Mitigating Write-ahead Log Contention on Shared Storage Devices

Lalitha Donga
Rochester Institute of Technology

Kayla Walton
San Jose State University

Fangmin Lyu
Facebook Research

Ben Reed
San Jose State University

Abstract

To preserve durability guarantees, write-ahead logs (WAL) must be written to stable storage before a client can receive a response. Applications with write-ahead logs encourage the use of a dedicated storage device to minimize latency and throughput problems caused by write contention from other processes. Unfortunately, many deployments of applications with write-ahead logs have only a single storage device. It is tempting to think that flash storage will make this problem go away since flash does not have an arm to contend with, but flash still has channel contention. We examine the problem of write-ahead log contention in a replicated system, ZooKeeper. We show how contention is a problem at both the macro-level (for the entire replicated system) and the micro-level (on a single replica). We also show techniques that take advantage of distributed systems and I/O priorities to solve the problem at both levels. While mixing small WAL writes with large writes brought WAL writes latency up to 45 ms, we showed that using separate processes and priorities with mixed writes brings the WAL writes down to 0.2 ms, similar to small WAL writes running on their own. In this paper, we show that by using the proper block scheduler along with dedicated high-priority WAL processes, we can significantly improve performance and achieve consistent WAL performance on a shared storage device.

1 Introduction

Write-ahead logs (WAL) are the key to providing durability guarantees while also achieving low latency. Any changes to the system are first written to the WAL before they are applied to data structures stored elsewhere. Because all changes are written sequentially and batched together, the WAL can optimize storage bandwidth and minimize latency. Once changes to the WAL are synced to storage and made durable, changes to the needed data structures stored elsewhere can be done in the background. If a system failure happens, the WAL is used to recover system state and carry out any data structure changes that did not make it to the storage before the failure.

Most systems that use WALs recommend that they are stored on a dedicated storage device [4, 5]. If both the WAL and the application data is stored on the same device, the background writes and the WAL writes will contend with each other. This contention will increase the latency and throughput of WAL writes and affect system performance.

Historically most of the storage contention overhead was due to a single hard drive arm [29] that needed to seek to locations on the hard drive. This arm contention was a mechanical limitation that was measured in milliseconds. Today, solid state storage is becoming more prevalent and does not have a mechanical arm. It is tempting to think that solid state storage makes the problems of WAL on a non-dedicated storage device a thing of the past, but we will show that there are still other sources of contention in solid state storage that still need to be dealt with to achieve low latency WAL writes.

From a purely academic point of view, requiring a dedicated storage device does not seem onerous. Storage devices are cheap, and system boards support multiple storage devices. Unfortunately, it is a problem in practice. Most storage devices have plenty of capacity, so for most requirements a single storage devices satisfies the storage needs of systems. That is why the volume purchases of most companies are for devices with a single storage device. Of course, it is a small modification to add another storage device for those applications that need it. While the initial modification is small, the impacts on purchasing, setup, provisioning, management, and application deployment are not. Due to these practical limitations, applications that recommend a dedicated storage device for a WAL are often deployed on systems with a single storage device.

In this paper, we examine the problem of write-ahead log contention on a shared device in a replicated system, Zookeeper, by exploring the following research questions:

1. What is the problem of write-ahead log contention at the **macro-level** in Zookeeper?
2. What is the problem of write-ahead log contention at the **micro-level** in Zookeeper?

3. Can we reproduce the problem of write-ahead log contention found in Zookeeper?

4. Can we use priorities to mitigate the problem of write-ahead log contention found in Zookeeper?

5. How much performance improvement do we gain from the use of priorities to mitigate the problem of write-ahead log contention in Zookeeper? Can we achieve consistent WAL performance on a shared storage device?

The work presented here reproduces the problems that result from using a WAL on a shared device. We show that the problems happen at both the operating systems layer, using micro-benchmarks, and the application layer, using benchmarks on ZooKeeper, a distributed system running on five machines. We explore various solutions to the resulting performance problems, and show that simple application of techniques such as IO priorities do not work, but we also show solutions that work by taking advantage of distributed systems as well as low-level solutions that achieve low latency and priority bandwidth on a single system. In our experiments, we consistently use machines with a single storage device. Some of our experiments examine the WAL performance in isolation, but we do not run experiments with multiple drives since that doubles the I/O bandwidth as well as provides isolation, which muddies the result.

To summarize, our main contributions in this paper are:

1. **Reproducing contention.** We run a benchmark that generates small durable WAL writes in the presence of large background writes to reproduce contention at the local level. We identify the latency spikes in this scenario.

2. **Testing priorities to mitigate contention.** We hypothesize that using priorities will mitigate the problem of contention at the local level. To validate our approach, we run a benchmark using this technique. *ioprio_set()* is used to boost the I/O priority of the thread or process containing small durable WAL writes.

3. **Discoveries about priorities.** Experimentation showed that although *ioprio_set()* has a thread parameter, we need to use processes to set the priority correctly, as the system call is likely running on a process level. Other conditions were also discovered to ensure that priorities are set correctly.

4. **Successfully mitigating contention.** Results show significant performance improvement and consistent latency after setting small durable WAL writes as high priority in the presence of large background writes. Because of this, we developed a Java library for integrating applications that use WALs.

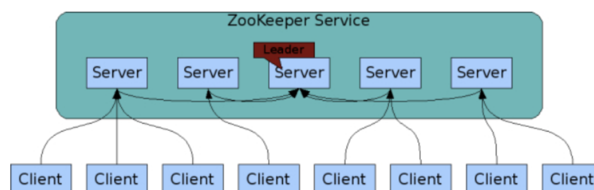

Figure 1: ZooKeeper Architecture

2 Problem Background

Production performance problems of ZooKeeper motivated this work. ZooKeeper is a popular open source distributed coordination service. The coordination data is maintained as an in-memory data tree that is replicated across a quorum of replicas. It provides strong consistency and durability guarantees that distributed applications build upon.

ZooKeeper achieves its durability guarantee with good performance by using a WAL. When ZooKeeper processes a change to the coordination state, the state change will be replicated to a quorum of replicas and persisted on non-volatile storage before the change is applied to the coordination state.

ZooKeeper replicates state to all the replicas of an ensemble. As shown in figure 1, the replicas will elect a leader to coordinate changes to the replicated data. The leader distributes the replica data and will let replicas know when they can commit the data once a quorum has persisted a change.

