Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks

Shengan Zheng, *Shanghai Jiao Tong University;* Morteza Hoseinzadeh and Steven Swanson, *University of California, San Diego*

https://www.usenix.org/conference/fast19/presentation/zheng

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Open access to the Proceedings of the
17th USENIX Conference on File and
Storage Technologies (FAST '19)
is sponsored by

Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks

Shengan Zheng[†*] Morteza Hoseinzadeh[§] Steven Swanson[§]
[†]*Shanghai Jiao Tong University* [§]*University of California, San Diego*

Abstract

Emerging fast, byte-addressable Non-Volatile Main Memory (NVMM) provides huge increases in storage performance compared to traditional disks. We present Ziggurat, a tiered file system that combines NVMM and slow disks to create a storage system with near-NVMM performance and large capacity. Ziggurat steers incoming writes to NVMM, DRAM, or disk depending on application access patterns, write size, and the likelihood that the application will stall until the write completes. Ziggurat profiles the application's access stream online to predict the behavior of individual writes. In the background, Ziggurat estimates the "temperature" of file data, and migrates the cold file data from NVMM to disks. To fully utilize disk bandwidth, Ziggurat coalesces data blocks into large, sequential writes. Experimental results show that with a small amount of NVMM and a large SSD, Ziggurat achieves up to 38.9× and 46.5× throughput improvement compared with EXT4 and XFS running on an SSD alone, respectively. As the amount of NVMM grows, Ziggurat's performance improves until it matches the performance of an NVMM-only file system.

1 Introduction

Emerging fast, byte-addressable persistent memories, such as battery-backed NVDIMMs [25] and 3D-XPoint [24], promise to dramatically increase the performance of storage systems. These non-volatile memory technologies offer vastly higher throughput and lower latency compared with traditional block-based storage devices.

Researchers have proposed several file systems [8, 10, 11, 29, 32] on NVMM. These file systems leverage the direct access (DAX) feature of persistent memory to bypass the page cache layer and provide user applications with direct access to file data.

The high performance of persistent memory comes at a high cost. The average price per byte of persistent memory is higher than SSD, and SSDs and hard drives scale to much larger capacities than NVMM. So, workloads that are cost-sensitive or require larger capacities than NVMM can provide would benefit from a storage system that can leverage the strengths of both technologies: NVMM for speed and disks for capacity.

Tiering is a solution to this dilemma. Tiered file systems manage a hierarchy of heterogeneous storage devices and place data in the storage device that is a good match for the data's performance requirements and the application's future access patterns.

Using NVMM poses new challenges to the data placement policy of tiered file systems. Existing tiered storage systems (such as FlashStore [9] and Nitro [23]) are based on disks (SSDs or HDDs) that provide the same block-based interface, and while SSDs are faster than hard disks, both achieve better performance with larger, sequential writes and neither can approach the latency of DRAM for reads or writes.

NVMM supports small (e.g., 8-byte) writes and offers DRAM-like latency for reads and write latency within a small factor of DRAM's. This makes the decision of where to place data and metadata more complex: The system must decide where to initially place write data (DRAM or NVMM), how to divide NVMM between metadata, freshly written data, and data that the application is likely to read.

The first challenge is how to fully exploit the high bandwidth and low latency of NVMM. Using NVMM introduces a much more efficient way to persist data than disk-based storage systems. File systems can persist synchronous writes simply by writing them to NVMM, which not only bypasses the page cache layer but also removes the high latency of disk accesses from the critical path. Nevertheless, a DRAM page cache still has higher throughput and lower latency than NVMM, which makes it competitive to perform asynchronous writes to the disk tiers.

The second challenge is how to reconcile NVMM's random access performance with the sequential accesses that disks and SSDs favor. In a tiered file system with NVMM and disks, bandwidth and latency are no longer the only

[*]This work was done while visiting University of California, San Diego.

differences between different storage tiers. Compared with disks, the gap between sequential and random performance of NVMM is much smaller, which makes it capable of absorbing random writes. Simultaneously, the file system should leverage NVMM to maximize the sequentiality of writes and reads to and from disk.

We propose Ziggurat, a tiered file system that spans NVMM and disks. 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.

We implement an efficient migration mechanism in Ziggurat to make room in NVMM for incoming file writes and accelerate reads to frequently accessed data. We first profile the temperature of file data and select the coldest file data blocks to migrate. During migration, Ziggurat coalesces adjacent data blocks and migrates them in large chunks to disk. Ziggurat also adjusts the migration policy according to the application access patterns.

The contributions of this paper include:

- We describe a synchronicity predictor to efficiently predict whether an application is likely to block waiting a write to complete.
- We describe a write size predictor to predict whether the writes to a file are both large and stable.
- We describe a migration mechanism that utilizes the characteristics of different storage devices to perform efficient migrations.
- We design an adaptive migration policy that can fit different access patterns of user applications.
- We implement and evaluate Ziggurat to demonstrate the effectiveness of the predictors and the migration mechanism.

We evaluate Ziggurat using a collection of micro- and macro-benchmarks. We find that Ziggurat is able to obtain near-NVMM performance on many workloads even with little NVMM. With a small amount of NVMM and a large SSD, Ziggurat achieves up to 38.9× and 46.5× throughput improvement compared with EXT4 and XFS running on SSD alone, respectively. As the amount of NVMM grows, Ziggurat's performance improves until it nearly matches the performance of an NVMM-only file system.

The remainder of the paper is organized as follows. Section 2 describes a variety of storage technologies and the NOVA file system. Section 3 presents a design overview of the Ziggurat file system. We discuss the placement policy and the migration mechanism of Ziggurat in Section 4 and Section 5, respectively. Section 6 evaluates Ziggurat, and Section 7 shows some related work. Finally, we present our conclusions in Section 8.

Technology	Latency		Sequential Bandwidth	
	Read	Write	Read	Write
DRAM	0.1μs	0.1μs	25GB/s	25GB/s
NVMM	0.2μs	0.5μs	10GB/s	5GB/s
Optane SSD	10μs	10μs	2.5GB/s	2GB/s
NVMe SSD	120μs	30μs	2GB/s	500MB/s
SATA SSD	80μs	85μs	500MB/s	500MB/s
Hard disk	10ms	10ms	100MB/s	100MB/s

Table 1: **Performance comparison among different storage media.** DRAM, NVMM and hard disk numbers are estimated based on [5, 19, 35]. SSD numbers are extracted from Intel's website.

2 Background

Ziggurat targets emerging non-volatile memory technologies and conventional block-based storage devices (e.g., SSDs or HDDs). This section provides background on NVMM and disks, and the NOVA file system that Ziggurat is based on.

2.1 Storage Technologies

Emerging non-volatile main memory (NVMM), solid-state drive (SSD) and hard disk drive (HDD) technologies have their unique latency, bandwidth, capacity, and characteristics. Table 1 shows the performance comparison of different storage devices.

Non-volatile memory provides byte-addressability, persistence and direct access via the CPU's memory controller. Battery-backed NVDIMMs [25, 26] have been available for some time. Battery-free non-volatile memory technologies include phase change memory (PCM) [22, 28], memristors [31, 33], and spin-torque transfer RAM (STT-RAM) [7, 20]. Intel and Micron's 3D-XPoint [24] will soon be available. All of these technologies offer both longer latency and higher density than DRAM. 3D-XPoint has also appeared in Optane SSDs [17], enabling SSDs that are much faster than their flash-based counterparts.

