ZNSCCACHE: ZNS Based Chunk Cache

John Ramsden
University of British Columbia
john@ece.ubc.ca

ABSTRACT

Cloud providers like AWS, Azure, and Google Cloud offer reliable ways to access remote data, but cloud-based storage can be slow when compared to local storage. To improve performance and reduce latency, expensive RAM caches are often used on client systems. When cache needs surpass physical RAM capacity or become too costly, SSDs are used. However, SSDs can introduce unpredictability, which can impact service providers' ability to meet latency targets. I explore a method to reduce unpredictability and tail latency in SSD-based caches using Zoned Namespace (ZNS) SSDs.

KEYWORDS

solid-state drives, zoned namespace, garbage collection

ACM Reference Format:
John Ramsden. 2024. ZNSCCACHE: ZNS Based Chunk Cache. In *Trade-offs in Designing Computer Systems, 2024, Vancouver, BC, Canada.* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

Cloud-based storage is an attractive option for data storage because the providers manage redundancy, security, scalability, and high availability. This enables applications to use storage without the need to worry about infrastructure and security. The usage of cloud-based storage is growing as a result. Unfortunately, cloud-based object stores can become expensive [3].

In data-intensive workloads, using a cache is a clear solution to reduce costs associated with frequent data retrieval, especially when dealing with remote cloud-based data and its inherent access latency, which can be detrimental to performance. A cache effectively mitigates this latency issue. Conventionally, data caching involves implementing a RAM-based cache, as depicted in Fig. 1 (left). However, the substantial cost of larger RAM capacities makes this approach financially costly, thereby rendering it unfeasible to create caches of sufficient size. To accommodate more substantial caching needs, the integration of SSD caches becomes viable, as illustrated in Fig. 1 (right). While significantly slower than RAM, using a fast SSD cache can be considerably faster and cheaper than traversing the network to access remote data.

CPEN 534, 2024, Vancouver, BC, Canada
2024.

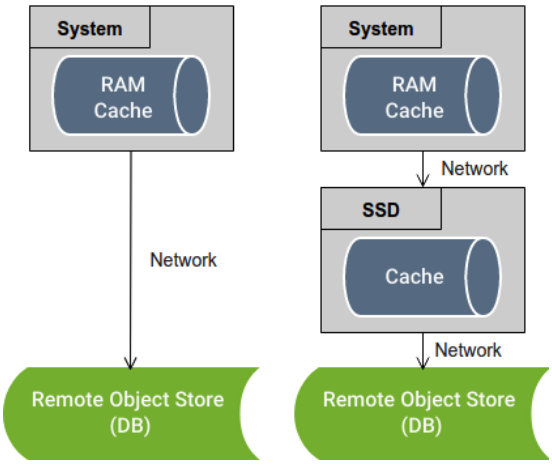

Figure 1: System with RAM cache (left), and system with SSD and RAM cache (right).

Flash-based SSDs do not allow overwriting data in-place. Instead, we must write a new copy of the data to a different location. Periodically, the drive cleans up old copies of data to free up space in a process known as garbage collection (GC) [4]. However, GC can make performance unpredictable [4]. New types of SSDs, called Zoned Namespace (ZNS) SSDs, do not perform GC, resulting in predictable performance (Fig. 2). They only allow sequential writes into large zones, typically ranging from hundreds of megabytes to low single-digit gigabytes [4]. This imposes additional restrictions on software designers. The research question I would like to address is: can I build a cache on a ZNS SSD that maintains predictable performance while overcoming the restrictions imposed by zones?

I focus on workloads that store data in cloud storage, such as AWS, and access this data remotely from compute instances. I explore how ZNS-based SSD caches on local compute instances can mitigate some of the drawbacks associated with SSD-based caching.

2 BACKGROUND AND RELATED WORK

2.1 Zoned Namespace (ZNS) SSD

On modern SSDs, the Filesystem Translation Layer (FTL) maintains mappings from logical to physical blocks, creating the illusion that an SSD functions like a traditional hard drive. This layer allows users to make in-place updates, a

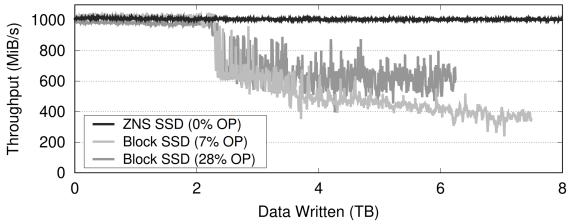

Figure 2: Throughput of a multi-threaded write workload that overwrites usable SSD capacity four times. The SSDs all have 2 TB raw media and share the same hardware platform [4].

process otherwise impossible on flash memory. Because the FTL emulates a traditional block device, it necessitates background
data moved during these updates [9]. Garbage collection significantly increases overhead and contributes to high tail latencies [8]. ZNS SSDs offer a way to eliminate this overhead [4] by imposing additional restrictions and responsibilities on the client. These SSDs eliminate hardware-level garbage collection but introduce several requirements.

