When Do Zoned Namespaces SSDs Matter? A Comparative Study of Cache Workloads

by

John Ramsden

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

April 2026

© John Ramsden 2026

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

When Do Zoned Namespaces SSDs Matter? A Comparative Study of Cache Workloads

submitted by **John Ramsden** in partial fulfillment of the requirements for the degree of **Master of Applied Science** in **Electrical and Computer Engineering**.

Examining Committee:

Alexandra (Sasha) Fedorova, Professor, Electrical and Computer Engineering, UBC
Supervisor

Mohammad Shahrad, Assistant Professor, Electrical and Computer Engineering, UBC
Supervisory Committee Member

Sathish Gopalakrishnan, Associate Professor , Electrical and Computer Engineering, UBC
Supervisory Committee Member

Abstract

Remote object stores are widely used in cloud workloads for their high availability, redundancy, and elastic scalability, which allow users to offload infrastructure management. However, accessing these remote stores introduces additional latency and cost. To mitigate these drawbacks, local caching layers are often employed to improve performance and reduce request overhead.

While DRAM-based caches offer high performance, they are expensive and impractical for large datasets. Flash-based (SSD) caches provide a more cost-effective alternative, but their performance can degrade under heavy workloads due to background garbage collection (GC) processes inherent to conventional SSD architectures.

Zoned Namespaces (ZNS) SSDs have recently emerged as a promising alternative to traditional flash storage. By eliminating the need for device-level GC, ZNS SSDs can provide more predictable performance. However, they impose new constraints on data placement and require significant changes in cache design and management.

In this work, we design and implement a ZNS-aware caching system to evaluate the performance trade-offs between ZNS and traditional block-interface SSDs in caching workloads. Through experimental analysis, we identify when ZNS devices can outperform conventional SSDs, and when the benefits may not justify the required software complexity. We conclude by providing design recommendations for caching system developers considering the adoption of ZNS SSDs.

Lay Summary

Local caching systems are commonly used to speed up access to remote data and reduce data access costs. This study explores a new storage technology called Zoned Namespaces (ZNS) SSDs, which differ from conventional solid-state drives. ZNS SSDs offer the potential to improve the performance and efficiency of local caches while lowering storage costs.

We design and implement a system to evaluate this technology and share our findings on its advantages and limitations. Ultimately, we provide practical recommendations for caching system designers - highlighting when adopting ZNS SSDs can deliver meaningful benefits and when the transition may not justify the cost.

Preface

The work presented in consists of original research conducted by the author, with collaboration and input from Sam Cheng, Matthias Bjørling, Hans Holmberg, and Alexandra (Sasha) Fedorova.

Dr. Fedorova provided supervision and guidance that shaped the overall research direction and methodology. Sam Cheng contributed feedback and assistance related to aspects of the caching system's design and implementation. Matthias Bjørling and Hans Holmberg provided technical advice and feedback throughout the development, evaluation, and writing process. The author was responsible for the core system design, the evaluation methodology, and for conducting and analyzing all experiments.

Generative AI tools were used in preparing this thesis: ChatGPT was used to assist with structuring and proofreading sections of the written text. Claude Code was used to generate supporting scripts for data visualization and parsing, as well as for implementation-level debugging and planning. The system design, experimental methodology, analysis, and written content of the thesis were produced by the author.

This research builds upon and extends work completed as part of the courses CPEN 534, CPSC 508, and EECE 571. Work conducted in CPSC 508 informed the evaluation methodology, while the project developed in EECE 571 served as the foundation for the caching system used to evaluate ZNS devices in this thesis.