2.2 The NOVA File System

Ziggurat is implemented based on NOVA [32], an NVMM file system designed to maximize performance on hybrid memory systems while providing strong consistency guarantees. Below, we discuss the file structure and scalability aspects of NOVA's design that are most relevant to Ziggurat.

NOVA maintains a separate log for each inode. NOVA also maintains radix trees in DRAM that map file offsets to NVMM locations. The relationship between the inode, its log, and its data pages is illustrated in Figure 2a. For file writes, NOVA creates *write entries* (the log entries for data updates) in the inode log. Each write entry holds a pointer to the newly written data pages, as well as its modification time (`mtime`). After NOVA creates a write entry, it updates the tail of the inode log in NVMM, along with the in-DRAM radix tree.

NOVA uses per-cpu allocators for NVMM space and per-cpu journals for managing complex metadata updates. This enables parallel block allocation and avoids contention in journaling. In addition, NOVA has per-CPU inode tables to ensure good scalability.

3 Ziggurat Design Overview

Ziggurat is a tiered file system that spans across NVMM and disks (hard or solid state). We design Ziggurat to fully utilize the strengths of NVMM and disks and to offer high file performance for a wide range of access patterns.

Three design principles drive the decisions we made in designing Ziggurat. First, Ziggurat should be *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 [14, 15] have done. Second, Ziggurat strives to be *frugal* by placing and moving data to avoid wasting scarce resources (e.g., NVMM capacity or disk bandwidth). Third, Ziggurat should be *predictive* by dynamically learning the access patterns of a given workload and adapting its data placement decisions to match.

These principles influence all aspects of Ziggurat's design. For instance, being fast-first means, in the common case, file writes go to NVMM. However, Ziggurat will make an exception if it predicts that steering a particular write in NVMM would not help application performance (e.g., if the write is large and asynchronous).

Alternatively, if the writes are small and synchronous (e.g., to a log file), Ziggurat will send them to NVMM initially, detect when the log entries have "cooled", and then aggregate those many small writes into larger, sequential writes to disk.

Ziggurat uses two mechanisms to implement these design principles. The first is a placement policy driven by a pair of predictors that measure and analyze past file access behavior to make predictions about future behavior. The second is an efficient migration mechanism that moves data between tiers to optimize NVMM performance and disk bandwidth. The migration system relies on a simple but effective mechanism to identify cold data to move from NVMM to disk.

We describe Ziggurat in the context of a simple two-tiered system comprising NVMM and an SSD, but Ziggurat can use any block device as the "lower" tier. Ziggurat can also handle more than one block device tier by migrating data blocks across different tiers.

3.1 Design Decisions

We made the following design decisions in Ziggurat to achieve our goals.

Send writes to the most suitable tier Although NVMM is the fastest tier in Ziggurat, file writes should not always go to NVMM. NVMM is best-suited for small updates (since small writes to disk are slow) and synchronous writes (since NVMM has higher bandwidth and lower latency). However, for larger asynchronous writes, targeting disk is faster, since Ziggurat can buffer the data in DRAM more quickly than it can write to NVMM, and the write to disk can occur in the background. Ziggurat uses its *synchronicity predictor* to analyze the sequence of writes to each file and predict whether future accesses are likely to be synchronous (i.e., whether the application will call `fsync` in the near future).

Only migrate cold data in cold files During migration, Ziggurat targets the cold portions of cold files. Hot files and hot data in unevenly-accessed files remain in the faster tier. When the usage of the fast tier is above a threshold, Ziggurat selects files with the earliest average modification time to migrate (Section 5.1). Within each file, Ziggurat migrates blocks that are older than average. Unless the whole file is cold (i.e., its modification time is not recent), in which case we migrate the whole file.

High NVMM space utilization Ziggurat fully utilizes NVMM space to improve performance. Ziggurat uses NVMM to absorb synchronous writes. Ziggurat uses a dynamic migration threshold for NVMM based on the read-write pattern of applications, so it makes the most of NVMM to handle file reads and writes efficiently. We also implement reverse migration (Section 5.2) to migrate data from disk to NVMM when running read-dominated workloads.

Migrate file data in groups In order to maximize the write bandwidth of disks, Ziggurat performs migration to disks as sequentially as possible. The placement policy ensures that most small, random writes go to NVMM. However, migrating these small write entries to disks directly will suffer from the poor random access performance of disks. In order to make migration efficient, Ziggurat coalesces adjacent file data into large chunks for migration to exploit sequential disk bandwidth (Section 5.3).

High scalability Ziggurat extends NOVA's per-cpu storage space allocators to include all the storage tiers. It also uses per-cpu migration and page cache writeback threads to improve scalability.

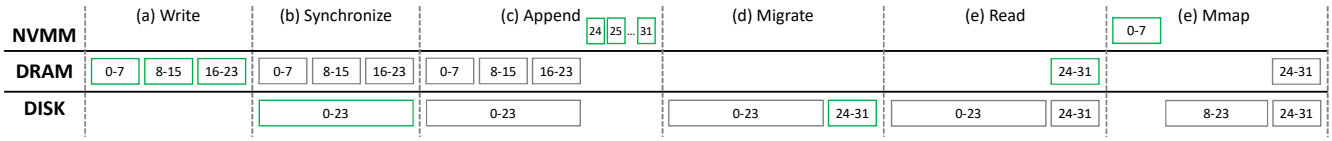


Figure 1: **File operations.** Ziggurat utilizes different storage tiers to handle I/O requests such as write, synchronize, append, migrate, read and mmap efficiently.

3.2 File Operations

Figure 1 illustrates how Ziggurat handles operations on files (write, synchronize, append, migrate, read, and mmap) that span multiple tiers.

Write The application initializes the first 24 blocks of the file with three sequential writes in (a). Ziggurat first checks the results from the synchronicity predictor and the write size predictor (Section 4) to decide which tier should receive the new data. In the example, the three writes are large and Ziggurat predicts that the accesses are asynchronous, so Ziggurat steers these writes to disk. It writes the data to the page cache in DRAM and then asynchronously writes them to disk.

Synchronize The application calls `fsync` in (b). Ziggurat traverses the write log entries of the file, and writes back the dirty data pages in the DRAM page cache. The write-back threads merge all adjacent dirty data pages to perform large sequential writes to disk. If the file data were in NVMM, `fsync` would be a no-op.

Append After the `fsync`, the application performs eight synchronous writes to add eight blocks to the end of the file in (c). The placement predictor recognizes the pattern of small synchronous writes and Ziggurat steers the writes to NVMM.

Migrate When the file becomes cold in (d), Ziggurat evicts the first 24 data pages from DRAM and migrates last eight data blocks from NVMM to disk using group migration (Section 5.3).

Read The user application reads the last eight data blocks in (e). Ziggurat fetches them from disk to DRAM page cache.

Memory map The user application finally issues a `mmap` request to the head of the file in (f). Ziggurat uses reverse migration to bring the data into NVMM and then maps the pages into the application's address space.

4 Placement Policy

Ziggurat steers synchronous or small writes to NVMM, but it steers asynchronous, large writes to disk, because writing to the DRAM page cache is faster than writing to NVMM, and Ziggurat can write to disk in the background. It uses two predictors to distinguish these two types of writes.

Synchronicity predictor The synchronicity predictor predicts whether the application is likely to call `fsync` on the file in the near future. The synchronicity predictor counts the number of data blocks written to the file between two calls to `fsync`. If the number is less than a threshold (e.g., 1024 in our experiments), the predictor classifies it as a synchronously-updated file. The predictor treats writes to files opened with `O_SYNC` as synchronous as well.

