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

Exposé for a Master’s Thesis

# 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.

# 0 OpenZFS Primer

OpenZFS is a combined volume manager and file system with many advanced features that we will not further elaborate on in this exposé. We recommend [this](https://en.wikipedia.org/wiki/OpenZFS) and [this](https://openzfs.org/wiki/Main_Page) web site for introductory information.

The OpenZFS on-disk format is a merkle tree rooted at the superblock (*uberblock*). It maintains on-disk consistency at all times through copy-on-write. However, to mitigate the inefficiencies of the “bubble up” effect that is typical to copy-on-write systems, the merkle tree is only updated every five seconds or if sufficient dirty data has accumulated. This periodic update of the on-disk format is commonly referred to as *txg sync* or *spa sync*. (A draft for the original on-disk spec can be found [here](http://www.giis.co.in/Zfs_ondiskformat.pdf).)

Asynchronous file system operations in ZFS only modify in-DRAM state and then return to user space immediately. Their modifications are persisted eventually by the txg sync background thread.  
For operations that have synchronous semantics such as writes to files opened with *O\_SYNC*, ZFS maintains the **ZFS Intent Log** (ZIL). The ZIL is a logical redo log which contains a sequence of non-idempotent replayable log entries. The ZIL chain consists of so-called log-write blocks (LWBs) each containing many log entries. The ZIL chain is rooted within the merkle tree but can be extended independent of the periodic txg sync. The system call can return to user space once all the LWBs that contain log entries created by the system call have been written to disk. The alternative to maintain synchronous semantics would be to wait for txg sync. Obsolete entries at the head of the ZIL chain are freed during txg sync. Note that a syscall with synchronous semantics carries out all the work that a non-sync syscall would. The time spent on writing the ZIL entry is thus strictly additive.

The ZIL’s on-disk blocks are allocated through ZFS’s regular block allocation mechanism. This implies that any of the block devices in the pool can hold ZIL blocks. To improve performance, ***separate log devices* (SLOGs)** can be added to the pool which are reserved for ZIL blocks and preferred for ZIL block allocation.

ZFS’s unified infrastructure for performing block IO to the operating system’s block device drivers is the ZIO pipeline. Many of ZFS’s distinguishing features are implemented in the ZIO pipeline, e.g. transparent data and metadata checksumming, compression, deduplication and encryption as well as redundancy mechanisms such as raidz. Both the IOs issued by txg sync and the ZIL IOs are made through the ZIO pipeline.

# 1 Motivation

Persistent Memory (PMEM) is an emerging storage technology that provides byte-addressable persistent storage. In contrast to NVMe devices, PMEM sits directly on the memory bus. For example, a 128GiB PMEM DIMM in AppDirectNonInterleaved occupies a 128GiB contiguous physical memory region. Code executing on the CPU stores data to the PMEM DIMM by issuing regular store instructions plus flushing the written cache lines. Reading from PMEM is done through load instructions. PMEM access latency is ~300ns, which is only 3x slower than DRAM access but at least 5x faster than the fastest NVMe devices on the market. (Learn more about PMEM [starting here](https://pmem.io/2019/12/19/performance.html)).

Linux can expose PMEM in several modes, the most common being *fsdax* where a PMEM DIMM shows up as /dev/pmemX block device. /dev/pmem can thus be used by any existing block device consumer. *bio\_* submissions are converted to synchronous writes and necessary cache flushes  
Further, Linux provides the kernel-internal DAX (“**D**irect **Acc**ess”) infrastructure which can be leveraged by existing block device consumers (read: file systems) to detect that they run on a PMEM device. These file systems can then implement PMEM-specific optimizations and a variant of *mmap*that establishes a direct mapping from user pages to PMEM pages. Both XFS and Ext4 support this mmap variant. On the block-device layer, the dm-writecache device mapper target uses DAX to provide a fast block-level write-back cache.

## 1.1 Benchmark: /dev/pmem as a SLOG device

The motivation for this thesis is a benchmark published by OpenZFS vendor [iXsystems who used FreeBSD’s equivalent of /dev/pmem as an OpenZFS SLOG device](https://www.snia.org/educational-library/real-world-performance-advantages-nvdimm-and-nvme-case-study-openzfs-2018).

They use battery-backed DRAM DIMMs (NVDIMM-N) in their storage products which have even lower latency / higher IOPS than PMEM. Their benchmark ‘Random Write 4KiB Thread Scaling - iSCSI’ achieves a total 22.5k IOPS with 8 threads writing to 8 different ZVOLs simultaneously. When scaling up to 4 threads per ZVOL (i.e. 32 concurrent writers), they achieve 40k IOPS.  
We conducted a similar experiment with Intel Optane DC memory on Linux with OpenZFS 2.0-rc2. The minimum single-threaded latency for a 4k synchronous write is ca. 80us, i.e. 12.5k IOPS. Parallelization using 8 fio threads with separate datasets per thread yields an almost linear speed-up to 77k IOPS. SMT is beneficial for throughput (110k IOPS at 16 threads) but latency rises to 143us.  
For async writes (sync=disabled), which only touch DRAM on their critical path, we observe 10us / 90k IOPS @ 1 thread and 20us / 165k IOPS @ 4 threads. The results of our benchmark are depicted in Figure 1.

