ZNCache - ZNS Workload Analysis

John Ramsden
University of British Columbia
Vancouver, Canada

Sam Cheng
University of British Columbia
Vancouver, Canada

ABSTRACT

Modern cloud-based environments commonly use remote data stores such as S3, but they typically suffer from high latency and high cost of I/O. Although RAM-based caches can mitigate this problem by reducing requests to the remote data stores, they also come with a very high cost. Alternatively, users can deploy block-interface SSDs, but these devices exhibit high tail latency and unpredictability due to *garbage collection* (GC). *Zoned Namespace* (ZNS) SSDs offer more consistent performance and lower tail latencies but impose usage constraints. In this paper, we evaluate various workloads and adapt traditional caches to comply with ZNS semantics. We study if and when ZNS SSDs outperform block-interface SSDs and where block-interface SSDs remain sufficient or preferred, particularly in the context of cache workloads. With these insights, software developers can make more informed decisions about which SSD type to use based on specific application needs.

1 INTRODUCTION

Users frequently rely on cloud-based remote storage due to its high availability, scalability, security, and redundancy; however, it comes with a significant cost (typically $0.005 per 1,000 requests on S3) which add up during high usage scenarios [1]. Remote access incurs significant latency (e.g., 5.4s for 512MiB; see Appendix A). To mitigate this issue, systems use caching to reduce the number of requests. Typically, administrators provision large amounts of RAM for caches, which can become prohibitively expensive as datasets grow. As an alternative, SSD-based caches offer ample storage at a relatively lower cost, with reasonable performance. However, SSDs and other flash-based media have inherent limitations.

Flash-based media do not support in-place writes. The devices instead write new data to new locations and mark stale data, such as deleted or modified pages, as invalid. Over time, garbage collection (GC) reclaims invalidated blocks by clearing large areas of flash called *erase blocks* (smallest unit of physical erasure), which operate at the granularity of several smaller *flash pages* (smallest unit for physical writes) [7]. This process can cause many additional writes, which lead to performance degradation, write amplification (WA), and ultimately high tail latency.

An emerging alternative is the Zoned Namespaces (ZNS) SSD, which eliminates device-side GC. ZNS SSDs maintain lower read latency and higher write throughput compared to conventional SSDs because device-side GC no longer occurs (Fig. 1). However, the interface introduces additional complexity for programmers. ZNS enforces sequential writes to *zones*, predefined regions which can only be erased entirely. This shifts management complexity to software.

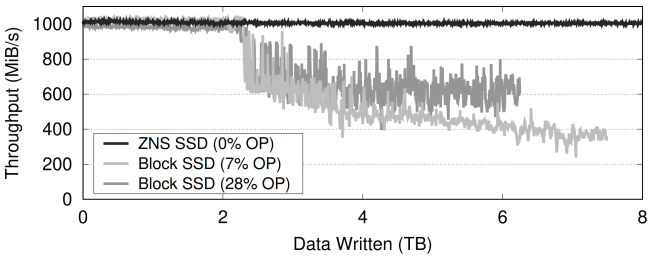

Figure 1: Throughput of a multi-threaded write workload that overwrites usable SSD capacity four times [2].

Handling these restrictions in the context of caching presents significant challenges, particularly during eviction. The primary issue is the difference between the granularity at which data can be erased and that at which it can be written. For example, if a zone contains 20 objects, and 2 of them need to be evicted, those 2 objects cannot simply be erased and written back over. The entire *zone* must be erased, and the valid data must be migrated elsewhere, or written back to the erased zone without the invalidated data. This situation can lead to host-side GC becoming necessary on ZNS SSDs. This situation is further explored in §3.

We investigate the following core question: *Under which workloads can ZNS SSDs outperform conventional block-interface SSDs, and under what conditions do block-interface SSDs perform the same or better?* Since adopting ZNS incurs costs, either through hardware investment or software adaptation, we aim to establish decision criteria for when transitioning to ZNS SSDs is beneficial and when it is inappropriate or unneeded[1].

Our findings demonstrate that under the right conditions where device-side GC occurs, block-interface SSDs can suffer a severe performance degradation of up to 67.75% in increased tail latency, and 568.2% in reduced throughput, while ZNS SSDs retain consistent performance.

[1]We are answering this question from a performance perspective and not a cost per gigabyte, or cost of software rebuild perspective even though ZNS drives have the potential to lower cost per gigabyte pricing when built at scale.

2 BACKGROUND AND RELATED WORK

Flash-based storage systems are commonly used as the backing store in persistent caches because they offer higher throughput and lower latency compared to HDDs or network storage. For example, MongoDB is exploring the idea of using local disk-based caches [14] in order to mitigate costly accesses, motivating the need for low-latency access in production workloads [16]. Compared to DRAM caches, flash-based storage is slower, but offers lower costs and data persistence in the event of power loss [11].

ZNS SSDs directly avoid the overhead of the block-interface [2]. The system must write to flash-based media sequentially, which is reset at the granularity of erase blocks. ZNS exposes this complexity to the programmer - random writes are forbidden and zones are aligned to the erase blocks of the underlying flash drive. In return, performance remains consistent even when nearing drive capacity (Fig. 1). In addition, since GC is no longer performed, ZNS drives expose more usable storage capacity due to the absence of over-provisioning previously required for GC.

Existing caching implementations can also map well to ZNS SSDs. Many cache engines group multiple cache entries into larger "regions" before flushing to disk, in order to improve I/O patterns which reduce GC and increase throughput [3, 15]. These regions can then be placed into zones exactly, allowing the cache to support ZNS SSDs transparently.

Yang et al. confirmed the viability of ZNS SSDs in caching workloads and examined various approaches for storing data in ZNS SSDs [15]. The approaches utilized CacheLib as the caching engine, which writes fixed-size regions to the cache. One approach mapped zones to regions 1-to-1. This enforced sequential writes to the disk, which allowed zero GC, but meant that some valid cache entries may be discarded as the entire zone must be evicted. The other approach mapped multiple regions to a single zone, allowing more granular eviction. To support this, the cache must perform host-side GC, which incurs extra complexity. However, GC policies can be co-designed along with the cache's eviction algorithms, and by doing so, the authors observed lower tail latencies and higher throughput compared to traditional SSDs. Similarly, Lv et al. [8] present a case study and ZNS aware cache system "ZonedStore", demonstrating another design for a ZNS backed cache. However, while both papers built a cache over ZNS and demonstrated simple policies, the workloads were limited, and did not sufficiently answer how different parameters for the cache affected performance or whether or not device-side GC occurred. Additionally, in the former paper, cache size was very small and not representative of workloads with large working sets, typical of SSD-based caches. We seek to answer these questions by exploring a wider range of workloads and parameters and examining their effects.

3 METHODOLOGY

We designed and implemented *ZNCache*, a concurrent key-value cache that fetches and caches data from a remote data store[2]. For the purposes of the project, we made several simplifications: (1) there is no persistence between cache restarts, and (2) we do not cache any actual real data. We simulate actual data stored in the cache with random bytes, and reference them with an integer key. To perform a simulated fetch with latency from the remote store due to a cache miss, we add a constant request time (measured from fetch requests from S3, averaged over 100 runs, see Appendix A). These simplifications remove significant cache functionality, but they are justified since our goal is to study ZNS vs block-interface SSD performance - not to build a full production system.

To distinguish between the types of blocks (erase blocks, unit of I/O, and blocks as the unit of caching), we refer to the unit of caching as a *"chunk"*. Based on the use-case, it is important to be able to have a variable chunk size (variable at initialization not runtime). Larger chunks are beneficial in certain workloads, or to limit access to remote storage for cost, and smaller chunks perform better on workloads where there is less eviction occurring [10]. Our implementation will make this modifiable so we can evaluate the effects. The intention is to cover a broad range of workloads, in order to get a characteristic of how the different SSD types behave under various scenarios.

ZNCache supports two backends: block-interface SSD and ZNS SSD. The codebase is functionally identical for both backends, with slight differences mostly consisting of additional operations for the ZNS backend that are required to execute ZNS operations such as zone erase. Both backends operate on zones. Although block-interface SSDs do not have the same concept of a zone in firmware, we logically break up the disk into regions on the block-interface SSD, leading to functionally identical behavior between the two backends. When we refer to zones in the context of our implementation, they may be ZNS zones or regions on block-interface SSDs representing zones. This allows us to explore performance on an even playing field, with a common codebase. The concept of splitting block-interface SSDs into regions is not a new one, and is a common methodology employed by caching systems [4].

The structure of our caching system is divided into three logical components. *Cachemap* (§3.1) consists of a hash-table that maps data IDs to the address of cached chunks on disk.

[2]We have identified that there are plenty of improvements we can make to performance, both in reducing lock contention, and decoupling workers that receive requests from workers that execute them. We leave this to future work.

Zone State Manager (ZSM) (§3.2) keeps track of the state of individual zones. *Eviction Policy* (§3.3) selects an eviction strategy to use when the cache is full. A thread pool spawns cache worker threads that service requests retrieved from a pre-generated workload file and operate on the cache concurrently. Fig. 2 describes the high-level control flow for a worker thread serving a request[3].

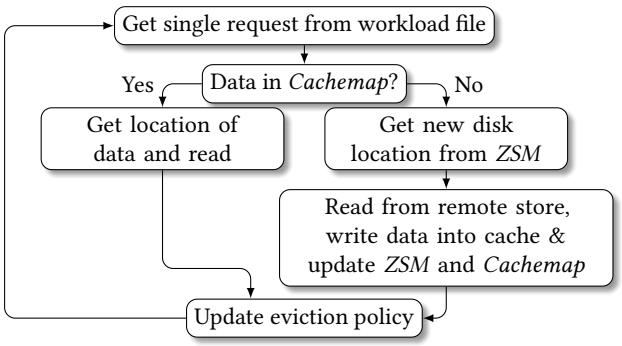

Figure 2: Control flow of a worker thread.

Eviction is controlled by a single thread that is periodically woken up to evaluate whether the cache needs to perform eviction. Eviction is triggered when cache capacity falls below a tunable threshold. Foreground eviction occurs when background eviction fails to keep up, leaving no available space[4].

3.1 Cachemap

Cachemap maps keys to their location on disk. The map is implemented as a hash table behind a lock (data ID to disk address)[5], with an additional map from zones to entries in the table for eviction.

A thread serving a request first looks up whether the data exists on disk in *Cachemap*. If it does, then the thread directly reads from the disk. If not, then the thread must read data from the "remote store", and write the data to the disk. Before unlocking, a temporary condition variable representing a write in-progress fills the map entry. New threads reading from the same data location wait on this condition variable. Once the original thread finishes the write, it updates the table and signals the condition variable, waking up any threads that need to read the data.

[3]We do not maintain a DRAM cache in front of the disk cache, however, this has been identified as a potential improvement that could allow us to buffer writes and then flush them in the background.

[4]This could also be implemented more efficiently with a broadcast mechanism wherein the background eviction task is woken up as needed.

[5]This has been identified as a potential cause of lock contention. Future work could consist of maintaining several "buckets", which can each be locked independently, leading to less lock contention.

3.2 Zone State Manager

The *Zone State Manager* tracks the state of all zones and updates them as needed. Zones start in the **empty** state and are placed in a queue. When selected for writing, a zone moves to the **active** state. Once full, it transitions to the **full** state, meaning no further writes are allowed.

Only one thread is allowed to write to a zone at one time in order to avoid race conditions that result in non-sequential writes to the zone; this is enforced through synchronization primitives. When an active zone is being written to, it is removed off the queue of active zones so that no other thread may write to it. Then the zone is returned when the thread has finished writing, or is moved to the full queue when it is full.

The ZSM differentiates between ZNS and block-interface SSDs when performing chunk eviction. During eviction, the cache erases ZNS SSD zones and copies valid cache entries from the zone back into the cache (host-side GC). In contrast, block-interface SSDs maintain a free list of invalid chunks that the device can overwrite later. This design allows us to avoid performing host-side GC on block-interface SSDs.