In general, the latency of the small writes can be significantly impacted when stuck behind a large, unimportant write or snapshot activity. This is especially true with systems like Zookeeper, where the writes being sent to nodes are FIFO ordered [15]. ZooKeeper, like most other applications that use a WAL, recommends that the WAL resides on a dedicated storage devices so that other traffic will not affect its write latency. Some of that other traffic will come from ZooKeeper itself. In addition to the WAL writes, a ZooKeeper replica periodically writes a local snapshot of its current state to storage to avoid replaying a long list of transactions from the WAL during recovery. The snapshot also allows us to periodically truncate WAL so that the WAL does not grow without bounds. Because ZooKeeper state can get quite large, on the order of gigabytes, the periodic snapshot can create a large write load on the storage device. However, snapshots can happen in the background and do not cause any delays in processing. If a failure happens while writing a snapshot, a previously written snapshot can be used. ZooKeeper makes sure to keep a few older snapshots and enough WALs around to recover from if the latest snapshot is not usable. While snapshots are not latency sensitive and can happen in the background, the load they create can affect the latency of WAL writes if both the WAL and snapshots are written to the same storage device.

ZooKeeper tries to takes advantage of the distributed nature of its replicas to mitigate the effects of snapshot and WAL

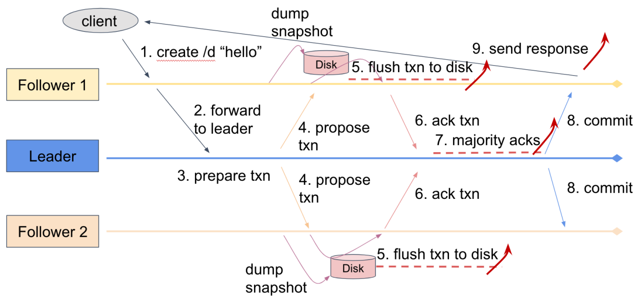

Figure 2: Write request flow

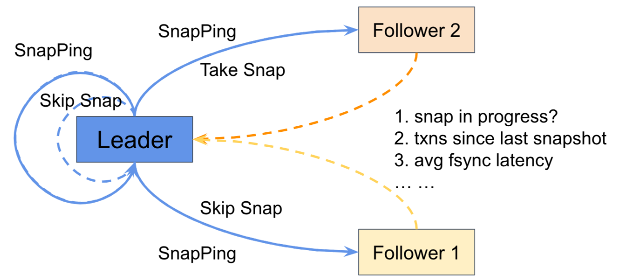

Figure 3: Snapshot scheduler

contention by randomly choosing when to take a snapshot. Even if a snapshot does increase the latency of a WAL write on the local replica, as long as a quorum of replicas is not taking a snapshot at the same time, the latency to accept a change should not be affected.

3 Snapshot scheduling to improve WAL latency

We realized that the existing randomized snapshot scheduling in ZooKeeper was insufficient to mitigate the problem of contention between snapshot and WAL writes. We need to develop something better. When we started deploying servers with flash devices, we thought our storage latency problems would be over. The latency of flash is so much lower than spinning hard drives, and we do not have to deal with arm contention. We knew that there could be some latency spikes while snapshots were happening, but we also had randomized snapshot scheduling helping us out.

Much to our dismay, and that of the applications using ZooKeeper, even when we moved to flash our latency spikes did not go away. In production, we noticed that write load of snapshots caused a significant increase in write latency and a corresponding drop in ZooKeeper performance. We were getting channel contention on the flash. The snapshots were large enough and thus lasted long enough that there was a quorum of replicas taking a snapshot at the same time often enough to affect the performance of our production systems.

3.1 Write request flows

To understand the problem and our solution, we will quickly review the flow of a write request in ZooKeeper shown in figure 2. When a client sends a write request, the replica the client is connected to will forward the request to leader. The leader will convert that request into an idempotent state change, which we call a transaction, or txn in the figure. The leader will send a PROPOSAL with txn to all replicas. When the replica receives the PROPOSAL, it will persist txn to the WAL and send an ACK to the leader. Once the leader has received a quorum of ACKs, it will send a COMMIT to all of the replicas. When the replicas receive the COMMIT, they will apply txn to the in-memory data tree and the replica connected to the client will send a response back to the client. Since snapshot and txns are writing to the same storage device, one disk IO will affect another. When a quorum of replicas are taking snapshots at the same time, the quorum transaction write latency will go up. As the snapshot size grows, it is more likely that the majority of replicas will take a snapshot at the same time even with random snapshot scheduling.

To stabilize the client write latency, we created the snapshot scheduler to coordinate the snapshot and prevent the majority from taking snapshot at the same time.

As shown in figure 3, rather than each replica independently and randomly selecting when to schedule a snapshot, the leader coordinates the scheduling of snapshots. The leader will periodically ping its followers to tell them when to take a snapshot. The followers provide information about any ongoing snapshot they have, transactions they have processed since the last snapshot, and metrics such as their average fsync latency.

3.2 Resulting Improvements

We saw a big improvement in our production performance due to the snapshot scheduler. To measure the change we designed a benchmark that creates a steady stream of write requests. We measured the resulting throughput and latency. The benchmark is done on a cross data center ensemble with five servers, each server having 18 Cores CPU, 64GB RAM and 256GB SSD.

Figure 4 shows the throughput as the data tree size increases. As the graph shows, this problem is real; at 3GB our throughput is less than a quarter of what it was at 100MB. The graph also shows that our scheduling made the throughput decrease only slightly due to a larger snapshot.

Figure 5 shows the latency improvements with snapshot

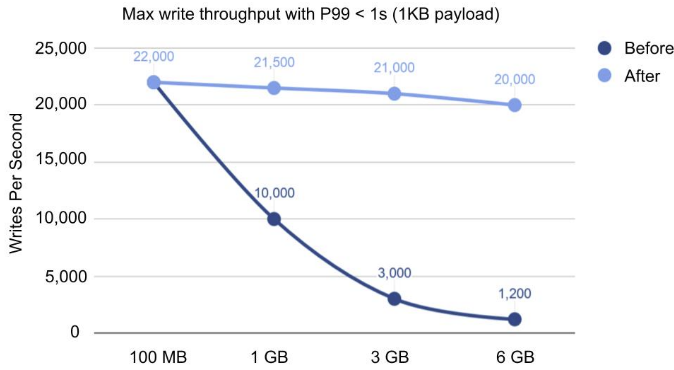

Figure 4: Throughput with snapshot scheduler

scheduling. The y-axis shows the maximum latency of a ZooKeeper write over 1 minute intervals. The spikes are quite high, with some writes taking over 3 seconds to complete. At time 14:45 we enable the snapshot scheduling. We see that the write latency decreases dramatically. All writes are well under 200ms.