|  |  |
| --- | --- |
|  |  |
|  |  |
| Figure 1: fio 4k 100% random write latency in a fresh zpool with one Intel Optane DC DIMM as /dev/pmem configured as SLOG device in OpenZFS 2.0.0-rc2. *sync* maps to the fio *--sync* flag, *numbjos* maps to the fio *--numjobs* flag (number of threads). Benchmark conducted on Supermicro X11DPi-NT, Intel(R) Xeon(R) Silver 4215 CPU @ 2.50GHz, Linux 5.8.10, OpenZFS commit 2.0.0-rc2, fio-3.23-28-g7064. | |

## 1.2 Benchmark: ext4 and dev-dax

Whereas the results from the previous section seem high in absolute terms, benchmarks with the same single PMEM DIMM in AppDirectNotInterleaved mode show that ZFS leverages very little of the possible PMEM performance:

* With Ext4 (mounted with or without -o dax, it does not matter), the same fio config that was used for the OpenZFS benchmark above achieves 100k IOPS @ 1 thread and up to 466k IOPS @ 4 threads.
* For the raw DIMM, fio’s *dev-dax* IO engine achieves 408k IOPS @ 1 thread and peaks at 511k IOPS @ 2 threads, then falls back to 460k IOPS @ 8 threads. These observations are in line with other reports from academia (Yang et al., “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory.”)

## 1.3 Analysis: /dev/pmem as SLOG

We conduct a superficial superficial analysis of the ZFS sync write code path to understand the distribution of latency.

As explained in Section 0, a sync write carries out all the work that an async write does, plus the work of writing the ZIL entry. A simplified model of sync write latency in the fio benchmark from Section 1.1 is thus given by

where is the size of the write and is *numjobs*, i.e., the number of threads performing those syn cwrites.  
  
The IOPS that can be achieved are

With regards to the ZFS benchmark in Section 1.1, is already given by the *sync=0* series. It starts with 10us @ 1 thread and grows approximately linearly up to the maximum measured of 250us @ 32 threads. Similarly, can be approximated by subtracting the latency of the *sync=0* from *sync=1* series.

Given these insights and the data from the benchmark we compute the latency added by the current ZIL implementation when using an Intel Optane DC Persistent Memory DIMM as a SLOG device. We refer to this number as . It is plotted in Figure 2.

|  |  |
| --- | --- |
|  |  |
| Figure 2: computed from the formulas above and the measurements depicted in Figure 1. | |

We use eBPF tracing to get a more detailed breakdown of for a single thread.

* The ITX layer’s contribution to overall latency is marginal.
  + Involved call stacks: *zfs\_log\_write* => *zil\_itx\_{create,assign}*
  + **1-6 us** total average latency for both *create* and *assign*
  + Note 1: *zil\_itx\_create* has an average latency of 2us, but the distribution is bimodal: if kmem\_cache\_alloc has a hit, it is in the 0.25-0.5us range, if kmem\_cache\_alloc has a miss, it is in the 8-16us range
  + Note 2: Lock contention is not a bottleneck (we measured up to 8 threads).
* *zil\_commit* is the function that stuffs ITXs into LWBs and writes out the LWBs to disk.
  + **~70us total average latency**
  + **~62us** of these ~70us spent in *zil\_commit\_waiter*; measured with  
    `./funclatency\_specialized -T -i 3 zil\_commit,zil\_commit\_waiter -S`
* *zil\_commit\_waiter* latency is distributed as follows:
  + 10%: waiting for LWB timeout, a mechanism that balances LWB space utilization against commit latency. Measured with  
    `./funclatency\_specialized -T -i 3 zil\_commit\_waiter,\_\_cv\_timedwait\_hires -S`
  + **85%**: **waiting for LWB write and then root ZIO to complete**; measured with  
    `./funclatency\_specialized -T -i 3 zil\_commit\_waiter,\_\_cv\_wait -S`
* ZIO latency is difficult to analyze using the *funclatency* tool due to the pipeline’s asynchronous design. However, we can compare the aggregate time spent in *zil\_commit\_waiter* vs time spent writing PMEM. Note that it does not make sense to compare latencies at this point because multiple ZIOs might be issued for one VFS-level 4k write due to size inflation by log entry metadata.
  + Measuring aggregate PMEM time:
    - Linux 5.10: ` ./funclatency\_specialized -T -i 3 pmem\_submit\_bio -S`
    - Linux 5.8: `./funclatency\_specialized -T -i 3 pmem\_do\_write -S`
    - (Note that PMEM latency depends on NUMA placement of DIMM vs zio pipeline thread that issues the bio. We conduct the experiment with both on the same NUMA node.)
  + Measuring aggregate waiting time for ZIO (same as above):
    - `./funclatency\_specialized -T -i 3 zil\_commit\_waiter,\_\_cv\_wait -S`
  + Result:  
    **Only** **15%** of the time spent waiting for the ZIO is spent in **pmem\_do\_write**.