Zones: In ZNS SSDs, the basic unit of erasure is known as a "zone", which, similar to flash-based erase blocks, can only be erased entirely. Writes within a ZNS SSD must be confined to a single zone and proceed sequentially from the beginning to the end, following a location known as the write pointer. Failing to write in this sequential order results in data loss. The programmer is responsible for ensuring that writes not only occur sequentially but also at the correct location within a zone. A limited number of zones can be "active" at the same time. A zone becomes implicitly active once it receives writes and remains so until it is "finished" and closed, restricting the number of writable zones before finalizing the data.

2.2 WiredTiger Chunk Cache

Much of the work done in this paper was inspired by the WiredTiger chunk cache [2]. The WiredTiger chunk cache's intended purpose is to reduce the requests to remote storage by caching data at a configurable granularity referred to as "chunks".

I performed WiredTiger performance experiments, examining the impact of different chunk sizes. WiredTiger uses an on-disk "chunk cache" to reduce data movement costs by storing retrieved data from remote storage in chunks within persistent storage. My findings underscore the need to support both large (zone granularity) and small (sub-zone size) chunks, as there is no one-size-fits-all solution for all workloads.

WiredTiger has two caches: the in-memory cache and the optional chunk cache. The in-memory cache stores data in a format that enables efficient searching, while the chunk cache stores portions of database data on disk. I enabled the chunk cache, and intentionally configured the in-memory cache to the small size of 2GB, which is notably smaller than the database size of 35GB. This deliberate choice ensures that WiredTiger frequently accesses storage, thereby making the use of the chunk cache necessary.

For these tests, I utilized the WiredTiger B-tree workload [1]. Both the database and chunk cache were stored on a local SSD. In scenarios where I specifically tested evictions, I limited the chunk cache capacity to a small enough size to ensure eviction occurred. In cases where I aimed to avoid evictions, I set the chunk cache capacity significantly larger than the database. To evaluate how different chunk sizes affect performance, I conducted experiments by varying the chunk size. Throughout my experiments, I ran the test workload for 1,000 seconds, measuring operations per second.

Fig.3 shows the performance without eviction, where larger chunks perform well. Conversely, in Fig. 4 with eviction, smaller chunks perform better, while larger ones perform poorly. This experiment demonstrates that the chunk size should be tuned depending on the cache size, and that ZNS-based caches must effectively support chunks of varying sizes.

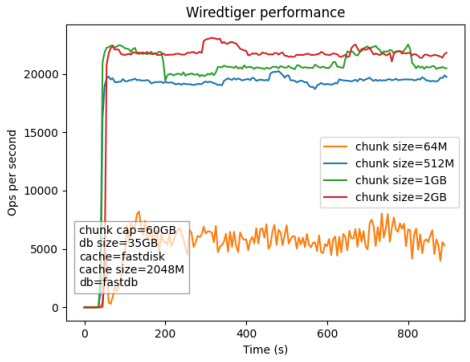

Figure 3: WiredTiger workload under no eviction.

3 DESIGN AND IMPLEMENTATION

3.1 Overview

ZNSCCACHE is a cache similar to the WiredTiger chunk cache. Given a object ID, size, and offset, the cache requests the relevant data from remote object storage and caches it on a local ZNS SSD. If the requested data is present in the local cache it is returned from the ZNS SSD.

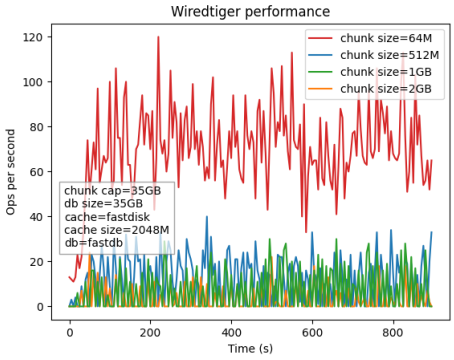

Figure 4: Wiredtiger workload under eviction.

3.2 Implementation

Zone Structure: Zones are broken up into chunks of a predetermined size, specified at run time, and static throughout the cache lifetime. These chunks cannot be larger than a zone in my existing implementation. Data is cached at the granularity of chunks.