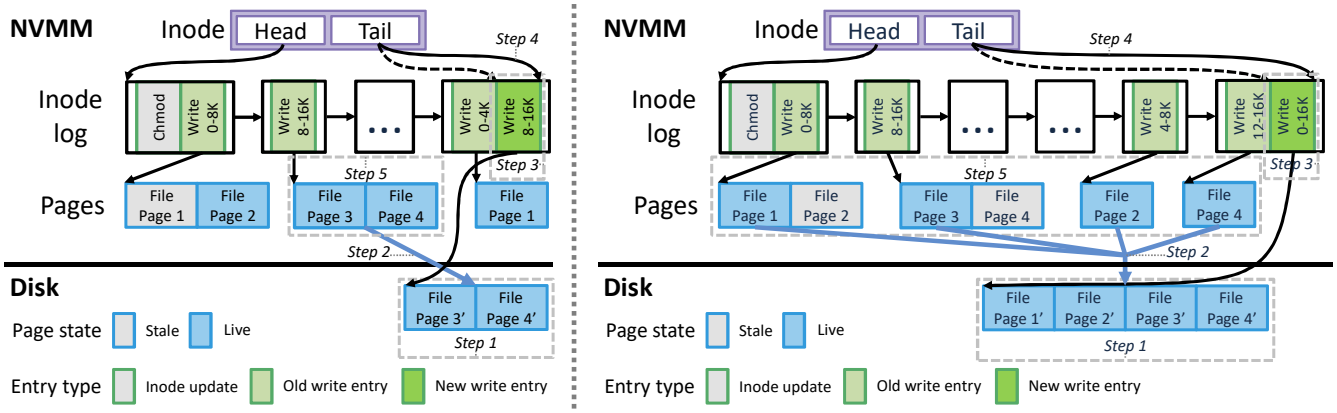
Write size predictor The write size predictor not only ensures that a write is large enough to effectively exploit disk bandwidth but also that the future writes within the same address range are also likely to be large. The second condition is critical. For example, if the application initializes a file with large I/Os, and then performs many small I/Os, these small new write entries will read and invalidate discrete blocks, increasing fragmentation and leading to many random disk accesses to service future reads.

Ziggurat's write size predictor keeps a counter in each write entry to indicate whether the write size is both large and stable. When Ziggurat rewrites an old write entry, it first checks whether the write size is big enough to cover at least half the area taken up by the original log entry. If so, Ziggurat transfers the counter value of the old write entry to the new one and increases it by one. Otherwise, it resets the counter to zero. If the number is larger than four (a tunable parameter), Ziggurat classifies the write as "large". Writes that are both large and asynchronous go to disk.

5 Migration Mechanism

The purpose of migration is to make room in NVMM for incoming file writes, as well as speeding up reads to frequently accessed data. We use *basic migration* to migrate data from disk to NVMM to fully utilize NVMM space when running read-dominated workloads. We use *group migration* to migrate data from NVMM to disk by coalescing adjacent data blocks to achieve high migration efficiency and free up space for future writes. Ziggurat can achieve near-NVMM performance for most accesses as long as the migration mechanism is efficient enough.

In this section, we first describe how Ziggurat identifies good targets for migration. Then, we illustrate how it migrates data efficiently to maximize the bandwidth of the disk



(a) The file structure of Ziggurat and basic migration

(b) Group migration

Figure 2: **Migration mechanism of Ziggurat.** Ziggurat migrates file data between tiers using its basic migration and group migration mechanisms. The blue arrows indicate data movement, while the black ones indicate pointers.

with basic migration and group migration. Finally, we show how to migrate file logs efficiently.

5.1 Migration Profiler

Ziggurat uses a migration profiler to identify cold data to migrate from NVMM to disk.

Implementation. Ziggurat first identifies the cold files to migrate. Ziggurat profiles the temperature of each file by maintaining *cold lists*, the per-cpu lists of files on each storage tier, sorted by the average modification time(`amtime`) computed across all the blocks in the file. The per-cpu cold lists correspond to per-cpu migration threads which migrate files from one tier to another. Ziggurat updates the cold list whenever it modifies a file. To identify the coldest blocks within an cold file, Ziggurat tracks the mtime for each block in the file.

To migrate data, Ziggurat pops the coldest file from a cold list. If the `mtime` of the popped file is not recent (more than 30 seconds ago), then Ziggurat treats the whole file as cold and migrates all of it. Otherwise, the modification time of the file's block will vary, and Ziggurat migrates the write entries with `mtime` earlier than the `amtime` of the file. Hence, the cold part of the file is migrated to a lower tier, and the hot part of the file stays in the original tier.

Deciding when to migrate. Most existing tiered storage systems (such as [6, 21]) use a fixed utilization threshold to decide when to migrate data to lower tiers. However, a higher threshold is not suitable for write-dominated workloads, since the empty space in persistent memory will be devoured by intensive file writes. In this case, the file writes have to either stall before the migration threads clean up enough space in NVMM, or write to disk. On the other hand, a lower threshold is not desirable for read-dominated workloads, since reads have to load more blocks from disks instead of NVMM. We implement a dynamic threshold for NVMM in Ziggurat based on the overall read-write ratio of the file system. The migration threshold rises from 50% to 90% as the read-write ratio of the system changes.

5.2 Basic Migration

The goal of basic migration is to migrate the coldest data in Ziggurat to disk. When the usage of the upper tier is above the threshold, a per-cpu migration thread migrates the coldest data in a cold file to disk. The migration process repeats until the usage of the upper tier is below the threshold again.

The granularity of migration is a write entry. During migration, we traverse the in-DRAM radix tree to locate every valid write entry in the file and migrate the write entries with `mtime` earlier than the `amtime` of the file.

Figure 2a illustrates the basic procedures of how Ziggurat migrates a write entry from NVMM to disk. The first step is to allocate continuous space on disk to hold the migrated data. Ziggurat copies the data from NVMM to disk. Then, it appends a new write entry to the inode log with the new location of the migrated data blocks. After that, it updates the log tail in NVMM and the radix tree in DRAM. Finally, Ziggurat frees the old blocks of NVMM.

To improve scalability, Ziggurat uses locks in the granularity of a write entry instead of an entire file. Ziggurat locks write entries during migration but other parts of the file remain available for reading. Migration does not block file writes. If any foreground file I/O request tries to acquire the inode lock, the migration thread will stop migrating the current file, and release the lock.

If a write entry migrates to a disk when the DRAM page cache usage is low (i.e., below 50%), Ziggurat will make a copy of the pages in the DRAM page cache in order to

accelerate future reads. Writes will benefit from this as well, since unaligned writes have to read the partial blocks from their neighbor write entries to fill the data blocks.

Ziggurat implements reverse migration, which migrates file data from disks to NVMM, using basic migration. Write entries are migrated successively without grouping since NVMM can handle sequential and random writes efficiently. File `mmap` uses reverse migration to enable direct access to persistent data. Reverse migration also optimizes the performance of read-dominated workloads when NVMM usage is low since the performance depends on the size of memory. If Ziggurat can only migrate data from a faster tier to a slower one, then the precious available space of NVMM will stay idle when running read-dominated workloads. Meanwhile, the data on disks contend for a limited DRAM. Reverse migration makes full use of NVMM in such a scenario.