3.3 Eviction

The *Eviction Policy* module defines a uniform interface for multiple eviction strategies. All threads invoke this interface after completing a read or write to disk. When eviction is triggered, the eviction thread queries the policy for a zone or chunk to evict - depending on the active strategy - which is then removed from the policy's internal data structures. The eviction thread subsequently updates bookkeeping structures to reset the zone and adjusts the relevant state in the ZSM.

We explored two primary eviction policies in this paper: Zone Promotion LRU (ZPLRU), which evicts at zone granularity, and Chunk LRU (CLRU), which evicts at chunk granularity. We chose these two cache algorithms based on our requirements of being able to demonstrate host-side GC (CLRU), and no host-side GC (ZPLRU). These algorithms are also ones we've seen frequently used in literature (LRU), or other similar caching systems [4].

ZPLRU maintains and updates a single LRU queue of zones. Whenever a single chunk is accessed within a zone via a read or write, the *entire* zone, and all chunks within are promoted to be recently accessed. When the eviction thread requests a zone to be evicted, the policy chooses the least recently used zone. We chose this algorithm for its simplicity and lack of host-side GC. Additionally, it is the method chosen for eviction by other region-based cache systems, demonstrating its applicability [4]. A tradeoff when compared to CLRU is its coarser granularity when performing eviction: a zone may contain both frequently and infrequently accessed chunks, resulting in recently used chunks being evicted. In practice

we have found the algorithm still maintains a high hit ratio (Appendix D).

CLRU separates eviction (marking data as invalid and reclaimable) from (host-side) GC (physically reclaiming space). It operates at the chunk level: when a threshold for available chunks is met, the least recently used chunks are evicted and marked invalid. On ZNS SSDs, GC is performed at the zone level. When the GC threshold is reached, CLRU selects the most invalidated zone and rewrites its remaining valid chunks into a new location, then erases the original zone to reclaim space.

In contrast, on block-interface SSDs, physical erasure is handled by the device, so host-side GC is not needed. Invalidated blocks can be overwritten directly, and CLRU only manages logical invalidation.

To track chunk access recency, CLRU maintains an LRU queue. Chunks are moved to the back of the queue whenever they are read or written. When eviction is required, CLRU selects the least recently accessed chunks from the front of the queue, marks them invalid, and updates a zone-level priority queue that tracks the proportion of invalid chunks in each zone. This priority queue is used to select zones for garbage collection (on ZNS only).

4 EVALUATION

The goals of our experiments (and the corresponding sections where evaluation occurs) are to:

(1) **Parameter Eval (§4.2.1)**: Evaluate how the different SSDs compare under workloads with varying parameters, such as data distribution and chunk size.
(2) **GC Eval (§4.2.2)**: Determine the effects of device-side GC on block-interface SSDs for cache workloads, and how device-side GC impacts throughput and tail-latency.
(3) **Eviction Eval (§4.2.3)**: Determine the effects of different eviction algorithms that include host-side ZNS SSD GC, and those that do not.

4.1 Experimental Setup

Our server consists of an Intel Server R2208WFTZSR with, 256GiB of RAM, and two 16 core Xeon(R) Silver 4216 CPU, running at 2.10 GHz running Ubuntu 22.04 with Linux 6.8.0.

We utilize a ZNS SSD, and a block-interface SSD. Both drives share the same hardware, with the difference simply being the firmware applied to them. We have confirmed this with the vendor. They are of different capacity due to the block-interface SSD requiring over-provisioning for device-side GC that does not need to occur on the ZNS SSD. This translates to more capacity being available for usage on ZNS.

ZNS SSD: (ZN540, Western Digital): 950.789GiB, 904 zones
Block-interface SSD: (SN540, Western Digital): 894.3GiB

All experiments use the *mq-deadline* scheduler on the ZNS SSD and the *none* scheduler on the block-interface SSD. The *mq-deadline* scheduler enforces strict I/O ordering [12], which is necessary to maintain write pointer ordering on ZNS devices in Linux 6.8.0[6]. In contrast, the *none* scheduler is preferred for block-interface SSDs, and is the default in Ubuntu 22.04, due to its lower latency and higher IOPS under a simple FIFO design where strict ordering is unnecessary [13].

4.2 Benchmarks

We test Zipfian and uniform random distributions for cache access patterns. We use the YCSB distribution generator [5] for generating the relevant workloads. We use the default Zipfian parameter (0.99) used by YCSB, which the authors determined to be a reasonable workload for database and cache workloads [6].

We planned to evaluate our two eviction types described in §2 : (1) CLRU, to evaluate the effects of host-side GC, and (2) ZPLRU, to evaluate how this policy performs with no GC. CLRU was implemented but bugs remain, meaning we were not able to complete experiments. For the purposes of evaluation CLRU is missing, and all experiments use ZPLRU.

Our workloads consist of executing a sequence of calls to the cache according to our experimental parameters. We pre-generate workloads corresponding to chunk accesses as binary files, load them into memory (to avoid the variability and latency of disk access), and execute them from start to finish. To measure metrics we inline nanosecond granularity timers directly in code and measure between specific code sections. For graphing we bin data over 60 second intervals, leading to more readable graphs. We do not use binning for latency, throughput and percentile calculations. We vary three parameters: **chunk size**, **distribution**, and **ratio**. Ratio refers to cache size to workload size ratio. All benchmarks were executed with 64 threads (corresponding to the number of hyperthreads available) to achieve the highest possible bandwidth.

4.2.1 Parameter Evaluation. Our parameter evaluation consists of modifying a broad range of parameters and evaluating the effects on throughput and latency. Throughput was low for 64KiB chunk experiments[7], resulting in very long-running experiments. Due to time constraints and the large number of experiments, we artificially reduce cache size based on chunk size for parameter evaluation. Specifically for chunk sizes of 64MiB we use 40 zones; for chunk sizes of 512MiB we use 200 zones. Ideally, we would run all experiments with a full-disk cache (904 zones), since under

[6]As of Linux 6.10.0, Zone Write Plugging was introduced [17], eliminating the need for a specific scheduler to enforce write ordering.
[7]This could be due to inefficiencies in our cache, and future work could consist of attempting to increase throughput through optimization.

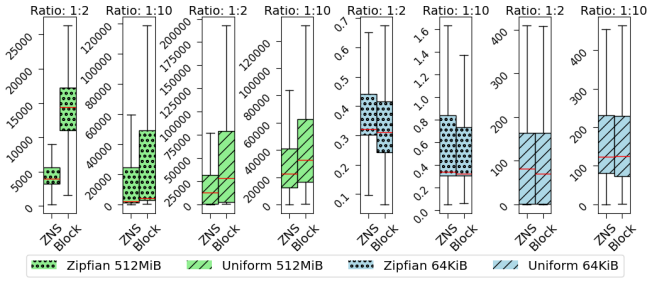

Figure 3: Cache Get Latency (ms), end to end latency including hits and misses comparing ZNS and block-interface (Block) SSDs

Figure 4: Cache Throughput (GiB/s) comparing ZNS and block-interface (Block) SSDs

typical usage we would expect a large portion of users to fully utilize their disks. Prior to all experiments we issue a preconditioning phase consisting of filling the disk to capacity to ensure a baseline for each disk type. This is the same strategy taken by Björling et al. (2021) [2]. The preconditioning phase helps simulate a fully utilized disk, mitigating some of the downsides of the reduced cache size. For the 512MiB and 64KiB chunk benchmarks, the total I/O is fixed at 3TiB and 600GiB, respectively, including both disk reads (cache hits) and writes (cache misses). The two I/O amounts correspond to the same amount of I/O over cache size (64KiB uses 40 zones, which is 20% of 512MiB's 200, and therefore it has 20% of the total I/O - 600GiB). We chose ratios of 1:2 and 1:10 for cache size to workload size to demonstrate the behavior of low eviction and high eviction, respectively. This allows us to see two typical use-cases with low eviction demonstrating a common scenario where the user has a large cache and wants to achieve a high hit ratio, and high eviction, which will stress our eviction algorithms. The high eviction scenario will be more likely to initiate SSD GC due to the higher quantity of writes.

As demonstrated in Fig. 3 (raw data in Appendix B), the ZNS SSD demonstrates significantly lower cache get latency with 512MiB chunks (on average 50.58% lower) compared to the block-interface SSD. Along with lower latency, significantly reduced variance is observed. P99 latency is also consistently lower across 512MiB ZNS workloads compared to SSD (on average 55.87% lower). We also see (Fig. 4) that throughput is maintained at a higher rate (on average 108% higher). We attribute this to GC, which appears to begin early in the run (see Appendix C). Notably, on the block-interface SSD, performance improves significantly over time following an initial GC-induced drop, with write latency decreasing steadily. This trend exceeds what would be expected from improved hit ratios alone. We hypothesize that, with a large 512MiB chunk size, the drive gradually defragments itself during the run, beginning from a heavily fragmented state due to preconditioning. We explore this hypothesis in more detail in §4.2.2.

With 64KiB chunks, block-interface SSDs match or slightly outperform ZNS SSDs - unlike the performance gap observed with 512MiB chunks (Figs. 3, 4). 64KiB chunks do not allow us to reach throughput necessary to exhibit GC, meaning the block-interface SSD is capable of keeping up with the workload without demonstrating performance degradation. It is possible more cache optimization could lead to sufficient throughput with smaller chunks, leading to a similar pattern of performance that we see with large chunk sizes; this remains as future work. On average across 64KiB workloads, we see 1.88% reduced performance on ZNS SSDs, 1.9% increased latency, and similar tail latency. We determine that GC is not occurring with 64KiB chunks due to the workloads not exhibiting the same pattern of performance degradation. In contrast, all 512MiB workloads exhibit clear signs of GC (Appendix C).

4.2.2 Garbage Collection Evaluation. Due to our preconditioning process, which attempts to put drives into a steady state, it can be difficult to detect the beginning of GC. Since we write the entire disk in full prior to our experiment, we may be at the drop off point displayed in Fig. 1 prior to even starting our cache, meaning we are already doing significant GC. This does correspond to some of the behavior we see in the 512MiB chunk size experiments we performed in §4.2.1 - we observe that throughput almost immediately drops on the block-interface SSD, leading us to believe GC is occurring immediately.

In this experiment we attempt to determine the before and after effects of device-side GC on block-interface SSDs under conditions where GC should be most apparent. We modify our initialization phase to give our workload the best chance of exhibiting GC. We perform a complete disk TRIM, which informs the block-interface SSD that all blocks are no longer in use and can be completely erased. This puts the SSD in the ideal performance scenario where GC should *not* be occurring, and therefore we can observe behavior at the beginning of our workload that does not demonstrate GC, and then the corresponding drop. At this point we execute a high write

workload (low ratio 1:10, uniform random, 256MiB chunks, 6TiB of I/O). We chose to use a slightly smaller chunk size of 256MiB instead of 512MiB to cause more internal fragmentation based on some of the findings we observed in §4.2.1, where the drive appeared to defragment itself over time. If our cache is capable of exhibiting a severe drop off it should then demonstrate it around the time it completely fills. Fig. 5 illustrates a clear reduction in throughput attributable to GC, with a pronounced drop occurring at the point of the first complete disk fill. In addition to throughput, get latency was significantly reduced on ZNS, as demonstrated in Fig. 6 (more results in Appendix E). We saw a latency increase with the block-interface SSD of 51.0%, a tail latency increase of 55.6%, and a throughput decrease of 35.4% compared to performance before GC.

