gen$. We make the most conservative assumption that $R_{i_{max}}.txg$ was the *open txg* in generation $open\_gen - 1$.
3. Find the counters for *quiescing* and *syncing txg*. We scan the *last table* twice to find counters for transaction groups $R_{i_{max}}.txg - 1$ and $R_{i_{max}}.txg - 2$. If the *last table* does not contain those counters, we use a value of zero.

Figure **??** contains an illustration of the process that converts the *last table* into the counters table. We provide an extensive example in Figure 5.5.

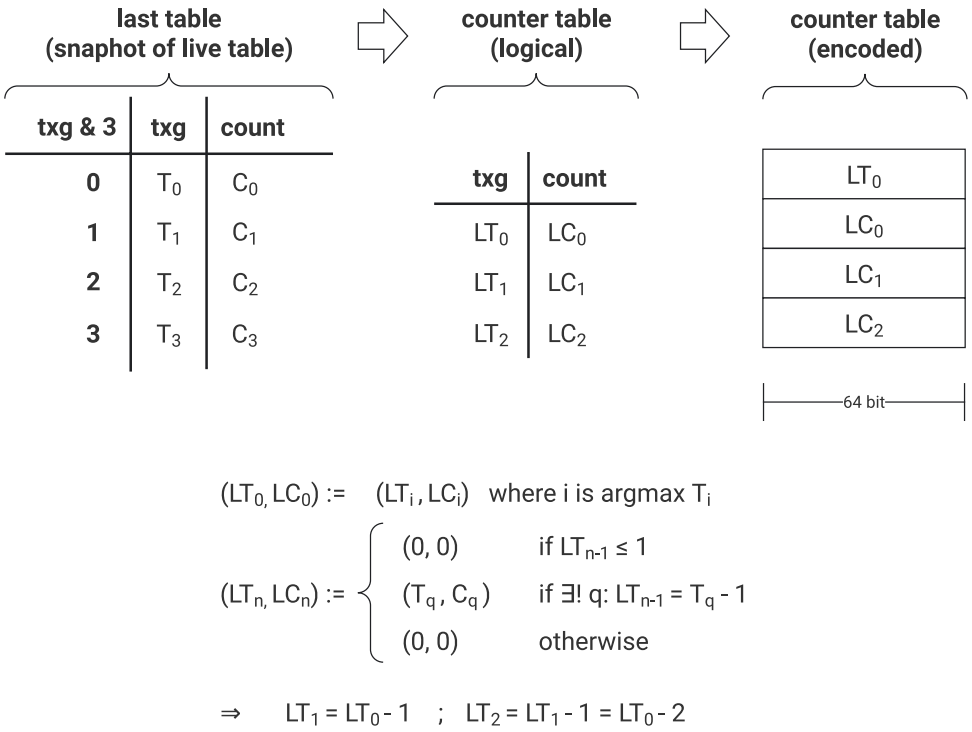
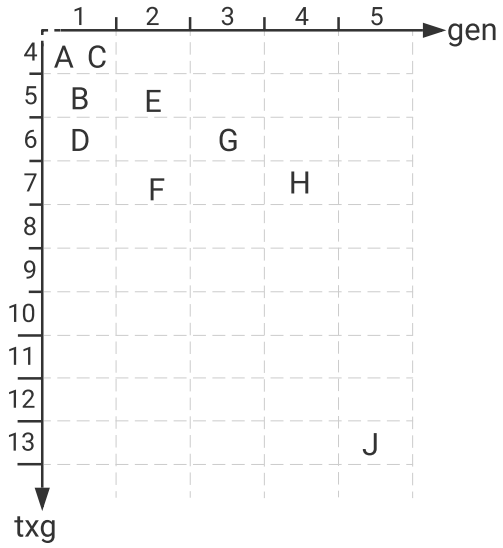


Figure 5.4: Visualization of how the counters table is computed from the *last table* structure.



live table over time

txg&3	txg	count
0	4|8	A,C | H
1	5|13	B,E | J
2	6	D,G
3	7	F

emitted counter tables for each generation

generation	(LT_i, LC_i) i=0		i=1		i=2	
1	0	0	0	0	0	0
2	6	1	5	1	4	2
3	7	1	6	1	5	2
4	7	1	6	2	5	2
5	8	1	7	1	6	2

each row represents one counter table

txg	count
6	1
5	1
4	2

Figure 5.5: Example for the computation of the counters that are stored in the entry. The following entries are written to the log: A,B,C,D in generation 1; E,F in generation 2; G in generation 3; H in generation 4; J in generation 5. Note that their txgs differ sometimes but stay in a corridor of at most three txgs. The table at the bottom shows the counter values in the entry headers: each row represents the counter table that is stored in the entry headers of one generation. The table at the upper right shows the *live table*'s content over time. Instead of counters, we show the entries that would cause the specific counter to be incremented. We visualize row-reuse by separating new row content with a "|" symbol in each cell of the reused row.

5.6 HDL: Model For Data Corruption

The use of counters to validate that no log entries are missing relies on the assumption that random data corruption is incapable of "forging" new log entries. If this were possible, a forged entry could compensate a lost entry in the counter table. The loss of the entry would go unnoticed and the replay of the forged entry would likely corrupt the dataset.

We believe that it is sufficiently unlikely for random data corruption to accidentally forge an entry. A forged entry would have to fulfill the following requirements to affect counter validation for a given log L:

- It must have been correctly added to PRB's data structures such that a scan of PRB will find it during recovery. (We address the PRB's data structure in Section 5.9.)
- Its *log GUID* value must match log L's log GUID.
- Its $(gen, gsid)$ must not collide with the original entries. Such a collision would be noticed when constructing the replay sequence. (Our implementation uses a b-tree that identifies and sorts nodes by $(gen, gsid)$).
- Its *txg* value must be in the correct corridor of unsynced txgs and generation. If the txg is too old or too young, the forged entry would be noticed by the dependency tracking code when re-computing the counters (see previous section).

Apart from random data corruption, we have considered the following scenarios and concluded that they are out of scope for ZIL-PMEM:

- Implementation errors on the write path.
- Firmware bugs, e.g., in the Optane DIMM's wear-leveling layers, that could make old entries re-appear.
- Deliberate injection of log entries by a malicious party with write access to PMEM. This needs to be part of the threat model if ZIL-PMEM is extended to support OpenZFS Native Encryption. Forged entries should be trivially detectable by cryptographically authenticating the entry header, e.g. using the AEAD ciphers that are already in use for the encryption feature.
- Accidental reuse of *log GUIDs*. If two HDLs use the same log GUID to write their entries, the HDLs will each claim their and the other HDL's entries. This scenario is very unlikely because we generate log GUIDs as 128 bit random numbers and check for collisions with other HDLs.

5.7 HDL: Replay Crash-Consistency

The PRB and its HDLs must be able to tolerate a system crash at any time in their lifecycle. With regards to recovery, this means that claiming must be restartable and replay resumable across crashes. Further, the restarted or resumed recovery procedure must be able to handle online and offline loss of entries due to data corruption.

Crash-consistency for claiming is trivial because it runs during pool import before txg sync starts. Thus, all changes made by claiming are accumulated in DRAM and atomically persisted by the first transaction group that is synced after pool import. From the perspective of HDLs that are in state *logging*, this first transaction group is $precrash\_txg + 1$. If the system crashes before that txg is synced, a subsequent import attempt restarts from the same state as the previous one. HDLs that are already in state *replaying* make no modifications during claiming — they only build up DRAM state, i.e., the set of entries held back for replay.

Replay is complicated by two additional aspects:

- Replay of an individual log entry is not idempotent.
- Replay is spread across several transaction groups and hence not atomic. (See this chapter's Section **??** on OpenZFS background.)

Our solution is to externalize all state of the replay algorithm, including the counters used for dependency tracking, to a structure called *replay state*. Whenever we replay an entry, we checkpoint *replay state*. We store the latest checkpoint in the ZIL header every time we replay an entry. If the system crashes, we recover *replay state* from the checkpoint in the ZIL header.

The following variables are stored in *replay state*:

$(\mathbf{gen_{last}}, \mathbf{gsid_{last}})$ The $(E.gen, E.gsid)$ of the last entry E that was replayed.

Live Table The *live table* at the time the entry was replayed, i.e., R_i with $i \in 0, 1, 2, 3$.

Last Table The *last table* at the time the entry was replayed.

E_{seal} An artificial entry header that is used to detect lost entries at the end of the log structure.

The crash-safe replay procedure performs the following steps:

1. Claiming reads the log GUID from the ZIL header, then scans the PRB and holds back the log's entries from garbage collection.

- If the HDL is in state *logging*, claiming transitions the ZIL header to *replaying* and initializes the *replay state* checkpoint in the ZIL header.
  - $(gen_{last}, gsid_{last})$ is $(0, 0)$.
  - *live table* and *last table* are zeroed out.
  - E_{seal} is determined as follows.
    (a) Construct a temporary replay sequence $S_{tmp} = \{E_s : s \in 1, \ldots, n\}$.
    (b) In DRAM, append an artificial entry E_{n+1} to S_{tmp}. $E_{n+1}.gen = E_n.gen + 1$, $E_{n+1}.gsid = 1$, and $E_{n+1}.txg = E_n.txg$.
    (c) Iterate over S_{tmp} and compute the counter tables. Validate the tables for $E_{1,\ldots,n}$. When arriving at E_{n+1}, assign the expected table to E_{seal}.
- If the HDL is already in state *replaying*, the ZIL header is not modified.

2. Replay recovers its *replay state* from the checkpoint C in the ZIL header.
3. Replay constructs a replay sequence S from the held back entries.
4. While replaying S, the following happens for each entry $E_s \in S$:
   (a) Skip E_s if it has already been replayed, i.e., $(E_s.gen, E_s.gsid) \leq (C.E_{last}.gen, C.E_{last}.gsid)$. Otherwise:
   (b) Create a backup checkpoint C_b of *replay state*.
   (c) Perform dependency tracking as described in the previous section.
   (d) Create a checkpoint C_{E_s}.
   (e) In one DMU transaction, replay E_s and store C_{E_s} in the ZIL header.
   (f) If replay fails or the DMU transaction fails, abort the transaction and roll back the in-DRAM version of *replay state* to C_b.

Note that steps 1 and 3 construct a *new* replay sequence every pool import. This allows handling of offline data corruption in PRB. Assume that we lose an entry E_m while the system is offline. If $E_m.txg \leq precrash\_txg$, the loss of the entry is unnoticed because E_m is by definition not part of the replay sequence. If E_m is skipped by step 4a, the loss of the entry might be worth reporting but does not affect replay because it has already been replayed. Otherwise, dependency tracking will eventually detect that E_m is missing if, and only if, any other entry depends on it. E_{seal} is an artificial dependency to ensure that lost entries in the last generation are be detected.

this is not yet implemented

We handle online data corruption as follows. Claiming and replay only operate on DRAM-buffered copies of the entry metadata called *replay node*. When replaying an entry, the replay callback is forced to use a special function that attempts to read the entry from PRB and buffers it in DRAM. Before allowing access to the entry, it ensures that the entry metadata still matches the data in the *replay node*. If either the read from PRB fails (e.g., due to checksum errors)

or if the metadata does not match, we know that the entry has been corrupted since the PRB scan.

Note that the last generation $E_{seal}.gen - 1$ of the log structure is not protected against re-appearing log entries. Assume that during initial claiming, entries $(42, 1)$ and $(42, 3)$ in generation 42 are observed as the last entries of the replay sequence. Then $(E_{seal}.gen, E_{seal.gsid})$ is $(43, 1)$ and the counter table of E_{seal} only accounts for these two entries, and all previous generation's entries. Now assume that another entry $(42, 2)$ had been written but was temporarily invisible during initial claiming, e.g., due to temporary data corruption. If for some reason $(42, 2)$ becomes visible again, and we start replaying, then $(42, 2)$ will be replayed even though it was not visible during claiming. And if $(42, 2)$ were to replace $(42, 1)$ or $(42, 3)$, we would not even notice the counter mismatch when arriving at E_{seal}. This behavior seems desirable in the sense that replay recovers as much committed data as possible. However, in combination with WR_INDIRECT ZIL log records, it can lead to pool corruption. ZIL-PMEM does not currently support WR_INDIRECT records due to this problem as well as another WR_INDIRECT-related issue that affects ZIL-LWB. Our proposed solution in the OpenZFS issue tracker could be used by ZIL-PMEM to properly detect changes to the last generation after claiming, and to support WR_INDIRECT if desired.

link github issue on leaking allocated blocks

5.8 PRB: DRAM Data Structure

The central requirements for PRB are

- persistence of log entries in PMEM,
- parallel insertion of log entries from multiple threads and HDLs,
- garbage collection of obsolete log entries,
- measures to detect data corruption.

Our design is built around the *chunk* abstraction. A chunk is a contiguous segment of PMEM that acts as an insert-only container for log entries. PRB's role is to mediate access between HDLs and chunks. When a thread writes an entry to the conceptual *set of entries* that is the PRB, it actually inserts the entry into one of the PRB's chunks. Once a chunk is full, PRB lets it sit unmodified until all entries in it are obsolete. Garbage collection then removes all entries in those obsolete-only chunks and makes them available for reuse by writers. The PRB scanning done by claiming and replay is implemented as an iteration over the entries in all chunks of the PRB. If a chunk contains at least one candidate entry for replay, the claming HDL holds the chunk back from the write path and/or

garbage collection. Note that this design fully decouples storage location (chunk) from log structure (encoded in entry metadata, see previous sections).

Each chunk is represented by a DRAM object that holds the following values:

PMEM location The chunk's start and end address in PMEM. These values do not change for the lifetime of the chunk.

Claim refcount The number of HDLs that claimed at least one entry in the chunk for replay. Chunks with non-zero claim refcount are exempt from garbage collection and the write path.

Max txg The maximum transaction group of the entries that were written to this chunk. This value is used by garbage collection to determine when all entries in the chunk are obsolete.

Write position The PMEM address where the next entry will be written. We dicsuss the PMEM layout of the chunk in more detail in Section **??**.

We move chunks between the following data structures to keep track of their current role within PRB.

Commit Slots PRB maintains a roster of *commit slots*, each of which has an associated *open chunk*. When a thread writes an entry through a HDL, the thread first *acquires* a commit slot to gain temporary exclusive access to its *open chunk*. Then, it writes the entry to the open chunk and immediately releases the slot so that other threads can aquire it.

Free List The *free list* holds empty chunks. Their *claim refcount* and *max txg* is zero and their write position is reset to the chunk's start address. A log writer that needs a new chunk for its commit slot puts the commit slot's current open chunk on the *full list* (see below) and gets a new chunk from the *free list*.

Full Lists The *full lists* contains chunks that wait for garbage collection. PRB maintains one full list for each unsynced txg T_i. The *full list* for T_i contains exactly those (full) chunks whose *max txg* is T_i. After T_i has been synced, the chunks on the corresponding *full list* are emptied and moved to the *free list*. Note that it is sufficient to maintain three full lists at any given time — one for each of the three unsynced transaction groups *open*, *quiescing*, and *syncing*. The three lists can be represented as an array of three lists that is indexed by $T_i \bmod 3$. Our implementation follows the common ZFS idiom of using four lists that are indexed by `T_i&3` where `&` is the *bitwise and* operation.

Wait Replay List During the claiming phase, all of the PRB's chunks are on the *wait replay list*. The HDL's scan the chunks on this list during claiming. If a chunk contains at least one entry that needs to be held back for replay, the HDL increments the chunk's *claim refcount*. Once claiming is done

and *txg sync* triggers the first garbage collection cycle, any chunk on this list that has a zero refcount is emptied and moved to the *free list*. The list is also scanned for zero refcounts during all future garbage collection cycles to garbage-collect chunks that are no longer held by HDLs because they finished replay. Note that it is critical for replay crash consistency to defer garbage collection until *after* the last txg of replay of the last HDL that held the chunk has synced. If we garbage-collected entries before the last holding HDL's ZIL header has transitioned to state *nozil* on disk and crashed inbetween, the garbage-collected entries might be missing when resuming replay.

Figure 5.6 visualizes transitions of the chunks over their lifetime. Figure 5.7 provides a corresponding example.

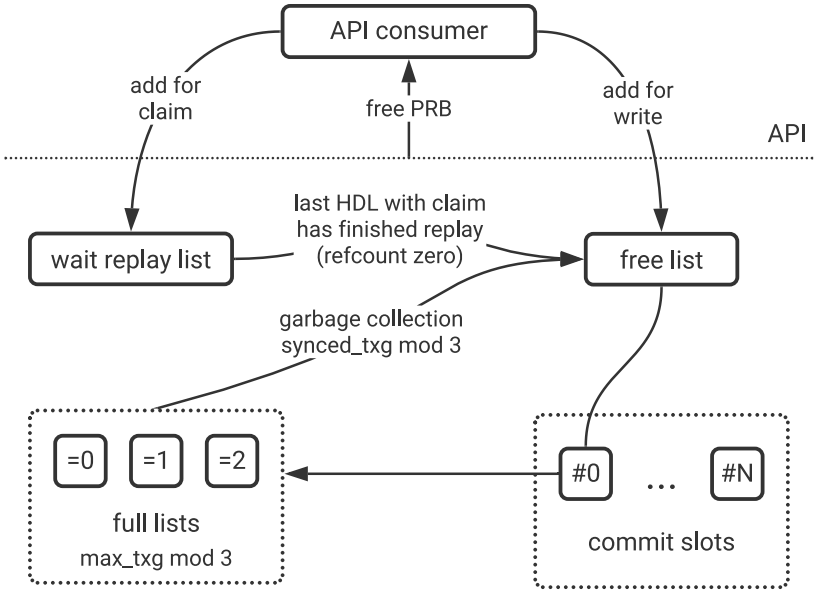


Figure 5.6: The different owners of a chunk and the events that cause ownership transitions.

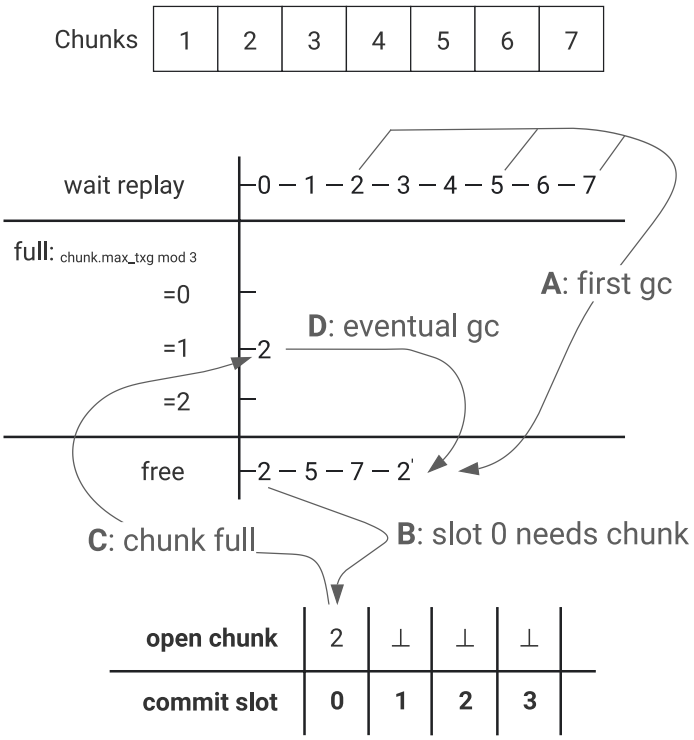


Figure 5.7: Example for chunk ownership transitions. The PRB is constructed with all eight chunks added as `_for_claim`. Claiming (*A*) determines that chunks 0, 1, 3, 4, and 6 need to remain on the *wait replay list* because they contain entries for HDL logs that need replay. Chunks 2, 5, and 7 only contain obsolete entries and move to the *free list*. We do not replay any of the logs. However, a log writer starts writing a new log in *B*. It finds that its commit slot (#0) has no active chunk. Thus, it and moves the first chunk on the *free list* (i.e., chunk 2) to the commit slot. After writing the entry to chunk 2 the log writer releases the commit slot. Another log writer aquires commit slot #0 and writes entries to it. Chunk 2's capacity is insufficient to hold the last entry (*C*). The thread places chunk 2 on the correct *full list* for chunk 2's *max txg* and finds a new chunk on the free list for the commit slot (not shown). Eventually *txg sync* triggers garbage collection for the txg that is chunk 2's *max txg* which resets chunk 2's in-PMEM and in-DRAM sequence and subsequently places it back onto the free list.

5.9 PRB: PMEM Data Structure

As explained in the previous section, PRB is built on top of contiguous slices of PMEM which we call *chunks*. However, PRB only consumes the chunks, it does

not create them. Chunk allocation is the responsibility of the PRB consumer, i.e., ZIL-PMEM. This design improvides modularity and testability because it decouples the following two concerns:

Resource Aquisition The PRB consumer is responsible for integrating PMEM SLOG vdev into the zpool, discovering its memory mapping, and partitioning the PMEM space.

PMEM Data Structure PRB only implements the persistent data structure that stores entries in the chunks (next section).

ref

With regards to persistent data structure, this separation of concerns relieves PRB from the need to define structures to track the partitioning of PMEM space into chunks. It relies on the consumer to provide the PRB with the same set of chunks every time the PRB is constructed.

Entries are variable-length records that are stored within the chunk as a contiguous sequence. Each entry is represented as a fixed-length 256 byte sized header and a variable-length body. The first entry starts at the chunk's start address which must be algigned to 256 bytes. The space after each entry is zero-padded to the next multiple of 256 bytes. The next entry starts after the padding. The sequence is terminated either explicitly by an invalid entry header or implicitly if the last entry has filled the chunk completely. Figure 5.8 provides an example chunk layout. Note that the ordering of entries within the chunk is has no semantic value as the logical view on the chunk is that of a set of entries.

The entry body is a verbatim copy of an opaque byte slice provided by the log writer. (In ZIL-PMEM, this byte slice is the log record structure that is shared among all ZIL kinds.) The entry header's contents are managed by the PRB and HDL. Its contents are as follows:

language

HDL-scoped metadata The metadata required for attribution of a log entry to a HDL and subsequent replay.

- Log GUID
- Generation
- Geneneration-Scoped ID
- Encoded Counters for dependency tracking.

Body Length We store the exact body length in bytes. The zero padding in the chunk sequence is not considered part of the entry itself.

fix that?

Body Checksum Fletcher checksum of the body data.

Header Checksum Fletcher checksum of the header to ensure data integrity of the metdata. If the header checksum is corrupted then none of the other header fields can be trusted.

Zero Padding The unused bytes in the header have the defined value of zero. Their value is part of the header checksum.

Chunks must be sufficiently large to hold at least one entry because there is no mechanism to split an entry across multiple chunks. The smallest chunk's size determines the maximum entry size that can be written to it. However, chunk sizes much larger than a single average entry are advisable for multicore-scalability, as we will elaborate on in Section 5.12.

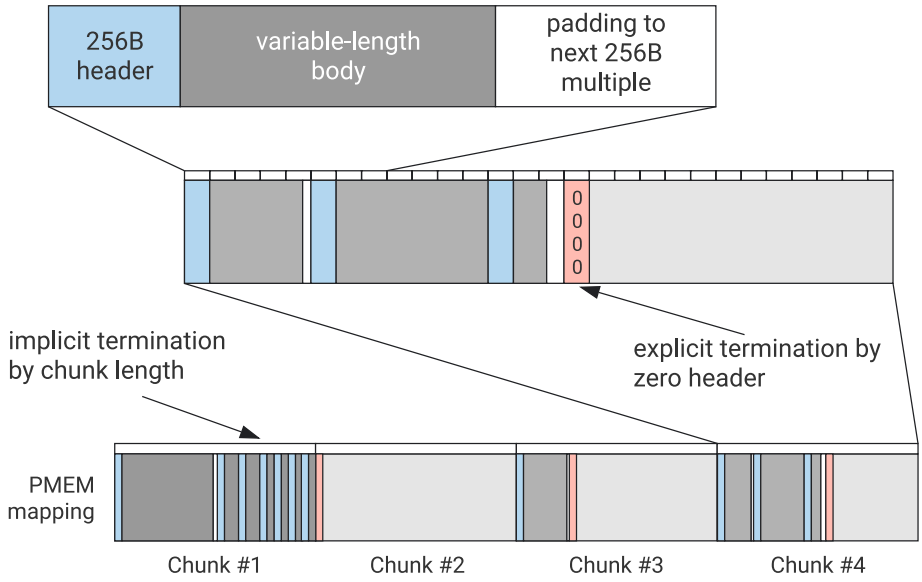


Figure 5.8: Example chunk layout. Note that although we partition the PMEM space very regularly in this example, the PRB consumer is free to use variable-sized chunks.