5.3 Group Migration

Group migration avoids fine-grain migration to improve efficiency and maximize sequential bandwidth to disks. Ziggurat tends to fill NVMM with small writes due to its data placement policy. Migrating them from NVMM to disk with basic migration is inefficient because it will incur the high random access latency of disks.

Group migration coalesces small write entries in NVMM into large sequential ones to disk. There are four benefits: (1) It merges small random writes into large sequential writes, which improves the migration efficiency. (2) If the migrated data is read again, loading continuous blocks is much faster than loading scattered blocks around the disk. (3) By merging write entries, the log itself becomes smaller, reducing metadata access overheads. (4) It moderates disk fragmentation caused by log-structured writes by mimicking garbage collection.

As illustrated in Figure 2b, the steps of group migration are similar to migrating a write entry. In step 1, we allocate large chunks of data blocks in the lower tier. In step 2, we copy multiple pages to the lower tier with a single sequential write. After that, we append the log entry, and update the inode log tail, which commits the group migration. The stale pages and logs are freed afterward. Ideally, the *group migration size* (the granularity of group migration) should be set close to the future I/O size, so that applications can fetch file data with one sequential read from disk. In addition, it should not exceed the CPU cache size in order to maximize the performance of loading the write entries from disks.

5.4 File Log Migration

Ziggurat migrates file logs in addition to data when NVMM utilization is too high, freeing up space for hot data and metadata. Ziggurat periodically scans the cold lists, and initiates log migration on cold files. Figure 3 illustrates how log migration is performed. Ziggurat copies live log entries from NVMM into the page cache. The log entries are compacted into new log pages during coping. Then, it writes the new log pages back to disk, and updates the inode metadata cache in DRAM to point to the new log. After that, Ziggurat atomically replaces the old log with the new one and reclaims the old log.

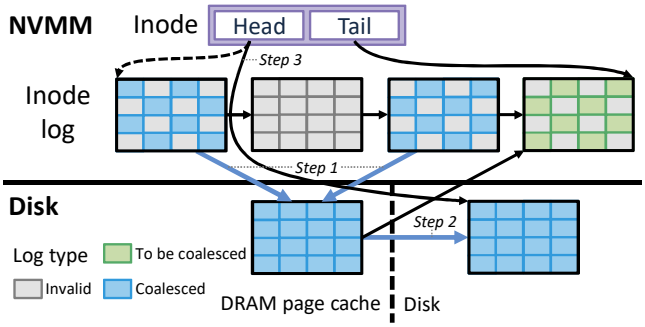


Figure 3: **Log migration in Ziggurat.** Ziggurat compacts logs as it moves them from NVMM to disk.

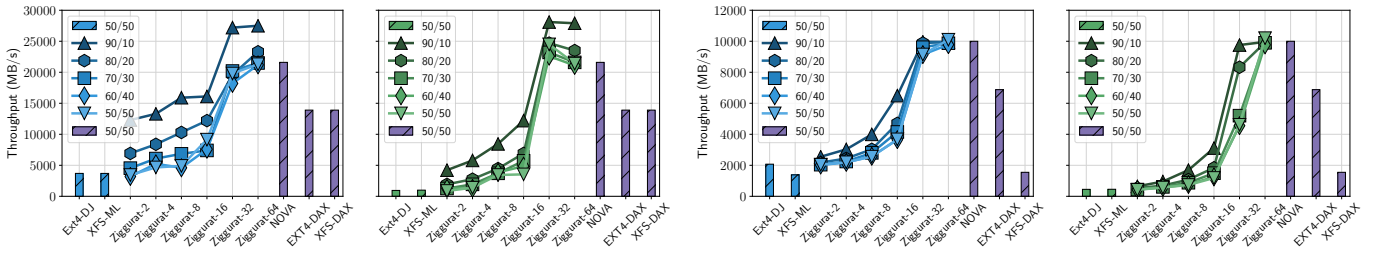
6 Evaluation

6.1 Experimental Setup

Ziggurat is implemented on Linux 4.13. We used NOVA as the code base to build Ziggurat, and added around 8.6k lines of code. To evaluate the performance of Ziggurat, we run micro-benchmarks and macro-benchmarks on a dual-socket Intel Xeon E5 server. Each processor runs at 2.2GHz, has 10 physical cores, and is equipped with 25 MB of L3 cache and 128 GB of DRAM. The server also has a 400 GB Intel DC P3600 NVMe SSD and a 187 GB Intel DC P4800X Optane SSD.

As persistent memory devices are not yet available, we emulate the latency and bandwidth of NVMM with the NUMA effect on DRAM. There are two NUMA nodes in our platform. During the experiments, the entire address space of NUMA node 1 is used for NVMM emulation. All applications are pinned to run on the processors and memory of NUMA node 0. Table 2 shows the DRAM latency of our experimental platform by Intel Memory Latency Checker [16].

We compare Ziggurat with different types of file systems. For NVMM-based file systems, we compare Ziggurat with NOVA [32], Strata [21] (NVMM only) and the DAX-based file systems on Linux: EXT4-DAX and XFS-DAX. For disk-based file systems, we compare Ziggurat with EXT4 in the data journaling mode (-DJ) and XFS in the metadata logging mode (-ML). Both EXT4-DJ and XFS-ML provide data atomicity, like Ziggurat. For EXT4-DJ, the journals are kept in a 2 GB journaling block device (JBD) on NVMM. For XFS-ML, the metadata logging device is 2 GB of NVMM.



(a) Fio read on Optane SSD and NVMM (b) Fio read on NVMe SSD and NVMM (c) Fio write on Optane SSD and NVMM (d) Fio write on NVMe SSD and NVMM

Figure 4: **Fio performance.** Each workload performs 4 KB reads/writes to a hybrid file system backed by NVMM and SSD (EXT4-DJ, XFS-ML and Ziggurat) or an NVMM-only file system (NOVA, EXT4-DAX and XFS-DAX).

Node	0	1
0	76.6	133.7
1	134.2	75.5

(a) NUMA latency (ns)

Node	0	1
0	52213.9	25505.9
1	25487.3	52111.8

(b) NUMA bandwidth (MB/s)

Table 2: **NUMA latency and bandwidth of our platform.** We use the increased latency and reduced bandwidth of the remote NUMA node to emulate the lower performance of NVMM compared to DRAM.

Locality	90/10	80/20	70/30	60/40	50/50
Parameter θ	1.04	0.88	0.71	0.44	0

Table 3: **Zipf Parameters.** We vary the Zipf parameter, θ, to control the amount of locality in the access stream.

We limit the capacity of the DRAM page cache to 10 GB.

For tiered file systems, we only do the comparison among Ziggurat with different configurations. To the best of our knowledge, Strata is the only currently available tiered file system that spans across NVMM and disks. However, the publicly available version of Strata only supports a few applications and has trouble running workloads with dataset sizes larger than NVMM size as well as multi-threaded applications.

We vary the NVMM capacity available to Ziggurat to show how performance changes with different storage configurations. The dataset size of each workload is smaller than 64 GB. The variation starts with Ziggurat-2 (i.e., Ziggurat with 2 GB of NVMM). In this case, most of the data must reside on disk forcing Ziggurat to frequently migrate data to accommodate incoming writes. Ziggurat-2 is also an interesting comparison point for EXT4-DJ and XFS-ML, since those configurations take different approaches to using a small amount of NVMM to improve file system performance. The variation ends with Ziggurat-64 (i.e., Ziggurat with 64 GB of NVMM). The group migration size is set to 16 MB. We run each workload three times and report the average across these runs.