## 1.4 The Case For ZIL-PMEM

Given the results of the latency analysis we see two alternatives to improve the situation:

1. Optimize ZIO pipeline latency.
2. Implement a ZIL-specific solution that does not use the existing LWB infrastructure and bypasses the ZIO pipeline completely.

In this exposé we propose to pursue the latter option. Our rationale is as follows:

**Research Value** There is little research value in optimizing the ZIO pipeline. In contrast, there is high research value in innovating in the PMEM space.

**Latency Is Insignificant To Most ZIO Pipeline Consumers** The ZIL is the only ZIO pipeline consumer for which end-to-end latency is significant. All other consumers are throughput-oriented, e.g. txg sync, dsl scan, and send/recv. They would not benefit significantly from a few microseconds of lower end-to-end latency.

**Ability To Leverage PMEM Features** An implementation that targets PMEM exclusively can take advantage of more distinguishing features than just low latency, e.g., the ability to do fine-grained persistent atomic updates at 8byte granularity.

**Maximum Performance** As we will point out in the remainder of this section, the maximum achievable speedup for the single-threaded 4k sync write use case is 1.89 when eliminating all ZIO pipeline overhead vs at least 3.65 with a PMEM-specific solution.

(We arranged all formulas to be on the same page)

We compute the ZIO pipeline’s and maximum speedup as follows:

(Total time spent on ZIL)

(Time spent on zil\_commit)

;

(Breakdown *zil\_commit\_waiter*)

(Breakdown *zil\_commit\_waiter*)

(Breakdown *zil\_commit\_waiter*)

Comparison of accumulated time spent in pmem\_do\_write vs. total :

Formulas and together yield:

Plugging this result intoyields the total :

Assuming 10us for based on the previous experiments, the total contribution of the ZIO pipeline to 4k sync write latency with a single thread is

Assuming all overhead can be eliminated, the achievable IOPS would be

which is a speedup of over the baseline (Figure 1).

**This speedup seems rather meager** considering that we observed 408k single-threaded 4k IOPS in the fio + dev-dax experiment in Section 1.2. We proceed by estimating the ideal IOPS achievable with a ZIL that does not use LWBs or the ZIO pipeline.

(Size of a log entry for a 4k write)

Assuming that the ITX layer is still used and 100% responsible for :

The ideal speedup in this model is .

# 2 Thesis Outline

Given out findings in the previous section we propose to develop a new type of ZIL called ZIL-PMEM that does not use the existing LWB infrastructure, bypasses the ZIO pipeline completely and targets PMEM exclusively. The research goal of this thesis is to explore the optimal use of persistent memory as a storage medium for the ZIL.

The following aspects are hard requirements for the project and subject to evaluation:

* Correctness
  + In the absence of hardware errors or bitflips, ZIL-PMEM must be able to replay all committed data in the same way as the current ZIL implementation does. Specifically:
    - Replay must respect the logical dependencies of log entries.
    - 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 replay must be able to resume. Resume must continue to respect logical dependencies of log entries.
  + Evaluation:
    - Formal modelling of full ZFS semantics is not required.
    - Application of Intel-provided tools to check correctness of writes to PMEM (cache flushes, etc.)
    - Unit- & Integration-Testing:
      * new tests specific to ZIL-PMEM
      * if possible the existing ZFS integration test suite
    - If sensible integration into the ZFS fault injection framework (e.g. for ZIL header persistence)
* Data Integrity
  + Bitflips within in-PMEM log entries must be detected during log replay using state-of-the-art techniques (error-detecting codes).
  + Time-of-check vs. time-of-use correctness: Bitflips in PMEM can happen at any time during system operation. This must be handled correctly.
  + Bitflips that are detected must be treated such that correctness is upheld (see previous major bullet point).
  + Minimum requirement: if a bitflip is detected the user is alerted and can decide whether they want to abandon the log or wait for the bitflip to disappear.
  + Ideal outcome: The user is informed about detected errors. Any log entry that can still be replayed correctly is replayed.
  + Evaluation: through unit & integration tests developed for ZIL-PMEM.
* Performance
  + Evaluation on real PMEM hardware, i.e., Intel Optane DC Persistent Memory.
  + Primary workload: 4k sync writes from user space to files and/or ZVOLs
  + Use of state-of-the-art standard benchmarks to evaluate regressions in other areas (e.g. synchronous metadata operations, fsync on directories, etc).
  + High degree of automation to execute benchmarks and gather raw data.

The design & implementation is constrained as follows:

* No changes to existing ZIL semantics.
  + The semantics and logical dependencies encoded in the write-ahead log remain unchanged. (“ITX layer”)
  + Rationale: changing semantics or guarantees would make the evaluation irrelevant as weaker guarantees naturally allow for greater performance.

The following aspects are explicitly out of scope for the thesis and subject to future work:

* Application of PMEM to anything not related to the ZIL (e.g. dbufs/arcbufs.)
* NUMA-Awareness
* Redundancy / Mirroring
* Protection against “scribbles” caused by code outside of the ZIL.
  + Scribbles are bugs in the system that cause overwrites of PMEM space.
  + Related work (e.g. NOVA-Fortis) has already presented highly-performant approaches to deal with scribbles.
* Compatibility with OpenZFS Native Encryption
  + But the design may accommodate this feature.