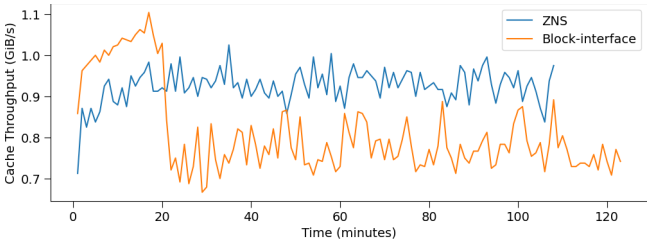

Figure 5: Cache throughput with GC (1:10 ratio, uniform random, 256MiB chunk, 6TiB of I/O workload)

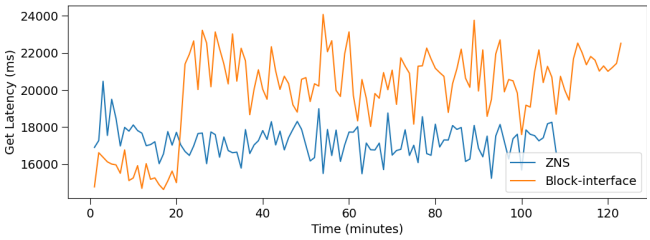

Figure 6: Get latency with GC (1:10 ratio, uniform random, 256MiB chunk, 6TiB of I/O workload)

We also ran this experiment on 512MiB chunks instead of 256MiB. Based on our hypothesis from §4.2.1 we believed we would not see GC at all with the large chunk size of 512MiB, and this was in fact what occurred - observable GC did *not* occur (Appendix E). We used all the same experimental parameters that we did for 256MiB in this section, performed the same amount of I/O, but only modified the chunk size. 256MiB chunk size in contrast seems to be small enough that fragmentation occurs, contributing to GC. This presents us with an interesting observation: large chunk sizes can potentially eliminate GC.

We isolated and tested the effects of 3.2s artificial latency (Appendix A) on GC. We executed the same 256MiB chunk GC workload with no latency, essentially simulating a very close, or local data store. Fig. 7 demonstrates that ZNS SSDs were able to take advantage of the time previously spent waiting, writing to the disk, significantly improving throughput and total runtime. The block-interface SSD in contrast saw next to no noticeable improvement by removing the artificial latency. This demonstrates that the block-interface SSD was already I/O bound due to GC, while the ZNS SSD was not.

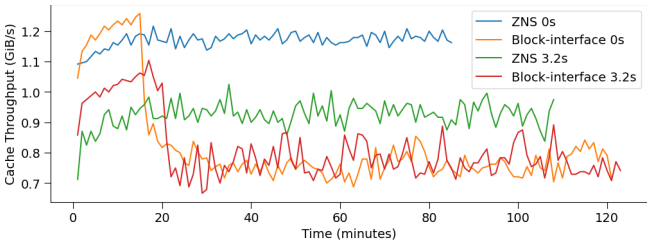

Figure 7: Throughput with GC (1:10 ratio, uniform random, 256MiB chunk, 6TiB of I/O workload) comparing artificial latency (3.2s) with no artificial latency (0s)

4.2.3 Eviction Algorithm Evaluation. For goal (3), we planned to compare CLRU and ZPLRU workloads, evaluating the overhead of executing host-side GC. As mentioned, CLRU was not in a state where we were ready to experiment; as such we do not have results. Given enough time we would have run all of the experiments mentioned in §4.2.1 for CLRU as well. We would have evaluated the overheads of doing host-side GC on ZNS (latency, and throughput) and evaluated how it compares to device-side GC. We had hoped to evaluate whether the increased potential hit ratio (due to us evicting only select data based on LRU, rather than everything within a zone with ZPLRU) would have translated to better performance.

5 CONCLUSION

We find that ZNS SSDs significantly outperform block-interface SSDs in caching scenarios where device-side GC is active and disks are fully utilized, delivering higher throughput, lower latency, and more consistent performance. On average, ZNS SSDs achieve a throughput increase of 107.9% and a latency reduction of 50.6% in these conditions. When GC is not a factor - such as under light workloads - performance differences are negligible, making the transition to ZNS less compelling given the associated development overhead. Our results also suggest that write granularity plays a critical role in performance: sufficiently large chunk sizes can mitigate or eliminate the impact of GC entirely. These insights suggest that developers can benefit from ZNS adoption in high-throughput environments, provided that the workload is tuned to ZNS constraints.

AVAILABILITY

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

6 FOOTNOTES

ChatGPT [9] was used for assisting with text re-structuring, and graphing. All prompts are available at 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] Amazon Web Services. 2025. Amazon S3 Pricing. https://aws.amazon.com/s3/pricing/ Accessed: February 10, 2025.

[2] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. 2021. {ZNS}: Avoiding the block interface tax for flash-based {SSDs} *(USENIX'21)*.

[3] CacheLib. 2025. Cache Library Architecture Guide. https://cachelib.org/docs/Cache_Library_Architecture_Guide/large_object_cache#inserts. Accessed: 2025-02-07.

[4] CacheLib. 2025. CacheLib – Pluggable caching engine to build and scale high performance cache services. https://cachelib.org. Accessed: 2025-02-07.

[5] Brian Cooper. 2019. *YCSB.* https://github.com/brianfrankcooper/YCSB/blob/ce3eb9ce51c84ee9e236998cdd2cefaeb96798a8/core/src/main/java/site/ycsb/generator/ZipfianGenerator.java Accessed: February 10, 2025.

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

[7] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a hybrid key-value cache that controls flash write amplification *(NSDI'19)*.

[8] Yanqi Lv, Peiquan Jin, Xiaoliang Wang, Ruicheng Liu, Liming Fang, Yuanjin Lin, and Kuankuan Guo. 2022. Zonedstore: A concurrent zns-aware cache system for cloud data storage. IEEE.

[9] OpenAI. 2025. ChatGPT: Assisting with Text Structuring and Restructuring. https://chat.openai.com. Accessed: 2025-02-07.

[10] John Ramsden. 2024. ZNSCCACHE: ZNS Based Chunk Cache. (2024).

[11] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. {RIPQ}: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*.

[12] Nick Tehrany and Animesh Trivedi. 2022. Understanding nvme zoned namespace (zns) flash ssd storage devices. (2022).

[13] Caeden Whitaker, Sidharth Sundar, Bryan Harris, and Nihat Altiparmak. 2023. Do we still need IO schedulers for low-latency disks?. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*.

[14] WiredTiger Project. 2025. Chunk Cache in WiredTiger. https://github.com/wiredtiger/wiredtiger.github.com/blob/062e0eb42ed1dc8777f8cf1b8651ca9eb6ac33ce/develop/chunkcache.html Accessed: February 10, 2025.

[15] Chongzhuo Yang, Zhang Cao, Chang Guo, Ming Zhao, and Zhichao Cao. 2024. Can ZNS SSDs be Better Storage Devices for Persistent Cache? *(HotStorage '24)*.

[16] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Transactions on Storage* (2021).

[17] Zoned Storage Project. 2025. Write Ordering Control. https://zonedstorage.io/docs/linux/sched#zone-write-plugging. Accessed: 2025-04-17.

A LATENCY EVALUATION

The following latency evaluation was completed on US West (Oregon) us-west-2, 11ms latency, accessed from the University of British Columbia campus.

Raw data: https://github.com/johnramsden/ZNCache/blob/f37149387436f91f27136464e22bc156fd44a865/docs/REMOTE_TRANSFER_EVAL.md

Results

Table 1: 64KiB Chunk size

Metric	Seconds	Microseconds
Mean latency	0.0406	40632
Geometric mean latency	0.0377	37745
Minimum latency	0.0282	28245
Maximum latency	0.3845	384506
Standard deviation	0.0350	35049

Table 2: 256MiB Chunk size

Metric	Seconds	Microseconds
Mean latency	3.2096	3209583
Geometric mean latency	3.1314	3131383
Minimum latency	2.6974	2697422
Maximum latency	7.7637	7763737
Standard deviation	0.8617	861663

Table 3: 512MiB Chunk size

Metric	Seconds	Microseconds
Mean latency	5.4138	5413781
Geometric mean latency	5.4129	5412910
Minimum latency	5.3835	5383455
Maximum latency	6.0413	6041250
Standard deviation	0.1003	100301

Table 4: 1GiB Chunk size

Metric	Seconds	Microseconds
Mean latency	11.5242	11524248
Geometric mean latency	11.5075	11507539
Minimum latency	11.3097	11309672
Maximum latency	16.5442	16544165
Standard deviation	0.6793	679328

B RAW THROUGHPUT AND LATENCY DATA

The percentages refer to the percentage increase in speed compared to the block device with the same parameters.

Table 5: Get Latency (End-to-end, reads and writes).

Name	Mean (ms)	P99 (ms)
ZNS-512M-UNIF-10	30434.25 (40.58%)	121594.3 (59.15%)
Block-512M-UNIF-10	51219.31	297634.35
ZNS-512M-UNIF-2	20797.86 (63.09%)	115668.18 (67.75%)
Block-512M-UNIF-2	56353.88	358656.23
ZNS-512M-ZIPF-10	17217.21 (55.22%)	120884.2 (62.36%)
Block-512M-ZIPF-10	38448.44	321128.22
ZNS-512M-ZIPF-2	14512.1 (43.42%)	140618.34 (34.22%)
Block-512M-ZIPF-2	25649.63	213772.96
ZNS-64K-UNIF-10	172.5 (-0.44%)	693.76 (0.01%)
Block-64K-UNIF-10	171.74	693.86
ZNS-64K-UNIF-2	115.92 (-3.32%)	629.02 (0.17%)
Block-64K-UNIF-2	112.2	630.11
ZNS-64K-ZIPF-10	45.52 (-0.86%)	488.2 (0.11%)
Block-64K-ZIPF-10	45.13	488.72
ZNS-64K-ZIPF-2	22.52 (-2.97%)	367.69 (-0.09%)
Block-64K-ZIPF-2	21.87	367.37

Table 6: Disk Write Latency

Name	Mean (ms)	P99 (ms)
ZNS-512M-UNIF-10	1467.7 (77.80%)	3973.54 (92.45%)
Block-512M-UNIF-10	6612.51	52602.35
ZNS-512M-UNIF-2	2804.45 (84.64%)	5500.31 (88.74%)
Block-512M-UNIF-2	18263.94	48869.13
ZNS-512M-ZIPF-10	4002.84 (76.03%)	6840.87 (87.46%)
Block-512M-ZIPF-10	16700.43	54572.31
ZNS-512M-ZIPF-2	13220.55 (54.49%)	25297.1 (61.22%)
Block-512M-ZIPF-2	29051.69	65231.09
ZNS-64K-UNIF-10	0.07 (0.00%)	0.12 (-16.7%)
Block-64K-UNIF-10	0.07	0.1
ZNS-64K-UNIF-2	0.07 (0.00%)	0.12 (-8.3%)
Block-64K-UNIF-2	0.07	0.11
ZNS-64K-ZIPF-10	0.07 (0.00%)	0.11 (0.0%)
Block-64K-ZIPF-10	0.07	0.11
ZNS-64K-ZIPF-2	0.07 (0.00%)	0.11 (9.1%)
Block-64K-ZIPF-2	0.07	0.12

Table 7: Disk Read Latency

Name	Mean (ms)	P99 (ms)
ZNS-512M-UNIF-10	451.61 (77.62%)	1278.2 (51.66%)
Block-512M-UNIF-10	2018.36	2644.16
ZNS-512M-UNIF-2	909.75 (65.14%)	2097.4 (42.32%)
Block-512M-UNIF-2	2609.59	3636.54
ZNS-512M-ZIPF-10	1367.41 (59.12%)	2752.36 (56.44%)
Block-512M-ZIPF-10	3344.54	6318.5
ZNS-512M-ZIPF-2	3701.99 (70.86%)	6935.28 (62.98%)
Block-512M-ZIPF-2	12702.44	18735.18
ZNS-64K-UNIF-10	0.31 (-3.33%)	0.63 (-1.61%)
Block-64K-UNIF-10	0.3	0.62
ZNS-64K-UNIF-2	0.32 (-6.67%)	0.65 (-4.84%)
Block-64K-UNIF-2	0.3	0.62
ZNS-64K-ZIPF-10	0.33 (-6.45%)	0.79 (-9.72%)
Block-64K-ZIPF-10	0.31	0.72
ZNS-64K-ZIPF-2	0.33 (-6.45%)	0.83 (-6.41%)
Block-64K-ZIPF-2	0.31	0.78

Table 8: Disk Get Throughput. Mean measured by dividing the total workload by the runtime, while P99 was obtained from periodic measurements averaged over 60 seconds.