This benchmark not only shows the problems that are caused, but it also shows that the snapshot scheduler was able to mitigate the problems. The benchmark also shows that the writes generated by snapshots are clearly affecting the performance of the WAL writes.

4 Mitigating Write-ahead Log Contention Locally

From our previous experiments, we know that WAL write contention is a problem, and we know that we can mitigate it in a distributed environment by scheduling snapshots. On the replicas that are doing a snapshot, contention is still happening. Can we mitigate that contention?

We hypothesized that we could achieve the WAL write performance close to that of a dedicated storage device if we could set the write priorities correctly. The performance of non-WAL writes would be affected by this approach, but those writes are background writes, so it is okay if they are delayed.

Our plan is conceptually simple, as shown in the left side of figure 6. We use priorities to give preference to WAL writes. Priorities allow us to get our important request quickly to the front of the I/O queue, so that important requests do not get delayed.

We are using NVMe flash; These flash devices have the notion of multiple queues, as shown in the right side of figure 6. When the device queues are full, requests are queued in the OS according using the I/O queue. Ideally, it would be nice to have important requests jump to the start of the device queues, but getting to the front of the operating system I/O queue is sufficient since the latency of the device queues are very low.

In theory, if WAL writes always have highest priority, they should get close to the latency that they would receive on a dedicated device. This is challenging because WAL writes often come from the same application that is generating the bulk background writes so it was not as simple as running the application with a system call named *ionice* [20], which sets the priority by application. So instead, we will be using *ioprio_set()* [21], a system call which seems to set the priority by threads or processes.

4.1 Measuring Latency

We started off by creating experiments that model WAL writes as small synchronous writes (4KiB) and background writes as large writes (536MiB) in C. We used an eight core AMD Ryzen 7 CPU with 32G RAM running Linux 5.8. We used a Samsung SSD 860 as our storage device.

In our first experiment, we ran continuous WAL writes by themselves on the SSD for 60 seconds. From this experiment, we see that small writes running on their own achieve very low latency, for an average between 0.05 ms to 0.15 ms, as shown in figure 7. This is expected because due to the nature of its size, small writes won't take up much time. Thus, no write gets stuck behind, resulting in low latency.

However, if large background traffic is mixed in with small writes, then the latency is likely to increase because large writes on its own are expected to take a very long time, resulting in the small writes getting stuck behind large writes, creating contention that we typically see in production. To confirm this, we ran the benchmark with the large background writes. Large writes running on their own achieve high latency, for an average of around 450 ms. The large writes are cached for a while, causing them to spike quickly from around 250 ms to 450 ms.

We reproduced the contention behavior that we see in production by running the large writes while concurrently doing the small WAL writes for 60 seconds. In figure 8, we can see the contention issue when small writes run with large background writes present. The orange line in our graph represents the rolling mean. For the first 10 seconds of the experiment, we only ran small writes. From the graph, we see that before the large writes enter, small writes achieve low latency, similar to small writes running on their own as seen in figure 7. After the 10 seconds, we introduce large writes in with the small writes. Immediately, we see that the latency of small writes began to increase once the large writes started, resulting in a much higher average latency: around 45 ms. We ended the large writes 10 seconds early in our program, and we can see the small writes go back to a low latency. This showed how drastic the contention problem is when small writes are mixed in with large writes. (These were the results using a NVMe storage device. Similar results with numbers

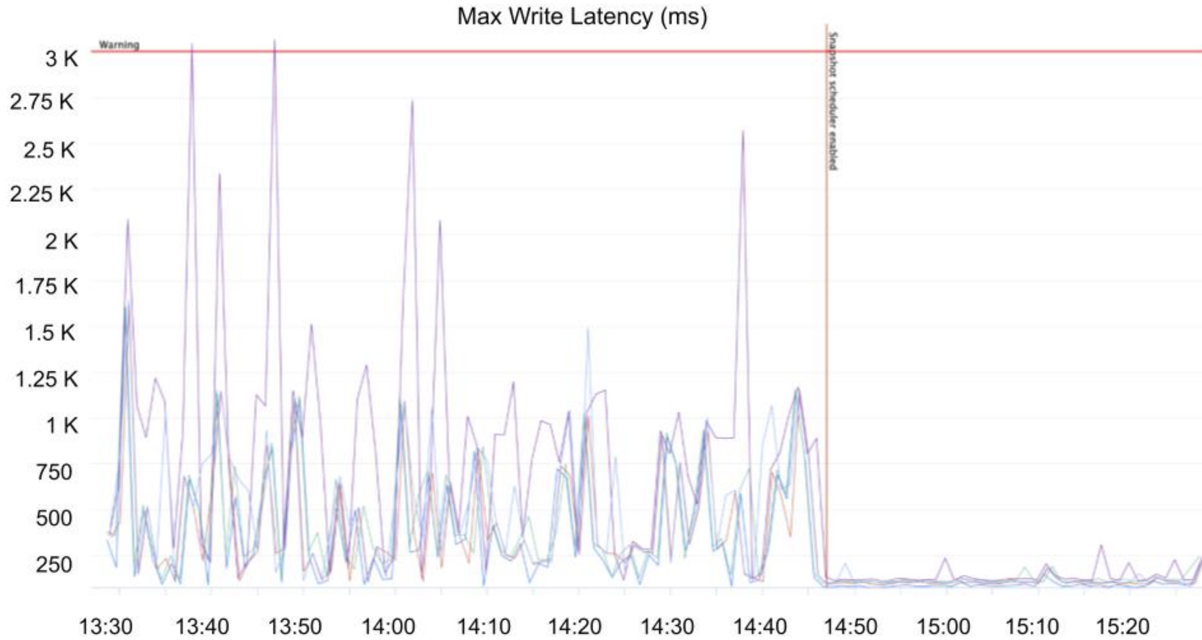

Figure 5: Latency improvement with snapshot scheduler. The verticle yellow light represents the moment that the snapshot scheduler is turned on.