5.10 PRB: Chunk Traversal

HDLs scan the PRB during claiming to put their holds on chunks that contain replayable log entries. During replay, the HDLs scan their held chunks again to construct the replay sequence. The building block for both procedures is the iteration over the entries of a chunk. We call this iteration process *chunk traversal* and present the algorithm in this section.

As a reminder, the data layout within the chunk is as follows:

- The first entry header starts at chunk offset zero. Its size is fixed (256 bytes).

- The variable-length body starts immediately after the header. Its length is stored in the header.
- The body is followed by zero padding to the next 256 byte multiple.
- The next entry's header starts there.
- An invalid header marks the end of the chain. A header is invalid if its header checksum is invalid, or if the log GUID is invalid. The (128 bit) log GUID is invalid if at least the upper or lower 64 bits are zero.
- Entries are never split between chunks.

In the absence of data corruption in PMEM, chunk traversal visits each valid entry in the sequence and stops at its end. For the case where PMEM is corrupted, we distinguish the following conditions:

word?

Machine check exception (MCE) If the PMEM hardware detects uncorrectable data corruption, it raises an MCE. The Linux kernel provides the an API (`memcpy_mcsafe`) which provides a memcpy-compatible interface but converts MCEs into error return values. We always use this API to buffer PMEM contents in DRAM before accessing them. The error handling for MCE errors is the same as for invalid checksums in entry header and body, which we describe below.

Header: Detected Data Corruption If the header checksum validation fails, the header was either corrupted by bitflitps or similar phenomena or never completely written. The latter case is critical for crash consistency on the write path as we will elaborate on in Section 5.12.3. Regardless of the cause, an invalid header's values cannot be trusted, and thus the traversal stops.

Header: Undetected Data Corruption If the header checksum does not detect data corruption in the header, the behavior is implementation defined. However, it is guaranteed that memory accesses are constrained to the entry's chunk's bounds.

Data corruption in the body The traversal algorithm does not read the body but instead returns a closure that can be invoked for this purpose. The closure reads the body into DRAM using `memcpy_mcsafe`, then validates the body checksum stored in the header. Validation failures are returned as an error. The caller can decide whether they want to iterate further or propagate the error up the call stack. Note that in our C implementation, the closure is replaced by the opaque struct `zilpmem_replay_node_t` and function `prb_replay_read_replay_node`.

Data corruption in the padding We require that the padding in the space that follows the body to the next 256 byte multiple consists only of zero. We validate this property in the closure that reads the body. Validation failure

results in a distinguished error being returned from the closure. Such an error is indicative of data corruption or an incorrect implementation on the write path. The caller of the traversal algorithm should surface it to the administrator, but may choose to proceed.

The following listing provides the pseudo code for chunk traversal.

```
Algorithm: iter_chunk() - Iterate over the entries in a chunk.

Inputs:

    ch_base    chunk's base address
    ch_len     chunk's length

Procedure:
    assert ch_base % 256 == 0
    assert sizeof(entry_header_t) == 256
    assert ch_len >= 256
    assert ch_len % 256 == 0

    e := ch_base
    while (e < ch_base + ch_len) {
        entry_header_t eh;

        // read header with a function that handles MCEs
        memcpy_mcsafe(&eh, e, sizeof(eh))?;

        // invalid hdr checksum or id terminate the sequence
        validate_header_checksum(eh)?;
        if eh.log_guid's upper or lower 64 bits are zero {
            return None;
        }

        body_ptr := e + sizeof(eh);

        if body_ptr + eh.body_len > ch_base + ch_len {
            return Err("""
              entry out of chunk boundary:
                a) writer implementation error
                b) undetected header corruption
            """);
        }
        e += sizeof(eh) + body_len;
        e = roundup_to_next_256byte_multiple(e);

        padding_ptr := body_ptr + eh.body_len
        padding_len := e - padding_ptr

        read_body := |buffer: *u8| {
            memcpy_mcsafe(buffer, body_ptr, eh.body_len)?;
            validate_body_checksum(eh, buffer, eh.body_len)?;
            pmem_is_zero(padding_ptr, padding_len)?;
            return Ok(());
        };
        yield Some((eh, read_body))
    }

Example:
```

```
    for (entry_header, read_body) in iter_chunk(...) {
        if entry_header.txg <= precrash_txg {
            continue;
        }
        buf := buffer of size entry_header.body_len
        read_body(buf)?;
        ...
    }
```

5.11 PRB: Garbage Collection

We already covered the essence of garbage collection in Section 5.8: if the capacity of an acquired commit slot is insufficient, the log writer puts it on the *full list* for the chunks' *max txg* and gets a new chunk from the *free list*. After *txg sync* has written out that *max txg*, it triggers garbage collection, which removes all entries in the chunks on the *full list* for *max txg*.

We remove all entries in a chunk in $O(1)$ time by invalidating the log GUID in the first entry header in the chunk (offset zero). As indicated by the This is sufficient to prevent chunk traversal in the future. The pseudo-code to invalidate a chunk is as follows:

```
Input:
    ch_base     chunk base address
    ch_len      chunk length
Steps:
    assert ch_len >= 256

    assert type of entry_header_t::log_guid is uint64_t[2]

    log_guid_addr := ch_base + offsetof(entry_header_t, log_guid)

    assert log_guid_addr is 64-bit aligned

    atomic 64-bit store of 0 to log_guid_addr
    atomic 64-bit store of 0 to log_guid_addr + 8

    flush modified cache lines
    sfence
```

5.12 The Write Path

A thread that writes an entry to a HDL needs to perform the following tasks:

- Find a target chunk using the *commit slot* mechanism. (Section 5.12.1)
- Determine the HDL-scoped metadata, i.e., log GUID, txg, gen, gsid, and counters.
- Compute header and body checksums.
- Insert the entry into the target chunk in a crash-consistent manner.

Our goal is a design with low write latency, good multicore-scalability, and CPU efficiency. We identify the following factors as particularly relevant:

Checksumming Checksumming adds latency but does not concern multicore scalability since no coordination is required between writers. The implementation should use ZFS's optimized implementations of the Fletcher checksum.

Dependency Tracking Counters We must update the dependency-tracking counters on every entry write operation. Since the counters are HDL-scoped, this only presents a scalability concern if a single HDL is written from multiple threads. Parallel writes to the same HDL do not happen in ZIL-PMEM proper but are relevant for our ZVOL-specific ITXG bypass (Section 6.3.5).

Commit Slot Aquisition & Chunk Replacement The PRB's commit slots and chunk lists are shared among all HDLs. All threads that write to any HDL of the PRB compete for these resources, making it a multi-core scalability challenge.

Optane Characteristics We develop and evaluate ZIL-PMEM for/on Intel Optane DC Persistent Memory. The performance characteristics of Optane DIMMs are significantly different from regular DRAM. Yang et al. have established that the Optane PMEM hardware is organized in units of 256 bytes. For example, the access granularity and kind of store and cache flush instruction have significant impact on the achievable write bandwidth. For size of log entries written by ZIL-PMEM, the use of AVX-512 non-temporal store instructions is recommend for highest possible performance. [Yan+20].

PMEM Bandwidth Limits & Multicore Scalability It is inherent to the programming model for persistent memory that wait time for PMEM I/O is spent on-CPU. For example, instructions that architecturally depend on a preceding store+cacheflush+sfence to PMEM will stall until the flushed cacheline reaches the power-fail protected domain of the CPU [Sca20]. This is problematic from a CPU utilization perspective: if multiple threads attempt to write at higher bandwidth than PMEM can sustain, they still appear busy towards the OS thread scheduler and waste CPU time that could be used more productively by other threads in the system. Yang et al. have shown that a single Optane DIMM's write bandwidth can be exhausted by one CPU core at 2 GB/s. Write bandwidth decreases to 1 GB/s at ten or more concurrently writing CPU cores [Yan+20]. Whereas excessive on-CPU waiting might be the right trade-off in certain userspace applications of PMEM, a kernel file system such as ZFS cannot make assumptions about the system's overall CPU priorities. We expect that PMEM write

bandwidth can be exhausted in real-world use cases for ZIL-PMEM. Our design must therefore find a way to limit concurrent access to PMEM and shift PMEM wait time off the CPU.

5.12.1 Commit Slots

Commit slots are our abstraction to enable multiple threads to write entries concurrently. The goal is to grant up to `ncommitters` parallel writers temporary exclusive access to a chunk into which they can write their log entry. For this purpose, a thread that wants to write an entry *aquires a commit slot* $S \in 0, 1, \ldots, ncommitters - 1$. Each thread that is simultaneously committing gets a different commit slot. If no commit slot is available, the function to aquire the commit slot blocks. Associated with each commit slot is a PMEM chunk which we refer to as *open chunk*. A thread that aquires a commit slot is allows to write its entry to the slot's *open chunk*.

The limitation to *ncommitter* parallel writers is desirable to avoid excessive on-CPU waiting on PMEM. Let us make the simplifying assumption of a fixed maximum write bandwidth of $B_max[byte/s]$ to PMEM before latency increases dramatically due to queuing. Then an equal distribution of that bandwidth yields $\frac{B_max}{ncommitters}[byte/s]$ of write bandwidth per writer. Assuming that writers do not exceed this limit, we can derive latency guarantees for writing entries, dependent on entry size.

We implement commit slots using a semaphore initialized to *ncommitters* and a bitmask with `ncommitters` bits. The thread that aquires a commit slot first enters the semaphore and then finds and flips a zero bit in the bitmask. The zero bit's index is the commit slot number. We use opportunistic spinning to find and flip the bit. The following pseudo-code explains the aquisition and release procedures.

```
Procedure For Commit Slot Aquisition:
  Input:
      sem     semaphore
      bm      pointer to PRB-wide bitmask with ncommitters bits
  Output:
      The commit slot number.
  Steps
    Enter semaphore.

    my_bm   <-  atomic_load(bm, SeqCst)
  'retry:
    idx     <- find first set bit index in (~my_bm)
    if idx == 0:
        panic: idx=0 indicates there is no free bit,
               but the semaphore guarantees that
    idx -= 1
    if idx >= ncommitters:
        panic: semaphore guarantees that there are
               free commit slots
```

```
    my_bm = my_bm | (1<<idx)
    if compare_and_swap(bm, &my_bm, SeqCst):
        // we won the race => return the commit slots
        return idx
    else:
        // we lost the race with another committer
        // => retry
        // (my_bm contains the actual value of bm)
        goto retry

---

Procedure For Commit Slot Release:

  Input:
     cslot     The commit slot number returned on aquisition.
  Steps:
    Atomic bitwise and of bm with ~(1<<idx)
    Exit Semaphore
```

The procedure above is all the PRB-wide coordination that is required for writing a log entry, iff the aquired slot's *open chunk*'s capacity is sufficient. If capacity is insufficient, the writing thread replaces the full *open chunk* with a new one. To accomplish that, it puts the current *open chunk* on the correct *full list* and gets a new *open chunk* from the *free list*. Access to the PRB's lists is protected by a PRB-wide mutex. The potential contenders for this mutex are up to *ncommitters* writer threads and the txg sync thread that performs garbage collection.

Getting a new chunk from the *free list* fails if the free list is empty. The log writer can choose whether to block and wait or fail the log entry write operation with an error. Block-and-wait is implemented through a condition variable that is signalled by garbage collection for every chunk that it puts back on the *free list*. If the log writer chooses to block and wait during chunk replacement, it must guarantee that it does not prevent *txg sync* from making progress in order to avoid a pool-wide deadlock. In practice, this means that the log writer must not hold a DMU transaction open when writing an entry. Note that this is the case for ZIL-PMEM proper because `zil_commit` is called after the DMU transactions have finished or failed. However, for the ITXG bypass for ZVOLs (Section 6.3.5), we write log entries from within the DMU transaction and thus need to use the non-blocking mode.

chapter?

Sharing the slots (and thus chunks) among all threads and HDLs in the PRB is of ambiguous value from perspectives other than bandwidth limitation:

Space Efficiency Sharing chunks causes entries to be packed into a small number of chunks, even more so if we implemented some form of *best fit* selection scheme for commit slots. Packing is beneficial for space efficiency

because the *spread* between mininum and maximum txg of the entries in a chunk is small, enabling timely garbage collection.

Blast Radius However, the concentration of entries in a chunk also increases the blast radius of data corruption due to the chunk's physical data structure which we discuss in Section 5.9. In particular, data corruption within the entry metatadata of one HDL's entry can render another HDL's entry unreachable during PRB scan if they are stored in the same chunk.

Cache Efficiency Our aquisition procedure deterministcally picks the lowest available commit slot. It is thus very likely that chunks bounce around CPU cores, without taking temporal locality into account. Note that, due to the use of non-termporal store instructions when writing log entries to PMEM, this only impacts the chunk DRAM object. We briefly experimented with per-core commit slots during development and found no noticable performance difference to the approach presented above.

one sentence on limited slog size, that it's not as relevant for PMEM, but maybe for NVDIMM-N?

this last paragraph doesn't feel right here...

5.12.2 HDL-Scoped Metadata

We have already-addressed the algorithm to compute the HDL-scoped entry metadata (Log GUID, txg, gen, gsid, Counters Table) in Section **??**. This subsection only addresses multithreading and scalability concerns.

check matches

First of all, HDL-scoped metadata does not pose a scalability concerns if entries are written sequentially. This is the case for ZIL-PMEM proper which uses a mutex to serialize `zil_commit` calls. (Remember from Section **??** that the ZIL's shared itxg structure defines the sequential *commit list* model.) However, for the ITXG bypass for ZVOLs (Section 6.3.5), the ZVOL dataset's HDL can be written in parallel.

Multiple threads are allowed to write to a single HDL simultaneously iff they do not start a new generation. Threads that start a new generation must wait for all threads that wrote entries to the previous generation to finish writing. The reason is that the new generation's entry logically depends on the previous generation's entry. It would be sufficient to prevent the *function calls* from returning out of dependency order while allowing the new generation's entry to be written in parallel with the old entries. Such a system is used by ZIL-LWB's *commit ITXs* which uses condition variables to wake `zil_commit`ting threads up when the last LWB that contains one of their commit list's records has been written. However, the *commit ITX* model was not directly applicable to ZIL-PMEM proper. For the ITXG bypass, we use a simple read-write-lock which we elaborate on in Section 6.3.5.

check content

Within the HDL, we use a spinlock to serialize access to the state used for dependency tracking (*live table*, *last table*). We only compute the encoded representation of the *last table* when the first entry is counted for a new generation. Subsequent writes for the same generation only need to `memcpy` the pre-computed encoded representation while holding the spinlock.

5.12.3 Crash-Consistent Insert

After the commit slot has been aquired, a target chunk been selected, and the HDL-scoped metadata determined, we are ready to insert the entry into the chunk. Remember from Section 5.9 that the chunk is a sequence of entries that is terminated by an invalid entry header. To insert the entry into the chunk, we must append the entry to the sequence in a way that is crash-consistent with regards to the traversal algorithm (Section 5.10):

> Appending an entry E_{n+1} to a chunk C that contains a sequence of entries $E_1, \ldots, E_n$ must be atomic from the perspective of traversal. After a system crash or power failure during the append operation, traversal of C must either find $E_1, \ldots, E_n$ or $E_1, \ldots, E_n, E_{n+1}$.

The following algorithm describes the procedure and invariants during the append operation. Figure 5.9 contains a step-by-step visualization.

```
Inputs:
    ch  The chunk into which we append
    e   The entry that we append to ch's sequence.
Procedure:
  Invariant 1: Traversal will stop at ch.pos because
               either: ch.pos points to PMEM space that
                       is an invalid header
               or: there is no space left in the chunk

  Phase 1:
    Write e.body to address ch.pos + 256
    Write trailing zero padding
    If there is still space in the chunk at
      address ch.pos + 256 + e.body_len + padding_len:
        Assert that the space is at least 256 bytes.
        Write an invalid follow header (256 zero bytes)
          to that address.
    Cacheflush + Sfence

  Corrolary 1: Neither body nor follow header
               will be visited by traversal
               because traversal still stops
               at address ch.pos (Invariant 1).

  Phase 2:
    Write e.header (256 bytes) to address ch.pos
    Cacheflush + Sfence

  Corrolary 2: Traversal visits the entry written
```

```
to ch.pos and stops at the follow
header.
```

Invariant 1 can be proven by induction: if the entry being written is the first entry in the sequence (base case) the chunk was fetched from the *free list* which is defined to only contain chunks that are *empty*. (*Empty* means that the write position (`ch.pos`) is at the chunk's base address and that the PMEM at the write position is an invalid header.) The induction step is that if an entry E_{n+1} is appended to an existing sequence E_n, invariant 1 holds as well. This is true because the space occupied by E_{n+1} contains an invalid follow header written by phase 1 when E_n was appended to the sequence. Stores made in Phase 1 for E_n are guaranteed to be persistent before any store for E_{n+1} happens due to the `cacheflush+sfence` at the end of Phase 1. Note that we do not need to address the case where the chunk is full because we cannot append E_{n+1} to a full chunk.

The first `sfence` in phase 1 is required for correctness iff we do not trust the body checksum to detect partial writes. If we omitted the `sfence` in phase 1, it would be possible that the entry header written in phase 2 reaches PMEM completely but the body written in phase 1 does not. In that case, after a crash, the traversal algorithm would observe a valid header but a body space with undefined content. We would rely fully on the body checksum to detect the partially written body. If the checksum is too weak, the replayer's interpretation of the undefined body contents determines subsequent behavior. Corruption of the dataset is a likely consequence.

The `sfence` (phase 2) is required for correctness because we must conservatively assume that the log writer's subsequent store instructions (in program order) depend on the entry having reached stable storage.

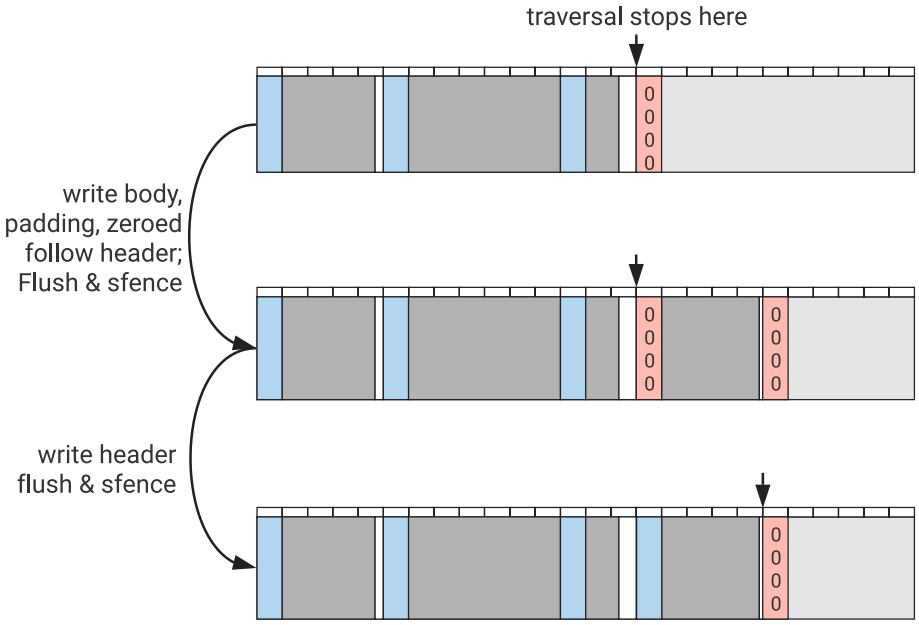


Figure 5.9: The crash-consistent append operation to a PMEM chunk. The traversal algorithm reaches the existing entries at all times and reaches the new entry only after its body and header have been written completely.

The following implementation details of the append operation are relevant for latency and efficient use of CPU time:

- For ZIL-PMEM, the use of `sfence` in phase 1 came with negligible impact on overall latency during development. We assume that this is because the wait time added by the `sfence` is neglible compared to the write time for the entry body. For example, the entry body and zero padding for a 4k sync write is $sizeof(lr_write_t) + data + padding = 4096 + 192 + 64 = 4352$ bytes large. Assuming 2 GiB/s write bandwidth for a single Optane DIMM [Yan+20], the write time for the entry body is $\sim$ 2 us. In contrast, the derived latency for a 256 byte write at that rate is $\sim$ 0.12 us. If we use this value as an approximation for the cost of the `sfence`, its latency contribution is only $\frac{0.12}{0.12+2} = 5.6\%$.
- All writes to PMEM happen in multiples of 256 bytes because 256 bytes is Optane's internal write unit size. Writes below this size cause read-modify-write cycles in the hardware and thus cost performance [Yan+20; Zha+21].

- We use AVX-512 non-temporal store instructions (`movnt`) instead of regular stores and cache flushes. Again, this addresses established performance properties of Intel Optane DC Persistent Memory [Yan+20].
- We also use ZFS's optimized implementation of the Fletcher checksum to compute body and header checksums. Whereas the best implementation is chosen by benchmark dynamically at runtime, it is safe to assume that some SIMD ISA extension such as AVX-512 will be used if available.
- OpenZFS cannot rely on the Linux kernel's interface for saving FPU state due to licensing issues []. Unless the FPU is used by a dedicated kernel thread, ZFS must temporarily disable preemption, mask local interrupts, and manually save FPU state. PRB is written directly from the task that calls `zil_commit` and thus incurs this overhead. We refactor ZFS's FPU state management abstraction so that FPU context is only saved once for both checksum computation and writing to PMEM. The time that is spent in this suboptimal state is bounded by the maximum log entry size.

we have no evaluation on impact of interrupt masking, future work

5.13 API Overview

We briefly discuss the PRB/HDL API that is used by the `zil_pmem.c` module. In the implementation, most types and functions are prefixed with `zilpmem_` or `zilpmem_prb_t` which we omit for brevity.

5.13.1 PRB Setup

```
prb_t* prb_alloc(size_t ncommitters);
void prb_free(prb_t *b, bool free_chunks);

chunk_t* chunk_alloc(uint8_t *pmem_base, size_t len);
void chunk_free(chunk_t *c);

void prb_chunk_initialize_pmem(chunk_t *c);
void prb_add_chunk(prb_t *prb, chunk_t *chunk);
```

The zpool import procedure allocates the PRB using the `prb_alloc` function. The returned `prb_t` is owned by the caller which is responsible for freeeing it using `prb_free` during pool export.

The PRB consumer allocates the chunk objects using `chunk_alloc`. It adds the chunk to the PRB using `prb_add_chunk`. The PRB assumes ownership of the chunks that are added to it. When the PRB is constructed for the first time, the PRB consumer must call `prb_chunk_initialize_pmem` to reset the PMEM sequence to an empty state.

The `free_chunks` argument to `prb_free` determines whether the PRB should free the chunks objects that were added to it, or whether the ownership moves back to the caller. Note that freeing the chunk object (`chunk_free`) does not alter the chunk's PMEM state.

HDL Setup

```
void zil_header_pmem_init(zil_header_pmem_t *zh);
hdl_t* prb_setup_hdl(prb_t *prb, const zil_header_pmem_t *hdr);
void prb_teardown_hdl(hdl_t *hdl,
    bool abandon_claim, zil_header_pmem_t *upd);
```

HDL recover their DRAM state from the ZIL header (`zil_header_pmem_t`). The setup `prb_setup_hdl` function takes a constant pointer to the last-synced header. The pointer is not internalized in HDL — the pointee's lifetime may be as short as the function call. The PRB consumer instantiates all dataset's HDLs during pool import or whenever a new dataset is created. For new datasets, the initial value for the ZIL header (state *nozil*) is set by `zil_header_pmem_init`.