Data Mapping: For keeping track of data location, a hash table is used. The purpose of the hash table is to map object UUIDs and data offsets (within bucket objects) to zone, and chunk location. In order to take into account UUID and offset, first the UUID is hashed, it is then bundled into a struct with an offset, and this entire struct is then hashed giving us a location in the hash table based on both the UUID and the offset.

Free Queue: To locate the next available location that should be written to, a "free queue" is maintained. Upon initialization, this queue is populated with the first chunk of each zone. When a chunk is popped off at the front of the queue for writing, if it is not the last chunk in its zone it is pushed onto the front of the queue so that the next write will pull the next chunk from that zone. In practice, this ends up meaning zones are filled from front to end sequentially. Zones are removed from the list when full and then put back on the list when they become available due to eviction occurring.

Eviction: Due to the constraints of ZNS, individual chunks within a zone cannot be erased without removing an entire zone. Marking certain chunks as invalid and moving valid chunks to a different zone when entire zones are erased would be a different form of garbage collection. For the purposes of this paper I chose the simplistic method of a modified version of an LRU based on an "epoch value" of a zone. This value is the average of every chunks "epoch value" within that zone. It is an estimation of how recent the data is in an entire zone. This method avoids garbage collection but has the downside of removing data that may have been recently used due to the zone being considered as a whole and not the individual chunks.

3.3 Example Walkthrough of a Get Request

Upon cache start, the epoch list is initialized, with each chunk's associated epoch being set to the same current value at initialization time. An average epoch value is also stored for each zone. The free queue is populated with all the available zones.

When data is requested from the cache in the form of *get(uuid, offset, size)*, the cache looks for the first zone and chunk by querying the hash table based on the UUID and offset. If the data is present in the cache, it is read from disk, and the cache continues on to the next chunk, repeating the process. When data is not present in the cache, a call to remote object storage is made, which requests the remaining data. The call to object storage does not simply request one chunk, but requests all of the remaining data that will later be written to other chunks to avoid subsequent expensive calls to remote object storage. When data is not in the cache and was requested from object storage, it is placed in the hash table, and written to disk at the next available zone from the free queue. Subsequently, the epoch list is updated marking the current time in the relevant chunks entry in the epoch list, and the zone's epoch average is updated.

Once the cache hits a user-defined percentage of capacity threshold for when eviction should begin, epoch averages are checked, and based on another user-defined variable, a set number of zones are evicted. This eviction process erases zones and updates the hash table removing invalid data. Eviction is a simple process and occurs in $O(n)$, where n is the number of zones.

4 EVALUATION

System Configuration: I evaluate the ZNS cache using a Western Digital Ultrastar ZN540 [7], which contains 1024 zones of 1GiB. My host machine consists of a Xeon(R) Silver 4216 CPU, running at 2.10 GHz.

4.1 Data Configuration

For generating workloads I used a benchmark modeled after YCSB [6], a standard tool used for benchmarking of storage systems [10]. I used the YCSB zipfian generator [5] to produce a distribution of values where some are more popular than others.

For my evaluation process I tested with four different configurations as shown in Table. 1. Each of the workloads was set up to run a maximum of 3 million requests. If the requests did not complete within 4 hours they were stopped. The no-evict workloads use a remote object store small enough to fit within the cache.

Workload	Cache	Object	Remote	Eviction
smallquery-noevict	10GiB	32KiB	7GiB	no
smallquery-evict	10GiB	32KiB	400GiB	yes
largequery-noevict	10GiB	1MiB	7GiB	no
largequery-evict	10GiB	1MiB	400GiB	yes

Table 1: Data configuration used for evaluation.

In order to avoid additional inconsistency in systems I do not control, I emulated the use of remote object storage. I added a compile-time flag to my system that allowed setting an arbitrary latency for object storage emulation, along with a flag that toggles between remote or emulated storage. During emulation, the code sleeps based on the predefined compile time latency and then returns a buffer containing garbage data. While this does not allow for testing of correctness of data returned, it does allow testing end-to-end latencies of the cache, which is the primary concern of this project.

To determine reasonable latencies based on the size of data retrieved, I made 100 calls to remote object storage (S3 calgary region) using the object sizes from Table. 1. I then took the standard deviation and the geometric mean of the calls (see Table. 2).

Object Size	Geo Mean (ms)	Std Dev (ms)
32KiB	39	21
1MiB	123	54

Table 2: Geometric mean and standard deviation for remote requests.

The calls sent to remote object storage have a high standard deviation. By using the geometric mean of request time, I remove one unpredictability aspect that is outside my control.