Table of Contents

Abstract iii

Lay Summary iv

Preface v

Table of Contents vi

List of Tables viii

List of Figures ix

List of Abbreviations xi

Acknowledgements xii

1 Introduction 1

2 Background and Related Work 3

3 Implementation 6
- 3.1 RemoteStore Module 8
- 3.2 ReaderPool and WriterPool Module 9
- 3.3 Server Module 9
- 3.4 ZoneList Module 9
- 3.5 Device Module 10
- 3.6 Eviction Module 11

4 Evaluation 13
- 4.1 Experimental Setup 13
- 4.2 Evaluation Goals 14
- 4.3 Parameter Evaluation 14

4.4 Real World Case Study 26
4.5 Summary of Findings 30

5 Conclusion 33

Bibliography 34

Appendices

A Object Storage Latency Evaluation 36

B Parameter Evaluation Read and Write Latency 38

List of Tables

4.1 Parameter space for evaluation. (*) For 1077 MiB chunks, only Zone LRU eviction is performed. 14
4.2 Consolidated P99 latency comparison. Percentages show relative latency change of best throughput on ZNS vs best on block-interface device. (ZNS / Block, ms) 23
4.3 Latency statistics for Zone LRU and Chunk LRU eviction policies. Each entry reports ZNS / block-interface latency, with the percentage indicating the relative change when using the block-interface device. (ms) 30

A.1 Latency evaluation for 64KiB Chunk size making requests to S3 36
A.2 Latency evaluation for 256MiB Chunk size Chunk size making requests to S3 36
A.3 Latency evaluation for 1077 MiB Chunk size Chunk size Chunk size making requests to S3 37

B.1 Total end-to-end latency request latency statistics for Chunk LRU eviction policy. Each entry reports ZNS / block-interface latency, with the percentage indicating the relative change when using the block-interface device. (ms) 38
B.2 Total end-to-end latency request latency statistics for Zone LRU eviction policy. Each entry reports ZNS / block-interface latency, with the percentage indicating the relative change when using the block-interface device. (ms) 38

List of Figures

3.1 High level Cache design. 8

4.1 Cache throughput (256 MiB chunk size, Zipfian distribution, cache size to workload ratio 1 to 10, Zone LRU eviction. Vertical lines show the location where the cache has first filled, and eviction begins. 17
4.2 Cache throughput (256 MiB chunk size, Zipfian distribution, cache size to workload ratio 1 to 10, Chunk LRU eviction. Vertical lines show the location where the cache has first filled, and eviction begins. 17
4.3 Hit ratios for the uniform and Zipfian distributions 18
4.4 Cache throughput for the uniform distribution. The boxplots show the median (center line), interquartile range (box), and whiskers extending to 1.5x the IQR. 19
4.5 Cache throughput for the Zipfian distribution. The boxplots show the median (center line), interquartile range (box), and whiskers extending to 1.5x the IQR. 19
4.6 End to end latency for the Uniform distribution. Vertical lines represent P99 latency. 22
4.7 End to end latency for the Zipfian distribution. Vertical lines represent P99 latency. 23
4.8 Read latency for the Uniform distribution at chunk granularity. Vertical lines represent P99 latency. 24
4.9 Read latency for the Zipfian distribution at chunk granularity. Vertical lines represent P99 latency. 24
4.10 Write latency for the Uniform distribution at chunk granularity. Vertical lines represent P99 latency. 24
4.11 Write latency for the Zipfian distribution at chunk granularity. Vertical lines represent P99 latency. 25

4.12 End-to-end throughput boxplots for WiredTiger workload. The boxplots show the median (center line), interquartile range (box), and whiskers extending to 1.5x the IQR. 28
4.13 End-to-end throughput for complete WiredTiger workload (Chunk LRU eviction). Vertical lines show the first eviction. . 28
4.14 End-to-end throughput for complete WiredTiger workload (Zone LRU eviction). Vertical lines show the first eviction. . . 29
4.15 End-to-end request latency for complete WiredTiger workload. Vertical lines represent P99 latency. 29

List of Abbreviations

GC	Garbage Collection
LRU	Least Recently Used
ZNS	Zoned Namespaces

Acknowledgements

Throughout my work, I have received support from many people. I would first and foremost like to thank Dr. Fedorova, whose mentorship has been invaluable in teaching me how to approach and conduct rigorous systems research. Without her guidance and encouragement, I would not have pursued research, and I am deeply grateful for the inspiration and support she provided throughout this process.

I am grateful to Sam Cheng for his input and support during the course of this research.

I would like to thank Matias Bjørling and Hans Holmberg for their advice and expertise, which greatly informed my work.

I would also like to thank my friends who reviewed drafts, listened to presentations, and provided thoughtful feedback along the way. In particular, Jeremy Shulman offered many detailed reviews and insights.