When the head dataset is destroyed or the pool is exported, the PRB consumer calls `prb_teardown_hdl` to destroy the HDL. The PRB must be freed before all of its HDL's have been torn down.

Claiming

```
check_replayable_result_t prb_claim(
    hdl_t *hdl, uint64_t pool_first_txg,
    zil_header_pmem_t *upd);
```

After the PRB is constructed and HDLs are set up, we must *claim* log entries of all dataset's HDLs. The corresponding function `prb_claim` is invoked by the pool import procedure for each HDL. The `pool_first_txg` is the pool's first new transaction group. For HDLs in state *logging*, `pool_first_txg - 1` becomes the $precrash\_txg$.

`prb_claim` returns an update to the ZIL header through the `upd` out-parameter. We employ this pattern throughout the entire PRB API. It is always the API consumer's responsibility to ensure that the update is correctly persisted in the correct transaction group.

Claiming can fail, e.g., if the procedure detects a missing entry for the HDL (claiming performs a dry-run of replay internally). The API consumer defines the error handling policy. It can either abort pool import or choose to abandon the log. To abaondon the log, the API consumer must first tear down the HDL

using `prb_teardown_hdl, abandon_claim=true, ...)` to release claims made during `prb_claim`. Then, the API consumer resets the header using `zil_header_pmem_init` and re-instantiates the HDL using `prb_setup_hdl`.

need third option to retry with an override flag that discards the unreplayable part of the log

After all HDLs have been claimed the pool import procedure starts *txg sync* which writes out the header updates made by claiming in the `pool_first_txg`. From that point on there is no need to coordinate HDL operations between different datasets.

5.13.2 Replay

```
typedef struct { ... } replay_result_t;

replay_result_t
prb_replay(hdl_t *hdl, replay_cb_t cb, void *cb_arg);

typedef struct replay_node replay_node_t;
typedef int (*replay_cb_t)(void *rarg,
    const replay_node_t *rn,
    const zil_header_pmem_t *upd);

typedef enum {
    READ_REPLAY_NODE_OK,
    READ_REPLAY_NODE_MCE,
    READ_REPLAY_NODE_ERR_CHECKSUM,
    READ_REPLAY_NODE_ERR_BODY_SIZE_TOO_SMALL,
} read_replay_node_result_t;

read_replay_node_result_t
prb_replay_read_replay_node(
    const replay_node_t *rn,
    uint8_t *body_out, size_t body_out_size,
    size_t *body_required_size);

void prb_replay_done(
    hdl_t *hdl, zil_header_pmem_t *upd);
```

When a dataset is mounted the mounting procedure must always call `prb_replay`. If the HDL is in state *nozil*, the call is a no-op. If the HDL is in state *replaying*, the function invokes the provided replay callback for each log entry that needs to be replayed in replay order. The callback must perform the following steps for crash-consistent replay for each replayed entry E.

1. Start a DMU transaction tx.
2. Load the entry E into a DRAM buffer using `prb_replay_read_replay_node`.
3. Apply the change encoded in E to the dataset.

4. Update the ZIL header to the value of `*upd`.
5. `dmu_tx_commit(tx)` the DMU transaction.

Note that the callback must use `prb_replay_read_replay_node` function because `replay_node_t` is an opaque type in the PRB API. The function forces the API consumer to do DRAM buffering and protects against online and offline data corruption internally, as dicussed in Section 5.7.

`prb_replay` can fail either due to an error returned by the callback, due to an error in the scanning phase, or due to log corruption. If the error is due to missing log entries, the struct returned by the API contains the *witness* log entry (see Section 5.5). The caller decides whether to retry replay or abandon the remaining unreplayable part of the log. End of replay must be acknowledged explicitly by calling `prb_replay_done`. Abandonding the log is done using `prb_destroy_log` (next section).

retry with an override flag that discards the unreplayable part of the log

5.13.3 Writing Entries

```
bool prb_create_log(hdl_t *hdl, zil_header_pmem_t *upd);
void prb_destroy_log(hdl_t *hdl, zil_header_pmem_t *upd);

int prb_write_entry(hdl_t *hdl,
    uint64_t txg, bool needs_new_gen,
    size_t body_len, const void *body_dram);
```

After successful replay, the HDL is always in the unwriteable state *nozil*. The log writer must use `prb_create_log` to create a new log idempotently. If the HDL is in state *nozil*, the function alloctes a log GUID and transitions the HDL to state *logging*. Otherwise, the HDL must already be in state *logging* and the call is a no-op. If a new log was created, the caller must ensure that the transaction group that persists the ZIL header update has synced to disk before starting to write log entries. The caller distinguishes the caes based on the function's return value.

Log writers use the `prb_write_entry` function to write log records to the HDL. The log record is treated as an opaque blob. PRB does not provide facilities to version different version of the encoding. In addition to the body (`body_dram`, `body_len`), the log writer must provide two pieces of metadata:

fix in upstream release?

txg The transaction group $T_{i_{txg}}$ of the DMU transaction T_i whose changes C_i are encoded in the log entry (Section **??**). Note: for ZIL-PMEM, this value is always the same as the log record's `lrc_txg` field.

needs_new_gen Indicates whether a new generation should be started for this log entry. It is the responsiblity of the caller to serialize the start of a new generation as described in Section 5.12.2. if a thread writes

an entry with `needs_new_gen` is `true`, that thread must be the only thread executing `zilpmem_prb_write_entry` for the given HDL. The inverse is not true: if `needs_new_gen` is `false`, multiple threads may write entries in parallel to the same HDL. Note that writers for different HDLs need not coordinate at all.

contrary?

language?

Garabge Collection

```
void prb_gc(prb_t *prb, uint64_t synced_txg);
```

Whenever *txg sync* has finished syncing a txg T to the main pool it must call `prb_gc` with `synced_txg = T`. Note that unlike writing or recovering an individual log, garbage collection is a PRB-level operation. Synchronization is handled internally.

Chapter 6

Integration into ZFS

In this chapter we describe how we integrate PRB/HDL into ZFS in three steps. First, in Section 6.1, we describe how we re-architect ZFS to support different ZIL implementations (*ZIL Kinds)* at runtime (Section 6.1). Then, in Section **??**, we present our approach to make ZFS's device management layer (VDEV) aware of persistent memory devices. Finally, in Section **??**, we describe how we combine PRB/HDL and PMEM SLOG VDEVs into the new *ZIL-PMEM* ZIL kind.

6.1 ZIL Kinds

Coexistence with the existing ZIL and preservation of ZFS's crash consistency guarantees are two requirements for ZIL-PMEM (see Section 4.1.1). Our solution to both of these problems is to re-architect ZFS to support different persistence strategies for the ZIL while sharing all code and data structures that ultimately define crash consistency semantics. In order to make the integration of ZIL-PMEM seamless to the end user (goal: simple administration), the persistence strategy is the same for all datasets in a pool. The variable that determines the pool's persistence strategy is its *ZIL kind*. The following sub-sections present how we refactor ZFS to support ZIL kinds. The existing ZIL, which uses LWBs for persistence (cf. Section **??**) becomes the first ZIL kind called *ZIL-LWB*. Note that some listings and figures in this section already mention ZIL-PMEM. However, in our implementation, all refactoring steps presented in this section are separate commits that precede the introduction of the ZIL-PMEM ZIL kind.

6.1.1 On-Disk State

ZIL-LWB keeps its persistent state in the ZIL header that is stored in the `objset_phys_t`. For ZIL kinds, we change the ZIL header to be a tagged union that uses the new `zh_kind_t` enum as a discriminant. The existing ZIL-LWB header fields are moved into the `zil_header_lwb_t` type. ZIL-PMEM's ZIL header, which we describe in Section 6, is the second member of that union. Figure 6.1 shows the relevant C structures before and after the changes described in this paragraph.

```
typedef struct zil_header {
  uint64_t zh_claim_txg;
  uint64_t zh_replay_seq;
  blkptr_t zh_log;
  uint64_t zh_claim_blk_seq;
  uint64_t zh_flags;
  uint64_t zh_claim_lr_seq;
  uint64_t zh_pad[3];
} zil_header_t;
```

```
typedef enum {
  ZIL_KIND_UNINIT,
  ZIL_KIND_LWB,
  ZIL_KIND_PMEM,
  ZIL_KIND_COUNT
} zh_kind_t;

typedef struct zil_header_lwb {
  /* fields of zil_header_t,
   * without zh_pad */
} zil_header_pmem_t;

typedef struct zil_header_pmem {
  /* introduced later */
} zil_header_pmem_t;

typedef struct zil_header {
  union {
    zil_header_lwb_t zh_lwb;
    zil_header_pmem_t zh_pmem;
  };
  uint64_t zh_kind;
  uint64_t zh_pad[2];
} zil_header_t;
```

Figure 6.1: The ZIL header structs (in DRAM and on disk) before and after the introduction of ZIL kinds.

Compatiblity

We register ZIL kinds as a *zpool feature* flag. Feature flags are OpenZFS's mechanism for expressing variants of the on-disk format.

Whenever we access the ZIL header, we first check the activation status of the feature flag. If the feature is not active, we implicitly know that the ZIL kind is ZIL-LWB and always access `zh_lwb`. If the feature is active, we use the `zh_kind` discriminant field to determine the ZIL kind.

ZIL kind implementations only access their sub-structure within the ZIL header. For example, ZIL-LWB only operates on the `zil_header_lwb_t` structure, not the

full `zil_header_t`. Hence, there is no difference to the ZIL-LWB implementation whether ZIL kinds are enabled or not.

The migration path for activating ZIL kinds is simple since all ZIL headers in the pool are guaranteed to be ZIL-LWB before the migration. The only change is to set `zh_kind = ZIL_KIND_LWB` for all ZIL headers. (We found the `ZIL_KIND_UNINIT` variant, which is the zero value, to be very helpful in catching initialization bugs and would not want to miss it.)

New ZIL kinds, such as ZIL-PMEM, will require their own *zpool feature*s that are marked as dependent on the ZIL kinds feature. However, this is not a solution for decentralizing the assignment of new `zh_kind_t` enum variants to identify ZIL kinds in `zh_kind`. We have not yet come to a satisfying solution for this problem.

6.1.2 Runtime State

In upstream ZFS, the ZIL runtime state is kept in the per-dataset object `zilog_t`. `zilog_t` holds the *itxg* structure that is used to track uncommitted ITXs. `zil_commit` drains the ITXs into the *commit list* and proceeds by packing their encoded representation (log records) LWBs. (See Section 2.3.3 and 2.3.4 for details.)

We observe the following properties of the upstream ZIL code:

- The *itxg* data structure defines the framework for ZFS's crash consistency semantics. Whereas ZPL and ZVOL code create the ITXs, the organization by `itxgs` and the code that assembles the commit list constraints what can be expressed in them.
- ITXs and even the commit list are independent of the LWB chain that is ultimately written to disk. The commit list merely defines the set and order of ZIL records that need to be persisted to some form of sequential log.
- The interface between the ITX- and LWB-related code is limited to the *commit list* and its contents. The responsibilities are thus already (conceptually) separated.

Given these insights we refactor the ZIL implementation (*zil.c*) as follows:

1. Move all non-ITX functions into a separate module *zil_lwb.c* and prefix them with `zillwb_`. If the function was part of the public ZIL API, add a wrapper function with the original name to *zil.c* that forwards the call to the `zillwb_` function in *zil_lwb.c*.
2. Virtualize calls to `zillwb_` functions in *zil.c*:
    - Define a struct `zil_vtable_t` that contains function pointers with the type signature of each of the `zillwb_` functions called from *zil.c*.

- Define `zillwb_vtable` as an instance of `zil_vtable_t` that uses `zillwb_` functions as values for the respective function pointer members.
- Add a member `zl_vtable` to `zilog_t` that is pointer to a `zil_vtable_t`.
- Replace all calls to `zillwb_FN()` in *zil.c* with indirect calls through the vtable, i.e., `zilog->zl_vtable.FN()`.

3. Make non-ITX state private to *zil_lwb.c* by turning it into a subobject.
   - Move the `zilog_t` members that are only used by the functions in *zil_lwb.c* into a separate structure called `zilog_lwb_t` that is private to *zil_lwb.c*.
   - Embed `zilog_t` as the first member in `zilog_lwb_t`.
   - Add member `zlvt_alloc_size` to `zl_vtable_t` that indicates the amount of memory to be allocated when allocating a `zilog_t`.
   - Add a *downcast* step to the start of each `zillwb_` function that casts the `zilog_t` pointer into a `zilog_lwb_t` pointer. The cast is safe because `zilog_t` is embedded as the first member of `zilog_lwb_t`. The majority of `zillwb_` functions operate only on the `zilog_lwb_t`-private state without accessing the embedded `zilog_t`.
   - Add constructor and destructor methods to the vtable that are called after allocating the `zlvt_alloc_sized` `zilog_t`. The `zillwb_` constructor and destructors initialize and deinitialize the private members of `zilog_lwb_t`.

The end result ist best described in the terminology of object-oriented programming: **`zilog_t` is an abstract baseclass** that implements the public ZIL interface as well as ITX-related functionality and defines abstract methods for persisting log records. These abstract methods must be implemented by concrete subclasses. `zilog_lwb_t` is such a subclass that implements the LWB-based persistence strategy. For ZIL-PMEM, the `zilog_pmem_t` struct which we introduce in Section 6 implements persistence directly to PMEM. Which subclass is instantiated at runtime is determined by the `zh_kind` field in the ZIL header. Figure 6.2 illustrates the changes described in this section.

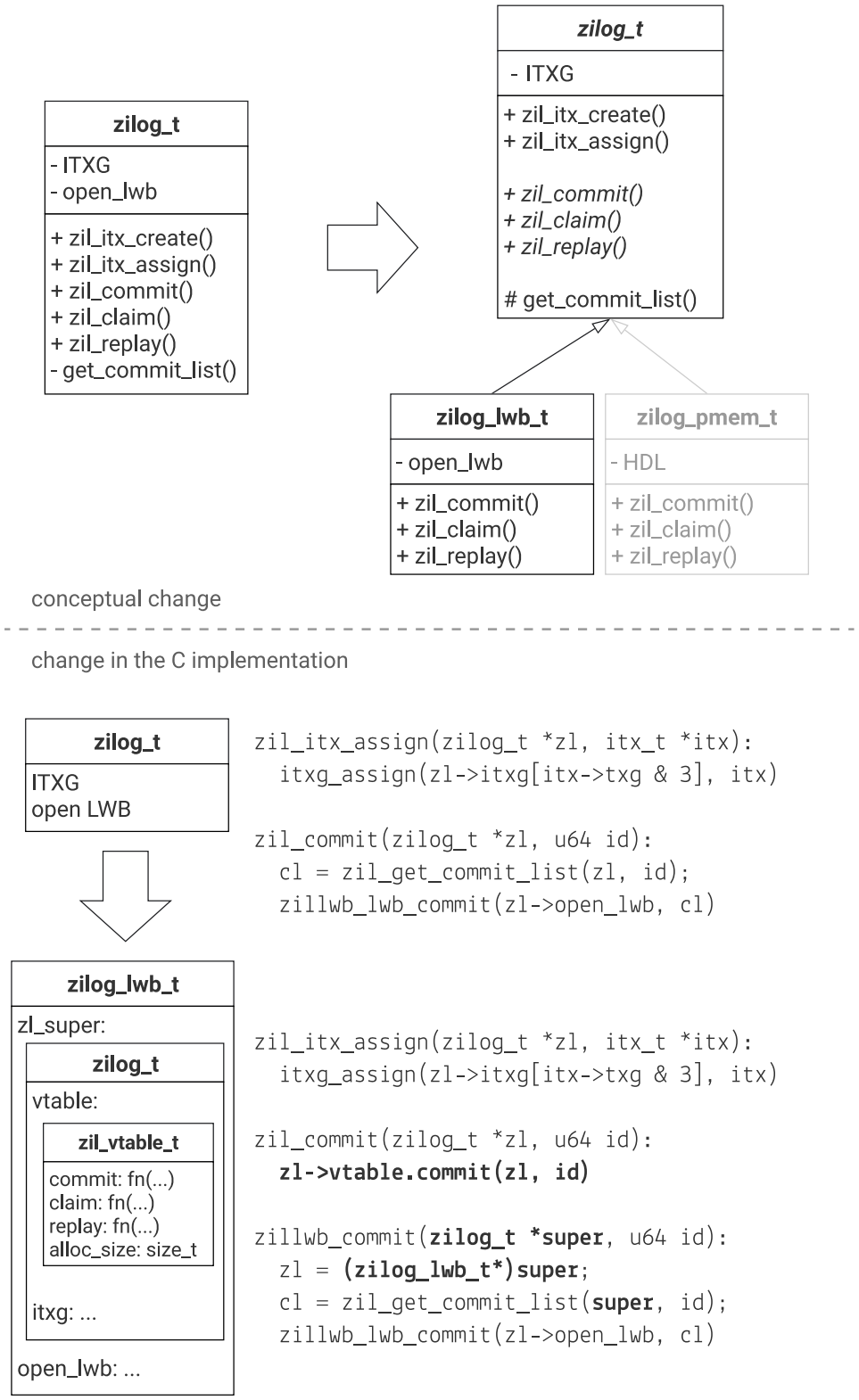


Figure 6.2: `zilog_t` before and after the introduction of ZIL kinds by example of the `zil_itx_assign()` and `zil_commit()` APIs.

6.1.3 Changing ZIL Kinds

A zpool's ZIL kind can be changed by switching over the `zh_kind` of every ZIL header in the pool. The following procedure enables online switching of the ZIL kind:

1. Ensure that all dataset's ZILs have been replayed. If not, cancel the procedure unless the caller specified to drop unreplayed logs.
2. Stop use of the ZIL API and wait until all active API calls to it have finished.
3. Wait for all ZIL entries to become obsolete by waiting for the current open txg to be synced.
4. Free all dataset's `zilog_t` instances.
5. Set all datasets' ZIL headers to the new ZIL kind's default value. The default value is the ZIL state that encodes the absence of log entries, e.g., *nozil* for ZIL-PMEM.
6. Allocate the new `zilog_t` instances and bring them into a usable state. The allocation routine uses `zh_kind` to select the (new) vtable, allocates `zlvt_alloc_size` bytes for the new `zilog_KIND_t` and runs the ZIL kind specific constructor.
7. Enable access to the ZIL API by unblocking the waiting callers.

Note that it is critical that Step 5 happens atomically for all ZIL headers in the pool. Otherwise, a crash could result in a pool with mixed ZIL kinds.

Due to time constraints we have **not yet implemented** the procedure outlined above. As a stop-gap solution, we instead implemented a simplified scheme where a zpool's ZIL kind is determined on pool creation time by a kernel module parameter (`zil_default_kind`). On subsequent pool imports, we derive the pool's ZIL kind from the root dataset's ZIL kind. We prevent attempts to switch the ZIL kind by preventing any changes to the SLOG vdev config in a ZIL-PMEM pool.

6.1.4 ZIL-LWB Suspend & Resume

The ZIL API provides the `zil_suspend` and `zil_resume` functions. `zil_suspend` pauses all ZIL activity and waits until all log entries are obsolete before returning to the caller. `zil_resume` reverts the state to normal operation. Compatibility code for versions of ZFS prior to the *fast snapshots* rely on ZIL suspend & resume for taking snaphots. For newer pool versions, the only consumer is `spa_reset_logs`: when removing a SLOG from the pool, the ZIL is temporarily suspended to ensure that the SLOG does not contain valid log entries. After the SLOG is removed, the ZIL is resumed and the metaslab allocator uses the remaining SLOGs or the main pool devices to allocated LWBs.

With regards to ZIL kinds, only the `spa_reset_logs` use case is relevant since ZIL kinds require a more recent pool version than the *fast snapshots* feature. Our design for changing ZIL kinds (Section 6.1.3) requires suspension of all ZIL activity and thus subsumes the ZIL-LWB specific `zil_suspend` and `zil_resume`. However, the compatibility code for pools before *fast snapshots* needs to be maintained in a way that does not depend on the ZIL kinds feature. Due to time constraints, we were **unable to address suspend & resume in our design**. We expect that the solution will be highly dependent on implementation-level constraints.

6.1.5 ZIL Traversal & ZDB

Whereas ZIL writing and replay are abstracted away by the `zilog_t` refactoring, there are several cases where the raw ZIL-LWB chain is traversed directly using the `zil_parse` function. `zil_parse` exposes several ZIL-LWB specific implementation details to its callers such as blockpointers and the concept of LWBs. This is problematic for ZIL kinds because not every conceivable ZIL kind uses these concepts — ZIL-PMEM being the obvious example. We investigate all users of the ZIL traversal code and come to the conclusion that there is no need for a generalized interface that every ZIL kind needs to implement. The basis for this decision is a manual audit of all `zil_parse` callers:

dmu_traverse This module implements a callback-based traversal of the zpool's data structures. It is used to implement many ZFS features, e.g., *zfs send*. If a dataset is traversed that is a head dataset (i.e., not a snapshot) and its LWB chain has been claimed, the LWBs are included in the traversal.

dsl_scan_zil During a *zpool scrub* (data integrity check of the entire pool), this function traverses claimed LWB chains.

spa_load_verify During pool import this function uses *dmu_traverse* to validate data structures that were modified in the last synced transaction groups.

zdb_il.c The *ZFS debugger* interprets the ZIL header of head datasets, traverses their LWB chain, and dumps its contents to stdout.

Most consumers of *dmu_traverse* operate on snapshots, not head datasets, and therefore do not trigger ZIL chain traversal. The *dsl_scan* and *spa_load* code only traverses the ZIL but does not access its data — the data integrity checks that are done for validation are implemented transparently in the ZIO read pipeline that is used to load the LWBs in the ZIL chain. One compatibility code path (`old_synchronous_dataset_destroy`) uses ZIL traversal to free the ZIL blocks, but can be replaced with a more recent API (`zil_destroy_sync`). *zdb* is an exception since its whole purpose is to interpret the ZIL chain for debugging purposes.

Given this analysis, we come to the conclusion that a generic ZIL traversal API is not necessary in practice. Hence, the ZIL vtable does not include such an API. To maintain pre-ZIL-kinds behavior, we make the following changes as a precursor to the refactoring of `zilog_t` which we described in Section 6.1.2. We change the `zil_parse` API to work directly on a `zil_header_lwb_t*` instead of `zilog_t`. We also rename the function to `zillwb_parse_phys` to reflect the fact that it is specific to ZIL-LWB and does not affect runtime state. We change the *dmu_traverse* API so that callers must be explicit about ZIL-LWB traversal. To avoid ZIL-LWB specifics in the DMU traversal callback, we change `old_synchronous_dataset_destroy` to use `zil_destroy_sync` instead of DMU traversal.

It is our impression that traversal of the ZIL-LWB chain for the purpose of data integrity checking is moot. The reason is that ZIL-LWB cannot distinguish data corruption from the end of the LWB chain because it relies on invalid checksums to detect the end of the chain. It is conceivable that, for claimed-but-not-replayed ZILs, lost LWBs could be detected and surfaces as errors to the user. However, the current ZIL implementation suggests that this case has never been a top priority of the ZIL design. For example, losing a claimed-but-not-replayed log entry can leak space in the main pool or cause a crash during pool import.

github issues + what I asked on slack today 2021-05-05

Other ZIL kinds or a future revision of ZIL-LWB might be able to detect the loss of log entries and handle such situations more gracefully. In that case, an abstract integrity check method on the vtable that operates only on the physical structure, not runtime state, might be advisable.

6.1.6 ZIL-LWB-Specific Callbacks

There are several callbacks in the ZIL API that are necessary for ZIL-LWB. They need to be considered for our ZIL kind refactoring.

zil_lwb_add_txg Necessary to keep the in-DRAM representation of an LWB alive when writing `WR_INDIRECT` blocks.

zil_lwb_add_block Necessary for an optimization that minimizes the amount of flush commands that are sent to the SLOG device.

zil_bp_tree_add During a ZIL traversal with `zil_parse`, this API was used to avoid doing operation more than once for a given blockpointer. It is an implementation detail of ZIL-LWB that is only a public ZIL API because it is used by *zdb*'s ZIL traversal code.