4.2 Results

4.2.1 Overview of Results. Table 3 shows an overview of the results. The workloads that underwent eviction had to be prematurely stopped before the 3 million requests could occur. The small queries had a slow fill rate, and eviction only started near the end of the workload, with the first eviction occurring at 03:29:21. This is in contrast with the large workload's first eviction occurring at 00:20:15. The slow throughput and request rate of the eviction workloads can therefore largely attributed to the low hit ratios (Table. 4), and not eviction.

4.2.2 Breakdown of Work. The workloads have varying sections that occupy the majority of time spent. Fig. 5 shows how time is distributed based on the different query types. For the *smallquery* workloads, and *largequery-evict* workload, the large majority of time was spent making calls to remote storage. The *largequery-noevict* workload in contrast spent most of its time returning cached requests. This distribution demonstrates that larger query sizes can greatly reduce the percentage of time spent making calls to remote object storage. The actual amount of time spent writing to disk was insignificant, under 4% in all workloads. Other work done such as eviction was not included in the breakdown due to the percentage of time spent being so low it did not register on the graph.

Workload	Throughput	Runtime	Requests
smallquery-noevict	8.52MiB/s	03:03:02	3000000
smallquery-evict	1.98MiB/s	04:00:00	912797
largequery-noevict	399.18MiB/s	02:05:15	3000000
largequery-evict	17.78MiB/s	04:00:00	256010

Table 3: Throughput of workloads, runtime in (HH:MM:SS).

Workload	Hit Ratio
smallquery-noevict	0.93
smallquery-evict	0.63
largequery-noevict	1.00
largequery-evict	0.57

Table 4: Hit Ratio for workloads.

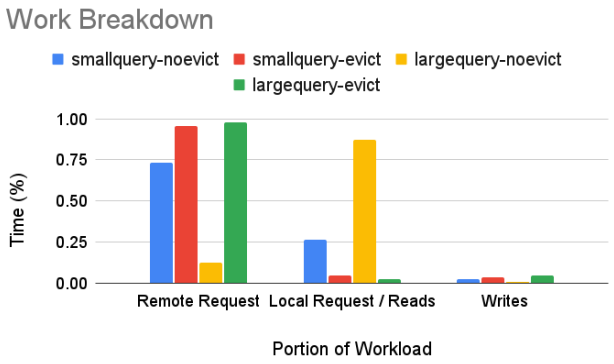

Figure 5: Work breakdown for different workloads.

Workload	Tail	Stddev	GMean	Mean
smallquery-noevict	40.68ms	9.82ms	1.33ms	3.66ms
smallquery-evict	40.78ms	19.06ms	4.26ms	15.77ms
largequery-noevict	3.42ms	6.22ms	2.19ms	2.50ms
largequery-evict	129.68ms	62.89ms	12.23ms	56.24ms

Table 5: Tail latency (99th percentile), standard deviation, geometric mean, and arithmetic mean for get requests.

Table. 5 shows tail latency, standard deviation, geometric mean, and arithmetic mean for get requests. A get request encompasses all time of remote object store or local read, caching data, and returning of the data to the user. Tail latency is significantly larger than the mean or geometric mean in all workloads except for *largequery-noevict*. This is largely due to workloads other than *largequery-noevict* spending the majority of their time retrieving remote data which exhibits a much higher latency than a cached result. I draw from this data that workload (cache size relative to data size, and request size) matters significantly when trying to reduce tail latency.

4.2.3 Read and Write Latency. Write latency exhibits an extremely variable latency as shown in Fig. 6. However, between moments of brief 50ms jumps, performance is fairly consistent (Fig. 7). What is occuring here is a "zone finish", which happens when writes reach the end of a zone, and the zone needs to be closed to avoid the active zone limit being reached. This operation exhibits a significant latency of around 50ms. It does not need to happen immediately, since several zones can be open simultaneously. Moving this operation to the background could significantly reduce tail latency of writes, allowing the cache to move on to different zones while previous zones are finished.

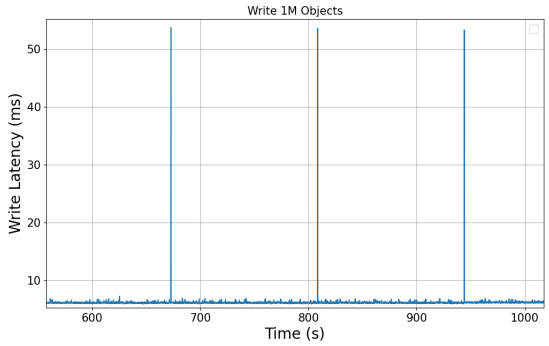

Figure 6: Write latency for *largequery-evict*.