Name	Mean (GiB/s)	P99 (GiB/s)
ZNS-512M-UNIF-10	1.043 (67.8%)	0.744 (568.2%)
Block-512M-UNIF-10	0.622	0.111
ZNS-512M-UNIF-2	1.520 (169.3%)	0.776 (242.2%)
Block-512M-UNIF-2	0.564	0.227
ZNS-512M-ZIPF-10	1.829 (121.8%)	1.243 (321.9%)
Block-512M-ZIPF-10	0.825	0.295
ZNS-512M-ZIPF-2	2.125 (72.7%)	1.564 (99.7%)
Block-512M-ZIPF-2	1.230	0.783
ZNS-64K-UNIF-10	0.023 (-0.5%)	0.021 (-0.1%)
Block-64K-UNIF-10	0.023	0.021
ZNS-64K-UNIF-2	0.034 (-3.2%)	0.022 (-0.2%)
Block-64K-UNIF-2	0.035	0.022
ZNS-64K-ZIPF-10	0.086 (-0.9%)	0.048 (-0.6%)
Block-64K-ZIPF-10	0.087	0.048
ZNS-64K-ZIPF-2	0.173 (-2.9%)	0.053 (-1.0%)
Block-64K-ZIPF-2	0.178	0.054

Table 9: Disk Read Throughput. Measurements obtained from periodic read throughput measurements averaged over 60 seconds.

Name	Mean (GiB/s)	P99 (GiB/s)
ZNS-512M-UNIF-10	0.102 (69.6%)	1.133E-11 (46.0%)
Block-512M-UNIF-10	0.060	7.761E-12
ZNS-512M-UNIF-2	0.743 (169.2%)	1.156E-10 (89.2%)
Block-512M-UNIF-2	0.276	6.108E-11
ZNS-512M-ZIPF-10	1.154 (121.9%)	5.420E-10 (214.9%)
Block-512M-ZIPF-10	0.520	1.721E-10
ZNS-512M-ZIPF-2	1.799 (72.5%)	9.215E-10 (48.5%)
Block-512M-ZIPF-2	1.043	6.205E-10
ZNS-64K-UNIF-10	0.002 (-5.3%)	2.893E-13 (-4.3%)
Block-64K-UNIF-10	0.002	3.025E-13
ZNS-64K-UNIF-2	0.013 (-8.0%)	1.080E-12 (-2.0%)
Block-64K-UNIF-2	0.014	1.102E-12
ZNS-64K-ZIPF-10	0.065 (-1.1%)	2.537E-11 (-1.0%)
Block-64K-ZIPF-10	0.066	2.563E-11
ZNS-64K-ZIPF-2	0.151 (-3.9%)	3.022E-11 (-1.9%)
Block-64K-ZIPF-2	0.157	3.080E-11

Table 10: Disk Write Throughput. Measurements obtained from periodic write throughput measurements averaged over 60 seconds.

Name	Mean (GiB/s)	P99 (GiB/s)
ZNS-512M-UNIF-10	0.941 (67.8%)	6.820E-10 (622.7%)
Block-512M-UNIF-10	0.561	9.437E-11
ZNS-512M-UNIF-2	0.786 (172.2%)	6.075E-10 (459.1%)
Block-512M-UNIF-2	0.289	1.087E-10
ZNS-512M-ZIPF-10	0.685 (122.3%)	5.531E-10 (493.8%)
Block-512M-ZIPF-10	0.308	9.313E-11
ZNS-512M-ZIPF-2	0.346 (74.8%)	2.207E-10 (129.5%)
Block-512M-ZIPF-2	0.198	9.616E-11
ZNS-64K-UNIF-10	0.021 (-0.0%)	1.946E-11 (-0.1%)
Block-64K-UNIF-10	0.021	1.948E-11
ZNS-64K-UNIF-2	0.021 (-0.0%)	1.945E-11 (-0.2%)
Block-64K-UNIF-2	0.021	1.948E-11
ZNS-64K-ZIPF-10	0.021 (-0.0%)	1.944E-11 (-0.2%)
Block-64K-ZIPF-10	0.021	1.948E-11
ZNS-64K-ZIPF-2	0.021 (0.0%)	1.945E-11 (-0.1%)
Block-64K-ZIPF-2	0.021	1.947E-11

Table 11: Get Latency (end-to-end including reads and writes) (GC workload, 3.2s latency for 256M chunks, 5.3s latency for 512M chunks)

Name	Mean (ms)	P99 (ms)
ZNS-256M-UNIF-10	17228.71 (12.55%)	68451.80 (20.01%)
Block-256M-UNIF-10	19701.04	85580.73
ZNS-512M-UNIF-10	30842.60 (-1.54%)	123933.37 (-1.16%)
Block-512M-UNIF-10	30374.74	122512.31

Table 12: Get Latency (end-to-end including reads and writes) (GC workload, 0s latency)

Name	Mean (ms)	P99 (ms)
ZNS-256M-UNIF-10	13533.11 (30.39%)	53076.77 (37.64%)
Block-256M-UNIF-10	19441.10	85112.27
ZNS-512M-UNIF-10	27305.66 (17.24%)	111870.09 (17.15%)
Block-512M-UNIF-10	32995.58	135029.59

Table 13: Disk Write Latency (GC workload, 3.2s latency for 256M chunks, 5.3s latency for 512M chunks)

Name	Mean (ms)	P99 (ms)
ZNS-256M-UNIF-10	674.14 (47.98%)	1614.04 (47.20%)
Block-256M-UNIF-10	1295.94	3056.81
ZNS-512M-UNIF-10	1523.29 (-3.52%)	3948.19 (4.87%)
Block-512M-UNIF-10	1471.53	4150.41

Table 14: Disk Write Latency (GC workload, 0s latency)

Name	Mean (ms)	P99 (ms)
ZNS-256M-UNIF-10	2817.61 (36.77%)	3433.91 (52.94%)
Block-256M-UNIF-10	4455.82	7296.20
ZNS-512M-UNIF-10	5697.63 (23.96%)	6767.06 (27.96%)
Block-512M-UNIF-10	7492.57	9392.91

Table 15: Disk Read Latency (GC workload, 3.2s latency for 256M chunks, 5.3s latency for 512M chunks)

Name	Mean (ms)	P99 (ms)
ZNS-256M-UNIF-10	199.70 (85.04%)	364.03 (88.66%)
Block-256M-UNIF-10	1335.24	3210.52
ZNS-512M-UNIF-10	430.28 (77.03%)	1192.47 (53.36%)
Block-512M-UNIF-10	1873.06	2556.69

Table 16: Disk Read Latency (GC workload, 0s latency)

Name	Mean (ms)	P99 (ms)
ZNS-256M-UNIF-10	1252.33 (73.78%)	1804.61 (78.51%)
Block-256M-UNIF-10	4775.61	8398.39
ZNS-512M-UNIF-10	3266.68 (-39.12%)	4275.98 (-51.13%)
Block-512M-UNIF-10	2348.17	2829.25

C THROUGHPUT AND LATENCY GRAPHS

The following section has detailed graphs for various metrics:

(1) Get throughput: Complete cache throughput (eg. user requests a 512MiB chunk, this is 512MiB of data contributing to throughput)
(2) Read throughput: Throughput contributions from only disk reads
(3) Write throughput: Throughput contributions from only disk writes
(4) Get latency: End-to-end latency including both hits and misses for the entire path required to "get" an object from the cache
(5) Read latency: Disk IO read latency
(6) Write latency: Disk IO write latency
(7) Hit latency: The entire code path executed when a cache hit occurs (includes read latency)
(8) Miss latency: The entire code path executed when a cache miss occurs (includes write latency)