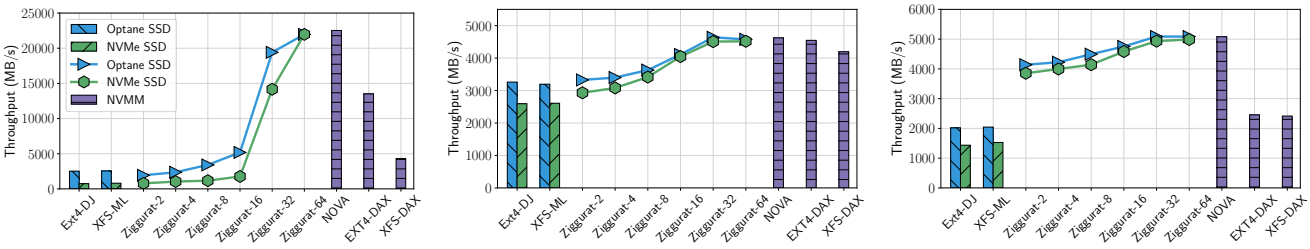
6.2 Microbenchmarks

We demonstrate the relationship between access locality and the read/write throughput of Ziggurat with Fio [1]. Fio can issue random read/write requests according to Zipfian distribution. We vary the Zipf parameter θ to adjust the locality of random accesses. We present the results with a range of localities range from 90/10 (90% of accesses go to 10% of data) to 50/50 (Table 3). We initialize the files with 2 MB writes and the total dataset is 32 GB. We use 20 threads for the experiments, each thread performs 4 KB I/Os to a private file, and all writes are synchronous.

Figure 4 shows the results for Ziggurat, EXT4-DJ, and XFS-ML on Optane SSD and NVMe SSD, as well as NOVA, EXT4-DAX, and XFS-DAX on NVMM. The gaps between the throughputs from Optane SSD and NVMe SSD in both graphs are large because Optane SSD's read/write bandwidth is much higher than the NVMe SSD's. The throughput of Ziggurat-64 is close to NOVA for the 50/50 locality, the performance gap between Ziggurat-64 and NOVA is within 2%. This is because when all the data fits in NVMM, Ziggurat is as fast as NOVA. The throughput of Ziggurat-2 is within 5% of EXT4-DJ and XFS-ML.

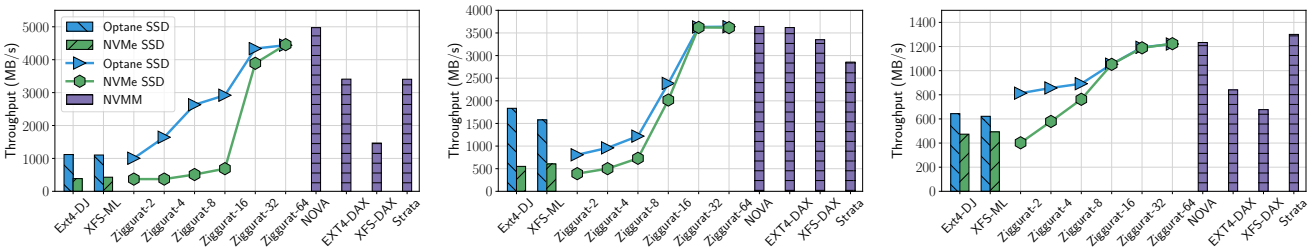
In Figure 4a and Figure 4b, the random read performance of Ziggurat grows with increased locality. The major overhead of reads comes from fetching cold data blocks from disk to DRAM page cache. There is a dramatic performance increase in 90/10, due to CPU caching and the high locality of the workload.

In Figure 4c and Figure 4d, the difference between the random write performance of Ziggurat with different amounts of locality is small. Since all the writes are synchronous 4 KB aligned writes, Ziggurat steers these writes to NVMM. If NVMM is full, Ziggurat writes the new data blocks to the DRAM page cache and then flushes them to disk synchronously. Since the access pattern is random, the migration threads cannot easily merge the discrete data blocks to



(a) Fileserver (b) Webproxy (c) Varmail

Figure 5: **Filebench performance (multi-threaded).** Each workload runs with 20 threads so as to fully show the scalability of the file systems. The performance gaps between Optane SSD and NVMe SSD are smaller than the single-threaded ones.



(a) Fileserver (b) Webproxy (c) Varmail

Figure 6: **Filebench performance (single-threaded).** For small amounts of NVMM, Ziggurat is no slower than conventional file systems running on disk. With large amount of NVMM, performance nearly matches that of NVMM-only file systems.

perform group migration in large sequential writes to disk. Therefore, the migration efficiency is limited by the random write bandwidth of disks, which leads to accumulated cold data blocks in NVMM. Increasing NVMM size, increasing locality, or reducing work set size can all help alleviate this problem.

We also measure the disk throughput of the random write workloads on Ziggurat-2 to show how Ziggurat fully utilizes disk bandwidth to achieve maximum performance. Although it is hard to merge the discrete data blocks to perform group migration, the per-CPU migration threads make full use of the concurrency of disks to achieve high migration efficiency. The average disk write bandwidth of Ziggurat-2 is 1917 MB/s and 438 MB/s for Optane SSD and NVMe SSD, respectively. These values are very close to the bandwidth limit numbers in Table 1.

6.3 Macrobenchmarks

We select three Filebench workloads: fileserver, webproxy, and varmail to evaluate the overall performance of Ziggurat. Table 4 summarizes the characteristics of these workloads.

Figure 5 shows the multi-threaded Filebench throughput on our five comparison file systems and several Ziggurat configurations. In general, we observe that the throughput of Ziggurat-64 is close to NOVA, the performance gap between Ziggurat-64 and NOVA is within 3%. Ziggurat gradually bridges the gap between disk-based file systems and NVMM-based file systems by increasing the NVMM size.

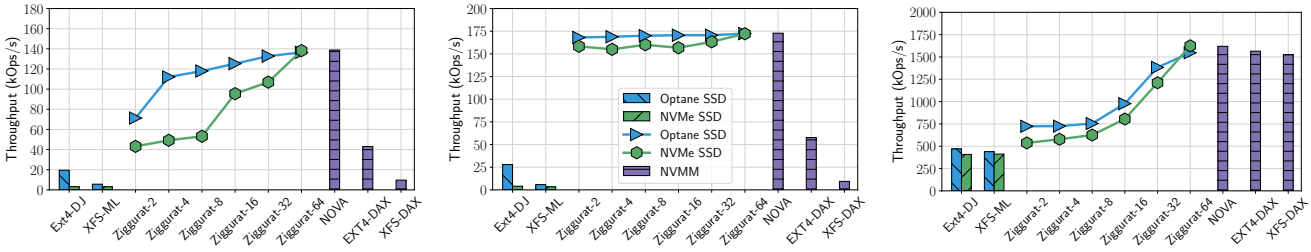
Workload	Average file size	# of files	I/O size (R/W)	Threads	R/W ratio
Fileserver	2MB	16K	16KB/16KB	20/1	1:2
Webproxy	2MB	16K	1MB/16KB	20/1	5:1
Varmail	2MB	16K	1MB/16KB	20/1	1:1

Table 4: **Filebench workload characteristics.** These workloads have different read/write ratios and access patterns.