None of them are necessary for ZIL-PMEM nor are they likely to be for other ZIL kinds. Therefore, we prefix the callback functions with `zillwb_` and move them to *zil_lwb.c*. The original call sites are unreachable with ZIL-PMEM which

allows us to ignore them for the remainder of this thesis. We recommend that future work replace these statically dispatched callbacks with dynamic callbacks through function pointers to enforce decoupling.

6.1.7 Reflection & Alternatives

The general concept of ZIL kinds and the vtable-based implementation add complexity to the ZIL code. In this section we discuss the alternatives that we considered for supporting multiple ZIL implementations at runtime.

Alternative #1: We considered to move the level at which we dispatch into ZIL kind specific code one API layer upwards. THis would make `zilog_t` and ITX a private implementation detail of ZIL-LWB. The API layer at which the virtual dispatch would take place are the `zfs_log_OP` and `zvol_log_OP` helper functions which create and assign ITXs for ZPL and ZVOL operations. The sequence diagram in Figure 6.3 provides an example of how they are used by a `write` system call. Declaring this API layer the interface for ZIL kinds would allow ZIL implementations to chose freely how they want to represent log records in DRAM and on stable storage. This additional freedom could be used by future ZIL kinds to implement a different log structure with better scalability or more fine-grained crash consistency guarantees. For example, an earlier design for ZIL-PMEM used a graph-based log structure where log entries for a file in the same dataset could be written in parallel. However, the additional freedom also has significant drawbacks that ultimately led us to the design presented in the previous subsections:

1. The `zfs_log_OP` family of functions only addresses the ZIL write path. There are no equivalent abstractions that wrap the `zilog_t` APIs for ZIL replay or traversal. In order to have a single clean abstraction at the level of `zfs_log_OP`, it would be necessary to extend this API so that `zilog_t` could be hidden as an implementation detail of ZIL-LWB. We were not confident in our ability to design this API extension without the risk of introducing a leaky abstraction.
2. ZIL kind specific functions for logging would also require ZIL kind specific replay functions. In ZIL-LWB this is a non-trivial amount of code. ZIL kinds such as ZIL-LWB that only implement a different persistence strategy would have to duplicate this code, pointlessly increasing maintenance cost.
3. We find it undesirable to allow ZIL-kinds to implement different crash consistency guarantees, in particular if ZIL kinds switch automatically depending on SLOG configuration as proposed in this chapter (Section 6.1.3). Centralizing the ITX code and forcing every ZIL kind to fit into the ITX

model is the best way to ensure that crash consistency guarantees are the same across ZIL kinds.

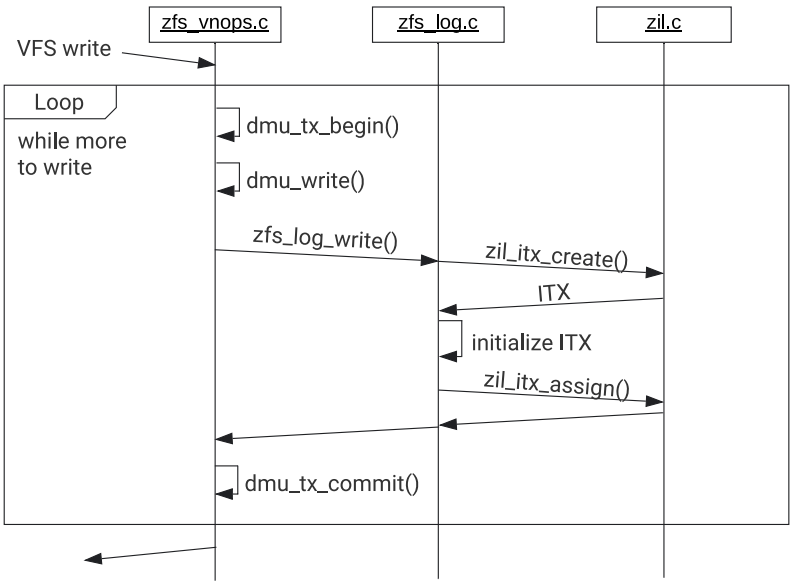


Figure 6.3: Sequence diagram of the APIs involved in creating the ITXs for a `write` system call.

Alternative #2: Since we identified the ZIO pipeline as the main source of latency in ZIL-LWB (Chapter 3), we considered sharing LWBs as a concept between all ZIL kinds. In that scenario, the ZIL kinds would merely be an alternative to the ZIO pipeline. A prototype that adopts this approach has been presented at the OpenZFS 2020 Developer summit by OpenZFS, targeting NVMe drives [Ope20] We found this layer of the ZFS software stack to be too restrictive for ZIL-PMEM:

1. The LWB timeout mechanism for packing multiple entries into a single LWB would add unnecessary latency overhead. This is in conflict with one of our requirements (see Section 4.1.1). This notion is shared by the database community which has deemed group commit schemes such as LWB timeouts unfit for PMEM (see Section 2.1.3 on disk-oriented DBMSs).
2. The design space for PRB/HDL would have been severly constrainted. In particular, fully parallel logging to the same HDL would not have been possible because the LWB chain is inherently sequential. With our ITXG bypass for ZVOLs (see Section 6.3.5), we explore whether parallel logging to PRB/HDL can increase the ZIL's scalability for a single dataset.

We believe that our design for ZIL kinds introduces ZIL kind specific behavior at a layer that allows for shared crash consistency semantics *and* flexibility in the storage model while minimizing code duplication and thus (hopefully) maintenance cost. The interface defined by the vtable is a clean abstraction although some design questions (see 6.1.3 and 6.1.4) as well as some LWB-specific APIs (see 6.1.5 and 6.1.6) remain. The state of each `zilog_KIND_t` is truly private do the ZIL kind's implementation. Neither ZIL-LWB nor ZIL-PMEM access the ITX-related state in the embedded `zilog_t` directly, but only through the `zilog_t` method that computes the commit list. If requirements change in the future, the design alternatives presented in the previous section should be considered.

6.2 PMEM-aware SPA & VDEV layer

Prior to the work presented in this thesis, ZFS had no concept of persistent memory. Fortunately, the requirements of ZIL-PMEM are very limited:

- `/dev/pmem` SLOGs must be recognized as PMEM SLOG vdevs when they are added to the pool.
- The PMEM SLOG's allocatable space must be directly accessible via the PMEM programming model (load/store instructions, cache flushing, ...). If direct access is not possible, the PMEM SLOG vdev must be considered faulty and pool import must be refused. We assume that direct access cannot be lost if it is established.

future work: memory hotplug

We add a new boolean attribute `is_dax` for `disk` VDEVs in the zpool config format. The attribute indicates whether the VDEV supports direct access through the Linux Kernel's DAX APIs. DAX capability is determined by the `zpool` command when creating or adding devices to a pool, using `libblkid`. When opening a vdev marked `is_dax`, the kernel module ensures that all of the block device's sectors are mappable as one contiguous range of kernel virtual address space. Failure to establish this mapping fails the onlining process, leaving the vdev in state `VDEV_STATE_CANT_OPEN`. By default, this state prevents the pool from being imported. Note that the `is_dax` feature applies to all VDEVs and is independent of ZIL-PMEM. It merely records the fact that a VDEV is required to be directly accessible via the DAX APIs. This ensures that future versions of ZFS can leverage DAX capability of main pool VDEVs without needing to change the on-disk format. Consequently, `is_dax` becomes an independent *zpool feature*.

6.3 The ZIL-PMEM ZIL Kind

The changes described in the preceding sections prepare the way for the introduction of the new ZIL-PMEM ZIL kind.

6.3.1 Activation

The ZIL-PMEM ZIL kind is used instead of ZIL-LWB based on the following rule:

> If the pool has exactly one SLOG and that SLOG is `is_dax`, the ZIL kind is ZIL-PMEM. Oterwise, it is ZIL-LWB.

This rule must be evaluated when the pool is imported or whenever the pool's SLOG vdevs are are about to change, i.e., during pool creation, on `zpool add` and `zpool remove`, and when `zpool import` is told to drop log devices via the `-m` flag. To determine whether change ZIL kinds, we compare the resulting desired ZIL kind $k_{desired}$ with the `zh_kind` value of any ZIL header in the pool. If they do not match, we change the ZIL kind of all headers as described in Section 6.1.3. This implicit activation & deactivation of ZIL kinds makes ZIL-PMEM completely transparent to the administrator, thereby fulfilling our requirement of simple administration. ZIL-PMEM is automatically used if it supports the user-specified SLOG device configuration, i.e., a single /dev/pmem0 SLOG VDEV. Otherwise, the system transparently falls back to ZIL-LWB.

Note: We have not yet implemented changing ZIL kinds after pool creation and thus not implemented the design described in the previos paragraph. Instead, the `zil_default_kind` kernel module parameter determines the ZIL kind of a newly created pool (ref. Section 6.1.3). If the ZIL kind is ZIL-PMEM, the implementation enforces that the pool config has exactly one SLOG VDEV that is `is_dax`, and prevents any changes to the SLOG vdev config.

Alternative Design The implicit activation and deactivation based on the rule above makes the rule effectively part of the zpool's on-disk format because it must deliver deterministic results for an unchanged pool, regardless of any future software changes. If the rule evaluation is not stable, the pool's ZIL kind could "flap" between imports which in turn would fail the import if there are unreplayed logs that prevent changing ZIL kinds. It is worth reconsidering whether full implicitness is actually desirable or whether surfacing ZIL kinds to the administrator is acceptable. For example, we could store the pool's *current* ZIL kind in the MOS and provide an zpool command or property to change it. Before the change is executed, the desired ZIL kind would check that the pool configuration is supported and fail the change operation if that is not the case. In the case of

ZIL-PMEM, this hook would evaluate the rule above. Any changes to the zpool topology (`zpool add`, `remove`, `import` with `-m`), would also invoke this hook.

6.3.2 PMEM Space Reservation & PRB Construction

If the pool's ZIL kind is ZIL-PMEM, we reserve all of the allocatable space on the PMEM SLOG vdev for PRB. To accomplish this, we introduce a new allocation class called `exempt` that is, by convention, never used for SPA allocations. In our current implementation, the `zpool` command assigns this allocation class automatically to `is_dax` SLOG VDEVs, instead of the `log` allocation class. Note that the VDEV labels, which attribute the VDEV to the zpool and store parts of the zpool config, are located outside of the allocatable space. Readers and updaters of VDEV labels continue to use block device IO (`zio_read_phys` and `zio_write_phys`).

maybe always allocate all of it so we can get rid of all the special cases?

ZIL-PMEM DAX-maps the allocatable space of the `exempt` SLOG VDEV and constructs the pool's PRB instance on top of it.

ref linux intro?

1. The pool import procedure allocates the `prb_t` and stores the pointer in the DRAM object that represents the imported pool (`spa_t`). When the pool is exported, the `spa_unload` procedure frees the `prb_t`. The `prb_t`'s lifetime is thus a contiguous sub-span of the lifetime of `spa_t`.
2. Immediately after allocating the PRB, pool import sets up the chunk objects and adds them to the PRB. If the pool is being created, we reset the chunks to a known PMEM state (`prb_chunk_initialize_pmem`) before adding them to PRB.

impl is a little different, doesn't matter?

We use a simplistic partitioning scheme to divide the PMEM space into chunks. The main advantage of the scheme is that it does not need to persist any metadata:

- Chunks have a hard-coded size of 128 MiB.
- We partition the PMEM SLOG vdev's allocatable space as a contiguous array of 128 MiB segments, starting at offset zero.
- Assuming A bytes of space, this yields `nchunks := A>>27` chunks and less than 128 MiB of wasted space at the end of the allocated space.
- We do not store `nchunks` anywhere. Instead, we prohibit online resizing of the PMEM SLOG vdev which allows us to deterministically re-compute the value.

Note: Additional experience gained during implementation and testing has taught us that instead of attempting to exempt the PMEM SLOG VDEV's space from regular allocation, it is probably preferrable to actually pre-allocate all the PMEM space on the SLOG VDEV from the SPA. The reason is that some lesser-known

comonents of ZFS expect that unallcoated SPA space can be overwritten in the background (e.g., the code that supports TRIM or the *vdev initialize* functionality). Also, there is existing infrastructure to prevent removal of SLOGs with allocations, for which we currently require a ZIL-PMEM specific special case.

6.3.3 Dataset & HDL Lifecycle Synchronization

language

Whereas `prb_t` and `spa_t`'s allocation lifecycles line up nicely, the same is not true for datasets and HDLs: HDLs must be set up during pool import and must not be torn down via `prb_teardown_hdl` until either their dataset destroyed or the zpool is exported. In contrast, the per-dataset runtime state (`dsl_dataset_t`, its `objset_t` and the objset's `zilog_t`) is allocated "on demand" when its consumer, the ZPL code, *holds* it by its object ID in the MOS. Specifically, the first holder performs the allocation and recovers state from the corresponding persistent structure (`dsl_dataset_phys_t`, `objset_phys_t` and, partially, `zil_header_t`). Conversely, if there was already another holder, the existing DRAM object is shared. Once a consumer no longer needs to hold a dataset, it *releases* its hold on the instance. The last consumer that releases the structure frees the DRAM object. (Holds and releases are implemented through reference counting.)

The mismatch of HDL and dataset object lifetimes is relevant to ZIL-PMEM because `zilog_pmem_t` needs access the corresponding HDL for claiming, replay, and writing log entries. Our solution is to add a search tree to `spa_t` which we call *HDL map*. It maps from the dataset's object set ID to its HDL. We add the necessary callbacks to synchronize the *HDL map* with the dataset layer.

- During pool import, after setting up the PRB but before claiming, we iterate over the head datasets[1] in the pool, set up the HDLs for each of them, and add them to the *HDL map*.
- When a new head dataset is created (`dmu_objset_create_impl_dnstats` and `dmu_objset_clone_sync`), we set up its HDL and add it to *HDL map*.
- When a head dataset is destroyed (`zil_destroy_sync`), we tear down the HDL and remove it from the *HDL map*.

Whenever the ZIL-PMEM implementation needs access to the HDL, it looks up the HDL in the *HDL map* and aquires a reference to it. We use reference counting to assert that there are no dangling references when the head dataset is destroyed and the HDL is torn down. We protect the *HDL map* against concurrent modifications using ZFS's *read mostly lock* which is a reader-writer-lock that is optimized for reads. Figure 6.4 visualizes the constellation of objects, their lifetimes, and

language

[1] Head datasets are ZPL filesystems, ZVOLs, and clones thereof. "Behind" a head dataset can be one or more snapshots.

reference relationships.

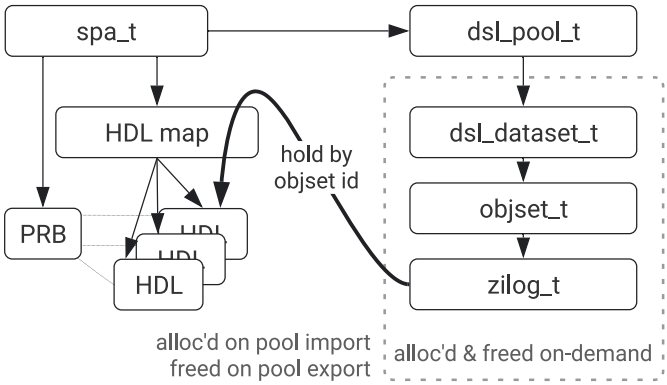


Figure 6.4: The different allocation lifecycles of HDLs and datasets, and how we bridge them for ZIL-PMEM.

6.3.4 ZILOG_PMEM_T

The `zilog_pmem_t` structure and its methods in the `zilpmem_vtable` implement the ZIL-PMEM ZIL kind. Its role is that of an adaptor between the shared high-level ZIL code (*itxg* structure) and the dataset's HDL that is associated with the dataset via the *HDL map*. Apart from the exceptions listed below, the methods in the vtable are thin wrappers around HDL's interfaces. The general pattern for a ZIL-PMEM method is as follows:

1. Aquire a reference to the dataset's HDL from *HDL map*.
2. Invoke the HDL method.
3. Release the HDL reference.

To avoid reference counting overhead for HDLs on the write path, we aquire a HDL reference once when the ZIL is opened during mounting and hold it in the `zilog_pmem_t` until the dataset is unmounted.

The following cases required additional code to adapt betweent the two domains:

Error Handling During Pool Import & Claiming We extracted the ZIL vtable from the original ZIL(-LWB), as described in Section 6.1.2. Consequently, it inherits ZIL-LWB's model for integrity checking and claiming of the ZIL during pool import: ZIL-LWB checks the LWB chain twice: the first pass (`zil_check_log_chain`) checks for log corruption to fail the pool import early if the log is corrupt. The second pass (`zil_claim`) does ZIL-LWB's version of

claiming and records the maximum claimed log record in the ZIL header. It discards traversal errors because it assumes that no online log corruption happend since the first pass. (Note that `zil_claim` does have a return value, but only due to software-technical reasons. It must always return zero.) ZIL-LWB's approach opens a window for time-of-check vs time-of-use bugs which we discovered and reported during development. In contrast, PRB/HDL does checking and claiming in a single pass (`prb_claim`), reports corrupted log state through its return value, and is designed to correctly handle online log corruption during claiming and replay (see Section 5.7).

ref

Due to time constraints, we have not refactored the zpool import procedure to allow claiming to fail. Until that shortcoming of our implementation is addressed, we trigger a kernel panic if `prb_claim` returns an error.

ZIL Header Updates Remember from Section 5.13.1 on the PRB/HDL API that the responsibility of *persisting* the ZIL-PMEM header is split: HDL APIs that update the ZIL header return the updated version through an *out*-parameter and the API consumer is responsible for persisting the update to the main pool.

Unlike DMU consumers, the ZIL implementation itself is responsible for queuing up changes that need to applied for the *open*, *quiescing* and *syncing* txg. When *txg sync* is syncing the `syncing_txg`, it invokes the `zil_sync` API of each `zilog_t`. The role of `zil_sync` is to modify the buffer of the ZIL header that is stored in `objset_phys_t::os_zil_header` the state that shall be persisted in the `syncing_txg`. *Txg sync* then uses this buffer's content when it writes out `objset_phys_t` to the on-disk tree structure.

ZIL-PMEM queues the updated header values returned by the HDL APIs in a 4-ary array that is indexed by `[txg&3]`. Each cell contains the tuple `(txg: u64, header: zil_header_pmem_t)`. A header update `(txg_u, header_u)` overwrites the cell at index `txg_u&3`. Updates must only be made from the *open* or *quiescing* txgs. `zilpmem_sync` then reads cell `C := updates[syncing_txg&3]`. If `C.txg == syncing_txg`, it updates `objset_phys_t::os_zil_header` to `C.header`. Otherwise, it does not modify it at all.

zil_commit We implement `zil_commit` as follows:

1. Aquire a `zilog_pmem_t`-wide mutex.
2. Get the *commit list* from the *itxg* data structure.
3. Invoke the HDL's `prb_write_entry` method for each ITX (i.e., log record) on the commit list. It starts a new generation for every ITX to encode the sequential log structure.
4. Release the mutex.

The mutex ensures that committers are serialized. This is critical for the correctness of *sync* or *fsync* which are cummulative. Assume two threads A, B that issue the *sync* system call. Assume A starts a *sync* system call and drains all ITXs from the *itxg* into its commit list. If A is now preempted by B which also starts a *sync* system call, B's commit list will be empty. B would return to userspace, giving the caller the impression that all dirty data has been synced, although they still need to be written to the HDL by A.

Starting a new generation for every entry ensures that the commit list's entries will be replayed in commit list order. The `zilog_pmem_t`-wide mutex already serializes the start of the new generations as required by the HDL API, see Section 5.13.3.

Serializing the entire `zil_commit` call allows for less parallelism than ZIL-LWB's *commit ITXs*. Commit ITXs implement a sort of pipelining by allowing log writers to issue LWB ZIOs in parallel while ensuring that they only return from `zil_commit` after all LWBs up to and including the log writer's last LWB have been written. Whereas a similar approach cold be applied to ZIL-PMEM in principle by mapping parallel writers to the same generation, the current implementation of commit ITXs is too tightly coupled to the concept of LWBs and the ZIO pipeline.

language, help

WR_NEED_COPY Chunking Remember from Section 2.3.3 that the ZIL allocates ITXs for *all* changes to a dataset in DRAM. The ITXs are queued in the *itxg* structure from where they are either `zil_commit`ted or freed when the txg syncs. To avoid unnecessary memory usage and performance overhead, ITXs that log large *write* operations do not always contain a copy of the written data. Instead, the ITX is marked as a `WR_NEED_COPY` ITX. Such ITXs only contain the object number of the DMU object and the byte range within the object that was written. The creation of the log record is deferred until `zil_commit` which creates the actual *write* records whose payload data is read back in from the DMU object.

To support WR_NEDD_COPY in ZIL-PMEM, we implement an isolated structure that turns a single `WR_NEED_COPY` record into an iterator over *write* log entries. `zil_commit` writes each entry yielded by the iterator to the HDL using `prb_write_entry`. Whereas the repeated chunk aquisition and dependency tracking during each HDL write incurs a small overhead, it also increases fairness among HDLs if the commit slots are contended.

6.3.5 ITXG Bypass For ZVOL

ZIL kinds are forced into the *commit list* model that defined by the *itxg* structure which is shared among all ZIL kinds. This model leads to an inherently sequential representation of the ZIL contents: ZIL-LWB is a long chain of log entries grouped in LWBs, and ZIL-PMEM starts a new generation for every entry on the commit list to encode it properly. We believe that this architecture is the best compromise for consistent behavior across ZIL kinds, performance, code duplication, and maintainablility. (See Section 6.1.7 for a reflection on alternatives.) Our evaluation in the next chapter shows that ZIL-PMEM yields significant latency benefits and comes close to saturating PMEM bandwidth when performing synchronous I/O from four threads to *separate* datasets. However, we also wanted to explore the performance potential of PRB/HDL if we eschew *itxg* in favor of a design that allows for parallel writing of entries for the same dataset. The result is an *itxg bypass* mode for ZIL-PMEM which we present in this subsection.

ok verb?

ZVOLs And The ZIL

The mode only works for ZVOLs which are sparsely allocated virtual block devices. Other block device consumers, e.g., virtual machines or other file systems, can treat the ZVOL as any other block device exposed by the kernel. ZVOLs are implemented as a dataset with a single DMU object that contains the virtual block device's data. When a block device driver accesses the ZVOL block device (*read, write, discard*), ZFS maps the block device operations to DMU operations on the object. The modifications are logged to the ZIL as *write* and *truncate* ITXs. If the block device operation has synchronous semantics, ZFS calls `zil_commit` before acknowledging completion of the block device operation. The following pseudocode describes how the ZVOL processes a block device operation.

```
Input:
    op      block device operation
    zv      zvol
Steps:

    if op.pre_flush:
        zil_commit(zv.dmu_object_id)

    match op {
        Read(...) => {
            dmu_read(zv.dmu_object, op.buf, ...)
        },
        Write(...) => {
            tx := dmu_transaction()
            dmu_write(tx, zv.dmu_object_id, ...)
            itx := zil_itx_create(...)
            zil_itx_assign(itx)
            dmu_tx_commit(tx)
        },
        Discard(...) => {
```

```
            // analogous
        }
    }

    if op.post_flush:
        zil_commit(zv.dmu_object_id)

    op.done() // acknowledge completion
```

The Linux kernel's model for block device I/O is asynchronous. The assumption is that the block device IO is submitted (`submit_bio`), processed asynchronously by the storage device, and eventually marked as completed (`BIO_END_BIO`). By default, ZVOL uses a thread pool (*taskq*s) to process block device I/O asynchronously and in parallel. The `zvol_request_sync=1` tunable changes this behavior to synchronous mode where the procedure outlined above is executed in synchronously in `submit_bio`. We will refer to this tunable in the Section 7.3.3 of the evaluation.