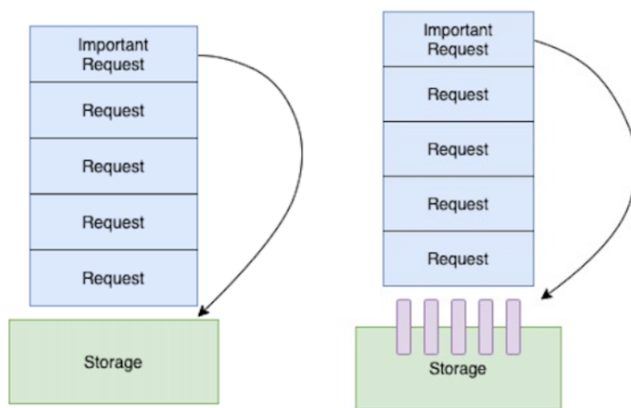

Figure 6: Priorities vs. Priorities with Multiple Queues

of different magnitudes were seen on SSDs that we tested on).

4.2 Using priorities to get deterministic latency

We wanted to set our WAL writes as the highest priority so that when the WAL (small writes) and background writes (large writes) run together, the WAL writes achieve a consistent latency, an average which is very close to or equivalent to small writes running alone (0.05 to 0.15 ms). Our first attempt used the system call *ioprio_set()* to boost the I/O priority of the WAL thread. *ioprio_set()* and *ioprio_get()* are Linux system calls that sets or gets the I/O scheduling class and priority [21].

Since WAL writes often come from the same application that is generating the bulk background writes, we did not use the system call *ionice*, which sets priority by application. *ioprio_set()* and *ioprio_get()* instead sets priority by threads or processes, which is more ideal for our experiment with WAL.

IOPRO_PRIO_CLASS(mask) is an ioprio macro where given a mask (an *ioprio* value), it returns its I/O class component, that is, one of the values *IOPRIO_PRIO_RT (highest priority class), IOPRIO_CLASS_BE, or IOPRIO_CLASS_IDLE (lowest priority class)* [21]. For our experiment, the *IOPRO_PRIO_CLASS* was set to *IOPRIO_PRIO_RT* at 0 (also referred to as RT/0), which is the highest priority class.

The priority being set was additionally verified using *iotop*, a simple I/O monitor which watches I/O usage information