C.1 Get Latency

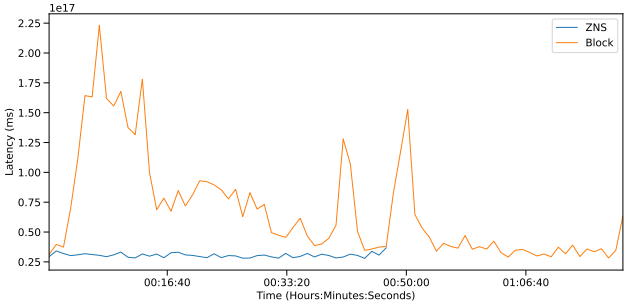

Figure 8: Cache Get latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

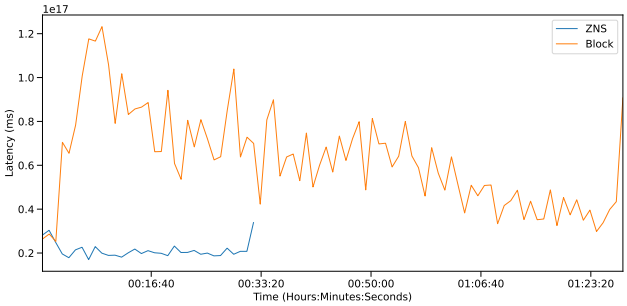

Figure 9: Cache Get latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

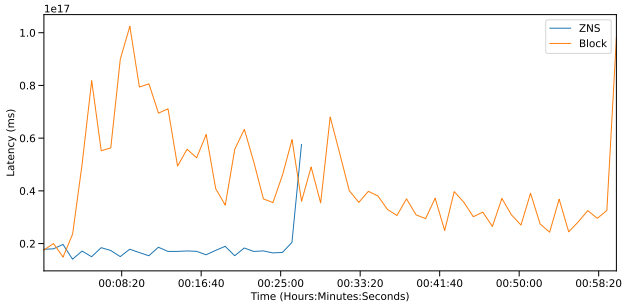

Figure 10: Cache Get latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

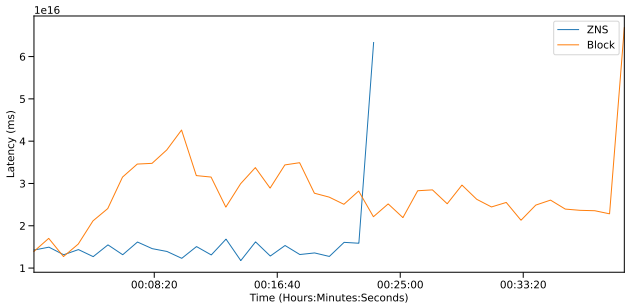

Figure 11: Cache Get latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

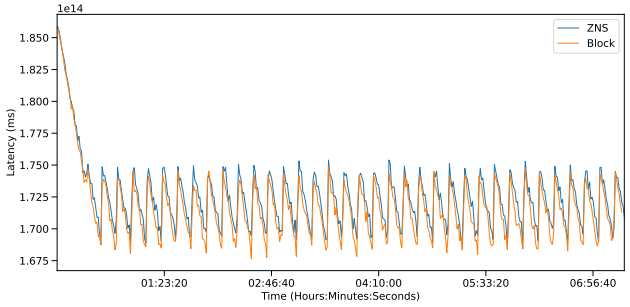

Figure 12: Cache Get latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

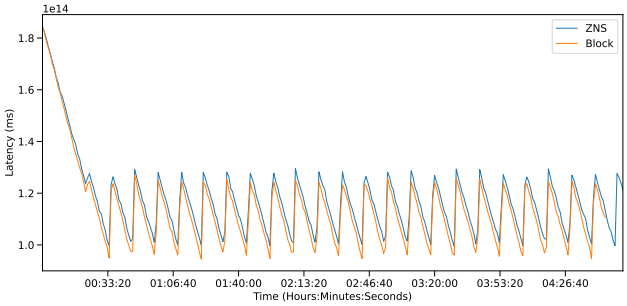

Figure 13: Cache Get latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

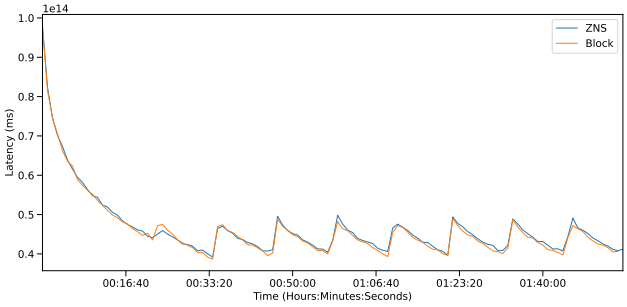

Figure 14: Cache Get latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

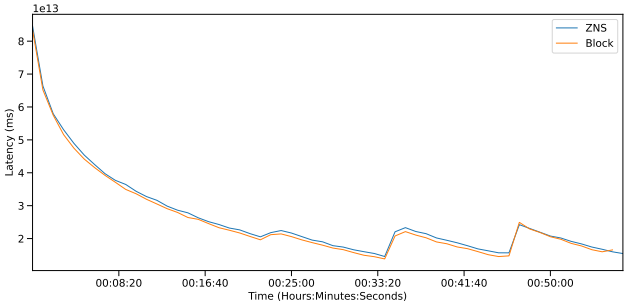

Figure 15: Cache Get latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

C.2 Hit Latency

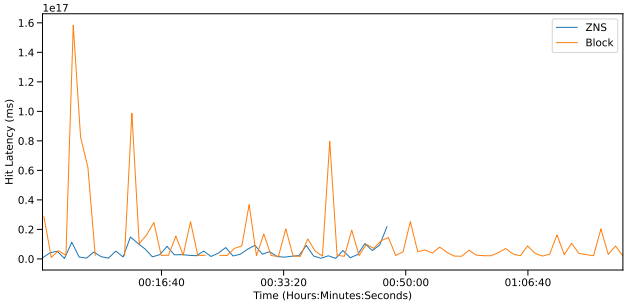

Figure 16: Cache Hit latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

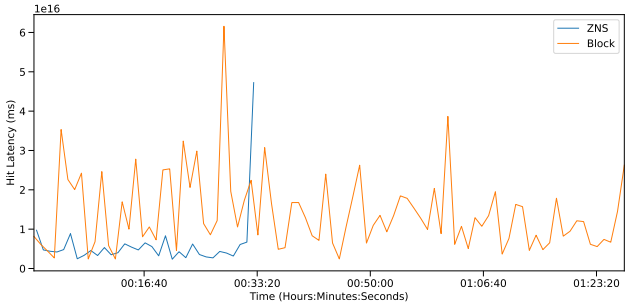

Figure 17: Cache Hit latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

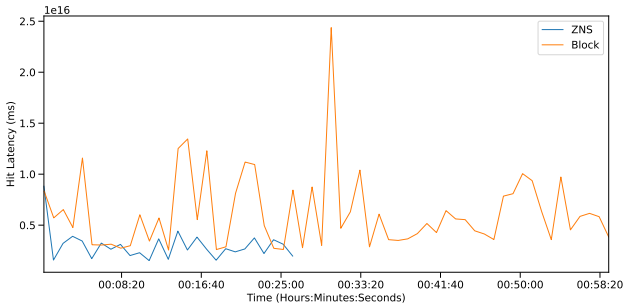

Figure 18: Cache Hit latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

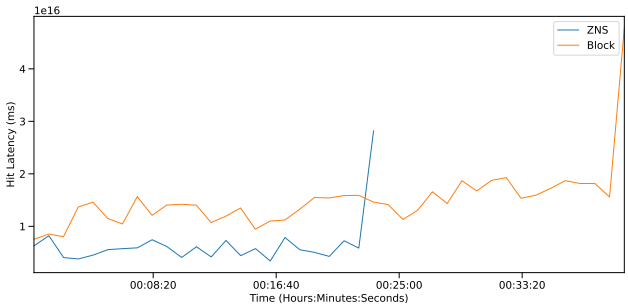

Figure 19: Cache Hit latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

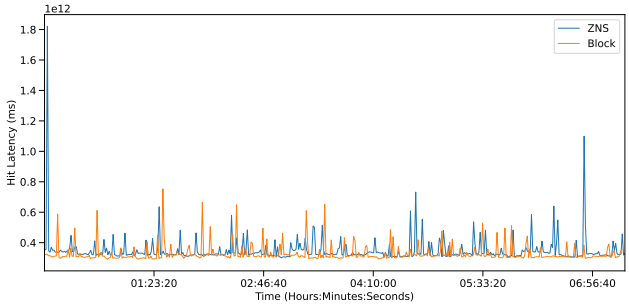

Figure 20: Cache Hit latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

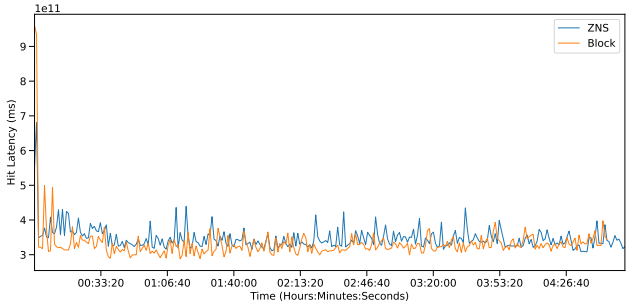

Figure 21: Cache Hit latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

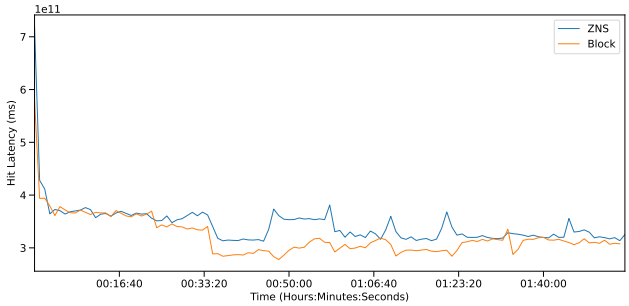

Figure 22: Cache Hit latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

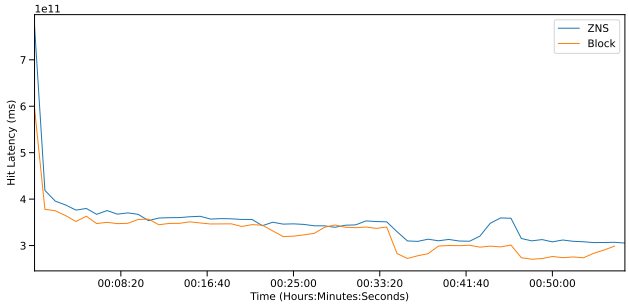

Figure 23: Cache Hit latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

C.3 Miss Latency

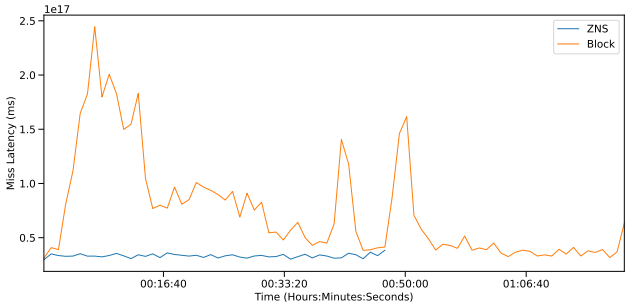

Figure 24: Cache Miss latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

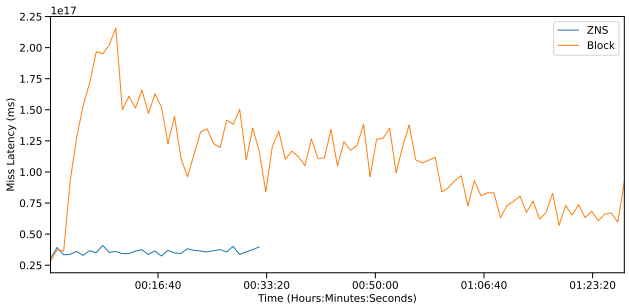

Figure 25: Cache Miss latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

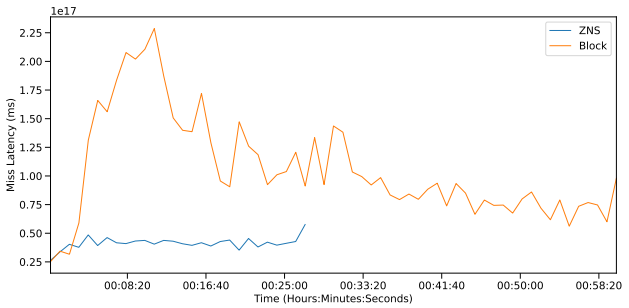

Figure 26: Cache Miss latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

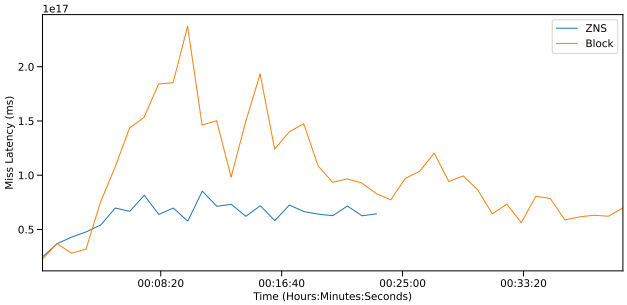

Figure 27: Cache Miss latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

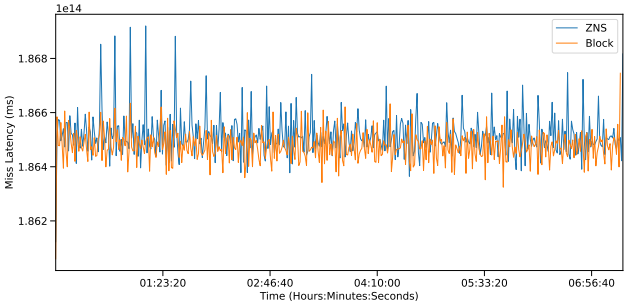

Figure 28: Cache Miss latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