ITXG Bypass

We observe that the ZIL's fully sequential structure is unnecessary for ZVOLs because block device consumers must already account for device-internal caching. If a block device consumer actually requires consistency at a given point in time, it already explicity indicates this to the block layer. In Linux, this indication is either given by the `bio` operation's type (`REQ_OP_FLUSH`) or the pre-flush or post-flush flags (`REQ_PREFLUSH`, `REQ_FLUSH`).

We leverage this observation as follows. We do not queue ITXs in *itxg* but write all log entries directly to the HDL from within the DMU transaction. By default, we do not start a new generation which allows the HDL to be written in parallel. If the block device consumer requests a flush, we serialize HDL access to start a new generation.[2]. We use a reader-writer-lock to achieve this behavior, as illustrated by the following pseudo-code.

```
Input:
    op      block device operation
    zv      zvol
      rwl       reader-writer-lock
      start_gen
Steps:
    if op.pre_flush || op.post_flush:
        zv.rwl.write_lock()
        zv.start_gen = true
    else:
        zv.rwl.read_lock()
        if zv.start_gen:
            zv.rwl.upgrade()
```

[2]Remember that external synchronization on generation start is a requirement of PRB/HDL, see Sections 5.12.2 and 5.13.3

```
match op {
    Read(...) => { ... }
    Write(...) => {
        tx := dmu_transaction()
        dmu_write(tx, zv.dmu_object_id, ...)
        itx := zil_itx_create(...)
        prb_write_entry(
            itx.into_log_entry(),
            needs_new_gen=zv.start_gen
        );
        assert zv.start_gen => zv.rwl.holding_write_lock
        if zv.start_gen:
            zv.start_gen = false
        dmu_tx_commit(tx)
    },
    Discard(...) => { /* analogous */ }
}

if op.post_flush:
    assert zv.rwl.holding_write_lock
    zv.start_gen = true

op.done() // acknowledge completion
```

The following figure provides an example sequence of two threads that issue block IO operations to the same ZVOL in parallel.

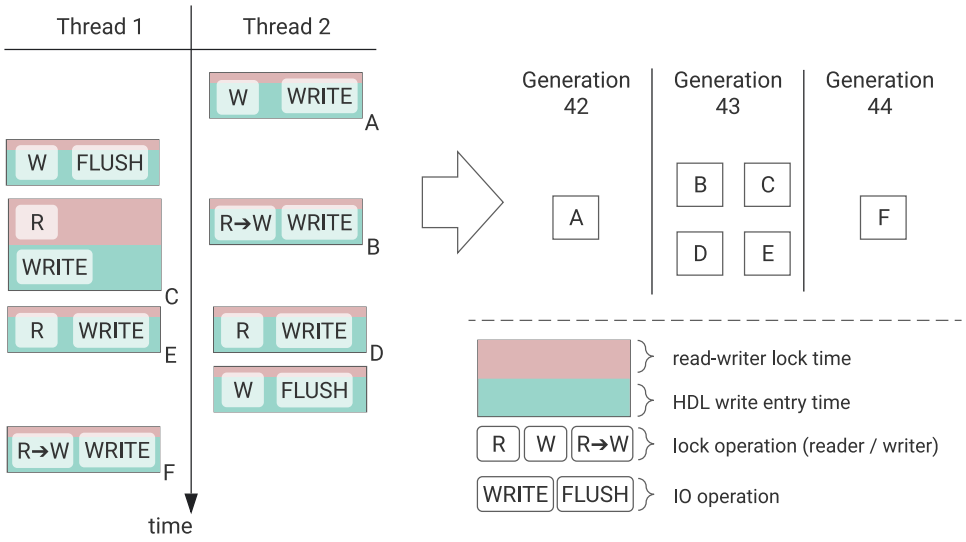


Figure 6.5: Example timeline for the IO operations of two threads that write to the same ZVOL, with the ITXG bypass mode enabled. The first write operation after a flush detects that `zv.start_gen` is set and hence upgrades its read-lock to a writer-lock so that it can write the first entry of the new generation sequentially. As soon as it is done, it sets `zv.start_gen = false` and relinquishes the writer lock. Subsequent writes until the next flush operation aquire a read lock, observe `zv.start_gen = false`, and write to HDL in parallel which results in multiple entries for generation 43. Entry F, which is the first entry after the second flush, starts a new generation.

Chapter 7

Evaluation

In this chapter we evaluate whether ZIL-PMEM meets the project goals established in Section 4.1.1.

7.1 Usability & Architecture

Our high-level requirements for ZIL-PMEM were: simple administration, sharing of PMEM as a pool-wide resource, same crash consistency guarantees, and coexistence with the existing ZIL(-LWB). Our design meets all of these requirements:

Simple Administration We have proposed a design where ZIL-PMEM is automatically activated if exactly one PMEM SLOG vdev is configured (Section 6.3.1). This would make ZIL-PMEM completely transparent to the administrator and not bring any user-visible change in ZFS's administrative tools. However, our implementation does not yet support changing the ZIL kind of a zpool after its creation (Section 6.1.3).

Pooled Storage PRB wraps the PMEM SLOG device and shares it as a pool-wide resource among all HDLs / datasets.

Coexistence The introduction of *ZIL Kinds* allows for coexistence of ZIL-LWB and ZIL-PMEM in code and at runtime. The layer at which ZIL kinds were introduced in the architecture allows for sharing of all code that deals with the ZIL's logical structure but enables coexistence of different persistence strategies.

Same Guarantees ZIL-PMEM maintains the same crash consistency guarantees as ZIL-LWB towards userspace, courtesy of the shared logical structure and PRB/HDL's guarantees.

7.2 Correctness

better verb? We use unit tests and integration tests to validate our implementation.

7.2.1 Testing Strategy for PRB/HDL

We test PRB/HDL's functionality in user space which is possible because PRB/HDL is included in the *libzpool* user-space library. *libzpool* contains the majority of the ZFS kernel module's code — it is used to by *zdb* (the ZFS debugger) and the *ztest* stress testing tool. We implement our unit tests in Rust to leverage its expressive type and macro system, rich standard library, and built-in testing harness. We use the popular *bindgen* crate to generate the bindings to *libzpool* and implement a few idiomatic wrappers around PRB/HDL to reduce boilerplate code in our tests.

API Walkthrough

We codify the walkthrough of the PRB/HDL API that we presented in Section 5.13 in a large test:

1. Allocate a ZIL header on the stack.
2. Create a chunk whose space is allocated from the heap.
3. Construct PRB and add the chunk to it.
4. Setup a HDL.
5. Create a log for the HDL.
6. Write two entries for $txg = 2$ body values 23 and 42 to the HDL.
7. Teardown the object set.
8. Destroy the PRB with `free_chunks=false`. This moves ownership of the chunk moves back from PRB to us.
9. Construct a new PRB instance with the same chunk.
10. Setup a HDL from the ZIL header.
11. Trigger claiming.
12. Trigger replay, and record the replay callback invocations. For each invocation, we read the body length and content, assert that no read error occurred. We record body content and the ZIL header update as records in a list. We report successful replay for every callback invocation.
13. After replay is complete, we compare the recorded list's contents to the expected replay order.

Correct Handling Of Obsolete Entries

PRB/HDL assumes that there are only three unsynced txgs at any given time. This manifests in frequent use of the `txg&3` indexing idiom that is common in ZFS. Whenever state needs to be kept for each of the unsynced txgs, a 4-ary array is used to represent it. The `txg&3`'th element in the array then contains the state for `txg`. The value of `txg` is repeated within the per-txg state to detect when an array element is re-used.

In the context of PRB/HDL, it is critical for correctness that an entry E_i for txg $E_i.txg$ is only written if it is either newer than txg_{open} or one of $\{txg_{open}, txg_{open}-1, txg_{open}-2\}$ (dependency tracking, see Section 5.5). We add tests that ensure that the `prb_write_entry` API returns an error if entries are written that do not meet this criterion. We also test the behavior of replay to ensure that, if the write path implementation were incorrect, replay would identify the problem with a distinct error code.

Replay Algorithm

We structure the PRB/HDL implementation such that it becomes possible to test the core replay logic that we described in Sections 5.4, 5.5, 5.6 and 5.7: The idea is to isolate the core logic in a function `replay_resume` that takes the following arguments as input:

NodeSet The set of entries, represented as *replay nodes*, that were discovered for the HDL.

ReplayState A pointer to the DRAM representation of replay state,

Callback A callback that receives as arguments a) the replay node and b) a pointer to the replay state that needs to be persisted to the ZIL header during replay.

`replay_resume` constructs the replay sequence from *NodeSet* and then invokes the *Callback* for each entry that needs to be replayed. The `prb_claim` API uses `replay_resume` for ZIL headers in state *logging* to determine E_{seal}, and for a dry-run of replay for headers in state *replaying* to detect missing entries early during pool import (see Section 5.7, Step 1). It provides its own *Callback* (`prb_claim_cb`) which checks that all replay nodes' bodies can be read without error, using the `prb_replay_read_replay_node` API that the actual replay callback is going to use as well. The `prb_replay` API uses `replay_resume` for actual replay. Its *Callback* (`prb_replay_cb`) serializes *ReplayState* to the representation stored in the ZIL header, builds the ZIL header update, and invokes the user-provided replay callback. Figure 7.1 visualizes this architecture.

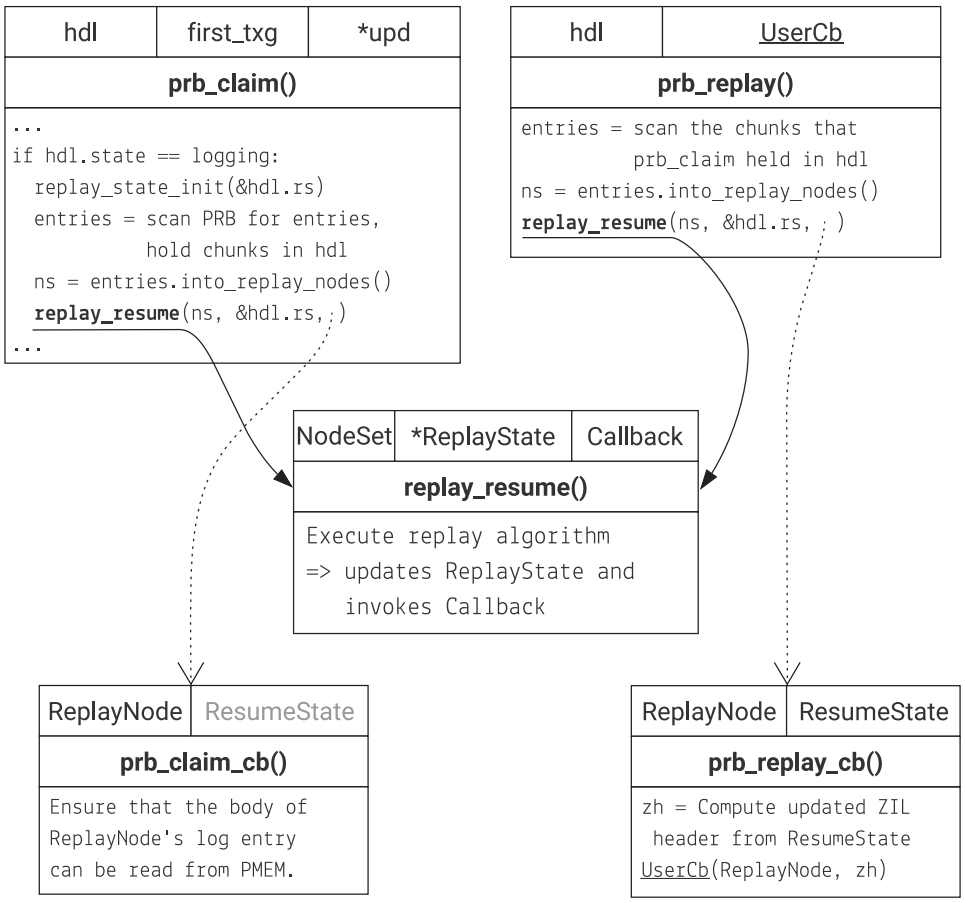


Figure 7.1: The architecture that enables code re-use and testability for the core replay logic.

This factorization of responsibilities provides maximum flexibility for testing because it decouples the storage substrate (PRB & chunks) from the per-HDL logical log structure:

Independence of PRB & Chunks The testing code does not need to mock chunks or PRB. It also does not need to allocate the actual entries. Instead, it is sufficient to allocate arbitrary *replay nodes* and to put them into the *EntrySet*.

Independence of ZIL-PMEM It is not necessary to mock or subsititute any behavior implemented in ZIL-PMEM.

Crash Consistency Testing To test crash consistency and correctness for resumed replay, the testing code can simply forge an arbitrary replay state in DRAM and invoke `replay_resume` with it.

We leverage Rust's macro system to define test cases in a concise and expressive manner. Our goal is to minimize the mental step from the two-dimensional grid visualization to a test case (ex. Figure 5.2). The following code snippet is an excerpt from the test suite:

```
TestSet {
  title: "I- shape",
  entries: maplit::btreemap! {
    // SynthEntry(txg,gen,gsid, (<counters>))
    "A" => SynthEntry(3,10,1, ([(0,0),(0,0),(0,0)])),
    "B" => SynthEntry(4,10,2, ([(0,0),(0,0),(0,0)])),
    "C" => SynthEntry(5,10,3, ([(0,0),(0,0),(0,0)])),
    "D" => SynthEntry(4,11,1, ([(5,1),(4,1),(3,1)]))
  },
  tests: vec![
    test! {
      "tail truncation ok",
      claim_txg = 1,
      // hide entry D during replay and assert that
      // replay does not complain
      stages = stages!(single, hide=&["D"], check=OK,),
      // expected replay callback invocations
      expect_replay = vec!["A", "B", "C"],
    },
    test! {
      "'A missing' detected, but rest of gen is replayed",
      claim_txg = 1,
      // hide entry A during replay and ensure that replay
      // complains about missing entries
      stages = stages!(single, hide=&["A"], check=M_ENTRIES,),
      // expected replay callback invocations
      expect_replay = vec!["B", "C"],
    },
  ],
}
```

Our tests check the following properties of the replay code:

Shapes We test replay for different "shapes" in the 2-D grid. For example, a "V"-shape with entries for txgs, or an "I-" shape as in snippet above.

Claim Txg We ensure that log entries for txgs $\leq precrash\_txg$ are not replayed because their effect is already part of the dataset's state. In the code base, it is more common to refer to $claim\_txg = precrash\_txg+1$.

Missing Entry Handling We test the handling of missing entries in both the already replayed and the still to-be-replayed parts of the replay sequence, as well as the special cases for missing entries in the last generation.

Resumability We test resumability of `replay_resume` for the cases of missing entries, re-appearing entries, and no change of *EntrySet* between resume attempts.

Entry Reappearance We test that re-appearing entries ordered *before* the last-replayed entry are ignored, and re-appearing entries in the *unreplayed* part are discovered and replayed.

Testing Runtime Assertions

We make heavy use of runtime assertions to protect against existing and future implementation errors. On the recovery path, which is not performance-critical, assertions are enabled in release builds. On the write path, most assertions are only enabled for debug builds. In *libzpool*, which is used by our user space tests, assertions are always enabled.

We modify *libzpool* such that our test suite can install a callback that is invoked when a runtime assertion fails. The callback receives the formatted panic message as an argument. We use this mechanism to implement *crash tests* which assert that a piece of test code triggers a runtime assertion. The panic message is used to identify the assertion since cross-FFI stack unwinding is not yet supported in Rust.

At this time, we use crash tests to ensure that incorrect usage of the PRB API is detected at runtime:

- Operations on a HDL in the wrong state, e.g., replay of a HDL which was not claimed.
- Attempts to set up a HDL that has already been set up.
- Writes that exceed the maximum allowed chunk size.
- Correct interplay of garbage collection and PRB during PRB allocation (critical for pool export).

7.2.2 Testing Strategy for ZIL-PMEM

Our changes to ZFS have less coverage than PRB/HDL. The OpenZFS project has the following existing facilities for testing:

ZFS Test Suite (ZTS) The ZFS Test Suite is a set of automated integration tests that assert behavior of the kernel module and CLI tools. ZTS is implemented in shell and provides a significant amount functional test that exercise both the data path and administrative layer of ZFS.

ztest The *ztest* tool is a user-space binary that links against *libzpool* and stress-tests core components of ZFS that are shared among all supported plat-

forms. It define routines that imitate the kernel module's use the core components' APIs. The stress-testing aspect comes from repeatedly dispatching these routines to a pool of threads. Additionally, IO failures are injected through ZFS's fault injection facilities (ref. Section 2.1.6), and system crashes are simulated by randomly killing the *ztest* process. *ztest* is particularly useful for exposing race conditions in the administrative layer that would not be discovered by the mostly sequential usage in the ZTS.

language

We identify three major challenges in adapting OpenZFS's testing infrastructure to ZIL-PMEM:

Dimensionality ZIL kinds add a new dimension in the configuration space that has not been anticipated by the designers of ZTS and ztest. Since all ZIL kinds should be functionally equivalent, the ideal testing strategy would execute all existing tests for each ZIL kind. However, the testing harnesses for ZTS does not have the infrastructure for this task.

File VDEVs Both ZTS and ztest rely heavily on the *file* VDEV type. *File* VDEVs are plain files that can be used in lieu of block devices when constructing a zpool. They are problematic for ZIL-PMEM because we only added support for DAX block devices, not files. To support *file* VDEVs for ZIL-PMEM, the ZFS kernel module would need to `mmap` the file in the kernel. We are unaware of a kernel API that allows an external module, such as ZFS, to accomplish this. For crash consistency, the memory mapping would further need to map directly to PMEM, i.e., be located on a Linux filesystem mounted with the *dax* mount option.

ref pmem in linux section

VDEV Management A significant number of tests in ZTS and ztest deals with mutations of the pool's VDEV config, e.g., adding, removing, growing, and offlining devices. This is related to ZIL-PMEM because our design goal is to transparently switch ZIL kinds based on the absence or presence of exactly one PMEM SLOG vdev. However, the current state of our implementation does not yet support changing ZIL kinds after pool creation and prevents changes to the SLOG VDEV config to ensure that the PMEM SLOG vdev is always present (Section 6.1.3). Thus, many tests that should work correctly in the final implementation currently fail when attempting to change the VDEV config.

language

We modify *ztest* as follows:

Mocking *Ztest* mocks the representation of datasets (`ztest_ds_t`) but uses the production implementation of the `objset_t` and `zilog_t` types and related APIs. Since ZIL kinds do not change the public ZIL API significantly, the parts of ztest that are concerned with mocking the dataset lifecycle, such

as dataset creation, snapshotting, destruction, and cloning, only required minor modifications.

ZIL-LWB Specific Assertions Some *ztest* routines assert ZIL-LWB specific state. We move these assertions behind `if` conditions that check whether the ZIL kind is ZIL-LWB and if so, downcast `zilog_t` to `zilog_lwb_t` to perform the check.

Configurable ZIL Kinds We add a command line switch to *ztest* to configure the default ZIL kind that is used for the test pool. This is equivalent to the `zil_default_kind` kernel module parameter (Section 6.1.3).

VDEV Management There are several *ztest* routines that randomly add and remove VDEVs, inlcuding SLOG devices. We change the function that selects the target VDEV such that it never returns a SLOG device if the pool's ZIL kind is ZIL-PMEM.

We have not added any ZIL-PMEM specific assertions or additional test routines to *ztest*. To get coverage for ZIL-PMEM and ZIL-LWB, it is required to run ztest twice with the respective values for the command line flag.

Regarding the ZTS, the following tests are relevant for the ZIL:

```
$ fdfind --print0 slog ./tests | xargs -0 grep log_assert | ...
slog_001_pos.ksh  Creating a pool with a log device succeeds
slog_002_pos.ksh  Adding a log device to normal pool works
slog_003_pos.ksh  Adding an extra log device works
slog_004_pos.ksh  Attaching a log device passes
slog_005_pos.ksh  Detaching a log device passes
slog_006_pos.ksh  Replacing a log device passes
slog_007_pos.ksh  Exporting and importing pool with log devices
slog_008_neg.ksh  A raidz/raidz2 log is not supported
slog_009_neg.ksh  A raidz/raidz2 log can not be added
                  to existed pool
slog_010_neg.ksh  Slog device cannot be replaced with spare
slog_011_neg.ksh  Offline and online a log device passes
slog_012_neg.ksh  Pool can survive when one of mirror log
                  device get corrupted
slog_013_pos.ksh  Verify slog device can be disk, file,
                  lofi device or any device
slog_014_pos.ksh  log device can survive when one of the
                  pool device get corrupted
slog_replay_fs_001.ksh   Replay of intent log succeeds
slog_replay_fs_002.ksh   Replay of intent log succeeds
slog_replay_volume.ksh   Replay of intent log succeeds
```

The `slog_*_{pos,neg}.ksh` tests cover VDEV management operations which are well-described by the excerpts above. The `slog_replay_*` tests exercise the replay code path as follows:

1. Create a zpool and a dataset in it.
2. Mount the dataset to start an on-disk ZIL chain.

3. Stop txg sync via *zpool freeze*. This keeps the current unsynced txgs open and inhibits garbage collection, both in ZIL-LWB and ZIL-PMEM. From now on, all modifications to the mounted file system queue up as dirty state in DRAM that is never synced out. However, ZIL entries can still be written since the ZIL is written independently of txg sync.
4. Perform synchronous file system operations that cause new ZIL entries to be written.
5. Archive the filesystem's contents via VFS, i.e., with *tar*.
6. Export the pool.
7. Import the pool again. The pool is now no longer frozen.
8. Mount the filesystem to trigger replay.
9. Use the *diff* utility to compare the archived state, including metadata such as *mtime* and extended attributes, against the replayed state of the filesystem. The test passes if no differences are found.

The `slog_replay_fs_00{1,2}` work directly on ZPL filesystems whereas `slog_replay_volume` exercises the ZVOL code paths by instantiating a platform-native file system on top of the ZVOL (ext4 on Linux, UFS2 on FreeBSD).

We modify ZTS as follows:

1. Add a mechanism to skip tests based on the value of the `zil_default_kind` module parameter (Section 6.1.3).
2. Skip all `slog_*_{pos,neg}.ksh` tests for ZIL-PMEM.
3. Add a new ZIL-PMEM specific test to assert the behavior of our current implementation. The test ensures that "ZIL-PMEM does not allow device removal, addition, replacing, offlining or pool splitting".
4. Replace `slog_replay_*` with ZIL kind specific variants:
   (a) Copy `slog_replay_*` to `slog_replay_*__lwb` and `slog_replay_*__pmem`.
   (b) Remove `slog_replay_*`.
   (c) Make `slog_replay_*__{lwb,pmem}` exclusive to either ZIL kind.
   (d) For `slog_replay_*_pmem`, hard-code `/dev/pmem0` instead of *file* VDEVs as a SLOG device.

To execute the tests, it is now required to load the ZFS kernel module and configure `zil_default_kind` before starting the ZTS test harness. To achieve full coverage, the ZTS must be run twice, once for ZIL-LWB and once for ZIL-PMEM.

7.2.3 Results

Our user-space tests for PRB/HDL enabled fast iteration on the implementation with high confidence in its correctness. The majority of test cases are the collection of edge-cases that we discovered while designing the PRB/HDL data struc-

ture. Naturally, we have addressed all of these edge cases in the design and implementation. Thus, thus all of our PRB/HDL tests pass.

Ztest helped us find many implementation errors in the integration code. Again, the high iteration speed and ease of debugging in user space proved very beneficial. We have run the *zloop.sh* script for X minutes for each ZIL kind and did not encounter any crashes.

value

After our changes to the ZIL-related ZTS tests, all of them pass (or are skipped) for both ZIL-LWB and ZIL-PMEM. They exposed significantly less implementation issues than *ztest*. (We used ZTS during initial development and only adopted *ztest* later in the process, hence we were able to experience the difference.)