# 

# 3 Related Work

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

### 3.1 PMEM File Systems

In this subsection we present research file systems that were explicitly designed for persistent memory. ZIL-PMEM integrates into ZFS, a production file system that was *not* designed for persistent memory. Hence we focus on techniques for crash consistency and data integrity which are applicable to our work.

#### 3.1.1 In-Kernel PMEM File Systems

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

**BPFS (2009)** is one of the earliest file systems expressly designed for PMEM. The file system layout in PMEM is inspired by WAFL and resembles a tree of multi-level indirect pages that eventually point to data pages. BPFS’s key contribution is the use of fine-grained atomic updates to ensure crash-consistency. Wherever possible, in-place atomic updates are used, e.g., updating *atime* or updating the file size after appending to a file. For cases where atomic operations are insufficient the authors use *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 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. This avoids “bubbling-up” of changes to the root of the tree. There is no further need for journaling in BPFS. The evaluation does not address correctness. (Condit et al., “Better I/O through Byte-Addressable, Persistent Memory.”)

**PMFS (2014)** is another research file system 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 (unit: number of bytes copied). PMFS also introduces protection against “scribbles” over file system data through *CR0.WP*. (Dulloor et al., “System Software for Persistent Memory.”)

**NOVA (2016)** is the most mature research PMEM file system. 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 *write* log entries which, judging from the text, are always at page granularity. The authors explain the recovery procedure and measure its performance but do not address correctness in the evaluation. (Xu and Swanson, “NOVA: A log-structured file system for hybrid volatile/non-volatile main memories.”)

**NOVA-Fortis (2017)** is a version of NOVA that introduces snapshots and hardening against data corruption. Snapshots are not relevant for this thesis. The data corruption countermeasures are manifold:

* Handling of *machine check exceptions (MCE*) in case the hardware detects bit errors. This is done by using *memcpy\_mcsafe*() for all PMEM access; PMEM data is always buffered in DRAM before it is read.
  + This also includes code to handle the kernel’s *poisoning* of physical address ranges and the ACPI interface to clear errors indicated by the hardware.
* Detection from metadata corruption through CRC32 checksums.
* Recovery from metadata corruption by comparing checksums of the primary and replica and restoring the variant with the matching checksum.
* Protection against localized unrecoverable data loss through RAID4-style parity.
  + Naturally this works only while a file is not DAX-mmapped.

The recovery capabilities are explained in detail but the evaluation focuses exclusively on the performance impact of the new protection features; it does not evaluate correctness or actual behavior during recovery. (Xu et al., “NOVA-Fortis.”)

#### 3.1.2 Hybrid PMEM File Systems

More recent publications explored the performance benefits of splitting responsibilities between the kernel and a user-space component in order to eliminate system call overhead.

**SplitFS (Kadekodi et al..”, 2019)** is a research file system 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. The authors put some emphasis on the optimizations such as the use of checksums andz aligned start addresses to find valid log entries instead of an explicitly linked list.  
The evaluation of correctness is limited to a comparison of user-observable file system state between SplitFS and ext4 in DAX mode. Recovery is evaluated only through the lens of recovery time (performance), not correctness.

**Strata (Kwon et al., 2017)** is a *cross-media* file system with both kernel and user-space components. Since its distinguishing feature is the intelligent migration of data between different storage media we will discuss it in the ‘cross-media storage systems’ section below.

#### 3.1.3 User-Space PMEM File Systems

**Aerie (Volos et al., 2014)** is a user-space file system 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 file system service periodically or when a global lock is released. It is our understanding that only log entries shipped to and validated by the file system 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.

**EvFS (Yoshimura, Chiba, and Horii, 2019)** 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 and appears to be out of scope: “EvFS is not a production-ready file system because it neither provides all the POSIX APIs or crash-safe properties”.

#### 3.1.4 In-Device File Systems

**DevFS (Kannan et al., 2018)** proposes to move the data path of the file system implementation into the storage device. The device exposes per-file I/O queues (reminiscent of NVMe command queues) that can be made accessible to user-space applications, thereby completely eliminating the kernel from the data plane. The concept itself is independent of PMEM as a storage technology. The system achieves compatibility with existing applications through a user-space library. Crash consistency is handled at the hardware level: “DevFS avoids logging to persistent storage by using device capacitors that can hold power until the device controller can safely flush data and metadata to storage.”

### 3.2 Cross-Media Storage Systems

Cross-media file systems are storage systems that 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 their high capacity, acceptable sequential read/write perf and low cost
* flash storage for low-latency random read/write workloads with small block sizes.

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 ZIL-PMEM we find it most useful to compare the overall system architecture.

#### 3.2.1 ZFS: /dev/pmem as a SLOG, L2ARC, Special, Or Dedup Device

ZFS was initially designed as a file system to support large pools of hard disks. Over its lifetime several cross-media file system features were added that, in many deployments, are essential to provide adequate performance.