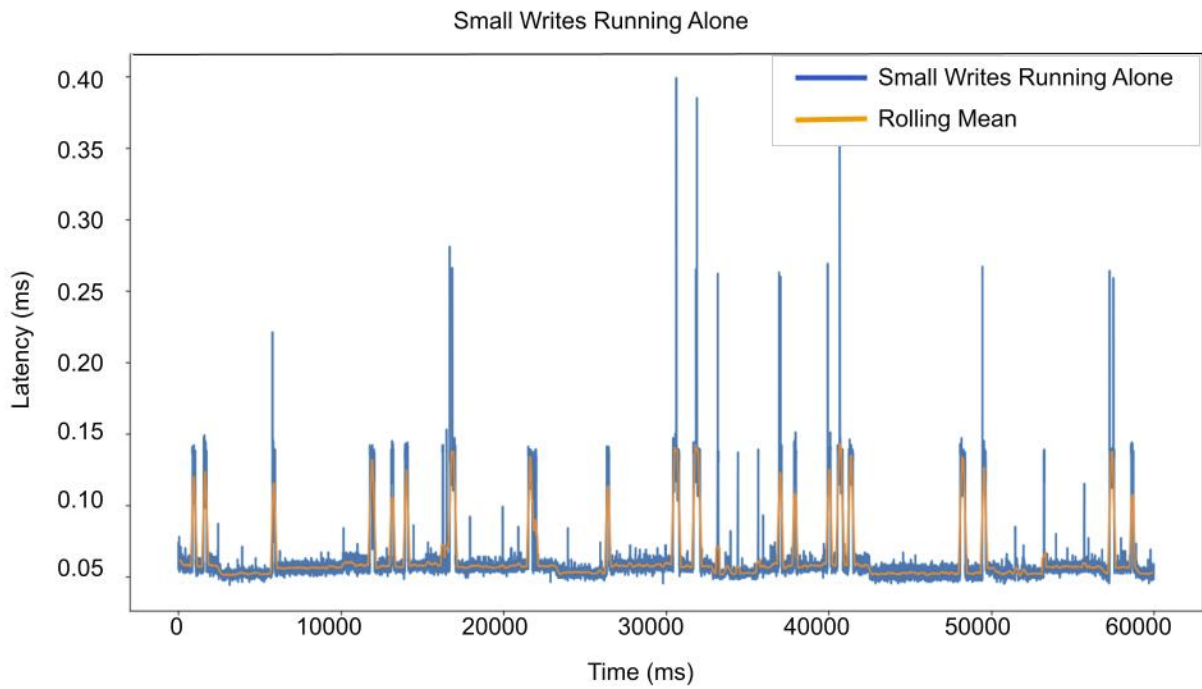

Figure 7: Small Writes Running Alone

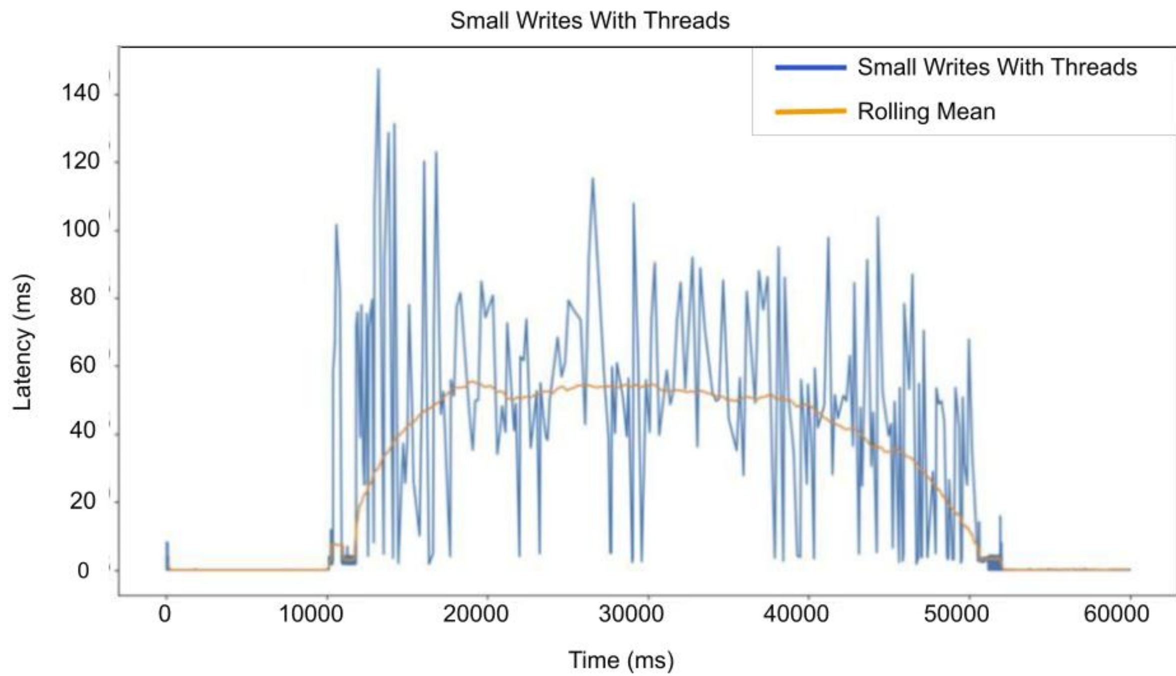

Figure 8: Small Writes Contention with Large Writes

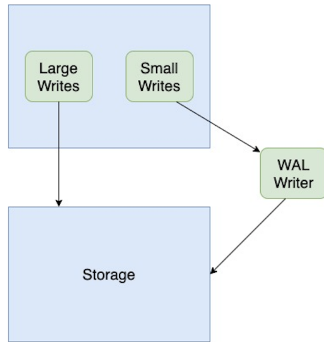

Figure 9: High Priority Experiment

output by the Linux kernel and displays a table of current I/O usage by processes or threads on the system [22]. For each process, its I/O priority (class/level) is shown and we were able to successfully find I/O priority RT/0 when mixed writes were ran.

Unfortunately, simply calling *ioprio_set()* on a WAL thread did initially not change the latency results. In diagnosing the absence of performance change, we made a few discoveries about priorities. First, although *ioprio_set()* has a thread parameter, it is running at a process level, so any change to I/O priority will affect the entire process. We need to use a separate process to do WAL writes so that we can set its I/O priority separately from the other writes of the application. Second, we need to use direct I/O to make sure the writes are done with the correct priority; this also meant that we needed to use aligned memory for the writes. Third, not all block schedulers honor priorities; we found that we needed to use either the CFQ or BFQ scheduler which recognize I/O priorities.

Currently, CFQ supports basic I/O priorities. We did not have access to CFQ on our machine, however, we did have BFQ. Although CFQ is the recommended scheduler [21] to use when using *ioprio_set()*, BFQ also supports I/O priorities and ended up yielding the expected results.

We are not sure what optimizations CFQ provides in comparison. CFQ places synchronous requests submitted by processes into a number of per-process queues and then allocates timeslices for each of the queues to access the disk. BFQ grants exclusive access to the device, for a while, to one queue (process) at a time, and implements this service model by associating every queue with a budget, measured in a number of sectors. BFQ may idle the device for a short time interval, giving the process the chance to go on being served if it issues a new request in time. CFQ does not idle the device. If BFQ is not yielding proper results on your machine, you can turn off idling on your scheduler.

Schedulers have tunable parameters, and we saw that BFQ and CFQ have almost the same tunable parameters. We can try to get BFQ to run like CFQ by setting slice idle = 0 and then rebooting the machine. When rebooting, it may be necessary to switch the scheduler back to CFQ or BFQ. The tunable parameters can be found in the /etc/sysctl.conf file. Latency can also be adjusted. Whether this makes a difference depends on the system. Since BFQ has low throughput, one should see overall improved throughput on faster storage devices like multiple SATA / SAS disks in hardware RAID config, as well as flash based storage with internal command queueing (and parallelism).

The *O_DIRECT* flag may impose alignment restrictions on the length and address of user-space buffers and the file offset of I/Os. To fix our alignment issues, allocating aligned memory was needed. This can be done using *posix_memalign()* or *aligned_malloc()* instead of *malloc()*. For our experiment, *aligned_malloc()* with a 4k offset was used. Any offset should work. Data alignment will increase the system's performance due to the way the CPU handles memory.

We were able to pull all of these discoveries together by creating a C program that does high priority writes to a file using direct I/O and *ioprio_set()* before sending the writes to storage, as we saw in figure 9. The application process that needs to do WAL writes will *fork()* and *exec()* the WAL writer and communicate data to be written through a pipe.

After these changes, we ran the WAL writes and large write benchmark again. We were able to obtain the similar consistent performance for WAL writes as we did when the WAL writes were running on their own (around .2 ms average), as shown in figure 10. The latency did increase slightly but nothing like the increase without our change.

In figure 11, we combined all of our results into one to show the latency improvement. The blue line on top are the latency's of small writes mixed with large background writes without priority (and using threads). The red line is a rolling mean of those latency's, which is around 45 ms. Along the bottom you can see our baseline, the small writes by themselves, along with another line showing the mixed writes using separate processes and priorities. All of the metrics except "Small Writes with Threads" and "Threads Rolling Mean" are all clustered together in the horizontal line at the very bottom of the graph. The two lines with their averages are there, showing around 0.2 ms, but we got them so close together the performance difference can't be seen in this graph. From the graph, we see that high priority small writes mixed with large writes run just as fast as small writes running on their own. Thus, creating separate processes and adding priorities created a drastic performance improvement.

4.3 Solving the Local Write Latency Problem

After seeing latency improvement in our experiment, we wanted to work on Java and C libraries for integrating applications that use WALs (key/value stores, databases, etc). Our first target is Apache ZooKeeper, a Java coordination