We use the code coverage analysis tools included in the OpenZFS and Linux kernel build system to get some quantitative results. The following table compares the code coverage by $\{\text{Upstream, Our Tree}\} \times \{\text{ztest + PRB/HDL tests}), \text{ZTS with kernel module}\}$.

todo

7.3 Performance

We evaluate the performance of ZIL-PMEM using an extensive set of benchmarks. We start with the familiar 4k synchronous random write workload that motivated this thesis (ref. Section 3), asking the following questions:

- How many IOPS does ZIL-PMEM achieve for this workload? What is the speedup over ZIL-LWB?
- How does ZIL-PMEM scale with an increasing number of writer threads?
- How homogenous is the service provided by ZIL-PMEM, i.e, what are the tail latencies and how do they compare to ZIL-LWB?

In Section 7.3.3, we expand our scope to more realistic workloads and different storage stacks:

- What benefit does ZIL-PMEM provide to applications that frequently perform synchronous I/O?
- How does ZFS with ZIL-PMEM perform compared to other Linux filesystems that were adapted to PMEM?
- Is ZIL-PMEM a viable alternative to Linux filesystems deployed on top of dm-writecache?
- To what degree do ZVOLs benefit from ZIL-PMEM?
- What is the benefit of the ITXG bypass for ZVOLs?

Finally, in Section 7.3.4, we examine PRB's *commit slot* mechanism:

- Do *commit slots* achieve their goal to limit on-CPU waiting for PMEM I/O?
- What is the mechanism's performance overhead?
- What is the effect of PMEM bandwidth through interleaving of Optane DIMMs?

7.3.1 Setup & Reproducibility

We have already described our primary evaluation hardware in detail in Section 3.1. In summary: we configure the system such that it effectively becomes a single-socket system with sufficient DRAM, two 128 GiB Optane DIMMs, and three Micron 960GB NVMe SSDs. We use this system for all benchmarks except for Section 7.3.4 where we evaluate the *commit slot* abstraction on a different system with higher core count and more Optane DIMMs.

To ensure reproducibility, we automate our performance evaluation using a custom benchmarking harness written in Python. It provides the following functionality:

Declarative PMEM Provisioning Declarative definition of the desired PMEM configuration (interleaving / regions, namespaces).

Hardware Resource Registry All hardware resources, e.g., NVMe partitions and PMEM namespaces, are registered in a global registry object by a label.

Unified Storage Software Configuration We abstract the setup and teardown of software storage systems such as ZFS or *dm-writecache* as Python *context manager* objects. The objects use the hardware resource registry to discover the storage devices. This decoupling enables portability between the two evaluation systems.

Unified Benchmark Abstraction We abstract all benchmarks presented in this section as Python objects with a unified interface for benchmark execution.

Pinned Software Versions We build benchmarks from pinned source code versions (git submodules) or use pinned binary versions.

Still need to do this

Result Storage For each benchmark invocation, we serialize a description of the storage stack, benchmark parameters, and results into JSON objects and store these objects as unique files in the filesystem. Results are grouped by a prefix for post-processing, e.g., files with prefix `app_benchmarks__v4` contain the results of the fourth iteration of our application benchmark design.

We also automate post-processing of the results using the popular Pandas framework and render plots using Matplotlib and/or Seaborn.

7.3.2 4k Synchronous Random Write Workload

The first pillar of our performance evaluation is the *fio* 4k synchronous random write workload that motivated the design of ZIL-PMEM. We use the same zpool layout as for ZIL-LWB (30 striped NVMe partitions & 1 non-interleaved Optane DIMM region / `fsdax` namespace as SLOG). We configure PRB with *ncommitters=3*, which is the most CPU-efficient settings for a single Optane DIMM as we will show in Section 7.3.4.

Figure 7.2 shows the cummulative IOPS and per-IOP average latency measured by fio for 1–8 writer threads ("`numjobs`"). With a single thread, ZIL-PMEM achieves 128k IOPS which is a speedup of 8 over ZIL-LWB (16k IOPS). ZIL-PMEM scales almost linearly to 400k IOPS at four threads where the speedup over ZIL-LWB is still 5.5x. Throughput does not increase further for higher thread counts. The constant offset to the *fsdax* curve for `numjobs` 4–7 suggests that performance is limited by PMEM write bandwidth. (Remember that *fsdax* shows the raw /dev/pmem block device's performance under the same workload.) The slight decline to 346k IOPS at eight threads correlates with a similar decline in the *async* curve

but not the *fsdax* curve, suggesting a CPU bottleneck (8 cores per socket) or scalability bottleneck in ZFS.

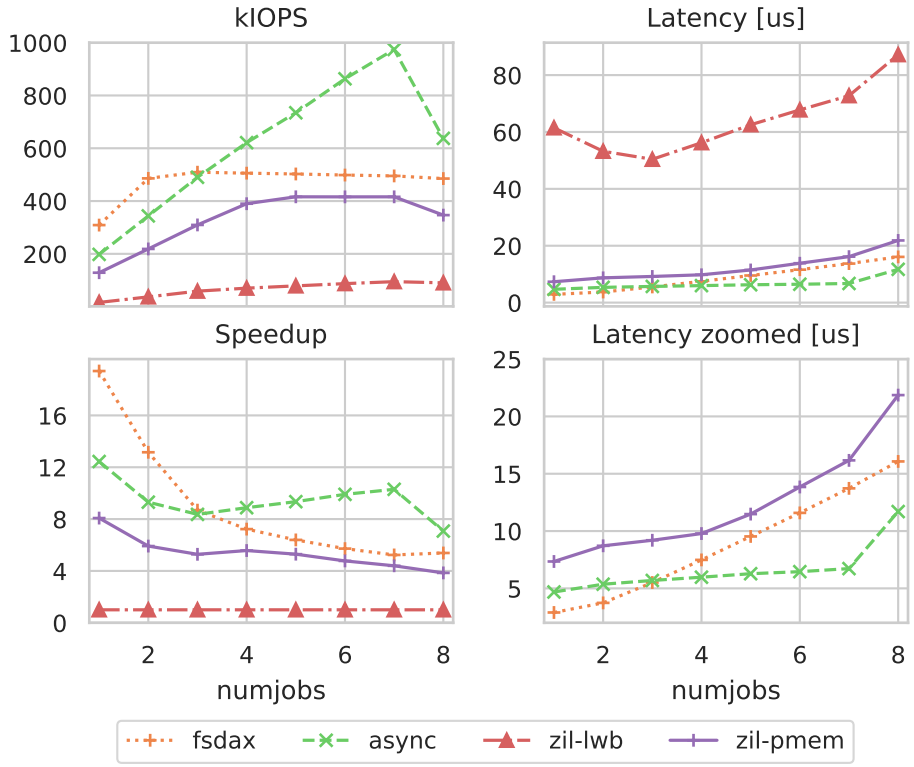


Figure 7.2: Mean IOPS and latency measured by *fio* for our 4k synchronous random write workload, by number of writer threads ("`numjobs`").

We observed that the speedup for `numjobs=1` and `2` varies significantly between benchmark runs but remains stable for higher values of `numjobs`. We investigate this issue by computing the coefficient of variation (CoV) of the different configurations based on the mean and standard deviation of IOPS reported by fio ($\frac{stddev}{mean}$). The results are displayed in Figure 7.3: The *zil-pmem* CoV remains close to the CoV of *async* until `numjobs=4` from where it starts to decline towards the CoV of *fsdax*. This phenomenon correlates with the stop of increase in IOPS at `numjobs=4`, supporting our assumption that *zil-pmem*'s behavior is dominated by the PMEM hardware for numjobs 4–7. In contrast, *zil-lwb*'s CoV is 5x that of *zil-pmem* for `numjobs=1` and 2x for `numjobs=2` before it starts to align closely with *async*. In absolute terms, for `numjobs=1`, the standard deviation for ZIL-LWB is 4.7k IOPS with a mean value of merely 16k IOPS. In contrast, for ZIL-PMEM, the standard deviation is 7.5k IOPS at a mean of 128k IOPS.

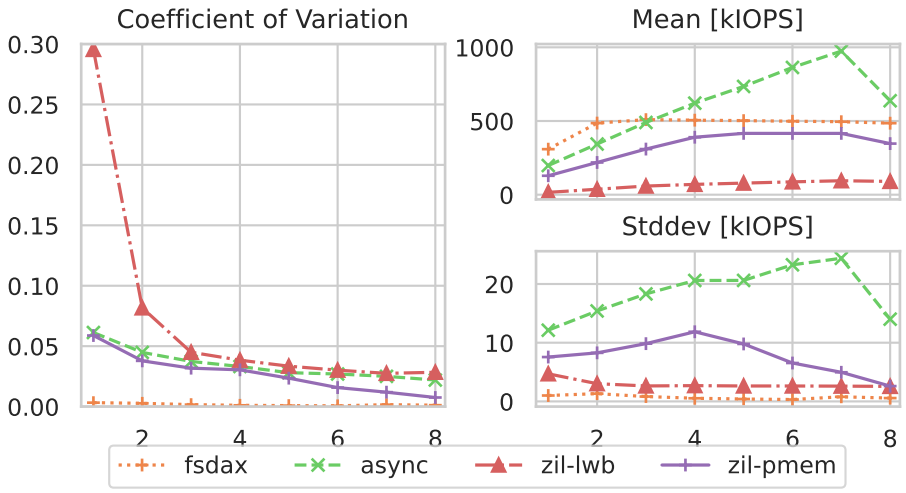


Figure 7.3: Comparison of the coefficient of variation ($\frac{stddev}{mean}$). The smaller plots display the absolute values.

To understand ZIL-PMEM's impact on tail latencies, we compare the 5th, 95th, 99.9th, 99.9th, 99.99th and 99.99th's latency percentiles reported by fio. *zil-pmem* follows *async* with a near-constant offset for all percentiles until `numjobs=4` where we observe the characteristical "knee in the curve" which is to be expected given that the system's peak IOPS are first achieved at this value of `numjobs`. In contrast, *zil-lwb*'s base latency (5th percentile, `numjobs=1`) is $\frac{36.6}{6.7}[\frac{us}{us}] = 5.46$ times higher than *zil-pmem*'s and the higher percentiles show a steeper growth with rising `numjobs`. Note that, by definition, the tail latencies for high percentiles at higher `numjobs` only have proportional and thus minor impact on the CoV.

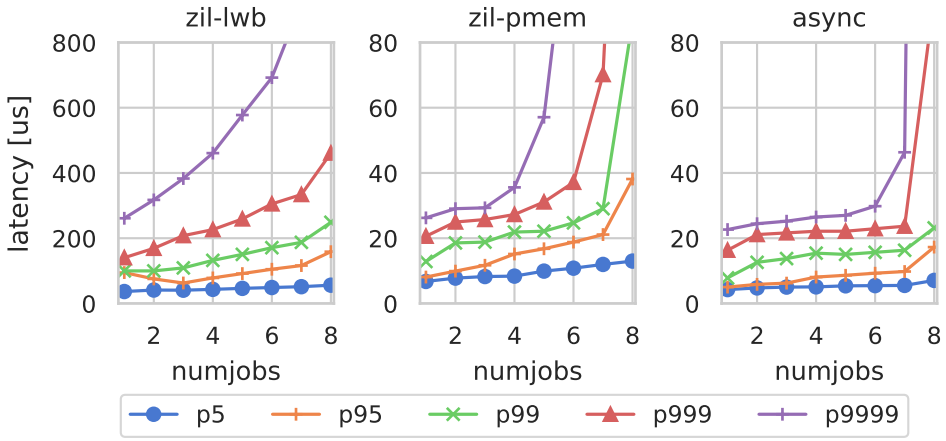


Figure 7.4: Comparison of completion latencies by percentile. Note the different y-axis scales.

To get an idea of the distribution of per-IOP latency in ZIL-PMEM, we generalize our eBPF instrumentation from Section 3.3.1 to support both ZIL-PMEM. We compare the results in Figure 7.5. The `zil_persistence` component supersedes the ZIL-LWB specific ones: it is defined as the overall time spent in `zil_commit` minus the time spent filling the commit list.

Whereas `zil_persistence` is the dominant factor for latency in ZIL-LWB (80% of average per-IOP latency, as observed in Section 3.3.1), ZIL-PMEM only spends approximately 25% on persistence to PMEM. For ZIL-PMEM, the dominant component with more than 50% is the *async* code, i.e, VFS-level work and modification of in-DRAM DMU objects. And in contrast to ZIL-LWB, the shared ITX code is a noticable component at 10%. The absolute *async* overhead for ZIL-PMEM is lower than for ZIL-LWB, particularly for small `numjobs` values. We have no satisfying explanation for this behavior.

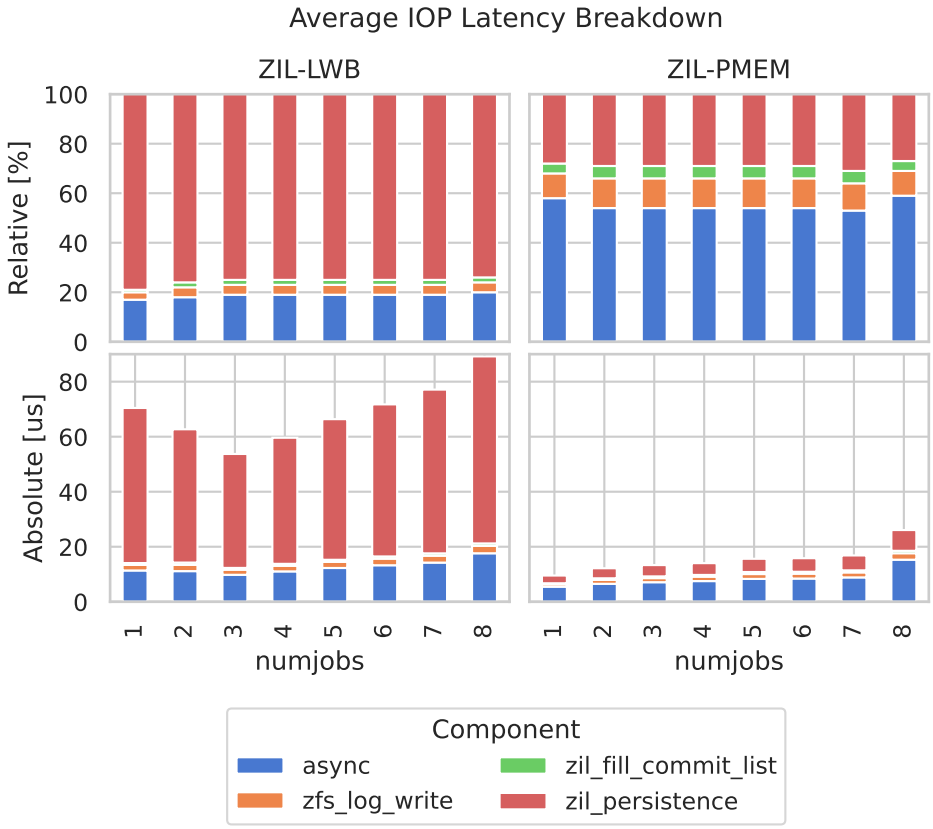


Figure 7.5: Comparison of relative and absolute latency contribution of functions executed by fio threads when they perform synchronous writes.

7.3.3 Application Benchmarks

We determine ZIL-PMEM's impact on real-world application performance through additional macro-level and application-level benchmarks. We execute each benchmark on top of different *storage stacks* that represent alternatives to ZIL-PMEM and compare the results in the subsequent sections.

We define a *storage stack* as a configuration of storage hardware and software that exposes a filesystem at a mountpoint M. Executing a benchmark *on top of* a storage stack means that the benchmark is configured to place all of its data within the filesystem mounted at M. We define the following storage stacks.

zfs-{lwb,pmem,async} A single ZFS dataset created on a zpool with the familiar hardware configuration (30 striped NVMe partitions and /dev/pmem0 as SLOG). For *zfs-lwb* and *zfs-pmem*, we configure the corresponding ZIL kind. For *zfs-async*, we configure the ZIL-LWB ZIL kind but set the `sync=disabled` property. All variants use a `recordsize=4k` and `compression=off` on the dataset since most of our benchmarks are synchronous workloads with small write sizes. For the *zil-pmem* variant, we configure PRB with `ncommitters=3`.

{xfs,ext4}{,-dax} on $BDEV Linux 5.9's *xfs* or *ext4* filesystems deployed on the *block device stack* `$BDEV`. If the "*-dax*" suffix is present, the `dax` mount option was set. We always perform benchmark runs without DAX, and perform an additional run if `$BDEV` is a DAX-capable device.

The *xfs* and *ext4* stacks are parametrized by the *block device stack* `$BDEV` which we define as a hardware and software configuration that provides a (potentially virtual) block device. We define the following block device stacks:

devpmem The raw `fsdax` namespace block device in devfs (/dev/pmem0).

dm-writecache The *dm-writecache* Linux Device Mapper target synchronously persists writes to PMEM and performs asynchronous write-back to an origin block device in the background. We configure *dm-writecache* with /dev/pmem0 as a cache device and a *dm-stripe* of the 30 NVMe partitions as the origin data store. Since most benchmarks consume relatively little space compared to /dev/pmem0's capacity (40GiB), we configure dm-writecache's *low watermark* to 0% and *high watermark* to 1% in order to trigger some write-back during benchmark execution. (See Section 2.1.2 for details on *dm-writecache*.)

zvol-lwb,rs={0,1} A ZVOL exposed by a zpool in the same config as *zfs-lwb*. We set `volblocksize=4k` on the ZVOL dataset. The *rs* variable controls the value of the `zvol_request_sync` tunable which, if enabled, processes block I/O requests (`struct bio`) synchronously instead of submit-

ting them to a thread pool. (Refer to Section 6.3.5 for more details on ZVOLs.)

zvol-async Like *zvol-lwb*, but with `sync=disabled`.

zvol-pmem,rs={0,1},byp={0,1} Like *zvol-lwb*, but configured with the ZIL-PMEM ZIL kind. PRB's `ncommitters` tunable is always set to 3. The *byp* variable controls whether the ITXG bypass is enabled (0 disabled, 1 enabled).

The following items are examples for storage stacks used in the evaluation:

ext4 on zvol-pmem,rs=1,byp=0 The Ext4 filesystem deployed on a ZVOL on a zpool with the ZIL-PMEM ZIL kind. The zpool is configured with `zvol_request_sync=1`, disabled ITXG bypass, and `ncommitters=3`. The ext4 file system is mounted without the `dax` mount option.

xfs-dax on devpmem The XFS filesystem, deployed directly on the PMEM block device (/dev/pmem0). It is mounted with the `dax` option.

zfs-pmem ZFS with the ZIL-PMEM ZIL kind and the default value of `ncommitters=3`. `Zvol_request_sync` and the ITXG bypass are only relevant for ZVOLs and therefore are not part of the configuration name.

We compare the performance of the storage stacks with the set of benchmarks described below. To facilitate the comparison, all benchmarks have the following properties:

Scaling Factor The scaling factor increases the degree of concurrent synchronous I/O operations issued by the benchmark. We run each benchmark with three scaling factor values: 1, 4, and 8.

Result Metric Each benchmark reports a single result metric per run. This allows us to compare performance across different storage stacks and scaling factor values. For all benchmarks in this evaluation, a numerically greater result is better.

What follows is a description of the benchmarks:

fio-growing The 4k synchronous random write workload that motivated our work, but with all worker thread files within the same filesystem. The scaling factor maps to the `numjobs` parameter which controls the number of worker threads. The working set (amount of modified data) grows with the scaling factor because each additional thread adds a private file with a constant size of 100 MiB. We report mean IOPS as the result metric.

fio-fixed The 4k synchronous random write workload, with all files on a single filesystem, but a fixed working set size, independent of `numjobs`.

We achieve the fixed working set size by setting the `size` parameter to $\frac{1\text{ GiB}}{numjobs}$. We report mean IOPS as the result metric.

filebench varmail Filebench is a benchmarking engine that executes arbitrary workloads described in a domain-specific language. The predefined *varmail* workload "emulates I/O activity of a simple mail server that stores each e-mail in a separate file (/var/mail/ server). The workload consists of a multi-threaded set of create-append-sync, read-append-sync, read and delete operations in a single directory. [...] The workload generated is somewhat similar to Postmark but multi-threaded" []. We use filebench in version `1.5-alpha1-33-g22620e6` with the upstream workload definitions. The scaling factor maps to the workload's `$nthreads` variable which determines the number threads that execute the operations described above. We use a runtime of 20 seconds per configuration and report filebench's *ops per second* value as the result metric.

filebench oltp The filebench *oltp* workload emulates database workloads. It "performs file system operations using Oracle 9i I/O model. It tests the performance of small random reads and writes, and is sensitive to the latency of moderate size (128k+) synchronous writes to the log file. By default [it] launches 200 reader processes, 10 processes for asynchronous writing, and a log writer." []. We leave all parameters at the default setting except for the `$ndbwriters` parameter for which we use the scaling factor's value instead of its default value of ten. It is our understanding that scaling the number of log writer threads to a number greater than one would not be feasible with the real DBMS and thus be unrealistic. We use a runtime of 20 seconds and report filebench's *ops per second* value as the result metric.

MariaDB/sysbench We deploy the MariaDB 10.5.9 Docker image in its default configuration (InnoDB storage engine) with the /var/lib/mysql directory bind-mounted to a sub-directory within the storage stack's mountpoint. We apply the *sysbench* benchmark's *oltp_insert* workload with its default parameters for ten seconds. The workload spawns a number of threads that inserts rows with random data into one or more tables. We use the default setting (a single table) and map the scaling factor to the number of threads that perform the insert queries. Each thread uses a private, long-lived connection to the MariaDB server. We use Docker's `--net=host` parameter when starting the MariaDB container to allow sysbench to connect via loopback TCP, avoiding the overhead of Docker's user-space proxy. The result metric is the number of transactions per second (tps) reported by sysbench.

Redis-SET Redis is a popular in-memory key value store. For persistence, it provides two mechanisms that are recommended to be used in com-

bination. First, the system periodically persists a snapshot of the Redis database (RDB). This process happens in the background in a forked child process. Second, Redis features a logical write-ahead log (*append-only file*, AOF) that is extended for every mutating operation and replayed after a crash. Redis supports three different behaviors for ensuring durability of the AOF, configurable through the `appendfsync` configuration variable. A value of `no` does not perform any fsyncs, `everysec` performs an fsync every second (default), and `always` performs an fsync operation for operation logged to AOF. []

We deploy Redis 6.2 (built from source). We configure RDB writeback to happen every second, regardless of the number of changes, We enable AOF and configure `appendfsync=always` which effectively turns our Redis deployment into a durable key-value store.

For benchmarking, we use Redis's own *redis-benchmark* tool []. The scaling factor maps to the `--threads` and `-c` parameters which control the number of parallel clients. We use the benchmark's *SET* workload to perform 10^6 *SET* operations with random keys and the default value size (3 bytes). We configure a keyspace size of 10^6 to reduce contention. We leave the values for pipelining and connection keepalive at their defaults (no pipelining, keepalive enabled). The result metric is the requests-per-second (rps) value reported by redis-benchmark.

We choose the number of 10^6 *SET* operations because it results in runtimes of ten seconds or more for most combinations of scaling factor and storage stacks. The only exception is *zfs-async* for which the shortest runtime is 8 s.

RocksDB-fillsync RocksDB is a popular key-value store developed by Facebook that is optimized for fast storage devices. It can be used directly by applications as an embedded database but is also the basis for the *MyRocks* storage engine for MySQL []. RocksDB uses log-structured merge trees for long-term storage which are written and rewritten in large units that are prepared in DRAM (memtables). Operations that request synchronous semantics using the `WriteOptions.sync` flag are logged to a write-ahead log file. [;]

We measure the impact of ZIL-PMEM on RocksDB WAL performance with the *fillsync* benchmark that is part of RocksDB's *db_bench* tool. *Fillsync* performs a fixed number of synchronous *Put* operations with random keys. We set the number of operations to 400k and map the scaling factor to the number of concurrently *Put*ting threads. The result metric is the *ops-per-sec* (operations per second) value reported by *db_bench*.