On the write path, separate log devices (**SLOG**) can be added to a storage pool to be used exclusively for persisting ZFS’s logical write-ahead log (the ZIL). Offloading latency-sensitive IO to the log device allows for greater throughput during txg sync to the main pool’s devices.

On the read path, the in-DRAM ARC, which is ZFS’s variant of the buffer cache, can be extended with a second-level victim cache on a block device, the L2ARC. L2ARC hits free up IOPS among the main pool’s devices which can be used for txg sync.

ZFS allocation classes are a more recent and generalized mechanism to limit the kinds of blocks that can be allocated from a given device in the storage pool. In addition to SLOG and L2ARC, there are allocation classes for small file blocks (**special**) and for deduplication table data (**dedup**).

All of these features are implemented at the block allocation and ZIO pipeline level which ultimately expects a block device interface. The ZIO pipeline includes a load balancing mechanism to level block devices bandwidth and storage utilization. There are also minimal automatic optimizations for non-rotating media in the ZFS.

Compared to the other cross-media file systems that were expressly designed for PMEM, the general approach in ZFS is that devices in each allocation *should* have relatively homogenous characteristics. It is then up to the administrator to maximize performance through allocation classes and tunables.

At the OpenZFS 2020 Developer summit, [a prototype for a ZIO pipeline bypass for the ZIL for fast SLOG devices was presented by Saji Nair of storage vendor Nutanix](https://openzfs.org/wiki/OpenZFS_Developer_Summit_2020_talks#ZIL_Design_Challenges_for_Fast_Media_.28Saji_Nair.29). The presentation mentions inefficiencies in the ZIO pipeline (context switching overheads) as well as software-technicalities and semantically unnecessary dependencies between I/O operations in the current ZIL implementation. The work has a strong focus on achieving more I/O parallelism through removal of the unnecessary dependencies between ZIO operations involved in writing the ZIL’s LWBs. The evaluation is limited to 4k sync write performance, citing up to 4x IOPS. Replay is only addressed superficially on one slide. We have reason to believe that the scheme proposed on the slide is incorrect or at least incomplete.

#### 3.2.2 Cross-Media Systems That Log To PMEM

**Strata (2017)** is a cross-media research file system. **“**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” (Kwon et al., “Strata: A Cross Media File System”).

We classified Strata as a hybrid file system in Section 3.1 because it consists of both a userspace library and an in-kernel component.  
The log, written from userspace, is an idempotent logical redo log, much like the ZFS intent log. The kernel component *digests* the userspace logs asynchronously and in parallel.

The kernel component performs aggregation of operations during digestion (e.g. same block written multiple times).  
Bulk digestion also permits the kernel component to issue “sequential, aligned writes” to the slower 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.
* The Strata paper suggests that it uses a single block size whereas ZFS supports variable block sizes (“Free Block Bitmap” in Strata vs. Spacemaps in ZFS).
* Strata seems to tune allocations for SSDs, e.g. allocating blocks in erasure block size to prevent write amplification. ZFS has no automatic optimizations that target write amplification in SSDs.
* ZFS guarantees data integrity and has provisions for redundancy.
* ZFS supports many devices at different performance class levels whereas Strata only supports a single device.
* 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.
* ZFS is a mature filesystem used at large scale in production whereas Strata is an unproven research prototype.

**Ziggurat (2019) “**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.” (Zheng, Hoseinzadeh, and Swanson, “Ziggurat: a tiered file system for non-volatile main memories and disks.”)

**Ziggurat compares to Strata as follows:**

* “Strata is a multi-tiered user-space file system that exploits NVMM as the high-performance tier, and SSD/HDD as the lower tiers. It uses the byte-addressability of NVMM to coalesce logs and migrate them to lower tiers to minimize write amplification. **File data can only be allocated in NVMM in Strata**, and they can be **migrated only from a faster tier to a slower one.** The profiling granularity of Strata is a page, which increases the bookkeeping overhead and wastes the locality information of file accesses. (Zheng, Hoseinzadeh, and Swanson, “Ziggurat.”)

**ZFS with and without ZIL-PMEM compares to Ziggurat as follows:**

* Ziggurat can *migrate* data *up* to PMEM. ZIL-PMEM does not address this concern at all and the ZFS architecture makes such a feature unlikely to happen (keyword: *blockpointer rewrite*).
* Ziggurat sends writes directly to the suitable tier through advanced prediction schemes. In ZFS the ZIL is only used if synchronous semantics are required and, as explained in the section on Strata, writes changes twice, once to the log and once to the main pool during txg sync. (The *WR\_INDIRECT* ZIL log entry type mitigates this for large writes).
* Both systems are fully in-kernel and thus require no modifications to user-space applications.
* Ziggurat builds on NOVA-Fortis and thus inherits the PMEM-specific data integrity / redundancy mechanism provided by it for the PMEM storage tier. ZIL-PMEM data integrity measures are more limited but could be expanded in the future.
* The paper does not mention data integrity measures or redundancy mechanisms for the layers beneath PMEM. Possibly, the existing Linux Device Mapper stack could be used to compensate. In contrast, ZIL-PMEM benefits from ZFS’s strong data integrity and redundancy mechanisms for all data *except* the log entries.
* Ziggurat is “fast-first. It should use disks to expand the capacity of NVMM rather than using NVMM to improve the performance of disks as some previous systems have done”. ZIL-PMEM approaches persistent memory from the opposite direction, starting with a file system strongly focussed on block devices and leveraging PMEM where appropriate.
* Ziggurat was not evaluated on actual persistent memory. The authors used memory on another NUMA node to simulate lower latencies.
* An examination of the Ziggurat code base shows that it only supports a single device per tier. The ‘lowest’ tier is always PMEM, higher tiers are always block devices.
* Ziggurat has been unmaintained since the publication. OpenZFS has a healthy community of both hobbyist and paid full-time contributors.