Finally, I would like to thank my family for their unwavering support throughout this endeavor, in more ways than I can count.

Chapter 1

Introduction

Cloud-based remote storage enables users to offload the burden of maintaining elastic, highly available, secure, and redundant storage. However, such convenience comes at a cost—both financial and performance-related. For example, in AWS S3 Standard, `PUT` requests cost $0.005 per 1,000 operations and `GET` requests $0.0004 per 1,000 [1]. Over time, these charges accumulate into substantial overhead. Moreover, accessing remote storage introduces additional latency, especially when requests originate outside the provider's network. In our experiments, transferring 256 MiB objects incurred an average latency of 3.2 s (Appendix A).

To mitigate these drawbacks, systems commonly utilize in-memory caches. While DRAM offers exceptional throughput and low latency, its cost scales poorly with capacity. As storage demands grow, flash-based storage is often introduced as a more affordable, higher-density caching tier.

Flash memory does not support in-place overwrites. Instead, new writes must target fresh locations, while internal mappings maintain the illusion of in-place updates. The smallest erasable unit on flash is an *erase block*, which contains multiple *pages* – the smallest writable units. As data is overwritten, stale pages accumulate. To reclaim space, the device periodically migrates valid data and erases entire blocks in a process known as garbage collection (GC). GC introduces additional writes, the so-called *write amplification*, and degrades performance.

Zoned Namespaces (ZNS) SSDs offer an emerging alternative to conventional block-interface SSDs. ZNS drives expose a similar interface but impose additional restrictions that allow the device to maintain higher throughput and lower latency by eliminating device-side GC [2]. While this enables a range of GC minimization strategies, including placement-aware writes, this work focuses on workload-aware eviction and cleaning policies for cache workloads.

In ZNS SSDs, data must be written sequentially within large, fixed-size regions called zones, typically ranging from hundreds of MiB to a few GiB in size. Zones are also the smallest erasable units; however, because they are much larger than the granularity at which data is written, space management becomes more complex. These restrictions simplify device-side management but shift the burden of complexity to the host. When used as a cache backing store, ZNS devices further complicate eviction, since data can only be erased at the granularity of entire zones. Consequently, depending on the eviction algorithm, host-side GC may again be required.

This work investigates the following core question: *Under which workloads do ZNS SSDs outperform conventional block-interface SSDs, and under what conditions do block-interface SSDs perform equally well or better?* Our goal is to identify criteria that help system engineers decide when adopting ZNS SSDs provides a tangible benefit - and when the additional software complexity and cost of deploying new hardware may not justify the transition. Our results show that throughput alone favors neither device, but tail latency consistently favors ZNS SSDs if used with zone-aware eviction.

Chapter 2

Background and Related Work

Flash-based storage is often used as a backing medium for persistent caches because it offers higher throughput and lower latency than HDDs or networked storage. Recent efforts, such as MongoDB's exploration of local disk caches [3], highlight the importance of reducing access costs in production environments [4]. While slower than DRAM, flash provides lower cost per byte and is non-volatile [5].

ZNS SSDs improve efficiency by reducing the overhead of the traditional *block-interface* [2]. They enforce sequential writes within predefined regions, called *zones*, which are reset at the *erase-block* granularity (the smallest unit of physical erasure). This model eliminates random in-place writes and removes the need for traditional device-side GC, where the device reclaims invalidated blocks by clearing erase blocks. As a result, ZNS SSDs deliver more predictable performance under high utilization and provide greater usable capacity by reducing over-provisioning. Bjørling et al. [2] explored the impact of using ZNS SSDs on latency and throughput by modifying the F2FS file system and RocksDB to specialize them for the ZNS interface. Inspired by their work, we instead focus on how ZNS can be leveraged in caching systems and examine the effects of different eviction policies, and various parameters.

In addition to the sequential write constraint, ZNS devices impose an *active zone limit*, specifying the maximum number of zones that can be written concurrently. A zone is considered active once it has been written to and remains so until it is either completely filled or explicitly closed. While ZNS provides efficiency and capacity benefits, it shifts responsibility to the host, requiring complex zone management policies in software.