We choose the number of 400k operations because it results in at least ten seconds of runtime for 90% of configurations. The configurations that achieve less than ten seconds of runtime are:

- All *zfs-pmem* configurations at scaling factor 1 (∼4 s runtime)
- *zfs-async* for scaling factors 1, 4, 8 (∼2, 6.8 and 8 s rt.)
- *xfs-on-devpmem* with dax mount option at scaling factor 1 (9.1 s rt.)

ZIL-PMEM vs. ZIL-LWB

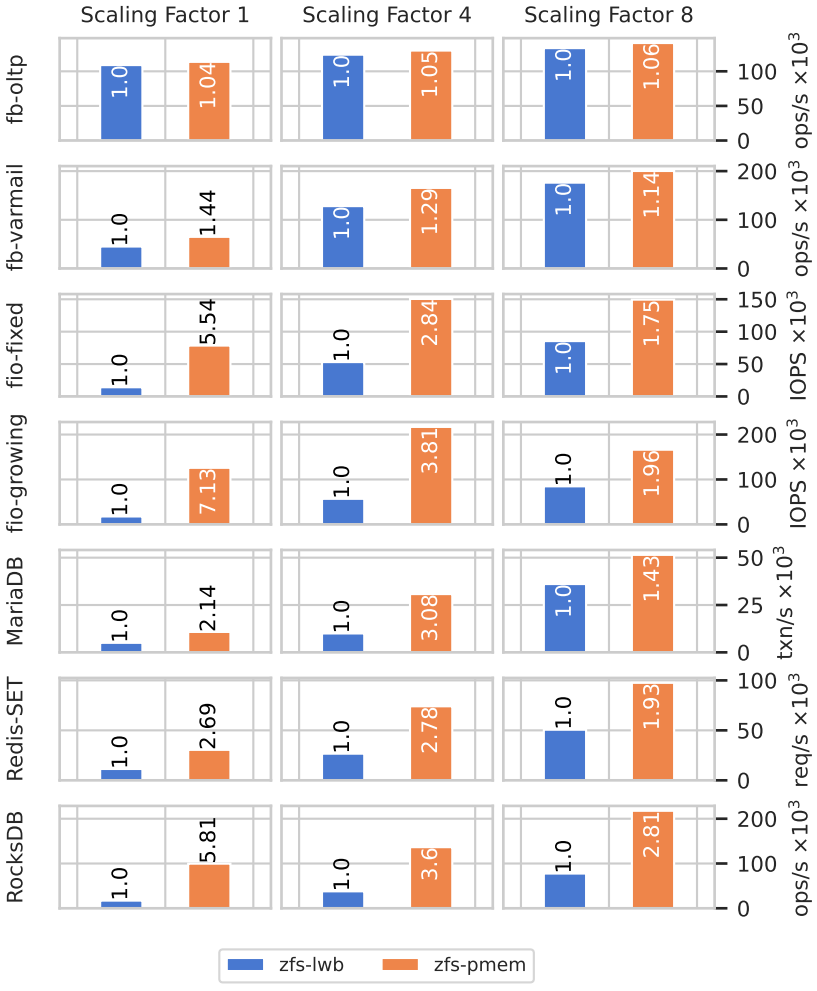


Figure 7.6: App Benchmarks: ZIL-PMEM vs. ZIL-LWB. The number on each bar indicates the speedup over *zfs-lwb*.

We first compare the speedup of ZIL-PMEM over ZIL-LWB. ZIL-PMEM outperforms ZIL-LWB in all benchmarks, but the speedup varies significantly for the

different workloads and scaling factors. The highest speedups are achieved at scaling factor 1 by the fio workloads (7.13x, 5.54x) and *RocksDB-fillsync* (100k ops-per-sec, a speedup of 5.8). *Redis-SET* achieves shows a speedup of 2.69 (30k rps) and MariaDB achieves 10k tps which is a speedup of 2.14. For scaling factor 4, the fio and RocksDB's speedup declines whereas Redis's grows to 2.78 and MariaDB's grows to 3.08. For scaling factor 8, all workloads show a reduction in speedup (RocksDB 2.81x, Redis 1.93x, MariaDB 1.43x).

The *filebench-oltp* workload shows no relevant speedup or slowdown for all scaling factor values. We observed during benchmark execution that the 200 reader processes always cause 100% CPU utilization.

The *filebench-varmail* workload only shows a speedup of 1.43 at scaling factor 1 which shrinks to 1.14 at scaling factor 8. A possible explanation for these meager results is the larger data volume (16k appends) compared to the fio workloads (4k writes). In combination with the large amount of metadata (file creation and deletion), ZIL-LWB might also be benefitting from the *commit ITXs'* pipelininig. In any way, *filebench-varmail* shows that the large speedups for small writes cannot be generalized to all filesystem workloads. Future work should investigate ZIL-PMEM's behavior during this benchmark in detail and determine how beneficial pipelining akin to *commit ITXs* could be.

ZIL-PMEM vs. XFS and Ext4 on PMEM

Our next set of results is the comparison between the Linux filesystems XFS and Ext4 and ZFS with both ZIL kinds (*zfs-lwb*, *zfs-pmem*). We also include *zfs-async* as a theoretical upper bound for any ZIL kind. Since both XFS and Ext4 are DAX-aware, we inlcude configurations with and without the *dax* mount option. Note that that the ZFS configurations only use PMEM for the ZIL but NVMe devices for permanent storage whereas the Linux filesystem configurations are PMEM-only.

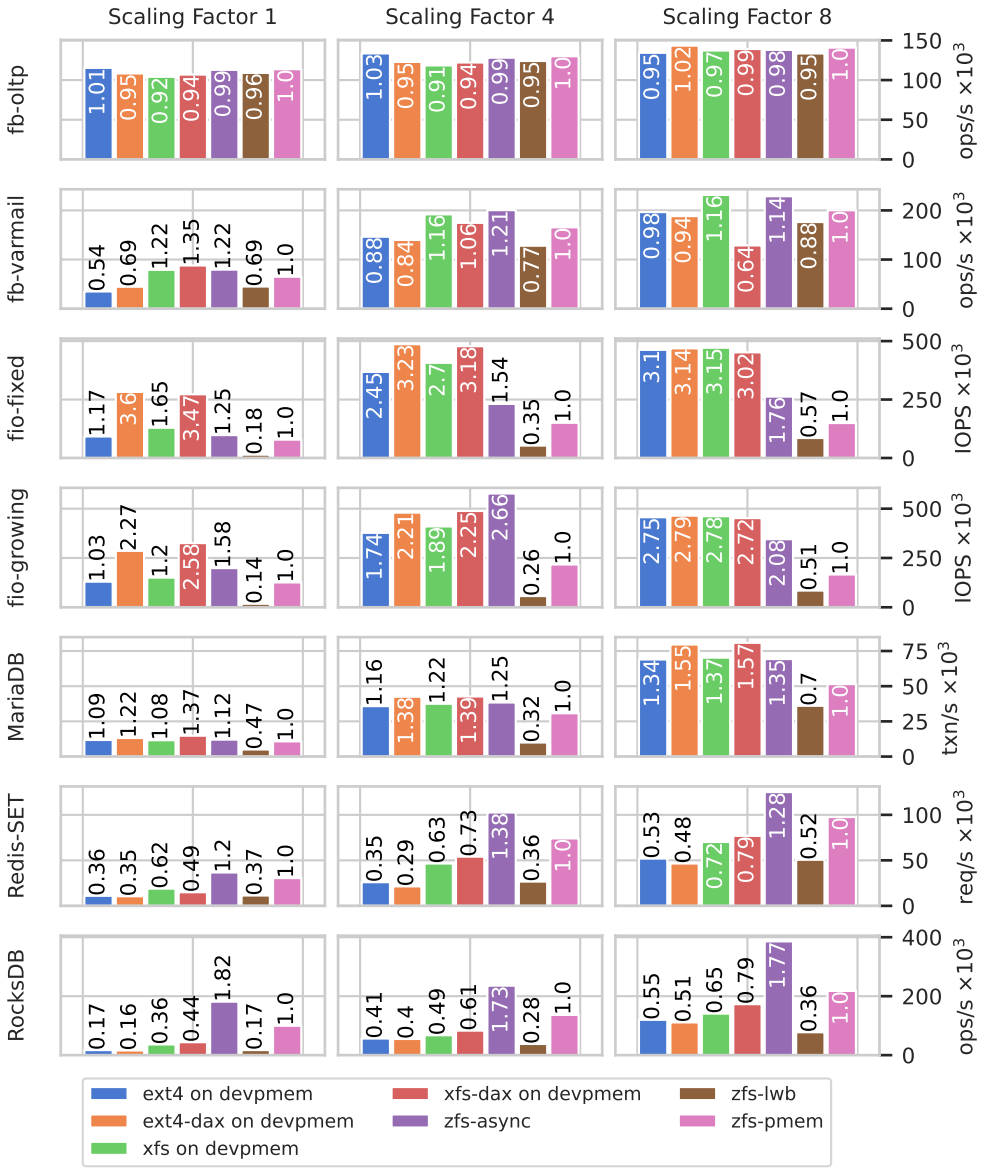


Figure 7.7: ZFS with ZIL-PMEM, ZIL-LWB and `sync=disabled` (*zfs-async*) as well as ext4 and xfs on /dev/pmem, with and without the *dax* mount option. The number on each bar indicates the speedup over *zfs-pmem* in the respective benchmark & scaling factor.

Our primary takeaway from this comparison is that, in contrast to ZIL-LWB, ZIL-PMEM is competitive with both ext4 and xfs at scaling factor 1 in most benchmarks. The fio workloads are the exception, where the Linux filesystems achieve a speedup of greater than 3 if the *dax* mount option is enabled. The reason for

this phenomenon is most likely lower CPU overhead at scaling factor 1 (compare *zfs-pmem* with *zfs-async*), and better multi-core scalability at higher scaling factors. Note that, since the benchmarks in this section all write to a single dataset, ZIL-PMEM's scalability is bounded by the sequential logical structure of the ZIL.

ZIL-PMEM performs exceptionally great for the *Redis-SET* and *RocksDB-fillsync* workloads at all scaling factors. The write-ahead log files written that these workloads write are consists of encoded representation of logical changes. The mutating requests issued by our benchmarks have small payloads and result in small `write+fsync` system calls.[1] We assume that ZIL-PMEM handles this IO pattern more efficiently than the Linux filesystems. First, in non-DAX mode, every small append + fsync operation on the Linux filesystems is necessarily amplified to block granularity, both for the data that is modified in place and for metadata that is additonally journaled. In contrast, ZIL-PMEM's PMEM write volume per append+fsync is most likely smaller. For example, a PRB entry for a small write system call only consumes $256 + 256 \times \lceil \frac{192+size}{256} \rceil$ bytes of PMEM. Second, even in DAX-aware mode, the implementations of Ext4 and XFS still assume block devices as the storage medium for their JBD2 journal (Ext4) or write-ahead log (XFS). Thus, both write amplification and unnecessary disk-oriented optimizations might be relevant here.

In the *MariaDB* benchmark, the DAX-aware Linux filesystem configurations achieve up to 1.57x more transactions per second than ZIL-PMEM. *Zfs-async* and the the non-DAX-aware configurations achieve 1.12x, 1.25x and 1.35x higher results at scaling factors 1, 4, and 8. Manual inspection of the CPU utilization shows a signficant amount of context switching but no clearly identifiable CPU bottleneck. The database server process performs 1024 byte sized `write + fdatasync` system calls to the InnoDB redo log file from a single thread. However, since *zfs-async* does not perform better than any of the systems that actually write to PMEM, we can rule out PMEM bandwidth as a possible bottleneck.

Regarding the *filebench-oltp* benchmark, no configuration performs exceptionally better than any other, including *zfs-async*. We observe high CPU utilization in all benchmark configurations, caused by the "hog" stage of the 200 processes that the benchmark uses to simulate reader threads. Thus, CPU time is most likely the bottleneck in all configurations.

[1] The RocksDB documentation and comments in the source code suggest that the WAL files are written in 32k blocks that are zero-padded if necessary [, db/log_writer.h]. We observed the contrary behavior with `strace`: each *Put* operation with `WriteOptions.sync` results in a `write` and `fdatasync` system call. The `write` system call's data size is proportional to the sum of the *Put* operation's key and value sizes.

For *filebench-varmail*, the highest speedup over ZIL-PMEM is 1.35 at scaling factor 1, achieved by XFS in DAX-aware mode, followed by ZFS with *zfs-async* with 1.22x. For higher scaling factors, XFS's speedup diminishes and even goes below 1 at scaling factor 8, albeit only in DAX-aware mode. We have no satisfying explanation for this behavior. Also, the fact that *zfs-async* achieves a lower speedup than XFS can be taken as an indicator that *filebench-varmail* is not bound by PMEM performance, contradicting our speculation in the previous section.

ZIL-PMEM vs. XFS and Ext4 on Dm-writecache

The comparison between ZIL-PMEM with NVMe storage and Linux filesystems on pure PMEM in the previous section is insightful but not a level comparison in terms of storage hardware performance and available functionality. A more practically relevant competitor to ZIL-PMEM is *dm-writecache* with a *dm-stripe* of NVMe partitions as the origin block device. In this section, we compare ZFS with ZIL-PMEM and ZIL-LWB to Ext4 and XFS on such a setup.

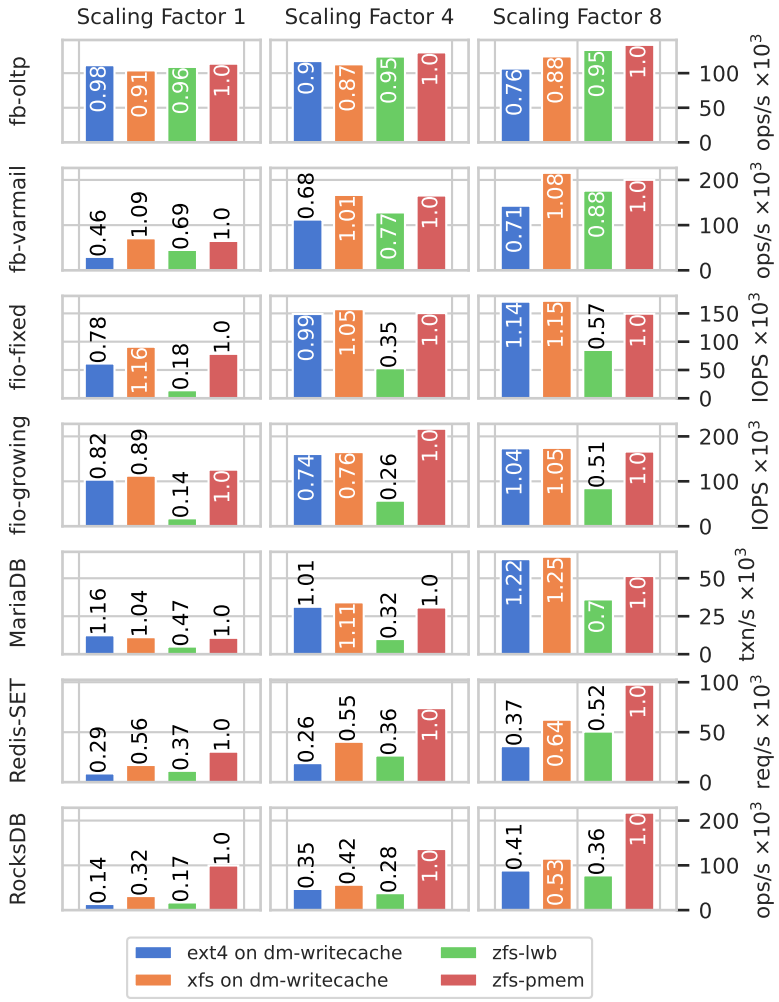


Figure 7.8: Comparison of the benchmark results for ZIL-PMEM and ZIL-LWB vs. Linux filesystems on dm-writecache. The number on each bar indicates the speedup over *zfs-pmem* in the respective benchmark & scaling factor.

ZIL-PMEM's performance is within +/-30% of XFS on *dm-writecache* at all scaling factors in all benchmarks except *Redis-SET* and *rocksdb-fillsync* where ZIL-PMEM performs significantly better. The 30% spread decreases slightly with higher scaling factors but does not change the overall picture. Our explanation for ZIL-PMEM's significantly better performance in the *Redis-SET* and *rocksdb-fillsync* workloads is (again) write amplification, possibly combined with the sub-optimal scaling behavior of dm-writecache that we describe below.

Ext4 performs signficantly worse than ZIL-PMEM (and XFS!) in some benchmarks, with only 46%, 29%, and 14% of ZIL-PMEM's performance in *filebench-*

varmail, *Redis-SET*, and *rocksdb-fillsync* at scaling factor 1. Write amplification might be a possible explanation as well.

We have manually observed the system while benchmarking the *dm-writecache* stack to ensure that the comparison with ZFS is actually fair. We found that, despite the very aggressive write-back configuration (high watermark = 1%, low watermark = 0%), the system performed very little write-back to the NVMe drives. The two *fio* workloads were the exception but the write load towards the NVMe devices was well below their maximum capacity. Thus, NVMe was not a bottleneck in the benchmark. However, we found that *dm-writecache* has severe multi-core scalability problems. For example, XFS on raw PMEM without the *dax* mount option achieves 150k IOPS at scaling factor 1 and 409k IOPS at scaling factor 4 in the *fio-growing* workload. In contrast, on dm-writecache, it is 112k IOPS at scaling factor 1 but only 165k IOPS at scaling factor 4. Using the *perf* tool, we observed severe contention at a lock that serializes access to the entire dm-writecache instance's state. In private communication, dm-writecache's maintainer Mikulas Patocka stated that "the purpose of dm-writecache is to decrease commit latency, not to increase throughput. There is not much that can be done with the lock contention". ZIL-PMEM has similar problems in *fio-growing* (125k IOPS at factor 1, 217k at factor 4) albeit due to the sequential structure of ZIL chain, not the scalability of PRB/HDL. (We know from Section 7.3.2 that we can achieve 400k IOPS with `numjobs` / scaling factor 4 if each writer thread operates on a seprate dataset. However, remember that for comparability with the Linux filesystems, the *fio-fixed* and *fio-growing* workloads operate on a single ZFS dataset when executed on a zfs- stack.) In the next section, we investigate whether a more relaxed log structure such as the ITXG bypass can improve this shortcoming of the ZIL structure.

Impact on ZVOL Performance & ITXG Bypass

In this section, we examine the impact of ZIL-PMEM on ZVOLs. We compare the performance of XFS deployed on a ZVOL in a zpool with varying ZIL kinds, `zvol_request_sync` (`rs_{0,1}`), and ITXG bypass setting (`byp_{0,1}`). The results are relative to *zvol-pmem,rs=0,byp=0* as the baseline which is the standard configuration for ZIL-PMEM pools. Note that deploying a filesystem on a ZVOL generally makes little sense in practice because ZFS filesystems provide the same basic service with more additional features at less overhead. However, ZVOLs are a popular choice for storage virtualization where ZVOLs are used as virtual hard disks, either on the same host or via a SAN (e.g., iSCSI, Fibre Channel). To simplify our evaluation, we avoid the overhead of a hypervisor or SAN and instead

instantiate XFS directly on top of the ZVOL, mount it, and execute the benchmarks.

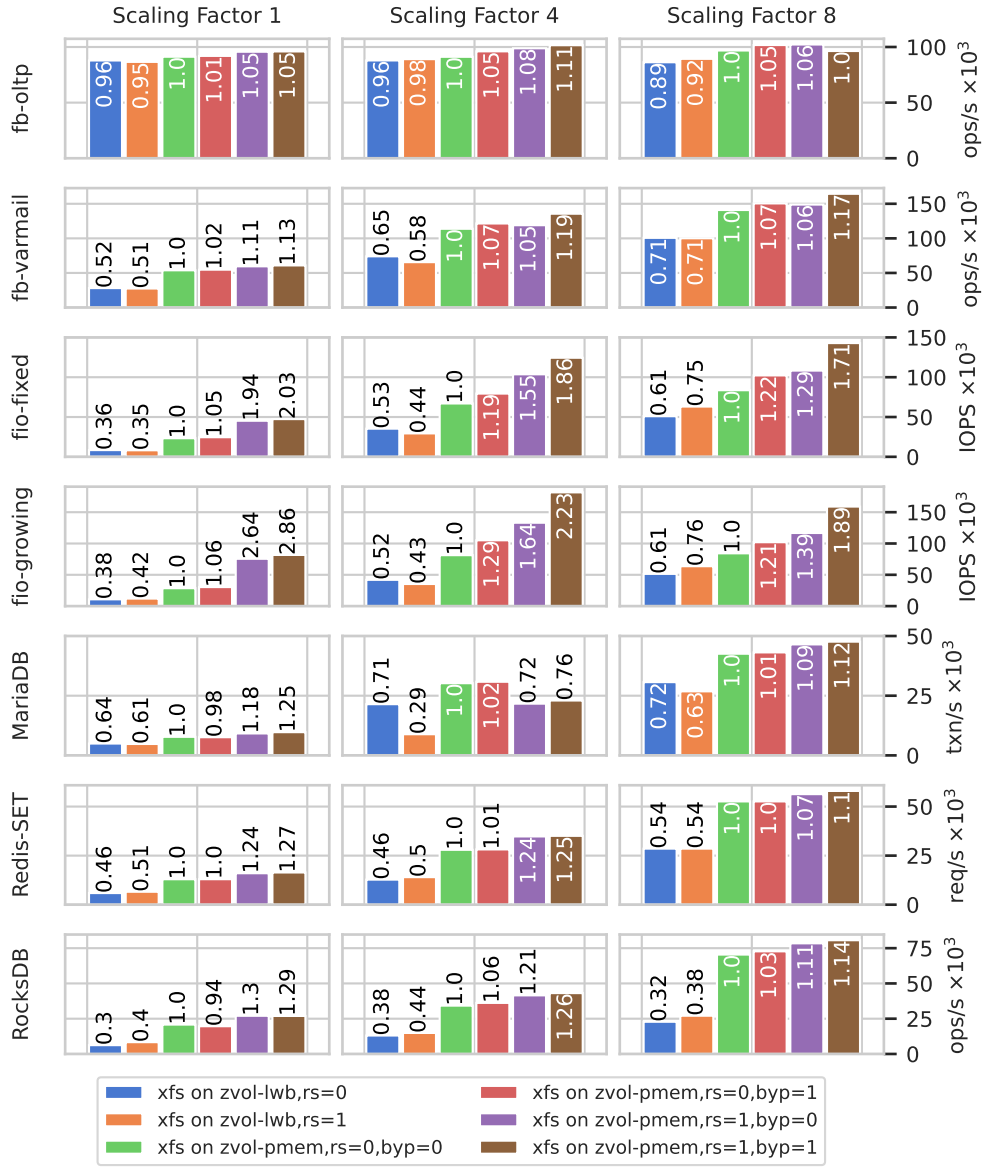


Figure 7.9: Comparison of the benchmark results for XFS on ZVOLs on different zpool configurations. The number on each bar indicates the speedup over ZIL-PMEM in the *zvol-pmem,rs=0,byp=0* configuration.

Our first observation is that the default configuration of ZIL-PMEM, *zvol-pmem,rs=0,byp=0*, delivers a significant speedup over ZIL-LWB. At scaling factor 1, it is approximately 2x for all workloads except *MariaDB* ($\frac{1}{0.64} = 1.56$) and *filebench-oltp*

($\frac{1}{0.96} = 1.04$). At higher scaling factors, the speedup declines for all benchmarks albeit less so with *Redis-SET* and *RocksDB-fillsync* than with the other workloads.

The effect of `zvol_request_sync=1` is ambiguous and not very signficant for ZIL-LWB but very beneficial for ZIL-PMEM in the *fio* workloads at scaling factor 1 with a 2x and 2.6x speedups over the standard configuration, respectively. *MariaDB*, *Redis-Set* and *RocksDB-fillsync* also benefit with speedups of 1.18x, 1.23x, and 1.30x over the default configuration. However, for scaling factor 4, *MariaDB* performs substantially worse if `zvol_request_sync` is set (30k tps vs. 20k tps). We also conducted the benchmarks with Ext4 instead of XFS (not shown in the plot). Ext4 exhibited worse performance in almost all configurations if `zvol_request_sync` was set, with the exception of the *fio* workloads.