#### 3.2.3 Block-Level Caches on PMEM

The Linux kernel provides two device mapper targets that can be used as caches for a slower block device. This is relevant to ZIL-PMEM because ZFS can emulate block devices through ZVOLs. ZVOLs use the same ZIL format as ZFS file systems do to implement block device requests with sync semantics. With regards to the evaluation of ZIL-PMEM we believe that performance metrics and redundancy features are most relevant.

**Linux Device Mapper with dm-cache** (pre 2018) is the established volume manager in Linux. With regards to volume management, Device Mapper supports striping (dm-linear), mirroring (dm-mirror) and different RAID levels (dm-raid). While different device classes can be put in the same *volume group*, it is inadvisable to use volumes of different device classes in the same dm-\* volume management abstraction.  
Instead, one can use **dm-cache** to configure a block device as a read/write cache, either in write-through or write-back mode. As such, /dev/pmem can be used in lieu of an NVMe or SSD device. Device Mapper’s redundancy mechanisms could theoretically be used to protect against data corruption or permanent errors in PMEM. We have not conducted a performance evaluation of dm-cache.

**Linux Device Mapper with dm-writecache** (since 2018) is a new Device Mapper module that provides a **low-latency write-back write-only cache** with explicit support for persistent memory.

* dm-writeache persists writes to PMEM before signalling completion, then performs write-back to the underlying block device or device mapper target in the background.
* dm-writecache uses *memcpy\_mcsafe()* to handle hardware-reported memory errors.
* dm-writecache does not take further provisions to guarantee data integrity.

In our experiments on Linux 5.8, dm-writecache delivers on the promise of low latency (100K IOPS @ 1 thread) but shows significant lock contention with multiple threads (peaks at 150k IOPS @ 4 threads). Compared to our proposal for a ZIL-PMEM in ZFS, dm-writecache might be able to handle overwrites more space-efficiently since the ZFS log semantics currently do not allow coalescing of overwrites into one log entry (e.g. on fsync).

### 3.3 Adaptation Of Block-Level (Sub-)Systems To PMEM

Several systems have been adapted to take advantage of PMEM when it is available, similar to what we plan to do with ZIL-PMEM. Again, we focus on the overall system architecture and performance numbers if applicable.

#### 3.3.1 On The File-System Level

**Linux DAX** is a Linux subsystem that allows file systems such as **ext4** and **xfs** to detect that the underlying storage medium is PMEM. The file system can then implement PMEM-specific optimizations and mmap files directly into user-accessible memory. This feature enables applications to bypass the operating system entirely for data persistence if all file blocks are pre-allocated and all pages pre-faulted. Intel’s PMDK builds all of its abstractions on top of this feature.

**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.” (<https://lwn.net/Articles/842385/>, 2020-02-03).

The article mentions ongoing work to leverage persistent memory for ext4 fast commits.

The approach is inspired by per-inode journaling as proposed by Park & Shin at Usenix ATC 2017.

**Linux jbd2 by example of ext4 data=journal on /dev/pmem** Linux provides the *journaling block device* (JBD2) used by several file systems in the kernel. As the name suggests, JBD2 is a reusable abstraction for a block-level file system journal. One of the consumers is ext4. Ext4 provides the ability to perform data journaling, and to use a separate block device for journaling. We used /dev/pmem as such a separate journaling device and conducted the 4k sync write experiments from Section 1. We observed that the system peaks at ~70k 4k sync write IOPS. It is bottlenecked by a single journaling thread in Linux’s jbd2 subsystem.

**Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory (2013)** 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 journal (e.g. one managed by JBD2) are indistinguishable from non-journal file system 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 file system 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.

#### 3.3.2 Block-Level Write-Ahead-Logs in Databases

Disk-oriented databases’ write-ahead logs are generally block-based. There is a significant body of prior work with PMEM in this area. Many of the aforementioned publications on file systems cite papers from the database community.  
Many databases implement some variant of ARIES which provides features that exceed the needs of file system journaling. Some techniques developed in the database community might be applicable to our work, though.

**Fang et al., “High Performance Database Logging Using Storage Class Memory.” (2012)** This is a relatively early paper outlining how databases should adjust their logs to better utilize PMEM. They explain that “Group Commit” is no longer necessary. Group Commit is a common technique to improve log throughput in databases. ZFS ZIL’s LWBs timeout mechanism (briefly mentioned in Section 1.3) are quite reminiscent of Group Commit.