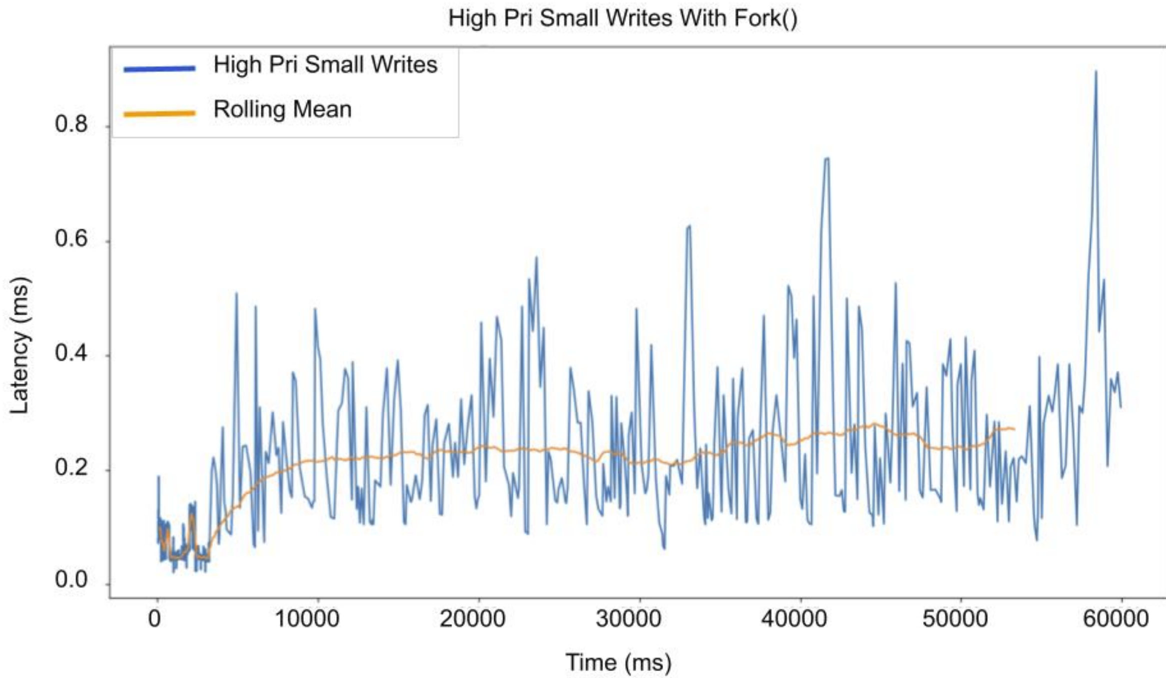

Figure 10: Consistent WAL Writes Using High Priority. High priority small writes in a forked process and normal priority large writes in the parent.

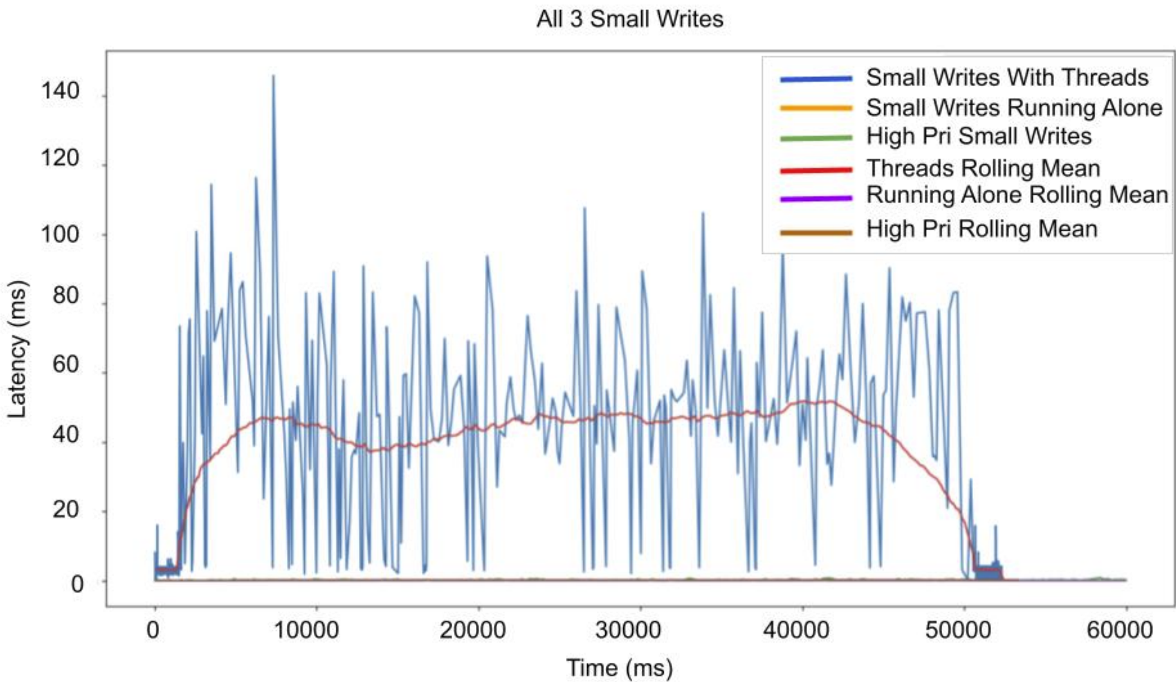

Figure 11: Latency Improvement with High Priority Small Writes. The metrics for the small writes with high-priority processes and small writes alone are all clustered at the very bottom.

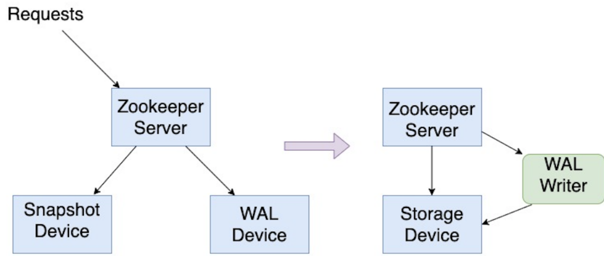

Figure 12: Zookeeper Performance Solution. The Zookeeper Server is split into two processes: the Zookeeper Server and the WAL Writer that runs with high priority.

service.

We developed a Java library that starts a WAL writer and sends WAL writes of various sizes through a pipe, as shown in figure 12. The WAL writer uses *ioprio_set()* to set its writes as high priority. The WAL writer uses direct I/O to do the high priority writes, so it ensures that writes are 4K aligned. When the writer receives writes that are not multiples of 4K, it will buffer old writes to always be able to write 4K aligned data.

ZooKeeper sends the number of bytes to write followed by the bytes through the pipe. the WAL writer sends back the last position written once a write has completed. We implemented a subclass of FileOutputStream that does the `exec` and implements the protocol so that we only needed to change a few lines of the ZooKeeper code. ZooKeeper still reads old logs and writes snapshots using the normal Java I/O classes. We only changed the code that implements the WAL.

After all of the high-priority WAL writes are written, they are sent straight to the storage device. With this, we are able to improve the performance of WAL writes in the presence of snapshots.

Our main target are applications that use WALs (key/value stores, databases, etc). However, our solution can be applied to any application that does a mix of latency sensitive writes and bulk writes.

5 Discussion

Although we show that high-priority writes done using a separate process can be used to lower latency, the approach is not optimal or convenient. The extra memory copies the result using a pipe between processes. This could be made more efficient using shared memory, but that would require an even more complicated implementation. Managing a separate process requires dealing with corner error cases like the logging program being in the correct location, the setuid bit and ownership setup correctly, abnormal process termination, etc. In theory, these complications could be addressed by making *ioprio_set()* work on the thread level rather than setting the priority for the whole process. However, is a process or thread the correct entity to attach the priority? For scheduling priorities, attaching scheduling priority to a thread is natural: the priority attaches to the thing being scheduled. We would argue that the natural entity for attaching I/O priorities is file descriptors. Our implementation would be much simpler and efficient if there was an ioctl to set the I/O priority of just the file descriptor of the transaction log.