Ordinarily, when using a ZNS device, writes within a zone are constrained by a sequential write pointer: only one thread can advance the write pointer

at a time. This typically requires careful tracking of the write pointer and synchronization across threads to ensure correctness. Zone Append offloads this management to the device [6]. Instead of specifying the write pointer location explicitly, the host issues a Zone Append with the data, and upon completion, the device returns the logical block address where the data was placed (i.e., the prior write pointer location). This mechanism not only simplifies write pointer management but also enables multiple threads to issue appends concurrently, each receiving the resolved address from the device. This effectively allows a queue depth greater than one within a single zone.

Existing cache systems such as CacheLib [7] often adopt region-based designs in which data is first written to memory regions and later flushed to disk in bulk. This approach amortizes flash write costs and aligns naturally with the sequential write and reset semantics of ZNS SSDs. Prior work has explored adapting such caches to ZNS devices. Yang et al. [8] extend CacheLib with multiple ZNS-based backends and evaluate eviction policies that either map regions directly to zones, eliminating software-managed GC, or multiplex multiple regions within a zone, trading increased cleaning overhead for lower latency. Similarly, Lv et al. [9] propose a ZNS-aware caching system designed to exploit zone semantics for improved scalability and concurrency. However, these studies primarily focus on how to build caches on ZNS SSDs and do not treat block-interface SSDs as a competing design point under identical cache architectures and eviction semantics. As a result, they do not address whether, for certain workloads, block-interface SSDs may exhibit comparable or superior behavior, nor do they isolate how eviction granularity and workload characteristics interact with device-side GC across the two interfaces. In contrast, this work asks a broader question: when and why do ZNS SSDs provide tangible benefits over block-interface SSDs for cache workloads? To answer this, we implement a unified cache architecture across both device types, apply identical eviction policies, and evaluate a wide range of workloads, cache sizes, and parameters. This enables direct, one-to-one comparisons that expose the conditions under which ZNS semantics meaningfully impact performance.

More recently, Flexible Data Placement (FDP) has emerged as an alternative NVMe interface for reducing write amplification. Unlike ZNS, which exposes zone semantics and shifts GC responsibility entirely to the host, FDP retains device-side control over logical-to-physical mapping and GC. The host instead provides optional placement hints that allow the device to segregate data with different lifetimes. This design preserves backward com-

patibility with conventional block-interface access and requires substantially less host-side software complexity than ZNS. However, because placement decisions are advisory rather than explicit, the host has less control over data layout and cannot fully eliminate device-side GC. Recent work has demonstrated that FDP can achieve near-ideal write amplification in production cache workloads [10], though Song et al. [11] show that effectiveness varies significantly across devices and workloads depending on opaque firmware-level policies. While FDP represents a promising direction that balances ease of adoption against GC reduction, it does not yet offer the same degree of host control that ZNS provides. Evaluating FDP as a cache backing store under the methodology used in this work is an interesting direction for future work.

Chapter 3

Implementation

To isolate the impact of device interface semantics on cache eviction behavior, we designed and implemented a purpose-built SSD cache. The cache provides a controlled environment in which identical (to the degree possible) eviction and admission policies can be evaluated across both block-interface and ZNS SSDs, while respecting the constraints and capabilities of each device type. By implementing the same cache architecture atop both interfaces, we enable direct, one-to-one comparisons that attribute observed performance differences to device semantics rather than to algorithmic or implementation artifacts.

The SSD cache leverages a remote backing store. Although the cache operates atop a non-volatile layer, it does not support persistence across server restarts. This omission does not alter the cache's core algorithms and is not required to answer the research question we seek to address. Adding persistence would primarily require retaining additional metadata and implementing a mechanism for crash consistency; this is orthogonal to our study. Our cache allows concurrent reads and writes (to fully saturate the disk bandwidth) and is implemented in Rust.

We define the unit of caching as a *chunk*. We request chunks from the remote store, and upon admitting data into the cache, we write it to the local disk at the granularity of a chunk. Smaller chunks can reduce read and write latency since they enable fine-grained access and reduce the amount of data that must be fetched or written for each request. Small chunks also experience reduced read amplification (where more data is fetched from the remote store than was actually requested) compared to larger sizes. However, they increase metadata overhead and fragment the underlying storage, eventually requiring GC (either host-side on ZNS or device-side on a block-interface SSD). In contrast, larger chunks amortize per-request overhead as well as GC overhead and improve sequential throughput by minimizing metadata management and better aligning with the write units

of flash-based storage. To study this trade-off, the system allows configurable chunk sizes at initialization, enabling evaluation of how varying chunk sizes affect performance.