**Pelley et al., “Storage Management in the NVRAM Era.”** also elaborate on NVRAM Group Commit.

### 3.4 Testing File System Crash Consistency

The publications that introduced the file systems presented in earlier subsections put little to no emphasis on evaluating or proving crash consistency. We expanded our literature survey to get an impression of the research that focuses not on raw performance but on crash consistency guarantees.

**All File Systems Are Not Created Equal (Pillai et al., 2014)** contributes a survey of the atomicity and ordering guarantees of several popular Linux file systems and provides a tool called ALICE to validate or derive the guarantees required by applications.

ZFS / 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. Also, ALICE could be used to determine whether ZFS’s guarantees exceed the requirements of the majority of applications which would indicate room for relaxation of requirements / higher performance. Due to the scope of this thesis (we maintain the semantics of the existing ZIL) these use cases are mostly relevant to future work.

**Specifying and Checking File System Crash-Consistency Models (Bornholt et al., 2016)** 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 file system behavior as a “litmus test”. A litmus test encodes expected behavior of a file system 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 file system to ensure that, iff the actual file system 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 file system 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 file system under test for recovery. After recovery the litmus test’s predicate must hold against the concrete file system state after recovery. If it does not hold the given permutation of the trace is proof that the file system does not match its model (assuming the litmus test passes execution against the model), or that the file system’s assumptions about the disk do not match the disk model.  
FERRITE appears to be a great tool to build a model of ZFS’s semantics (ZFS has not been evaluated by the authors). A model would be helpful to ensure that ZIL-PMEM implements the same semantics as the existing ZIL. This would require a “disk model” for persistent memory.

**Using Model Checking to Find Serious File System Errors (Yang et al., 2006)** The authors present an “implementation-level model checker” that removes the need to define a formal model for the file system. Instead, the model is inferred by running the OS with the file system 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 file system state and compares it to the expected “stable file system” 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 (applies to FERRITE as well, see above)
* The recovery checker produces a projection of the file system state (e.g. only names and content but no atime). Thus, the system can only check guarantees at the projection level.
* Also, baked in assumptions such as “Events should also have temporal independence in that creating new files and directories should not harm old files and directories” seem quite restrictive.

The idea of using the file system’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.  
We think that the results would be useful as a starting point for a formal model, e.g., as initial litmus tests for use with FERRITE.

We do not think that we can apply the presented approach to ZFS / ZIL-PMEM with reasonable effort.

### 3.5 PMEM-specific Crash-Consistency Concerns

**Yat: A validation framework for persistent memory software (Lantz et al., 2014)** “Yat is a hypervisor-based framework that supports testing of applications that use Persistent Memory [...] By simulating the characteristics of PM, and integrating 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 (summarized in an earlier subsection).  
In contrast to many other publications presented in this section, the hypervisor-based approach of Yat makes it the ideal tool for evaluating ZIL-PMEM.

Sadly, Yat has never been published and remains an Intel-internal project.

**pmemcheck (**[**https://pmem.io/2015/07/17/pmemcheck-basic.html**](https://pmem.io/2015/07/17/pmemcheck-basic.html)) is a tool in Intel’s Persistent Memory Development Kit (PMDK). pmemcheck integrates into Valgrind, a popular tool for dynamic analysis. The application code must be modified to inform Valgrind about which memory regions are PMEM mappings as well as *every* pmem-related operation. pmemcheck can check for a variety of a variety of properties, depending on options passed to the tool:

* Correct use of instructions for persistence (e.g. check for CLWB after write to PMEM)
* Multiple overwrites of the same memory location without an intermediate persistency guarantee (indicative of the assumption of sequential crash consistency)
* Redundant flushing (relevant for performance)

Notably, the PMDK (including libpmem) already includes all the necessary instrumentation for Valgrind which means that the tool is minimally invasive to application source code that uses the PMDK.

ZIL-PMEM runs in the kernel and cannot use pmemcheck directly. However, if ZIL-PMEM code can be ported to user space as has been done for much of the ZFS kernel code to simplify testing, ZIL-PMEM could benefit from pmemcheck.

**PMTest: A fast and flexible testing framework for persistent memory programs** **(Liu et al., 2019)** PMTest is a userspace tool that is significantly faster than pmemcheck. It ships with two built-in checkers to “test two fundamental characteristics of all CCS [crash-consistent software]: the ordering and durability guarantee. These checkers can also serve as the building blocks of other application-specific, high-level checkers.”  
Notably, PMTest also provides limited support to test crash-consistency of code that runs as a kernel module. Sadly, one of the limitations is that only a single thread kernel module thread can be checked. However, ZIL-PMEM might still benefit from PMTest given that we plan to issue all writes to PMEM from the thread that makes the syscall.

### 3.6 Robustness Against Hardware Failures

**Model-based failure analysis of journaling file systems (Prabhakaran et al., 2005)** 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 idea behind this approach is that a kernel component intercepts block writes and injects write failures. If the file system code 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 file system specific which already shows in the adjustments required for IBM JFS.