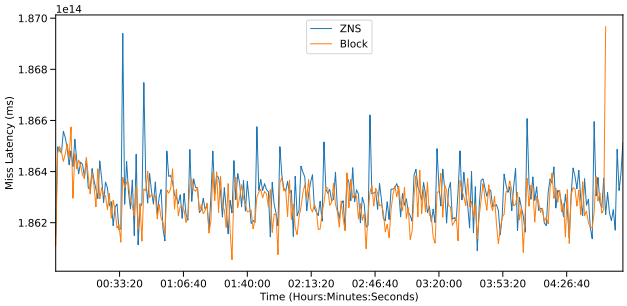

Figure 29: Cache Miss latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

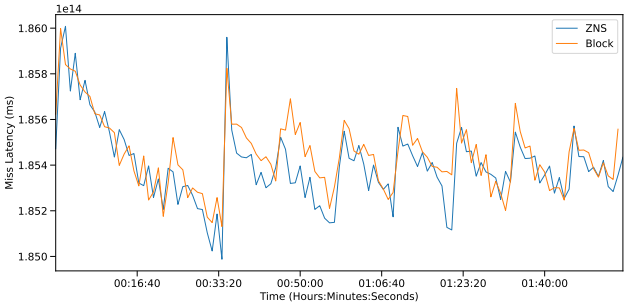

Figure 30: Cache Miss latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

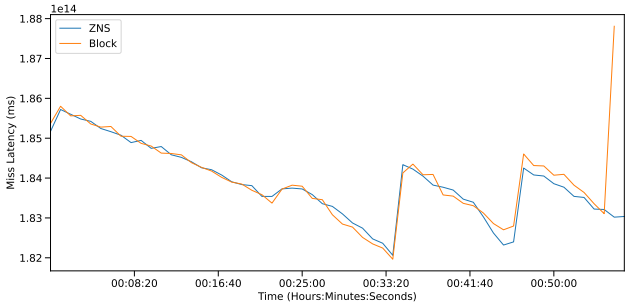

Figure 31: Cache Miss latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

C.4 Read Latency

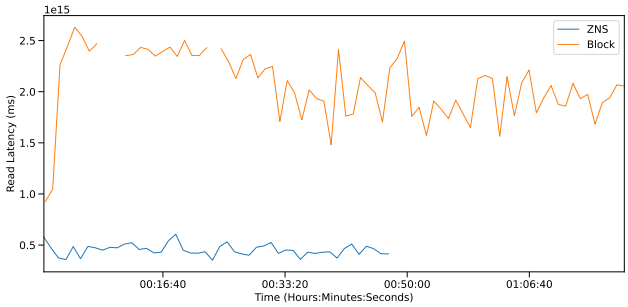

Figure 32: Disk Read latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

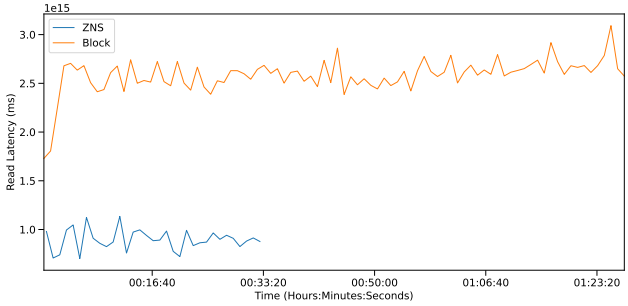

Figure 33: Disk Read latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

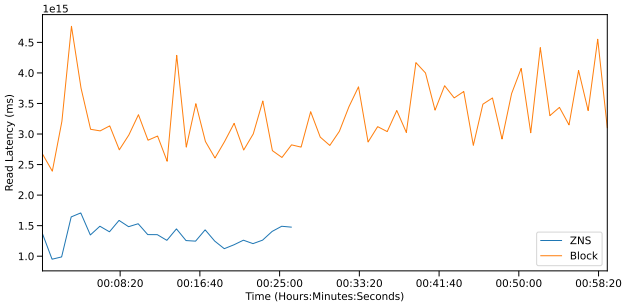

Figure 34: Disk Read latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

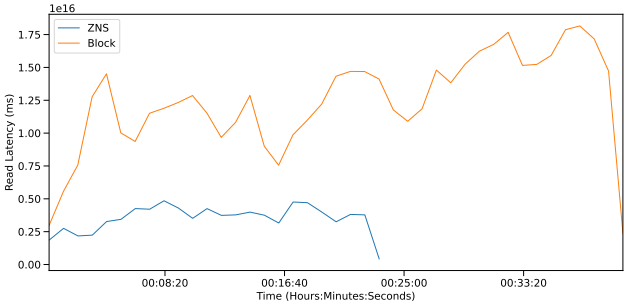

Figure 35: Disk Read latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

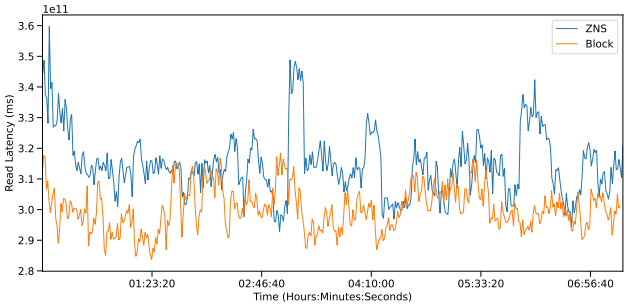

Figure 36: Disk Read latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

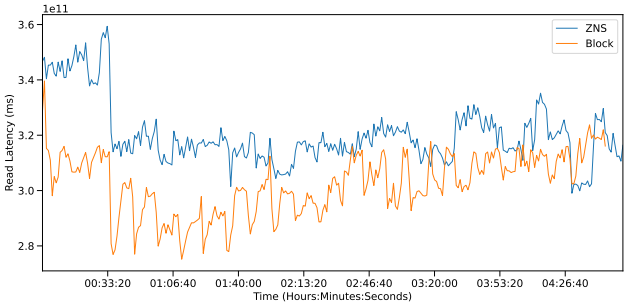

Figure 37: Disk Read latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

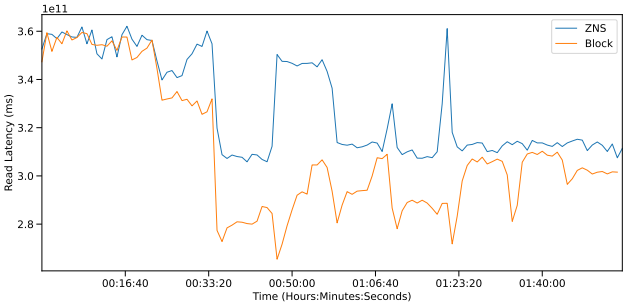

Figure 38: Disk Read latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

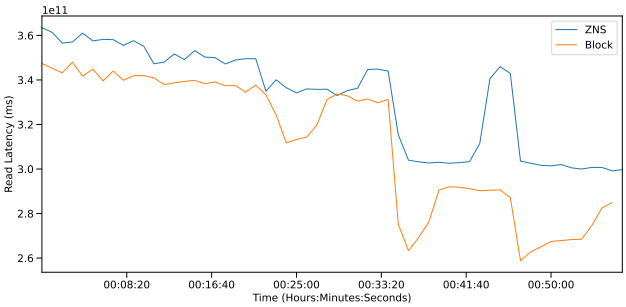

Figure 39: Disk Read latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

C.5 Write Latency

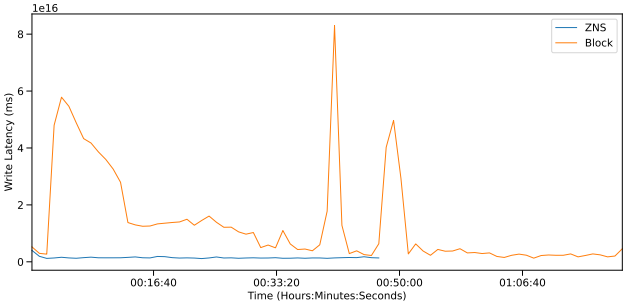

Figure 40: Disk Write latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

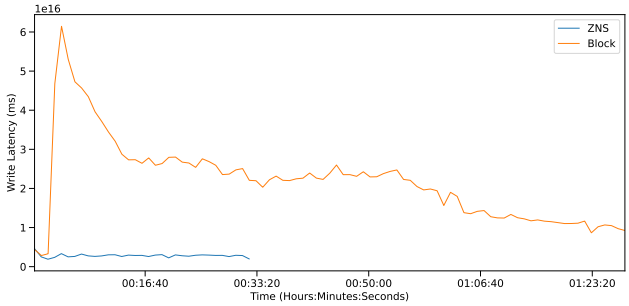

Figure 41: Disk Write latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

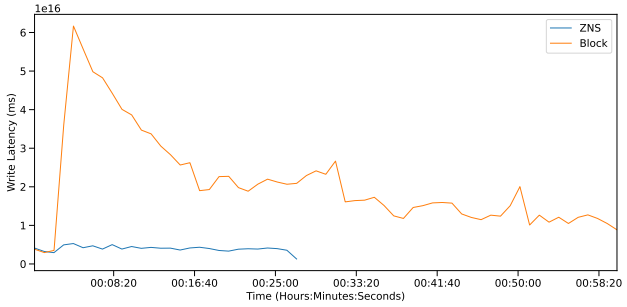

Figure 42: Disk Write latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

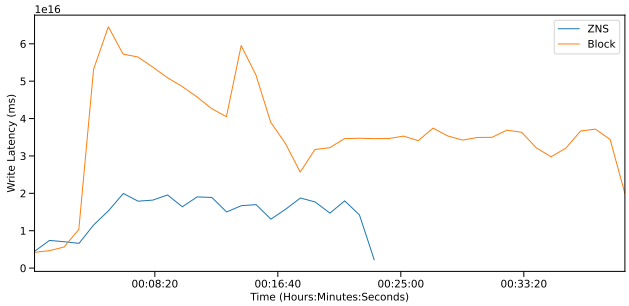

Figure 43: Disk Write latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

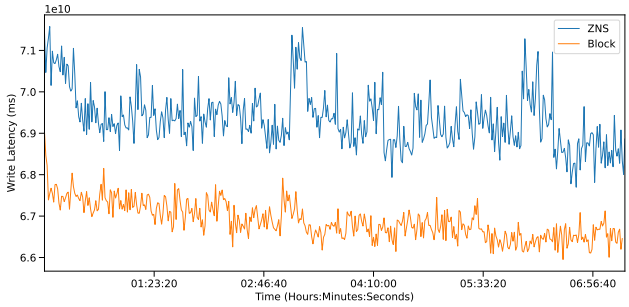

Figure 44: Disk Write latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

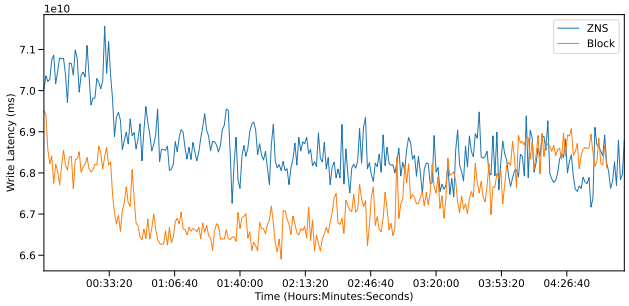

Figure 45: Disk Write latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

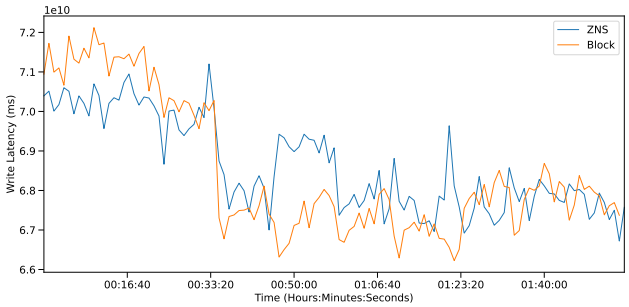

Figure 46: Disk Write latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

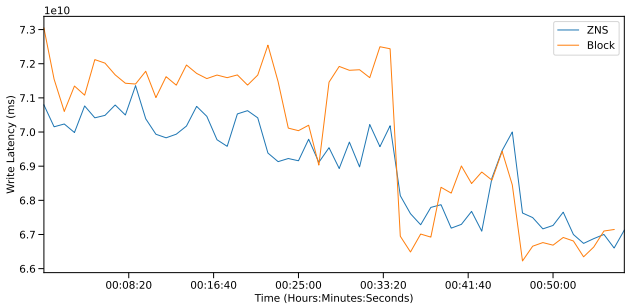

Figure 47: Disk Write latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

C.6 Get Throughput

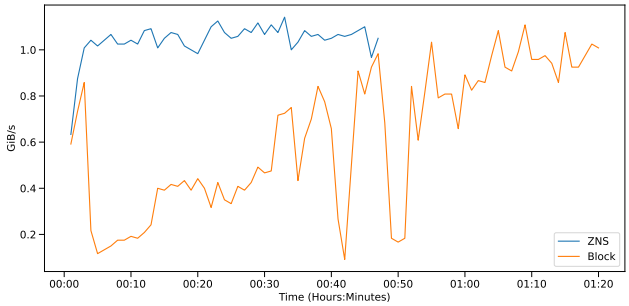

Figure 48: Cache Get Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:10 ratio.

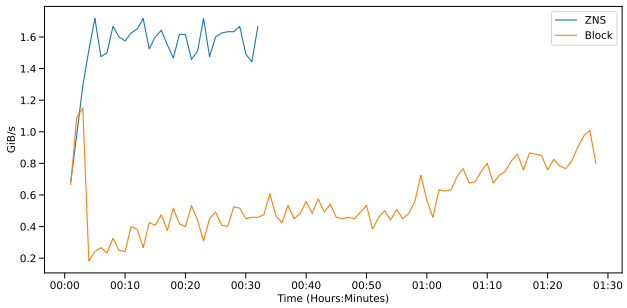

Figure 49: Cache Get Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:2 ratio.