The ITXG bypass only has marginal effects with and without `zvol_request_sync`, except for the *fio* workloads at scaling factors 1 and 4. For example, *fio-growing* at scaling factor 8 shows an increase in IOPS from 84k to 102k (21%) for `zvol_request_sync=0` and from 117k to 159k (35%) with `zvol_request_sync=1`. For Ext4, *filebench-varmail* also shows a 20% improvement with enabled ITXG bypass and `zvol_request_sync=0`. Activation of the ITXG bypass does not appear to have a negative impact for XFS in the remaining workloads. For Ext4, we observed some performance degradatations in setups whether both ITXG bypass and `zvol_request_sync` were enabled.

In summary, ZIL-PMEM provides a signficant performance advantage for ZVOLs but is less effective than with ZFS filesystems (see Sections 7.3.2 and 7.3.3). This is particularly noticable in the *fio* workloads and *RocksDB-fillsync*. Their speedup with ZIL-PMEM on XFS+ZVOL is only half the speedup of what we observed for ZFS at scaling factor 1. Write amplification cannot be responsible for this behavior because fio already writes at 4k block size. The most likely cause is latency overhead added by the filesystem that affects all workloads equally.

7.3.4 CPU-Efficient Handling Of PMEM Bandwidth Limits

PRB's *commit slot* mechanism limits the amount of parallel writers to PMEM to `ncommitters`. The goal is to avoid wasteful on-CPU stall cycles which inevitably occur if the aggregate write bandwidth to PMEM exceeds the hardware capacity. (See Sections 4.1.1 and 5.12 for a more detailed explanation.)

To determine the mechanism's effectiveness, we use our 4k synchronous random write workload with separate ZFS filesystems per fio thread. We add low-overhead per-CPU counters to ZIL-PMEM to sum up the total mount of time that is spent writing PMEM (T_{pmem}), as well as the number of write operations performed during the benchmark (N_{ops}). We then compute the *average PMEM write time per IOP* $T_{pmem_{iop}} = \frac{T_{pmem}}{N_{ops}}[\frac{s}{op}]$. The most efficient value for `ncommitters`

for a given system depends on the workload and efficiency goals. In general, `ncommitters` should be the minimal value where $T_{pmem_{iop}}$ is tolerable but the system's primary performance metric is not impacted.

We compute $T_{pmem_{iop}}$ for 1 to 18 numjobs, different values of `ncommitters` and two different PMEM configurations. The first PMEM configuration is a single, non-interleaved Optane DIMM. The second configuration are four interleaved Optane DIMMs. Regardless of how interleaving is configured, we create a 40 GiB-sized *fsdax* namespace on the resulting region and use it as the PMEM SLOG for ZIL-PMEM. We use a machine with higher core count for the benchmarks. The system configuration is as follows:

System Supermicro SYS-1029U-TRT
Mainboard Supermicro X11DPU, Version 1.10
CPU 2 x Intel(R) Xeon(R) Gold 5220 CPU 2.20GHz
18 cores per socket, 2-way SMT per core
DRAM 12 x 32GiB SK HYNIX DDR4-2666, HMA84GR7CJR4N-VK
PMEM 8 x Intel Optane DC Persistent Memory, 128 GB, (NMA1XXD128GPS),
4 per socket.
NVMe System rootfs only, no dedicated NVMe hardware.
Kernel Linux Kernel, Fedora 5.11.15-100.fc32.x86_64
Userland Fedora 32
fio fio-3.21

As with our main evaluation system (described in Section 3.1), we leave SMT enabled. We disable the second socket in software using the `isolcpus=18-35,54-71` kernel command line parameter. Since the system does not have NVMe devices available, we provision the disabled socket's Optane DIMMs in `AppDirectNotInterleaved` mode with one PMEM region and a single `fsdax` namespace per DIMM. We use these namespaces as the zpool's main (non-SLOG) vdevs.

Figure 7.10 visualizes the results of this experiment. For the non-interleaved configuration, we observe that the system's peak IOPS is 400k, first achieved with `ncommitters=3`. For `ncommitters=3`, IOPS quickly decline to 263k IOPS for higher $numjobs$. With `ncommitters=8`, the system is able to sustain the 400k IOPS for the for $numjobs \in 7 \ldots 13$. The price is significantly more on-CPU PMEM time per IOP: looking at $numjobs = 8$, the $T_{pmem_{iop}}$ is $3.3, 4.3, 6.5$ us for `ncommitters=3`, `4` and `8`. And at $numjobs = 12$, $T_{pmem_{iop}}$ climbs to $11.6, 20.6$ us for `ncommitters=8` and `12` whereas `ncommitters=3` is successfully limited to 2.7 us of PMEM write time per IOP. The `ncommitters=12` and `24` configurations do not achieve higher peak IOPS than `ncommitters=8` but decline to lower values for high $numjobs$, e.g., only 237k IOPS for `ncommitters=24`, $numjobs = 18$. This performance decline at high degrees of concurrency is an established property of the Intel Optane

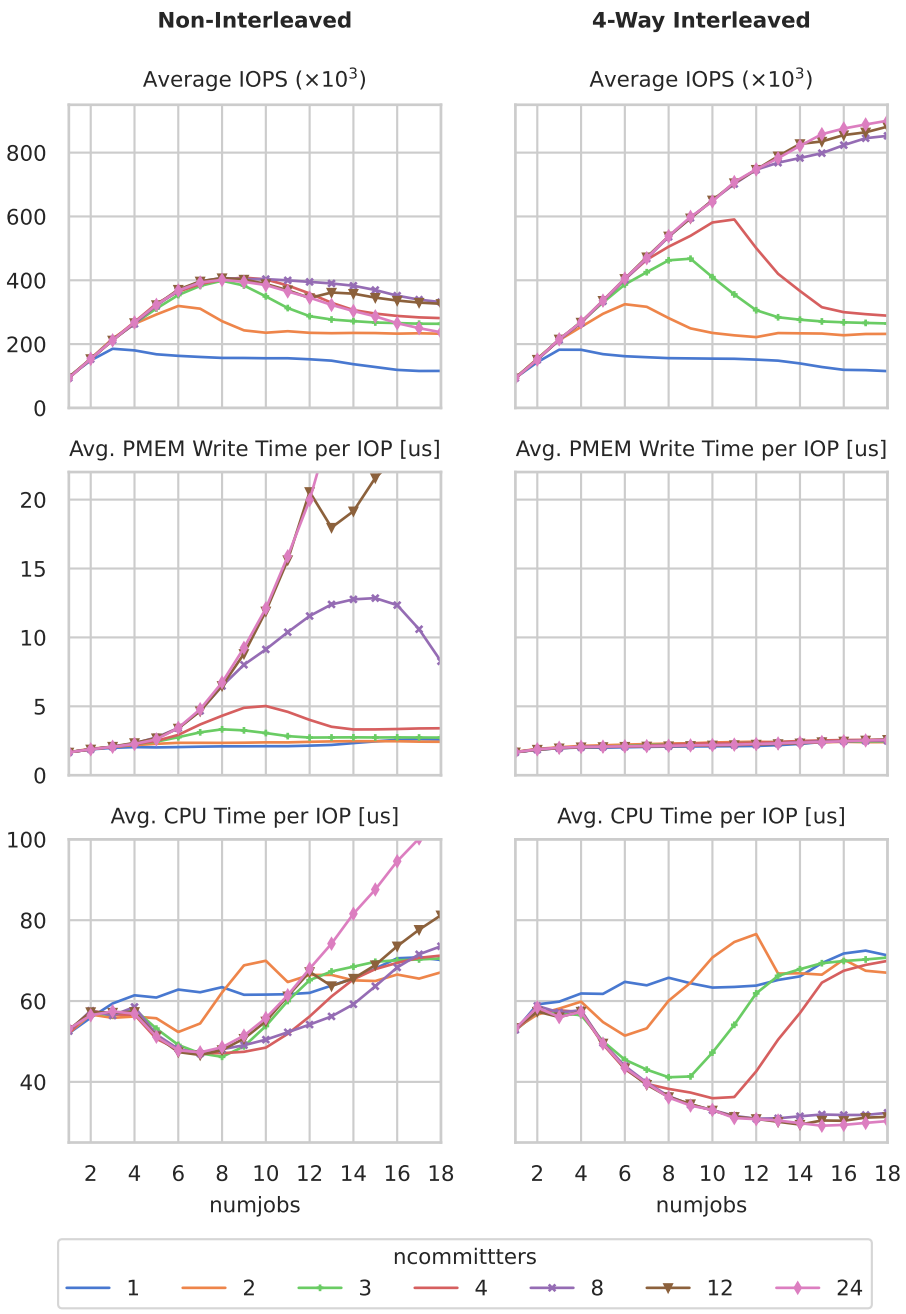


Figure 7.10: Comparison of performance and PMEM write time per IOPS for different values of `ncommitters` in the two PMEM configurations.

PMEM hardware [Yan+20]. The limitation to `ncommitters=8` mitigates this effect (331k vs. 237k IOPS at $numjobs = 18$).

The four-way interleaved configuration exhibits a very different behavior. We achieve the highest IOPS for the configuration where the *commit slot* mechanism has no effect, i.e., `ncommitters=24`, $numjobs = 18$ with 900k IOPS. Given that the IOPS curve has not reached a plateau at this point, a higher value might be possible with higher $numjobs$. $T_{pmem_{iop}}$ is the same for all `ncommitters` values at a given $numjobs$ value (+/- 0.1 us). It is 1.66 us for $numjobs = 1$ and grows slightly unevenly towards 2.56 us at $numjobs = 24$. This is an increase of merely 54% which is likely to be acceptable in any setup, given the 9.6x increase in IOPS (93k to 900k) that is possible with `ncommitters=24`.

To determine how the *commit slot* abstraction impacts performance in `ncommitters` $> \max numjobs$ configurations, we add additional instrumentation that measures the average latency of commit slot aquisition and release. Figure 7.11 visualizes the results for `ncommitters=24`. The absolute overhead is approximately the same for both PMEM comfigurations. It starts at 100 ns for $numjobs = 1$, climbs to 240 ns at $numjobs = 4$, and then scales linearly to 385 ns at $numjobs = 18$. In the interleaved configuration, for any $numjobs > 4$, this corresponds to approximately 11–13% of the overall write latency per log entry. Without *commit slots*, `zil_commit` could thus be $\frac{1}{1-0.13} = 14.9\%$ faster. However, due to the overhead of the other ZFS components, the contribution to overall IOP latency is only 2%, which would constitute a rather meager improvement.

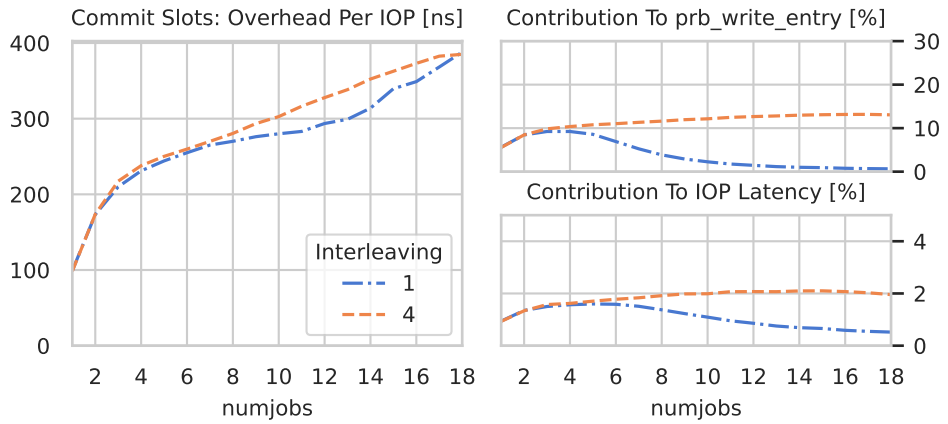


Figure 7.11: Overhead of the *commit slot* for 24 = `ncommitters` $> \max numjobs$. Note the different y-axis scales. Also note that the relative latency contribution for the non-interleaved configuration only shrinks because the PMEM write time (not shown) grows significantly.

We draw the following conclusions from our observations made above:

- The *commit slot* mechanism successfully limits PMEM write time.
- The *commit slots* mechanism's multi-core scalability is acceptable and contributes little relative overhead to each IOP compared to other ZFS components.
- Regarding the non-interleaved configuration (single Optane DIMM):
  - Our 4k synchronous write workload is able to achieve peak performance with 400k IOPS as early as `ncommitters=3`.
  - Setting `ncommitters=8` extends the peak to a plateau with regards to $numjobs$, but consumes $\frac{6.5 \text{ us}}{3.3 \text{ us}} = 1.96$ times more (on-CPU!) PMEM write time per IOP.
  - Note that we used `ncommitters=3` as the configuration for ZIL-PMEM in all of the previous benchmarks. Since our main evaluation system on which these benchmarks were executed only has 8 cores (2x SMT), this setting ensured CPU-efficient execution at all times. We believe that an efficiency-oriented default configuration is the preferrable approach for an in-kernel filesystem such as ZFS whcih provides a system-wide OS service whose overall priority in the system is not known a priori. However, for systems whose primary role is that of an NFS/SMB server or that of a SCSI target (ZVOLs), it may be appropriate to allow more on-CPU waiting time.
- Regarding the four-way interleaved configuration (4 Optane DIMMs):
  - The *commit slot* mechanism is not useful for the observed range of $numjobs$ values.
  - The reason is that ZIL-PMEM cannot saturate the Optane DIMMs' write bandwidth due to per-IOP overhead added by other components of ZFS.
  - However, with `ncommitters` $> \max numjobs$, the overhead per IOP is negligible.

7.4 Discussion

Storage stacks that we investigated but showed subpar perf:

- xfs with separate log
- ext4 with separate log

Future work:

Correctness:

- ensure + regression testing for PMEM memory model
- fault injection in ztest, equivalent to zio fault injection, should be easy
- fault injection through ndctl-inject for kernel module & ZTS: Although we force the replay callback to use the `prb_replay_read_replay_node` function which handles *machine check exceptions (MCE)* through `memcpy_mcsafe()`, we should somehow ensure that such errors are actually handled.

Performance:

- proof our write amplification claims (measure number of bytes of PMEM written as well as pmem write time, for all configurations. Needs instrumentation of pmem block device emulation.)
- ext4/xfs data journaling mode (ext4 has it, xfs not sure)
- xfs pmem-aware log (Christoph Hellwig has patches...)
- Scalability comparison of ZIL-LWB and ZIL-PMEM within the same dataset. Expectation: ZIL-PMEM's mutex eventually becomes the bottleneck, but likely still better than ZIL-LWB due to massive latency headroom.

Chapter 8

Summary

8.1 Future Work

Evaluation Of Replay Performance

Evaluation of Chunk Size

Fuzzing

ZTS run entire ZTS, not just ZIL-related tests

Appendix

Bibliography

[] *[Ndctl PATCH v2 3/6] Ndctl: Add an Inject-Error Command - Linux-Nvdimm - Ml01.01.Org*. URL: https://lists.syncevolution.org/hyperkitty/list/linux-nvdimm@lists.01.org/message/UAEXUT4RBYFUFTXIFPMH6OZM6FWYD7A4/ (visited on 04/13/2021).

[] *Fast Commits for Ext4 [LWN.Net]*. URL: https://lwn.net/SubscriberLink/842385/ea43ae3921000c72/ (visited on 01/16/2021).

[] *Filebench GitHub Wiki / Predefined Personalities*. GitHub. URL: https://github.com/filebench/filebench/wiki/Predefined-personalities (visited on 03/30/2021).

[] *FlushWAL; Less Fwrite, Faster Writes*. RocksDB. URL: http://rocksdb.org/blog/2017/08/25/flushwal.html (visited on 10/03/2020).

[] *How Fast Is Redis? – Redis*. URL: https://redis.io/topics/benchmarks (visited on 05/15/2021).

[] *Linux 5.0 Compat: SIMD Compatibility · Openzfs/Zfs@e5db313*. GitHub. URL: /openzfs/zfs/commit/e5db31349484e5e859c7a942eb15b98d68ce5b4d (visited on 05/04/2021).

[] *Metadata Allocation Classes by Don-Brady · Pull Request #5182 · Openzfs/Zfs*. GitHub. URL: https://github.com/openzfs/zfs/pull/5182 (visited on 04/13/2021).

[] *MyRocks | A RocksDB Storage Engine with MySQL*. MyRocks. URL: http://myrocks.io/ (visited on 05/16/2021).

[] *Ndctl-Inject-Error Man Page - Ndctl - General Commands*. URL: https://www.mankier.com/1/ndctl-inject-error (visited on 04/13/2021).

[] *Redis Persistence – Redis*. URL: https://redis.io/topics/persistence (visited on 05/15/2021).

[] *RocksDB GitHub Wiki: WAL Performance*. GitHub. URL: https://github.com/facebook/rocksdb/wiki/WAL-Performance (visited on 05/15/2021).

[] *Valgrind*. URL: https://www.valgrind.org/ (visited on 04/13/2021).

[] *Writecache Target — The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/writecache.html (visited on 04/13/2021).

[] *Zinject OpenZFS Man Page*. GitHub. URL: https://github.com/openzfs/zfs/blob/65c7cc49bfcf49d38fc84552a17d7e8a3268e58e/man/man8/zinject.8 (visited on 04/13/2021).

[Bon+03] Jeff Bonwick et al. "The Zettabyte File System." In: *Proc. of the 2nd Usenix Conference on File and Storage Technologies*. Vol. 215. 2003.

[Bor+16] James Bornholt et al. "Specifying and Checking File System Crash-Consistency Models." In: *ACM SIGPLAN Notices* 51.4 (Mar. 25, 2016), pp. 83–98. ISSN: 0362-1340. DOI: 10.1145/2954679.2872406. URL: https://doi.org/10.1145/2954679.2872406 (visited on 02/03/2021).

[Con+09] Jeremy Condit et al. "Better I/O through Byte-Addressable, Persistent Memory." In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. New York, NY, USA: Association for Computing Machinery, Oct. 11, 2009, pp. 133–146. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629589. URL: https://doi.org/10.1145/1629575.1629589 (visited on 02/04/2021).

[DeW+84] David J. DeWitt et al. "Implementation Techniques for Main Memory Database Systems." In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. 1984, pp. 1–8.

[Dul+14] Subramanya R. Dulloor et al. "System Software for Persistent Memory." In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. New York, NY, USA: Association for Computing Machinery, Apr. 14, 2014, pp. 1–15. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592814. URL: https://doi.org/10.1145/2592798.2592814 (visited on 02/04/2021).

[Fan+11] Ru Fang et al. "High Performance Database Logging Using Storage Class Memory." In: *2011 IEEE 27th International Conference on Data Engineering*. 2011 IEEE 27th International Conference on Data Engineering. Apr. 2011, pp. 1221–1231. DOI: 10.1109/ICDE.2011.5767918.

[HLM94] Dave Hitz, James Lau, and Michael A. Malcolm. "File System Design for an NFS File Server Appliance." In: *USENIX Winter*. Vol. 94. 1994.

[Joh+10] Ryan Johnson et al. "Aether: A Scalable Approach to Logging." In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 681–692.

[Kad+19] Rohan Kadekodi et al. "SplitFS: Reducing Software Overhead in File Systems for Persistent Memory." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. New York, NY, USA: Association for Computing Machinery, Oct. 27, 2019,

pp. 494–508. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359631. URL: https://doi.org/10.1145/3341301.3359631 (visited on 10/01/2020).

[Kwo+17] Youngjin Kwon et al. "Strata: A Cross Media File System." In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. New York, NY, USA: Association for Computing Machinery, Oct. 14, 2017, pp. 460–477. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132770. URL: https://doi.org/10.1145/3132747.3132770 (visited on 10/01/2020).

[Lan+14] Philip Lantz et al. "Yat: A Validation Framework for Persistent Memory Software." In: *2014 ${$USENIX$$ Annual Technical Conference (${$USENIX$$$${$ATC$$ 14)*. 2014, pp. 433–438.

[LBN13] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. "Unioning of the Buffer Cache and Journaling Layers with Non-Volatile Memory." In: *Presented as Part of the 11th ${$USENIX$$ Conference on File and Storage Technologies (${$FAST$$ 13)*. 2013, pp. 73–80.

[Liu+19] Sihang Liu et al. "Pmtest: A Fast and Flexible Testing Framework for Persistent Memory Programs." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 411–425.

[MNW14] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. 2nd ed. Addison-Wesley Professional, 2014. 928 pp. ISBN: 978-0-321-96897-5.

[Ope16] OpenZFS, director. *Matt Ahrens - Lecture on OpenZFS Read and Write Code Paths*. Mar. 4, 2016. URL: https://www.youtube.com/watch?v=ptY6-K78McY (visited on 05/28/2021).

[Ope20] OpenZFS, director. *ZIL Performance Improvements for Fast Media by Saji Nair*. Oct. 12, 2020. URL: https://www.youtube.com/watch?v=TnXwrigwF7I (visited on 04/13/2021).

[PAA05] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Model-Based Failure Analysis of Journaling File Systems." In: *2005 International Conference on Dependable Systems and Networks (DSN'05)*. IEEE, 2005, pp. 802–811.

[Pel+13] Steven Pelley et al. "Storage Management in the NVRAM Era." In: *Proceedings of the VLDB Endowment* 7.2 (2013), pp. 121–132.

[Pil+14] Thanumalayan Sankaranarayana Pillai et al. "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications." In: *11th ${$USENIX$$ Symposium on Operating Systems Design and Implementation (${$OSDI$$ 14)*. 2014, pp. 433–448.

[PS17] Daejun Park and Dongkun Shin. "iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call." In: *2017 ${$USENIX$$ Annual Technical Conference (${$USENIX$$${$ATC$$ 17)*. 2017, pp. 787–798.

[Sca20] Steve Scargall. "Persistent Memory Architecture." In: *Programming Persistent Memory: A Comprehensive Guide for Developers*. Berkeley, CA: Apress, 2020, pp. 11–30. ISBN: 978-1-4842-4932-1. DOI: 10.1007/978-1-4842-4932-1âĆĆ. URL: https://doi.org/10.1007/978-1-4842-4932-1%E2%82%82.

[SDM413] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3. 325462-074US. Apr. 2021. Chap. 4.1.3 Paging-Mode Modifiers.

[Tad+19] Rajesh Tadakamadla et al. "Accelerating Database Workloads with DM-WriteCache and Persistent Memory." In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 2019, pp. 255–263.

[Vol+14] Haris Volos et al. "Aerie: Flexible File-System Interfaces to Storage-Class Memory." In: *Proceedings of the Ninth European Conference on Computer Systems*. 2014, pp. 1–14.

[XS16] Jian Xu and Steven Swanson. "${$NOVA$$: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories." In: *14th ${$USENIX$$ Conference on File and Storage Technologies (${$FAST$$ 16)*. 2016, pp. 323–338.

[Xu+17] Jian Xu et al. "NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System." In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 478–496.

[Yan+06] Junfeng Yang et al. "Using Model Checking to Find Serious File System Errors." In: *ACM Transactions on Computer Systems (TOCS)* 24.4 (2006), pp. 393–423.

[Yan+20] Jian Yang et al. "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory." In: 18th {USENIX Conference on File and Storage Technologies ({FAST 20). 2020, pp. 169–182. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/yang (visited on 04/26/2021).

[YCH19] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. "EvFS: User-Level, Event-Driven File System for Non-Volatile Memory." In: 11th {USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19). 2019. URL: https://www.usenix.org/conference/hotstorage19/presentation/yoshimura (visited on 10/02/2020).

[Zha+10] Yupu Zhang et al. "End-to-End Data Integrity for File Systems: A ZFS Case Study." In: *FAST*. 2010, pp. 29–42.

[Zha+21] Wenhui Zhang et al. "ChameleonDB: A Key-Value Store for Optane Persistent Memory." In: (2021).

[ZHS19] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. "Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks." In: *17th ${$USENIX$$ Conference on File and Storage Technologies (${$FAST$$ 19)*. 2019, pp. 207–219.