The high-level design of our cache is depicted in Fig. 3.1. The cache supports two backends, which we evaluate: a block-interface backend and a ZNS backend. Although block-interface SSDs lack a hardware notion of zones, we adopt a zone abstraction to unify logic between the two backends (§3.4). On block-interface SSDs, data is logically partitioned into fixed-size regions, resulting in an interface that closely resembles that of a ZNS device. This approach allows us to maintain nearly identical implementations across both backends, eliminating differences in cache design. Consequently, we can apply comparable eviction algorithms operating at region granularity. Furthermore, enforcing sequential writes within regions improves write locality and reduces internal fragmentation on block-interface SSDs. Similar abstractions have been used in prior caching systems [7]. The details of logical partitioning are further described in §3.4.

Throughout our design we took advantage of Rust's stackless coroutines (also known as user threads) to manage concurrent requests to the cache, and async operations to remote stores. This approach avoids the heavy allocation cost of kernel threads, and context switching between user threads is faster as there is less state to save and restore. Additionally we frequently utilize Rust channels for communication (and as a synchronization method) between various threads.

The system follows a client–server architecture in which each incoming request is handled by a newly spawned user thread. If the requested data is not present in the cache, the user thread issues a query to a remote data store. The retrieved data is immediately returned to the server and forwarded to the client. In parallel, the data is dispatched to a pool of writer threads responsible for persisting the response in the cache. If the data is requested again before the data has been persisted to disk, the data is served directly from RAM where it is already resident. Once the write operation completes, the request user thread updates the relevant data structures and then terminates. For requests that can be satisfied by the cache, the user thread delegates the read to a reader thread pool, which returns the cached data to the server, and the server relays the response to the client.

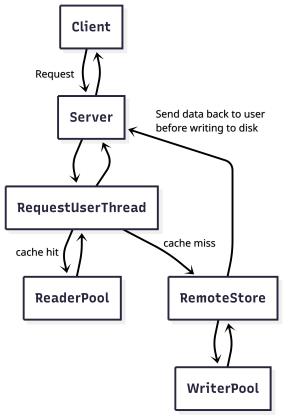


Figure 3.1: High level Cache design.

The cache is composed of several key modules. The *server* module (§3.3) acts as the entry point to the cache and spawns user threads to service client requests. The *RemoteStore* module (§3.1) provides an interface for interacting with remote data stores. The *device* module (§3.5) abstracts underlying storage devices, exposing a unified interface for both ZNS and block-interface devices. The *eviction* module (§3.6) manages cache eviction and invalidation through both periodic and foreground eviction. The *readerpool* (§3.2) and *writerpool* (§3.2) modules handle read and write requests, respectively. The *ZoneList* module (§3.4) manages zone state.

3.1 RemoteStore Module

The `RemoteStore` module defines an interface for implementing different backends such as S3. In our design, we implement an emulated backend to support evaluation. This backend simulates a remote store, including optional artificial latency (Appendix A) to mimic response times from an object store. It generates data (within the artificial latency period) based on the requested key, followed by random but deterministic bytes derived from a seed. This design ensures we can verify data correctness even though the actual contents are not meaningful.

3.2 ReaderPool and WriterPool Module

The reader and writer pools provide a simple interface for the server to perform disk operations. Each pool spawns a configurable number of threads and waits for requests on channels. When a request arrives, the pool calls the device through the device module, waits for completion, and then sends the response back to the caller. This design simplifies device access by abstracting away concurrency management and delegating parallelism to the pools.

3.3 Server Module

The server module functions as the core of the cache, accepting requests, and coordinates communication between various modules, and ultimately is responsible for sending a response back to the client.