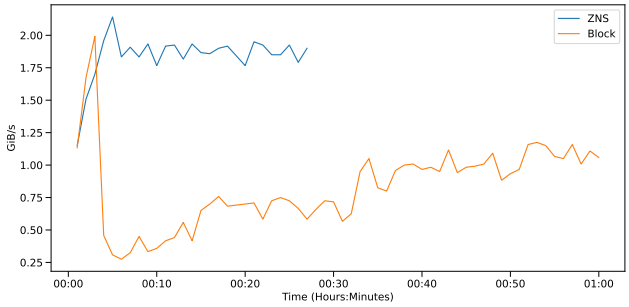

Figure 50: Cache Get Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

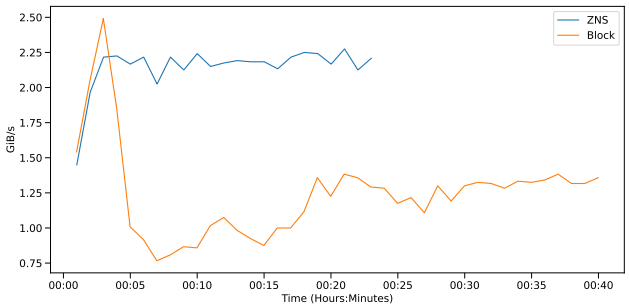

Figure 51: Cache Get Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

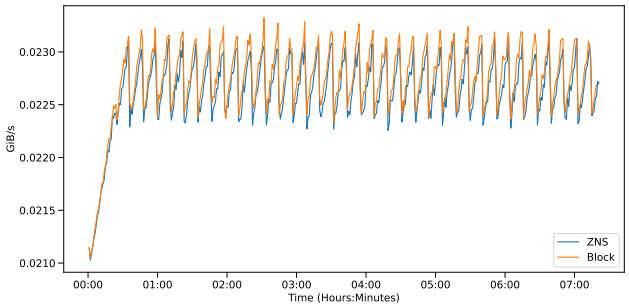

Figure 52: Cache Get Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:10 ratio.

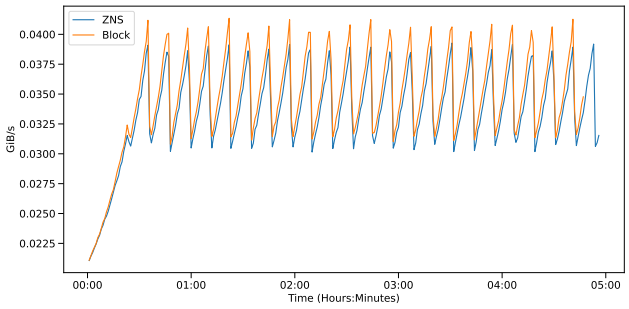

Figure 53: Cache Get Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:2 ratio.

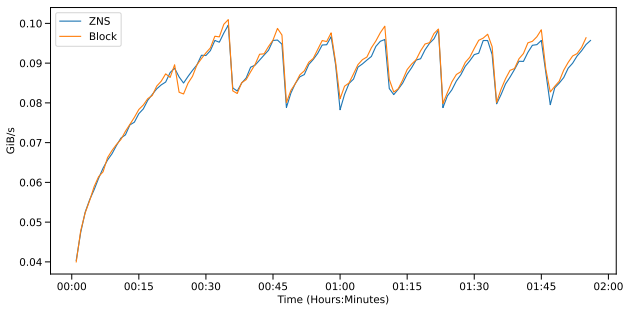

Figure 54: Cache Get Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

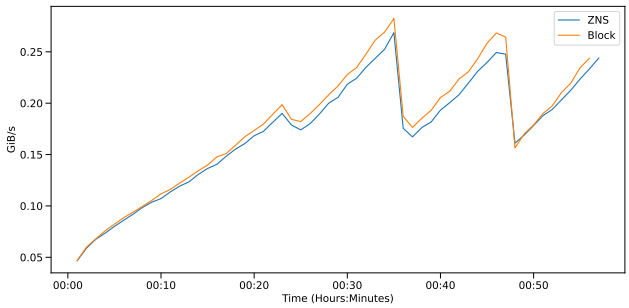

Figure 55: Cache Get Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

C.7 Read Throughput

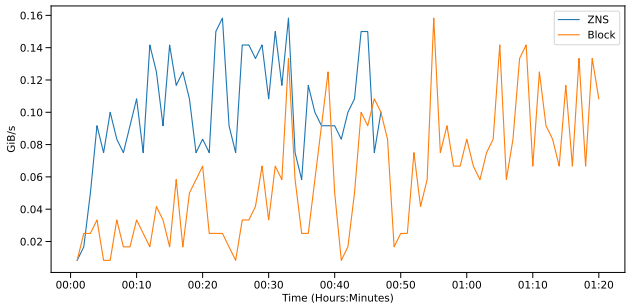

Figure 56: Cache Read Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:10 ratio.

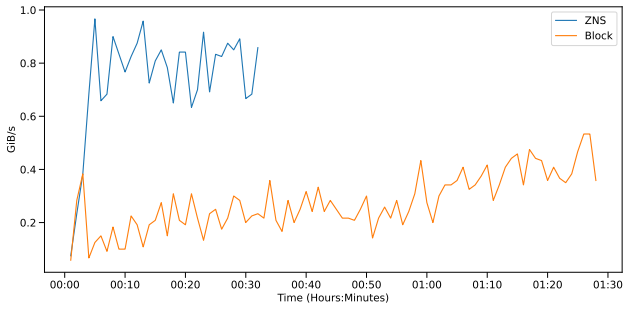

Figure 57: Cache Read Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:2 ratio.

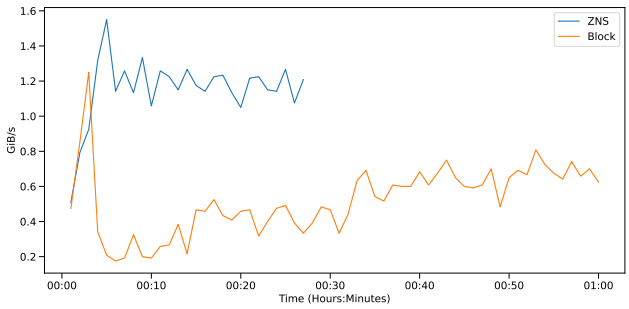

Figure 58: Cache Read Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

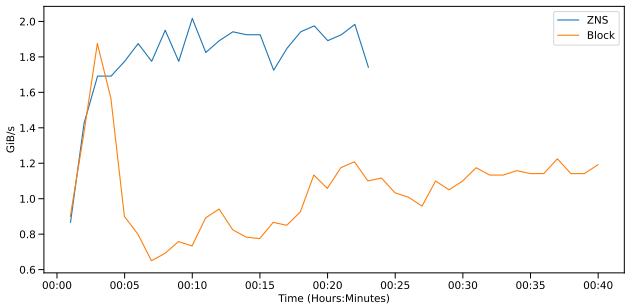

Figure 59: Cache Read Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

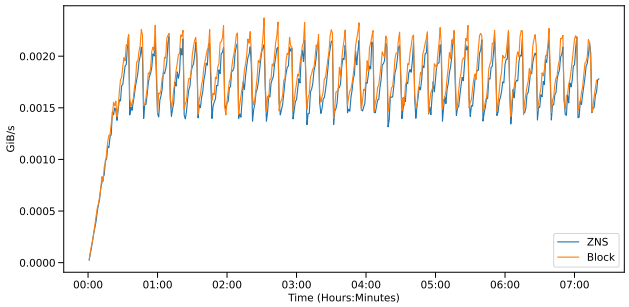

Figure 60: Cache Read Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:10 ratio.

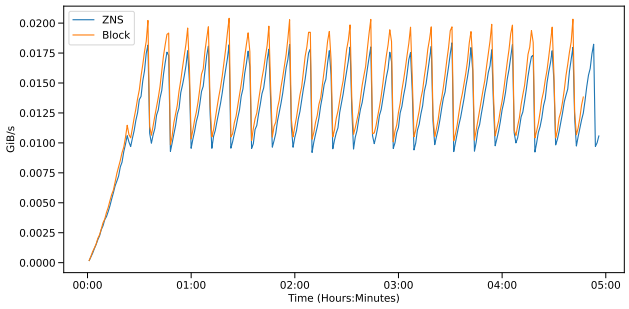

Figure 61: Cache Read Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:2 ratio.

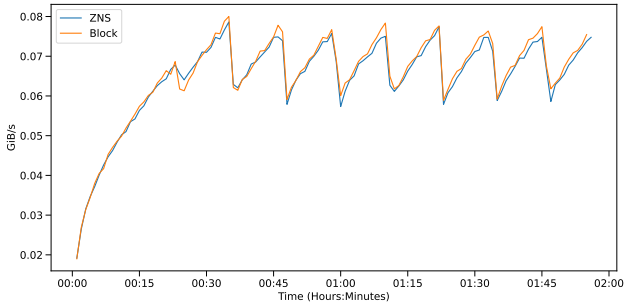

Figure 62: Cache Read Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

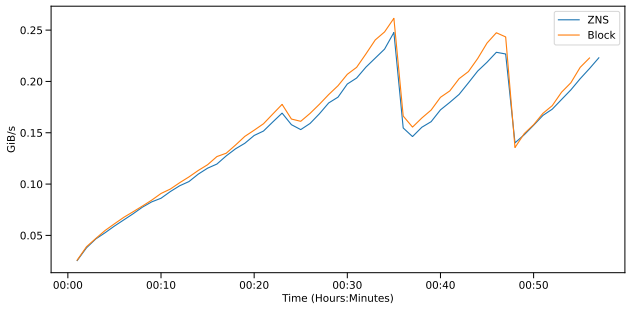

Figure 63: Cache Read Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

C.8 Write Throughput

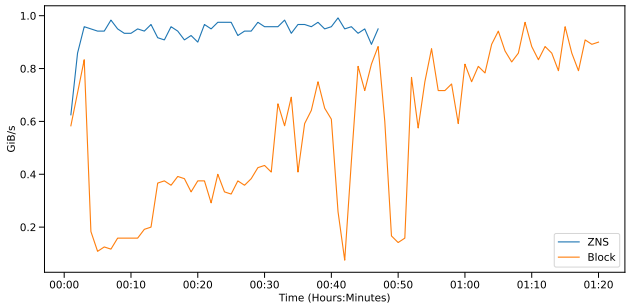

Figure 64: Cache Write Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:10 ratio.

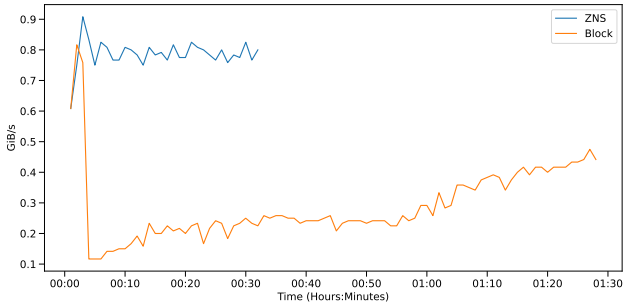

Figure 65: Cache Write Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:2 ratio.

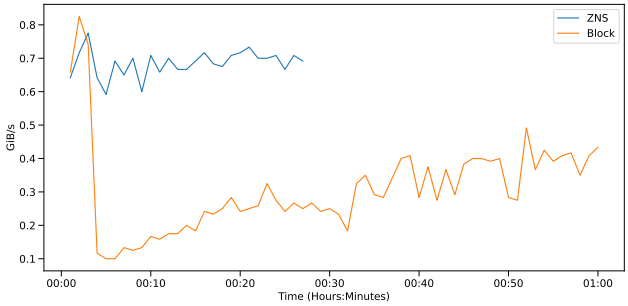

Figure 66: Cache Write Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

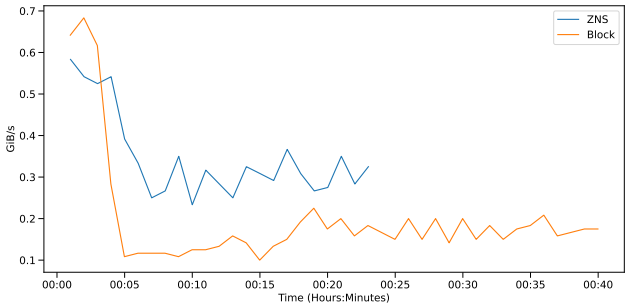

Figure 67: Cache Write Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

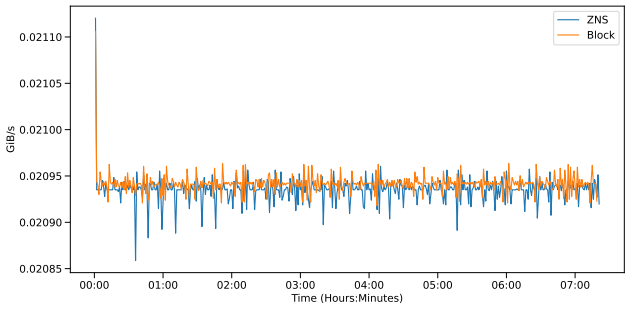

Figure 68: Cache Write Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:10 ratio.