Reads unlike writes are unaffected by the zone finish latency. Fig. 8 shows that performance is reasonably predictable throughout the workload, still showing some variance, but standard deviation and tail latency are low (0.17ms and 2.71ms respectively).

5 FUTURE WORK

A key functional improvement would involve moving zone finishes to the background, which could significantly decrease tail latency when zones are full. Exploring methods for preserving data during eviction without reducing performance significantly is also necessary. A comparison with

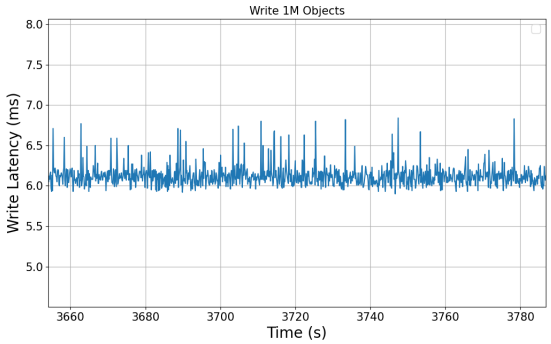

Figure 7: Write latency for *largequery-evict* zoomed in.

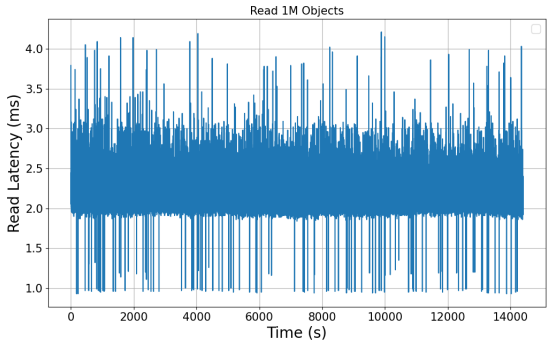

Figure 8: Read latency for *largequery-evict*.

traditional SSDs is also essential. My tests show consistent read and write performance with ZNS SSDs; however, integrating an SSD backend that undergoes garbage collection during benchmarking has not yet been tested to see if ZNS exhibits similar behavior. Furthermore, conducting experiments where a ZNS drive reaches full capacity and comparing the performance impacts with those of SSDs would further clarify differences in predictability. Throughout all of my workloads, performance never declined as usage of a ZNS drive increased, the primary problem SSD drives exhibit. Future work needs to occur in order to claim that ZNS drives can remove the unpredictability that traditional SSDs exhibit.

6 CONCLUSION

ZNS drives provide a way to avoid the overhead of garbage collection but require significant work from the programmer to essentially re-implement what hardware already accomplishes on traditional SSDs. ZNS drives have the potential to improve workloads with strict requirements on tail latency as long as the programmer can efficiently avoid the use of garbage collection.

AVAILABILITY

All source code for the projects described in the paper can be found at `https://github.com/johnramsden/znsccache`. Raw experiment data is available upon request.

ACKNOWLEDGMENTS

This work was done under the supervision of Professor Alexandra (Sasha) Fedorova Department of Electrical and Computer Engineering, University of British Columbia.

REFERENCES

[1] https://github.com/wiredtiger/wiredtiger/blob/b10d24b99e3efd75484f42b6bd94382fce3acc02/bench/wtperf/runners/evict-btree.wtperf.

[2] https://jira.mongodb.org/browse/WT-9812, 2022.

[3] Amazon. https://aws.amazon.com/s3/pricing, 2024.

[4] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. Zns: Avoiding the block interface tax for flash-based ssds. In *USENIX Annual Technical Conference*, pages 689–703, 2021.

[5] Brian Cooper. https://github.com/brianfrankcooper/YCSB/blob/ce3eb9ce51c84ee9e236998cdd2cefaeb96798a8/core/src/main/java/site/ycsb/generator/ZipfianGenerator.java.

[6] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[7] Western Digital. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/data-sheet/data-sheet-ultrastar-dc-zn540.pdf, 2024.

[8] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: A revelation from millions of hours of disk and {SSD} deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, 2016.

[9] Jongsung Lee, Donguk Kim, and Jae W Lee. Waltz: Leveraging zone append to tighten the tail latency of lsm tree on zns ssd. *Proceedings of the VLDB Endowment*, 16(11):2884–2896, 2023.

[10] Muntasir Raihan Rahman, Wojciech Golab, Alvin AuYoung, Kimberly Keeton, and Jay J Wylie. Toward a principled framework for benchmarking consistency. In *Eighth Workshop on Hot Topics in System Dependability (HotDep 12)*, 2012.