The server module keeps the "state map" (a mapping from keys to on-disk locations), as requests come in. If the requested data is not present in the state map, the server fetches it from the remote store, and immediately returns to the client. Since the data has not yet been persisted to the cache, the state is marked as "in progress". In this way, if a subsequent request is made for the same data, it is served from a temporary RAM buffer. The data is then sent to the writer pool. Once the data is persisted to disk, the entry is updated in the map as complete, and subsequent requests will be served from disk.

3.4 ZoneList Module

One of the main challenges in zone management is the *active zone limit*. To address this, we developed the `ZoneList` module, which maintains the state of all zones and provides an interface for managing their availability. The module allocates a zone when one is available, or notifies the caller when the device is full and eviction is required. It also supports returning zones after use and updates their state appropriately when eviction occurs.

In addition to tracking zone state, `ZoneList` implements the abstraction layer required to emulate zones on block-interface SSDs. This is achieved by maintaining a list of logical regions, each containing a set of available "chunks". When a new write is issued, the system requests a "zone" from the `ZoneList` module - specifically, a region containing a free chunk. The

module grants exclusive access to the selected region and internally marks it as active. Once the requester completes its operation, the zone is returned to the pool and its state is updated accordingly. This design is lightweight, consisting primarily of constant-time operations, and allows the system to preserve a consistent interface and set of algorithms across both ZNS and block-interface SSDs.

3.5 Device Module

The device module provides an abstraction layer over both ZNS and block-interface devices. We implemented two device types: `Zoned`, representing a ZNS device, and `BlockInterface`, representing a conventional block-interface device. Each implementation handles the specifics of its underlying device type.

Both of our device backends interface with their respective hardware through `libnvme`. This library provides low-level direct access (no kernel caching) to NVMe devices (both ZNS and block-interface). `libnvme` also exposes support for Zone Append.

The `Zoned` device implementation provides a few important functions required for ZNS - functionality to reset and "finish" zones (mark them as no longer active). The device module holds a ZoneList, and uses it to manage the different states zones may live in.

The `BlockInterface` implementation introduces a logical zoning abstraction over the block-interface device to enable a unified cache architecture across both block-interface and ZNS SSDs. Importantly, this abstraction does not impose the write or reset constraints of ZNS on the block-interface device: zones are logical groupings used solely for resource accounting and eviction, and the block-interface SSD remains free to perform in-place updates and internal data relocation as dictated by its flash translation layer (FTL).

Rather than disadvantaging the block-interface SSD, this design reflects common practice in SSD-backed caches [7], which often organize data into large, sequentially written regions to reduce GC overhead. By encouraging sequential writes within logical regions, the abstraction may in fact improve block-interface SSD behavior by aligning host write patterns with FTL expectations. As a result, our design does not penalize block-interface SSDs.

3.6 Eviction Module

The *eviction module* hosts an OS thread which periodically (based on a configurable interval) wakes up and performs eviction if needed. The module also listens on a channel for foreground eviction requests in case the system completely runs out of capacity before the periodic eviction can sufficiently evict.

Eviction Algorithms: We implement two eviction algorithms, each serving a distinct purpose: *Zone LRU* and *Chunk LRU*. The intention here is to cover the granularity spectrum, where we are doing fine-grained and coarse-grained (at the granularity of a Zone) eviction.

Zone LRU is a simple greedy [12] eviction policy that operates at the zone level and eliminates the need for host-side GC. When a zone becomes full, it is inserted into a zone-level LRU queue. If any chunk within that zone is accessed, the entire zone is "promoted" within the queue. During eviction, the least recently used zones are reset, and all data within them is discarded. This coarse-grained approach avoids host-side GC overhead but can slightly reduce hit ratio, as demonstrated in §4, because a zone can have a mix of "hot" and "cold" data. This design closely resembles zone-aligned region eviction policies explored in prior ZNS cache systems, including the one-to-one region-to-zone mappings evaluated by Yang et al. [8] and the zone-based eviction strategies employed by ZonedStore [9].

Zone LRU exposes two tunable thresholds: *high-evict* and *low-evict*. *high-evict* defines the number of occupied zones where eviction is triggered, while *low-evict* specifies the target number of occupied zones once eviction completes. As the cache fills and the number of occupied zones reaches *high-evict*, eviction proceeds until the count is reduced to *low-evict*.