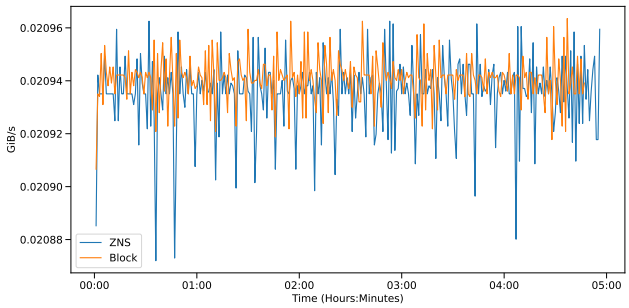

Figure 69: Cache Write Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:2 ratio.

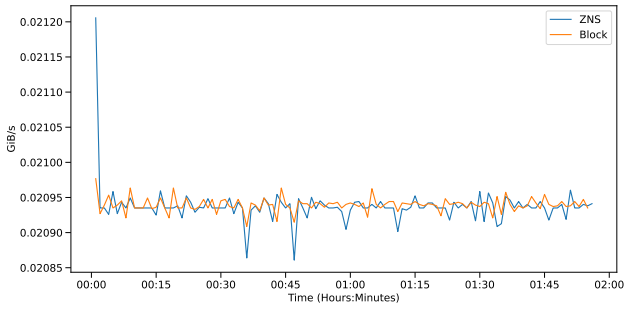

Figure 70: Cache Write Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

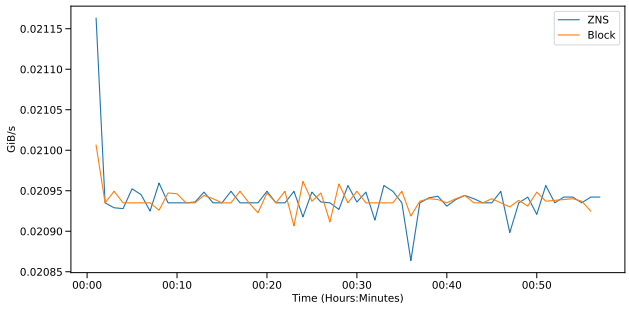

Figure 71: Cache Write Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

D HIT RATIO GRAPHS

CLRU was not evaluated, but we expected it to have a higher hit ratio compared to ZLRU.

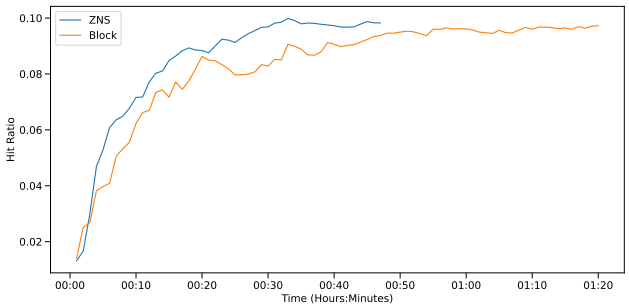

Figure 72: Cache Hit Ratio for 512M chunk size, Uniform distribution, and 1:10 ratio.

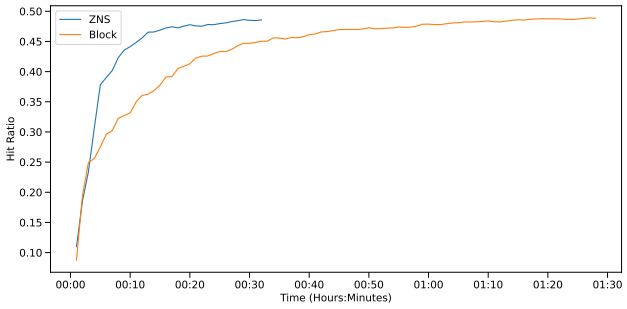

Figure 73: Cache Hit Ratio for 512M chunk size, Uniform distribution, and 1:2 ratio.

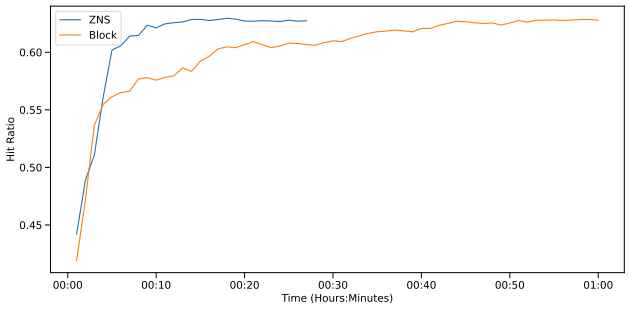

Figure 74: Cache Hit Ratio for 512M chunk size, Zipfian distribution, and 1:10 ratio.

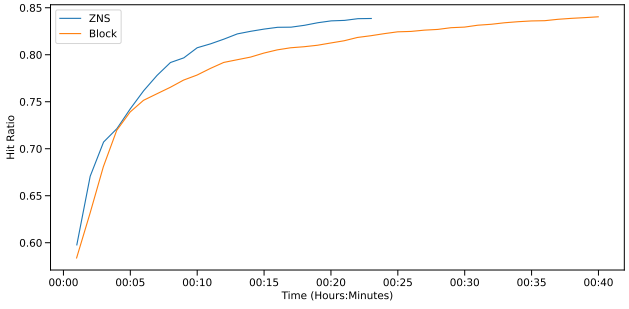

Figure 75: Cache Hit Ratio for 512M chunk size, Zipfian distribution, and 1:2 ratio.

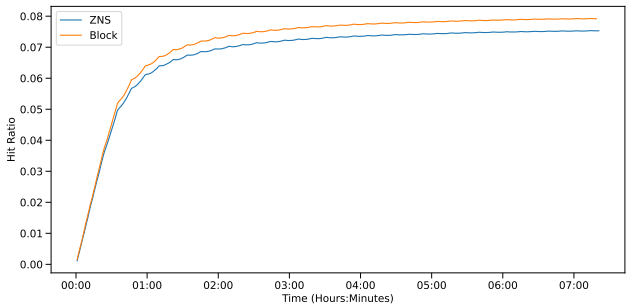

Figure 76: Cache Hit Ratio for 64K chunk size, Uniform distribution, and 1:10 ratio.

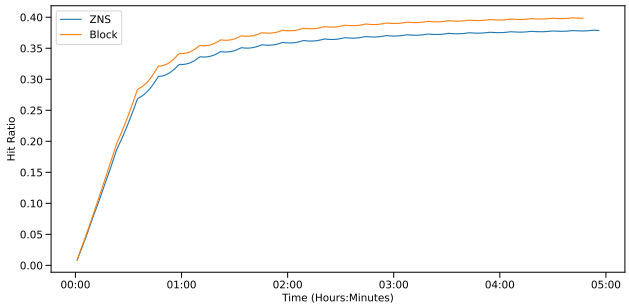

Figure 77: Cache Hit Ratio for 64K chunk size, Uniform distribution, and 1:2 ratio.

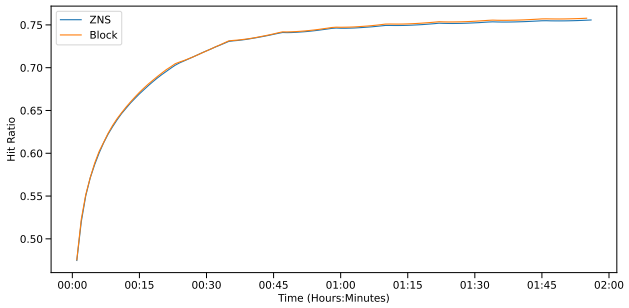

Figure 78: Cache Hit Ratio for 64K chunk size, Zipfian distribution, and 1:10 ratio.

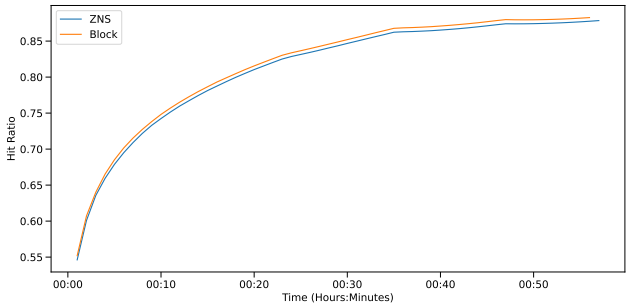

Figure 79: Cache Hit Ratio for 64K chunk size, Zipfian distribution, and 1:2 ratio.

E GC EXPERIMENT GRAPHS

No preconditioning phase was performed in these experiments in order to observe the effects of GC.

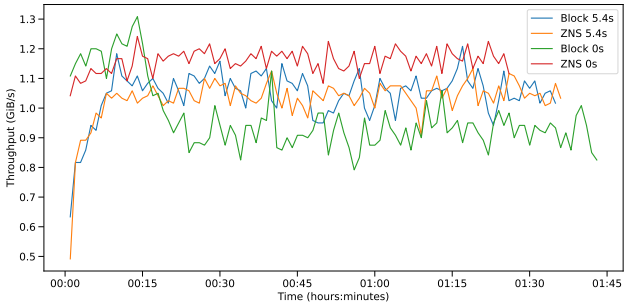

Figure 80: Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:10 ratio.

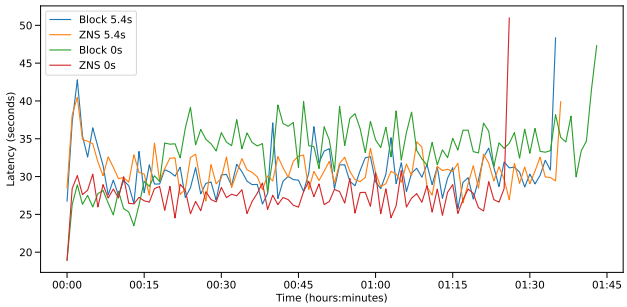

Figure 81: Latency for 512M chunk size, Uniform distribution, and 1:10 ratio.

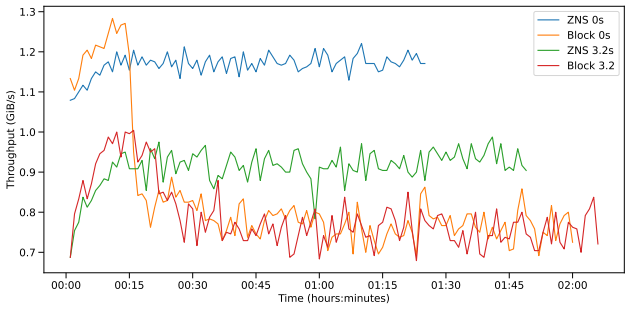

Figure 82: Throughput (GiB/s) for 256M chunk size, Uniform distribution, and 1:10 ratio.

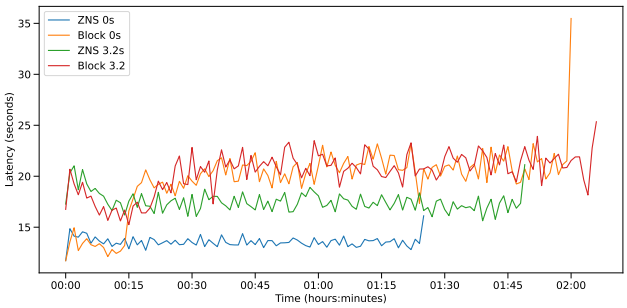

Figure 83: Latency for 256M chunk size, Uniform distribution, and 1:10 ratio.