ZFS’s architecture does not translate to the journaling modes presented in this document. However we can imagine that for ZIL-PMEM, a fault injection mode for persistent memory writes and access could be useful.

**ndctl-inject-error (https://pmem.io/ndctl/ndctl-inject-error.html)** is a subcommand of the administrative tooling for PMEM on Linux. It provides the ability to inject errors that manifest as **machine check exceptions** (MCE) which are indicative of a bad block. In userspace, Linux converts these errors into SIGBUS signals. In the kernel, *memcpy\_mcsafe()* must be used to avoid a kernel panic.  
We plan to use this tool during ZIL-PMEM development. We are still uncertain whether it is amenable for integration into automated regression testing.  
The following web page has more details on error recovery in PMEM applications: <https://software.intel.com/content/www/us/en/develop/articles/error-recovery-in-persistent-memory-applications.html>

**ZFS Fault Injection** The ZFS tool **zinject** allows for targeted injection of ZIO failures. The user can scope the injected errors to specific logical data objects. The ZFS integration test suite makes use of the command for high-level features such as automatic hot spares.

ZIL-PMEM does not use ZIO to access PMEM and thus cannot hook into the zinject infrastructure. Most of the semantics are tied to ZFS’s zbookmark and blkptr structures which we do not plan to use for ZIL-PMEM. We expect that proper adaptation of zinject to ZIL-PMEM will be difficult.

Scrach Pad

## 1.1 Estimated Performance Improvements

As outlined above we believe that ZFS could benefit immensely from a ZIL that is specialized for Persistent Memory (PMEM). In this document, we propose a design for this **new ZIL type called ZIL-PMEM**.

To quantify the potential speedup, we have implemented *pmem\_ringbuf*, a user-space prototype of the crash-consistent error-detecting data structure that ZIL-PMEM uses on the performance-critical write path. The following table (Figure 3) shows the scalability of this data structure for 4k writes. *numjobs* is the number of threads and *ncommitters* is the number of threads that are allowed to write to PMEM concurrently (implemented using a pthreads semaphore). The unit is *nano seconds*.

|  |
| --- |
|  |
| Figure 3: pmem\_ringbuf 4k write experiment. |

Plugging these values into the previously explained formula

yields the IOPS and speedup presented in Figure 4. The **baseline** for speedup is *ZIL-LWB* which is the configuration used in the introductory benchmarks where we evaluated the current ZIL implementation with /dev/pmem SLOGs. As an **upper bound** we include the measurements for sync=disabled which turns zil\_commit into a no-op.

|  |
| --- |
|  |
| Figure 4: Anticipated IOPS based on pmem\_ringbuf. |

## 3.1 Taxonomy

* Whole Filesystems adjusted to PMEM
  + Single Media vs. Tiered / Hierarchical Storage
  + History / Evolution: nvmm-first, hdd-first, hybrid-nvmm-first-then-blockdev, hybrid-blockdev-then-nvme , ...
  + Production-grade / Research
  + In-Kernel (Nova, DAX) | libFS + kernelFS (Strata) | libFS-only (SplitFS) | pmem-App
  + Consistency mechanism: block-level WAL vs. logical WAL
* PMEM for acceleration of unmodified systems
  + Block-Layer **Cache**: dm-writecache, bcache, flashcache
  + Block-Layer **Journaling**: (ext4 + pmem data journal, Unioning of the buffer cache …)
  + Database WALs

Ziggurat:

* An inspection of the publicly available NOVA + Ziggurat code base on GitHub reveals the following details not mentioned in the paper:
  + (Use `git log --all --graph --date-order -- fs/nova` to get an impression of the code flow / commit history)
  + Code History / Gossip
    - There is a ‘Ziggurat’ branch and a ‘tiering’ branch. The ‘tiering’ branch is based on Linux 4.13, whereas the Ziggurat branch is based on Linux 5.1.
    - Ziggurat has never been merged into the main NOVA code base.
    - The second-to-last commit in the Ziggurat branch is 12 days before the papers conference’s initial submission deadline.
    - According to Shengan Zheng’s GitHub site (first author in the list), he was a ‘Aug. 2017 - Sept. 2018 Visiting scholar, University of California, San Diego’
    - The second author, Morteza <[mhoseinzadeh@ucsd.edu](mailto:mhoseinzadeh@ucsd.edu)>, seems to have worked on the ‘tiering’ branch until 2017, which then appears to have been “merged” (code copied, no actual merge commit) into the Ziggurat branch. Since then, Morteza and Zheng seem to have collaborated in the Ziggurat branch.
  + Technical
    - Ziggurat has been called ‘tiered NOVA’ or ‘NOVA tiered’ internally for most of its lifetime.
    - **Ziggurat only supports a single device per tier. The ‘lowest’ tier is always PMEM, higher tiers are always block devices.**
    - The latency simulation mechanism (fs/nova/vpmem.c) has only been added for Ziggurat.
      * Likely because they didn’t have another system available?
    - **It is our (subjective) impression that there is a vast quality difference between the `master` branch and the `Ziggurat` branch.**
  + => This survey of the codebase suggests that Ziggurat has not been touched since the publication. We believe the project is dead and is not being pursued further.