Chunk LRU operates at a finer granularity and requires host-side GC on ZNS devices. On block-interface SSDs, it behaves as a conventional greedy LRU policy, invalidating least-recently-used chunks directly and reusing their space. However, ZNS devices do not permit partial overwrites, requiring a two-phase process of *invalidation* and *cleaning*.

On ZNS Chunk LRU is inspired by the cost-benefit cleaning of Rosenblum et al. [12], introducing heuristics that prioritize reclaiming highly invalid zones while avoiding expensive data movement with minimal benefit. A global chunk-level LRU maintains recency across the entire cache, while a per-zone priority queue tracks validity. Similar approaches are explored in prior work

to trade increased software-managed GC overhead for improved hit ratio and eviction precision [8].

As chunks are invalidated by the global chunk-level LRU, we update a per-zone priority score equal to the number of invalid chunks within the zone. Zones with higher priority (i.e., fewer remaining valid chunks) are selected first during cleaning, as reclaiming these zones requires migrating less live data to new zones. This cost–benefit cleaning strategy minimizes situations in which large amounts of data are migrated while reclaiming little free space.

The algorithm exposes three tunable thresholds: *high-evict*, *low-evict*, and *clean*. *high-evict* specifies the number of in-use valid chunks that trigger invalidation, *low-evict* defines the target number of valid chunks after invalidation, and *clean* controls how many unused chunks should be available once cleaning completes. Separating the *clean* threshold from invalidation enables cost-benefit heuristics to skip low-reward cleaning opportunities and focus on the most space-efficient zones.

On zoned drives, eviction proceeds as follows:

1. Invalidation phase: invalid chunks are marked, and their zones' validity scores are updated in a priority queue.
2. Cleaning phase: zones are selected based on priority. For each candidate zone, valid chunks are first buffered in memory. The zone is then erased and returned to the zone state manager for reuse. Buffered chunks are subsequently written to new locations by the existing writer pool with elevated priority to prevent blocking. Once rewritten, the chunk locations are remapped. During the cleaning phase, if requests to mid-transfer chunks are made, they are served from RAM until they have been completely persisted to disk.

This design leverages the existing writer pool to handle eviction-related writes, reducing system complexity and reusing the same mechanisms for tracking zone availability.

On block-interface SSDs, GC is performed transparently by the device firmware using a dedicated embedded processor. When GC responsibility is shifted to the host, as required by Chunk LRU on ZNS devices, these operations use host CPU cycles. This represents an additional cost beyond the I/O overhead of data relocation.

Chapter 4

Evaluation

4.1 Experimental Setup

All experiments perform read-only workloads at the client level. We evaluate a cache that admits data on read misses, where writes to the underlying device occur only when data is fetched from the remote store and inserted into the cache. As a result, reported write latencies correspond exclusively to cache-admission writes rather than client-initiated writes. We focus on this configuration because it reflects our target use case. We defer the analysis of other caching policies to future work.

All experiments were conducted on an Intel server equipped with 256 GiB of RAM and two 16-core Intel® Xeon® Silver 4216 processors clocked at 2.10 GHz. The system runs Ubuntu 24.04 with Linux kernel version 6.14.0. All experiments were performed using the noop I/O scheduler.

Both storage devices used in the evaluation share the same underlying hardware platform, differing only in the applied firmware:

- **ZNS SSD** (Western Digital Ultrastar® DC ZN540): 950.8 GiB, 904 zones
- **Block-interface SSD** (Western Digital Ultrastar® DC SN540): 894.3 GiB

The discrepancy in usable capacity arises from additional space reserved internally by the block-interface SSD to support device-side GC.

Before running the core experiments, we conducted a parameter sweep to determine suitable eviction thresholds. For each workload distribution and eviction policy, thresholds were tuned experimentally until performance stabilized.

4.2 Evaluation Goals

Our evaluation is guided by the following research questions:

RQ1: Device Comparison under GC Pressure: Determine under which workloads and eviction policies ZNS SSDs outperform block-interface SSDs, and when block-interface SSDs achieve comparable or superior performance.

RQ2: Cost of Host-side Garbage Collection: Quantify the performance overhead introduced by shifting GC responsibilities from the device to the host in ZNS-based caches. When ZNS requires host-side GC, does it still outperform a block-interface SSD performing device-side GC, or do its advantages disappear?