Fileserver emulates the I/O activity of a simple file server, which consists of creates, deletes, appends, reads and writes. In the fileserver workload, Ziggurat-2 has similar throughput to EXT4-DJ and XFS-ML. The performance increases significantly when the NVMM size is larger than 32 GB since most of the data reside in memory. Ziggurat-64 outperforms EXT4-DAX and XFS-DAX by 2.6× and 5.1×.

Webproxy is a read-intensive workload, which involves appends and repeated reads to files. Therefore, all the file systems achieve high throughputs by utilizing the page cache.

Varmail emulates an email server with frequent synchronous writes. Ziggurat-2 outperforms EXT4-DJ and XFS-ML by 2.1× (Optane SSD) and 2.6× (NVMe SSD)



(a) Random insert (b) Sequential insert (c) Sequential read

Figure 7: **Rocksdb performance.** Ziggurat shows good performance for inserting file data with write-ahead logging, due to the clear distinction between hot and cold files and its migration mechanism.

on average. Varmail performs an `fsync` after every two appends. Ziggurat analyzes these synchronous appends and steers them to NVMM, eliminating the cost of most of the `fsync`s.

Figure 6 illustrates the single-threaded Filebench throughputs. Strata achieves the best throughput in the varmail workload since its digestion skips many temporary durable writes which are superseded by subsequent writes. However, Ziggurat-64 outperforms Strata by 31% and 27% on the fileserver and webproxy workloads due to the inefficiency of reads in Strata.

6.4 Rocksdb

We illustrate the high performance of updating a key-value store with write-ahead logging (WAL) on Ziggurat with Rocksdb [13], a persistent key-value store based on log-structured merge trees (LSM-trees). Every update to RocksDB is written to two places: an in-memory data structure called memtable and a write-ahead log in the file system. When the size of the memtable reaches a threshold, RocksDB writes it back to disk and discards the log.

We select three Rocksdb workloads from db_bench: random insert (FillUniqueRandom), sequential insert (FillSeq), and sequential read (ReadSeq) to evaluate the key-value throughput and migration efficiency of Ziggurat. We set the writes to synchronous mode for a fair comparison. The database size is set to 32 GB.

Figure 7 measures the Rocksdb throughput. In the random insert workload, Ziggurat with 2 GB of NVMM achieves 8.6× and 13.2× better throughput than EXT4-DJ and XFS-ML, respectively. In the sequential insert workload, Ziggurat is able to maintain near-NVMM performance even when there are only 2 GB of NVMM. It achieves up to 38.9× and 46.5× throughput of EXT4-DJ and XFS-ML, respectively.

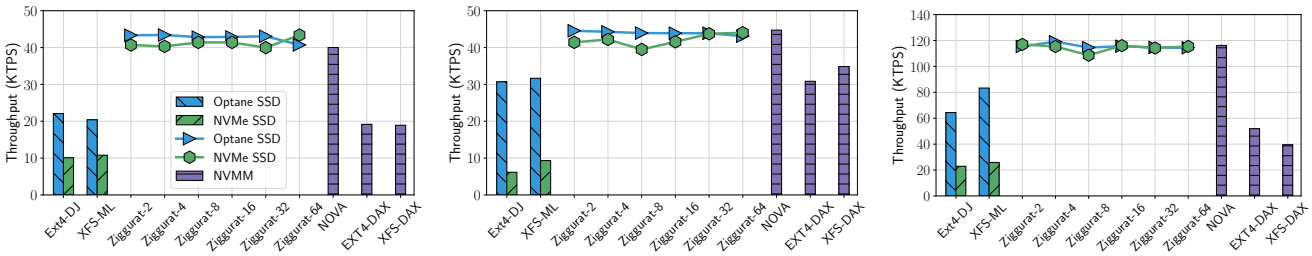
WAL is a good fit for Ziggurat. The reason is three-fold. First, since the workload updates WAL files much more frequently than the database files, the migration profiler can differentiate them easily. The frequently-updated WAL files remain in NVMM, whereas the rarely-updated database files are subject to migration.

Second, the database files are usually larger than the group migration size. Therefore, group migration can fully-utilize the high sequential bandwidth of disks. Moreover, since Rocksdb mostly updates the journal files instead of the large database files, the migration threads can merge the data blocks from the database files and perform sequential writes to disk without interruption. The high migration efficiency helps clean up NVMM space more quickly so that NVMM can absorb more synchronous writes, which in turn boosts the performance.

Third, the WAL files are updated frequently with synchronous and small updates. The synchronicity predictor can accurately identify the synchronous write pattern from the access stream of the WAL files, and the write size predictor can easily discover that the updates to these files are too small to be steered to disk. Therefore, Ziggurat steers the updates to NVMM so that it can eliminate the double copy overhead caused by synchronous writes to disks. Since the entire WAL files are hot, Ziggurat is able to maintain high performance as long as the size of NVMM is larger than the total size of the WAL files, which is only 128 MB in our experiments.

Comparing Figure 7a and Figure 7b, the difference between the results from random and sequential insert of Ziggurat is due to read-modify-writes for unaligned writes. In the random insert workload, the old data blocks of the database files are likely to be on disk, especially when the NVMM size is small. Thus, loading them from disks introduces large overhead. However, in the sequential insert workload, the old data blocks come from recent writes to the files which are likely to be in NVMM. Hence, Ziggurat achieves near-NVMM performance in the sequential insert workload.

In sequential read, Ziggurat-2 outperforms EXT-DJ and XFS-ML by 42.8% and 47.5%. With increasing NVMM size, the performance of Ziggurat gradually increases. The read throughputs of Ziggurat-64, NOVA, EXT4-DAX, and XFS-DAX are close (within 6%).



(a) DELETE (b) PERSIST (c) WAL

Figure 8: **SQLite performance.** Ziggurat maintains near-NVMM performance because the hot journal files are either short-lived or frequently updated, so Ziggurat keeps them in NVMM. Migrating the cold database file with group migration in the background imposes little overhead to foreground file operations.

6.5 SQLite

We analyze the performance of different logging mechanisms on Ziggurat by measuring SQLite [2], a popular lightweight relational database that supports both undo and redo logging. It hosts the entire database in a single file, with other auxiliary files for logging (rollback or write-ahead log). We use Mobibench [18] to test the performance of SQLite with three journaling modes: DELETE, PERSIST and WAL. DELETE and PERSIST are rollback journaling modes. The journal files are deleted at the end of each transaction in DELETE mode. The PERSIST mode foregoes the deletion and instead overwrites the journal header with zeros. The WAL mode uses write-ahead logging for rollback journaling. The database size is set to 32 GB in the experiments. The experimental results are presented in Figure 8.

For DELETE and PERSIST, the journal files are either short-lived or frequently updated. Therefore, they are classified as hot files by the migration profiler of Ziggurat. Hence, Ziggurat only migrates the cold parts of the database files, leaving the journal files in NVMM to absorb frequent updates. The performance gain comes from accurate profiling and high migration efficiency of Ziggurat. With an efficient migration mechanism, Ziggurat can clear up space in NVMM fast enough for in-coming small writes. As a result, Ziggurat maintains near-NVMM performance in all configurations. Compared with block-based file systems running on Optane SSD, Ziggurat achieves 2.0× and 1.4× speedup for DELETE and PERSIST on average, respectively. Furthermore, Ziggurat outperforms block-based file systems running on NVMe SSD by 3.9× and 5.6× for DELETE and PERSIST on average, respectively.