ZooKeeper is an in-memory database, but the split between latency sensitive writes and non-latency sensitive bulk writes also happen in other database applications as well.

LSM trees, which we talked about in section 6.2, use a WAL to record recent transactions. The periodic merges done in the background can generate a large number of writes that do not need low latency but can interfere with its own WAL writes. We would like to apply these techniques to applications using RocksDB [24].

Classic database applications also need low latency WAL writes to maintain their durability guarantees with good performance. The updates to their data storage structures, such as B-trees, are generally written to storage buffers that are later flushed to the storage. While these structures generally avoid the large bursts of writes generated by ZooKeeper's snapshots and LSM's compactions, they can cause short bursts of contention on the storage device and adversely affect the latency of the WAL write if the WAL is not on a dedicated device.

One big advantage of the techniques presented here is that they work with mainstream Linux kernels and schedulers. Applications can use these techniques now. This work shows that there is a need for accommodating the different I/O latency needs of applications. Enhancing the kernel file systems interface to convey these needs to the kernel would make this functionality much easier and more convenient to use.

6 Related Work

Improvements to CPUs, hardware, and I/O schedulers already in place have all led to performance gains for logging. There has been work done in refactoring databases like Scylla (a rewrite of Cassandra) [3] or MongoDB [9], among many others, and file systems like Ceph [1] to take advantage of the existing logging improvements.

Recent improvements in the performance of storage devices come from improvements to the kernel I/O subsystems, how applications use I/O, and major improvements in the storage hardware.

6.1 IO Schedulers

I/O scheduling has always been an exceedingly difficult problem to solve and has often required users to choose the algorithm that best fits their needs and workloads. Originally,

gaining access into the block layer was the biggest bottleneck of the system due to the single request queue. As the technology in block devices improved, so did the schedulers and drivers that made use of them [8]. Currently, Linux has several different algorithms for both multi-queue and single-queue scheduling, depending on kernel version being used.

CFQ, deadline, and noop are single-core algorithms that don't scale well with multi-core applications. CFQ [19] is "completely fair queuing" that provides predetermined queues for synchronous and asynchronous requests while recognizing ionice and priority for processes. Deadline is used when latency performance is favored over throughput, serving read and write queues based on an algorithm that weights how long a request waits in queue. Noop serves I/O in a simple FIFO queue and is often used by SSDs that use their own block layer or I/O frameworks [1, 3] that aim to control the data path more fully [8]. CFQ was the default scheduler prior to Linux 5 [2].

In multi-queue scheduling, the tasks are split among multiple software queues that are then merged into hardware queues in order to take advantage of multiple CPU cores. The three common algorithms are BFQ, mq-deadline, and none. None is the multi-queue version of noop, providing a simple FIFO queue for situations like NVMe devices or I/O frameworks [7]. In addition to none, the mq-deadline scheduler is similar to deadline in functionality, but optimized for write requests from multiple cores. Finally, budget-fair queuing (BFQ) follows CFQ design and while BFQ is good for slow devices, bad for fast devices, and has a higher overhead than mq-deadline [31], it is the only modern scheduler that recognizes ionice and has seen a fair amount of interest in improving performance for this reason [30]. CFQ and BFQ both support priorities, and due to this, we will be using BFQ in our application because we did not have a machine with CFQ available.

6.2 Log-Structured Merge-Trees

Atomicity, Consistency, Isolation, and Durability (ACID properties) are well-known to be fundamental to relational database systems, with transaction logging serving the purpose of protecting transaction integrity and accuracy. The drawback to transaction logging is bigger performance limitations when scaling for larger workloads and their associated traffic [25].

NoSQL database systems have been developed to scale horizontally and still maintain performance; these key properties have come to be known as BASE (Basically Available, Soft State, Eventual consistency). However, the logging methods used in NoSQL systems still come with throughput limitations from maintaining their durability guarantees [10]. Log-Structured Merge-Trees (LSM-trees) are one such method.

LSM-trees are data structures that keep recent changes in-memory but writes them to a WAL for recoverability. They were introduced to reduce the load on in-memory data structures to keep transaction logs, although they do not keep time-series data like traditional transaction logs [26]. While valuable, LSM-trees suffer from significant write-throughput performance reduction due to compaction activity and hot zones that trickle down to the commit log devices [6, 33]. Periodically, LSM-trees write sorted changes from memory into sorted key/value files, while also periodically merging those files. The merging creates a large amount of background traffic that contends with the WAL writes which need low latency performance [6, 33]. Since this behavior matches the applications we are targeting, LSM-trees should also benefit from our application.

6.3 General Computational Improvements

Concurrency has enabled databases to improve performance by interleaving transaction executions and reduce the overall idle time. Multi-core processors give the ability to leverage parallelism - running multiple concurrent processes at once with asynchronous writes seeing the most benefit. Logging relies on synchronization to ensure that the important writes get written to disk as quickly as possible to guarantee durability [10]. An increase in parallelism has a negative impact on the ability to control synchronization due to the lack of control over write ordering [27]. Some system frameworks like Seastar [3], BlueStore [1] and storage engines like WiredTiger [9] have focused on multi-core scaling and optimizations for concurrency while enabling durability at scale; these optimizations can require significant refactoring to existing code bases.

At some point, the amount of writes being sent to storage will introduce a bottleneck. Ideally, the bottleneck created by processing transactions on the CPU gets shifted to somewhere else in the system - typically to the disk - to allow the CPU to continue to do more work. Many implementations have been successful in improving concurrency, but generally require sacrificing some aspect of latency or durability to gain higher parallelism [10, 23, 25].

6.4 Hardware Improvements

Improvements in block devices, including lower latency and higher overall performance, have led to the general recommendation that logs be placed on a separate device, or at least one that can leverage parallel write technology [1, 17]. By creating a dedicated device, the mixed write traffic problem is resolved since both log and data writes will have their own distinct queues to work with; neither stream/write queue will mix. This dedicated device is now possible because of improvements to storage and server capacities. Larger capacity drives, where the physical storage itself is able to store more data, extends the life of existing server solutions. Larger capacity servers can refer to the actual storage itself, up to

petabytes in size, or the ability to house more drives. While increasing the storage capacity of a server does not always lead to performance improvement, the ability to increase either the number of drives or their capacities increases the likelihood that one drive can be set aside as a log device.