In WAL mode, there are three types of files: the main database files and two temporary files for each database: *WAL* and *SHM*. The *WAL* files are the write-ahead log files, which are hot during key-value insertions. The *SHM* files are the shared-memory files which are used as the index for the WAL files. They are accessed by SQLite via `mmap`.

Ziggurat's profiler keeps these hot files in NVMM. Meanwhile, the cold parts of the large database files migrate to disks in the background. This only introduces very small overhead to the foreground database operations. Therefore, Ziggurat maintains near-NVMM performance even when there's only about 5% of data is actually in NVMM (Ziggurat-2), which outperforms block-based file systems by 1.61× and 4.78×, respectively. Ziggurat also achieves 2.22× and 2.92× higher performance compared with EXT4-DAX and XFS-DAX on average.

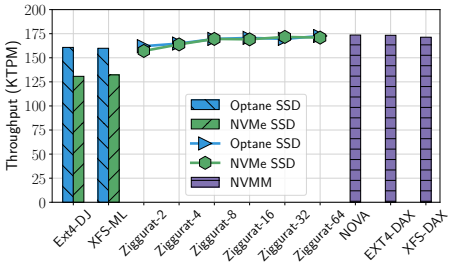
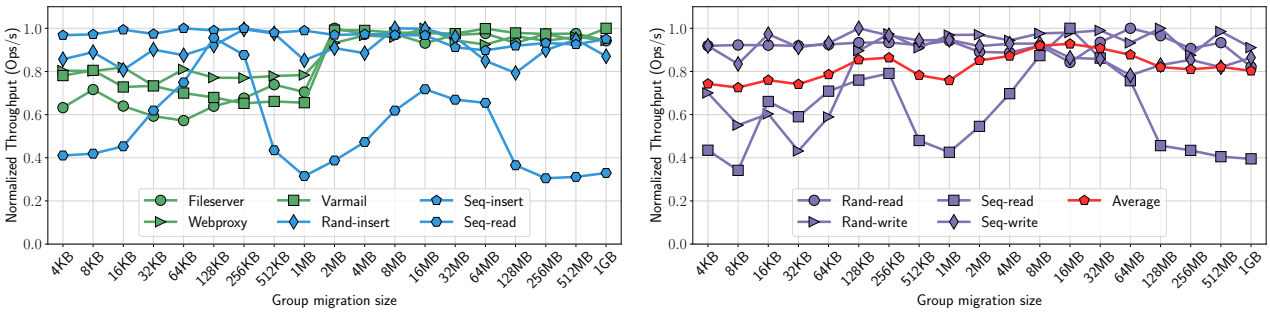


Figure 9: **MySQL performance.** Ziggurat manages to keep high throughput even with little NVMM.

6.6 MySQL

We further evaluate the throughput of databases on Ziggurat with MySQL [27], another widely-used relational database. We measure the throughput of MySQL with TPC-C [3], a representative online transaction processing (OLTP) workload. We run the experiments with a data set size of 20 GB.

Figure 9 shows the MySQL throughput. The performance of Ziggurat is always close to or better than EXT-DJ and XFS-ML. On average, Ziggurat-2 outperforms disk-based file systems by 1% (Optane SSD) and 19% (NVMe SSD). During the transactions, Ziggurat steers most of the small updates to NVMM. Since the transactions need to be processed in DRAM, Ziggurat is capable of migrating the data blocks to disks in time, which leaves ample NVMM space to



(a) Filebench and Rocksdb

(b) Fio

Figure 10: **Performance impact of group migration size.** We use Filebench (green), Rocksdb (blue) and Fio (purple) as our benchmarks. The average throughput (red) peaks when the group migration size is set to 16 MB.

receive new writes. Thus, Ziggurat maintains near-NVMM performance even with little NVMM.

6.7 Parameter Tuning

We illustrate the impact of the parameter choices on performance by measuring the throughput of workloads from Fio, Filebench and Rocksdb with a range of thresholds. We run the workloads with Ziggurat-2 on NVMe SSD.

Group migration size We vary the group migration size from 4 KB to 1 GB. The normalized throughputs relative to the maximum performance are shown in Figure 10. In general, larger group migration size provides better performance for write-dominated workloads, such as Rand-write from Fio and Rand-insert from Rocksdb.

For read-dominated workloads, such as Seq-read from Fio and Seq-read from Rocksdb, the throughputs peak when the group migration size is set to 128 KB and 16 MB. This is because the maximum I/O size of our NVMe SSD is 128 KB and the CPU cache size of our experimental platform is 25 MB. Note that the group migration size is also the granularity of loading file data from disk to DRAM since we fetch file data in the granularity of write entry. On one hand, if the group migration size is too small, Ziggurat has to issue multiple requests to load the on-disk file data into DRAM, which hurts performance. On the other hand, if the group migration size is too large, then a small read request will fetch redundant data blocks from disk, which will waste I/O bandwidth and pollute CPU cache. As Figure 10b shows, the average throughputs of all ten workloads peak when the group migration size is set to 16 MB.

The throughputs of Filebench workloads are saturated when the group migration size reaches 2 MB because the average file size of the workloads is 2 MB. During the migration of the Filebench workloads, the data blocks of a file are coalesced into one write entry, which suits the access pattern of whole-file-reads.

Synchronous write size threshold We vary the synchronous write size threshold from 4 KB to 1 GB. The performance results are insensitive to the synchronous write size threshold throughout the experiments. The standard deviation is less than 3% of the average throughput. We further examine the accuracy of the synchronicity predictor given different synchronous write size thresholds. The predictor accurately predicts the presence or absence of an `fsync` in the near future 99% of the time. The lowest accuracy (97%) occurs when the synchronous write size is set between the average file size and the append size of Varmail. In this case, the first `fsync` contains the writes from file initialization and the first append, while the subsequent `fsyncs` only contain one append. In general, the synchronous write size threshold should be set a little larger than the average I/O size of the synchronous write operations from the workloads. In this case, the synchronicity predictor can not only identify synchronously updated files easily, but also effectively distinguish asynchronous, large writes from rest of the access stream.

Sequential write counter threshold We vary the sequential write counter threshold of Ziggurat from 1 to 64. We find that different sequential write counter thresholds have little impact on performance since the characteristics of our workloads are stable. Users should balance the trade-off between accuracy and prediction overhead when running workloads with unstable access patterns. A higher threshold number improves the accuracy of the sequential predictor, which can effectively avoid jitter in variable workloads. However, it also introduces additional prediction overhead for Ziggurat to produce correct prediction.

7 Related Work

The introduction of multiple storage technologies provides an opportunity of having a large uniform storage space over

a set of different media with varied characteristics. Applications may leverage the diversity of storage choices either directly (e.g. the persistent read cache of RocksDB), or by using NVMM-based file systems (e.g. NOVA, EXT4-DAX or XFS-DAX). In this section, we place Ziggurat's approach to this problem in context relative to other work in this area.

NVMM-Based File Systems. BPFS [8] is a storage class memory (SCM) file system, which is based on shadow-paging. It proposes short-circuit shadow paging to curtail the overheads of shadow-paging in regular cases. However, some I/O operations that involve a large portion of the file system tree (such as moving directories) still impose large overheads. Like BPFS, Ziggurat also exploits fine-grained copy-on-write in all I/O operations.

SCMFS [30] offers simplicity and performance gain by employing the virtual address space to enable continuous file addressing. SCMFS keeps the mapping information of the whole available space in a page table which may be scaled to several Gigabytes for large NVMM. This may result in a significant increase in the number of TLB misses. Although Ziggurat similarly maps all available storage devices into a unified virtual address space, it also performs migration from NVMM to block devices, and group page allocation which reduces TLB misses.