There have also been improvements to how drives perform I/O by leveraging parallelism. HDDs are being developed to have multiple actuators, allowing for separate I/O operations to be performed in parallel, effectively splitting the drive into *n* chunks, where *n* is the number of actuators available [12]. Recent improvements to SSDs include capacity increases as mentioned above, but also include implementing new protocols to speed up drive access times [14]. In particular, NVMe (Non-Volatile Memory Express) has been built from the ground up to improve device performance, taking advantage of the faster PCIe bus available. NVMe achieves increased throughput performance, reduced latency, and, in certain cases, enables threads or processes to have their own I/O queue [11]. Recently introduced multiple write-streams, allowing a host to specify a stream for every write, have also given users more control over the write path. Multiple write-streams allow for better data placement, more efficient garbage collection, and splits the drive into *n* chunks, where *n* is the number of streams that are supported [16, 18]. With good partitioning and detail to where a write operation is being sent, a single physical drive can be treated as two logical ones.

Creating a dedicated logging device, either physically or logically, solves the initial problem of mixed writes by splitting traffic. However, it introduces another complexity to managing new hardware for data centers that may already struggle to manage what they have. A specific example of this is Zookeeper, which specifically recommends a dedicated device for its logs in any production environment [13]. However, users will still choose to run production environments where the log is on a shared device. The reasoning for ignoring recommended setups has many factors including cost, maintainability, and performance benefits. Additionally, choosing the right scheduler for a system depends heavily on workload and available resources.

Unfortunately, even with high performance flash, we still channel contention [28, 32]. This flash channel contention leads us to our production problems.

7 Conclusion

Write-ahead log contention is a real problem even with high performance flash devices. We have seen problem in production due to contention when we deploy equipment with a single flash storage device. Our distributed benchmark with ZooKeeper shows at a distributed systems level the problem of contention. We were also able to mitigate the problem of contention in a distributed environment using snapshot scheduling. Snapshot scheduling is already running in production with hundreds of ensembles supporting large data size and high write traffic, which scales the total data sizes that ZooKeeper can support without latency spikes.

We also examined the problem at the local level by reproducing contention with a benchmark program that generated both small durable writes and large background writes. We then used the benchmark to validate our approach to mitigating contention using priorities. Unfortunately, using priorities was much more difficult than invoking a single system call, but using the techniques developed in the paper we were able to mitigate the contention problem locally even when using Java. With the proper block scheduler and dedicated high-priority WAL processes, we can achieve consistent WAL performance on a shared drive.

References

[1] Bluestore. https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/3/html/administration_guide/osd-bluestore.

[2] Kernel's i/o schedulers. https://wiki.archlinux.org/index.php/Improving_performance#Kernel's_I/O_schedulers.

[3] Seastar documentation. http://docs.seastar.io/master/index.html.

[4] *MySQL 8.3 Reference Manual*, 8.3 edition, 2023. 10.5.8 Optimizing InnoDB Disk I/O.

[5] *ZooKeeper Administrator's Guide*, 3.9 edition, 2023. Single Machine Requirements.

[6] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, Santa Clara, CA, July 2017. USENIX Association.

[7] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, page 22. ACM, 2013.

[8] Neil Brown. Block layer introduction part 2: The request layer. November 2017. https://lwn.net/Articles/738449/.

[9] Michael Cahill. A technical introduction to wiredtiger, mongodb, 2015.

[10] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[11] NVM Express. Nvme overview. 2016. https://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf.

[12] Jason Feist. Multi actuator technology: A new performance breakthrough. march 2019. https://blog.seagate.com/craftsman-ship/multi-actuator-technology-a%2dnew-performance-breakthrough/.

[13] Apache Software Foundation. Zookeeper documentation, 2019. https://zookeeper.apache.org/documentation.html.

[14] Kyuhwa Han, Hyukjoong Kim, and Dongkun Shin. Wal-ssd: Address remapping-based write-ahead-logging solid-state disks. *IEEE Transactions on Computers*, 2019.

[15] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[16] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association.

[17] Adam Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, 2008.

[18] Heerak Lim. Application-driven flash management: Lsm-tree based database optimization through read/write isolation. 2018. http://2018.middleware-conference.org/wp-content/uploads/mwds18-paper68.pdf.

[19] Linux. CFQ (complete fairness queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[20] Linux. *ionice*. https://man7.org/linux/man-pages/man1/ionice.1.html.

[21] Linux. *ioprio_set()*. https://linux.die.net/man/2/ioprio_set.

[22] Linux. *iotop*. https://man7.org/linux/man-pages/man8/iotop.8.html.

[23] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The fuzzylog: A partially ordered shared log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 357–372, Carlsbad, CA, October 2018. USENIX Association.

[24] Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proc. VLDB Endow.*, 13(12):3217–3230, August 2020.

[25] C Mohan, D Haderle, B Lindsay, H Pirahesh, and P Schwarz. Aries - a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *Acm Transactions On Database Systems*, 17(1):94–162, 1992. https://cs.stanford.edu/people/chrismre/cs345/rl/aries.pdf.

[26] Patrick O'Neil, Edwar Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). https://www.cs.umb.edu/~poneil/lsmtree.pdf.

[27] Stan Park and Kai Shen. Fios: a fair, efficient flash i/o scheduler. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 13–13. USENIX Association, 2012.

[28] Roman Pletka, Ioannis Koltsidas, Nikolas Ioannou, Saša Tomić, Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Aaron Fry, and Tim Fisher. Management of next-generation nand flash to achieve enterprise-level endurance and latency targets. *ACM Trans. Storage*, 14(4), December 2018.

[29] Alexander Thomasian and Chang Liu. Disk scheduling policies with lookahead. *SIGMETRICS Perform. Eval. Rev.*, 30(2):31–40, September 2002.

[30] Paolo Valente and Arianna Avanzini. Evolution of the bfq storage-i/o scheduler. In *2015 Mobile Systems Technologies Workshop (MST)*, pages 15–20. IEEE, 2015.

[31] Paolo Valente and Fabio Checconi. High throughput disk scheduling with fair bandwidth distribution. *IEEE Trans. Comput.*, 59(9):1172–1186, September 2010.

[32] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[33] Ting Yao, Jiguang Wan, P. Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. A light-weight compaction tree to reduce i / o amplification toward efficient key-value stores. 2017.