PMFS [11] is another NVMM-based file system which provides atomicity in metadata updates through journaling, but large size write operations are not atomic because it relies on small size in-place atomic updates. Unlike PMFS, Ziggurat's update mechanism is always through journaling with fine-grained copy-on-writes.

Dong *et al.* propose SoupFS [10], a simplified soft update implementation of an NVMM-based file system. They adjust the block-oriented directory organization to use hash tables to leverage the byte-addressability of NVMM. It also gains performance by taking out most synchronous flushes from the critical path. Ziggurat also exploits asynchronous flushes to clear the critical path for higher write throughput.

Tiering Systems. Hierarchical storage Management (HSM) systems date back decades to when disks and tapes were the only common massive storage technologies. There have been several commercial HSM solutions for block-based storage media such as disk drives. IBM Tivoli Storage Manager is one of the well-established HSM systems that transparently migrates rarely used or sufficiently aged files to a lower cost media. EMC DiskXtender is another HSM system with the ability of automatically migrating inactive data from the costly tier to a lower cost media. AutoTiering [34] is another example of a block-based storage management system. It uses a sampling mechanism to estimate the IOPS of running a virtual machine on other tiers. It calculates their performance scores based on the IOPS measurement and the migration costs, and sorts all possible movements accordingly. Once it reaches a threshold, it initiates a live migration.

Since the invention of NVDIMMs, many fine-grained tiering solutions have been introduced. Agarwal *et al.* propose Thermostat [4], a methodology for managing huge pages in two-tiered memory which transparently migrates cold pages to NVMM as the slow memory, and hot pages to DRAM as the fast memory. The downside of this approach is the performance degradation for those applications with uniform temperature across a large portion of the main memory. Conversely, Ziggurat's migration granularity is variable, so it does not hurt performance due to fixed-size migration as in Thermostat. Instead of huge pages, it coalesces adjacent dirty pages into larger chunks for migration to block devices.

X-Mem [12] is a set of software techniques that relies on an off-line profiling mechanism. The X-Mem profiler keeps track of every memory access and traces them to find out the best storage match for every data structure. X-Mem requires users to make several modifications to the source code. Additionally, unlike Ziggurat, the off-line profiling run should be launched for each application before the production run.

Strata [21] 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.

8 Conclusion

We have implemented and described Ziggurat, a tiered file system that spans across NVMM and disks. We manage data placement by accurate and lightweight predictors to steer incoming file writes to the most suitable tier, as well as an efficient migration mechanism that utilizes the different characteristics of storage devices to achieve high migration efficiency. Ziggurat bridges the gap between disk-based storage and NVMM-based storage, and provides high performance and large capacity to applications.

Acknowledgments

We thank our shepherd Kimberly Keeton and anonymous reviewers for their insightful and helpful comments, which improve the paper. The first author is supported by the National Key Research and Development Program of China (No. 2018YFB1003302). A gift from Intel supported parts of this work.

References

[1] Fio: Flexible i/o tester. http://freecode.com/projects/fio.

[2] Sqlite. https://www.sqlite.org/.

[3] Tpc benchmark c. http://www.tpc.org/tpcc/.

[4] AGARWAL, N., AND WENISCH, T. F. Thermostat: Application-transparent page management for two-tiered main memory. *ACM SIGARCH Computer Architecture News 45*, 1 (2017), 631–644.

[5] ARULRAJ, J., PAVLO, A., AND DULLOOR, S. R. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 707–722.

[6] CANO, I., AIYAR, S., ARORA, V., BHATTACHARYYA, M., CHAGANTI, A., CHEAH, C., CHUN, B. N., GUPTA, K., KHOT, V., AND KRISHNAMURTHY, A. Curator: Self-managing storage for enterprise clusters. In *NSDI* (2017), pp. 51–66.

[7] CHEN, E., APALKOV, D., DIAO, Z., DRISKILL-SMITH, A., DRUIST, D., LOTTIS, D., NIKITIN, V., TANG, X., WATTS, S., AND WANG, S. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics 46*, 6 (2010), 1873–1878.

[8] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.

[9] DEBNATH, B., SENGUPTA, S., AND LI, J. Flashstore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1414–1425.

[10] DONG, M., AND CHEN, H. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA* (2017), pp. 719–731.

[11] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 15.

[12] DULLOOR, S. R., ROY, A., ZHAO, Z., SUNDARAM, N., SATISH, N., SANKARAN, R., JACKSON, J., AND SCHWAN, K. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 15.

[13] FACEBOOK. Rocksdb. http://rocksdb.org.

[14] FANG, R., HSIAO, H.-I., HE, B., MOHAN, C., AND WANG, Y. High performance database logging using storage class memory.

[15] HITZ, D., LAU, J., AND MALCOLM, M. A. File system design for an nfs file server appliance. In *USENIX winter* (1994), vol. 94.

[16] INTEL. Intel memory latency checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[17] INTEL. Intel optane technology, 2018. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[18] JEONG, S., LEE, K., HWANG, J., LEE, S., AND WON, Y. Androstep: Android storage performance analysis tool. In *Software Engineering (Workshops)* (2013), vol. 13, pp. 327–340.

[19] KANNAN, S., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., WANG, Y., XU, J., AND PALANI, G. Designing a true direct-access file system with devfs. In *16th USENIX Conference on File and Storage Technologies* (2018), p. 241.

[20] KAWAHARA, T. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Design & Test of Computers 28* (2010), 52–63.

[21] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 460–477.

[22] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 2–13.

[23] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A capacity-optimized ssd cache for primary storage. In *USENIX Annual Technical Conference* (2014), pp. 501–512.

[24] MICRON. 3d-xpoint technology, 2017. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[25] MICRON. Battery-backed nvdimms, 2017. https://www.micron.com/products/dram-modules/nvdimm#/.

[26] NARAYANAN, D., AND HODSON, O. Whole-system persistence. *ACM SIGARCH Computer Architecture News 40*, 1 (2012), 401–410.

[27] ORACLE. Mysql. https://www.mysql.com/.

[28] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News 37*, 3 (2009), 24–33.

[29] WILCOX, M. Add support for nv-dimms to ext4, 2017.

[30] WU, X., AND REDDY, A. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 39.

[31] XU, C., DONG, X., JOUPPI, N. P., AND XIE, Y. Design implications of memristor-based rram cross-point structures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011* (2011), IEEE, pp. 1–6.

[32] XU, J., AND SWANSON, S. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST* (2016), pp. 323–338.

[33] YANG, J. J., STRUKOV, D. B., AND STEWART, D. R. Memristive devices for computing. *Nature nanotechnology 8*, 1 (2013), 13.

[34] YANG, Z., HOSEINZADEH, M., ANDREWS, A., MAYERS, C., EVANS, D. T., BOLT, R. T., BHIMANI, J., MI, N., AND SWANSON, S. Autotiering: automatic data placement manager in multi-tier all-flash datacenter. In *Performance Computing and Communications Conference (IPCCC), 2017 IEEE 36th International* (2017), IEEE, pp. 1–8.

[35] ZHANG, Y., YANG, J., MEMARIPOUR, A., AND SWANSON, S. Mojim: A reliable and highly-available non-volatile memory system. In *ACM SIGARCH Computer Architecture News* (2015), vol. 43, ACM, pp. 3–18.