RQ3: Workload and Parameter Sensitivity: Identify which workload characteristics (access distribution, chunk size, cache pressure) most strongly influence GC behavior and performance.

RQ4: Real-world Applicability: Evaluate whether conclusions drawn from synthetic workloads hold under realistic workloads where this cache is likely to be employed.

4.3 Parameter Evaluation

We evaluate several parameters to determine how different drives perform under varying conditions. The parameters include cache-to-workload ratio, access distribution, chunk size, and eviction algorithm. We run all experiments across the entire device to avoid artifacts caused by reduced device-side GC when under-utilizing block-interface SSDs.

Table 4.1 summarizes the parameter space.

Table 4.1: Parameter space for evaluation. (*) For 1077 MiB chunks, only Zone LRU eviction is performed.

Parameter	Values
Distribution	Uniform random, Zipfian
Chunk size	64 KiB, 256 MiB, 1077 MiB*
Cache to Workload ratio	1:10, 1:2
Eviction algorithm	Zone LRU, Chunk LRU

Distribution: We select uniform random and Zipfian distributions for the following reasons. Uniform random access provides a baseline workload with no locality, representing a worst-case scenario for caching and eviction mechanisms. This allows us to isolate system and device behavior, including GC overhead, without the influence of hot-key effects. In contrast, the Zipfian distribution models workloads that more closely approximate real-world behavior, where a small subset of entries is accessed frequently while others remain rarely accessed. We generate Zipfian workloads using the YCSB Zipfian generator with the default parameter of 0.99 [13, 14].

Chunk size: We evaluate three chunk sizes: a small chunk size of 64 KiB, a mid-size of 256 MiB, and a maximally large size of 1077 MiB, corresponding to the full zone capacity of our ZNS SSD [1]. We select relatively large chunk sizes to model a cache operating in front of a remote cloud store. In these settings, frequent access to the remote store incurs both latency and monetary cost. Larger chunks allow the cache to retrieve extensive regions from the remote store per request, reducing the rate of access and consequently lowering both performance overhead and cost. With such large chunks, even on block-interface SSDs, GC is minimized due to reduced fragmentation and mostly sequential writes. Larger chunks have the downside that they can end up resulting in over fetching from a remote, where the user only requires a small subset of a chunk, and the rest may go unused, therefore depending on the workload the chunk size must be tuned accordingly.

For chunk sizes equal to zone size (1077 MiB), Zone LRU and Chunk LRU both essentially become the same algorithm due to there only being a single chunk within each Zone. Therefore we only perform experiments on Zone LRU for 1077 MiB chunks.

Cache to Workload ratio: We consider two cache-to-workload ratios. A 1:10 ratio represents a scenario where only 10% of the workload fits in the cache, resulting in frequent evictions. A 1:2 ratio represents a scenario where 50% of the workload can be cached. These ratios allow observing how devices behave at different chunk sizes under conditions of both high and low cache pressure relative to the workload size. With a higher hit ratio the performance of the drives would not be as prominent due to reduced writes, and therefore we do not explore very high hit ratio workloads.

We focus in particular on how eviction algorithms affect performance. By

[1] 1077 MiB is the zone size for our evaluation platform, but this can vary on alternate platforms

comparing algorithms that require host-side GC (Chunk LRU eviction on ZNS) against those that do not (Zone LRU eviction, and block-interface Chunk LRU eviction), we aim to quantify the overhead of shifting GC responsibilities from hardware to software. Specifically, we ask (RQ2): when ZNS requires host-side GC, can it still outperform a block-interface SSD performing device-side GC? What do the effects of host-side GC look like in practice, and how do they impact throughput and latency?

With respect to GC, our goal (RQ3) is to identify configurations that most likely trigger device-side GC on block-interface devices. Device-side GC can cause severe performance degradation and increased latency. By pinpointing workloads that tend to activate device-side GC, we can determine which workloads stand to benefit the most from ZNS devices.

Our workloads consist of executing a sequence of calls to the cache according to our experimental parameters. We pre-generate workload traces 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 gather metrics we inline timers directly in code and measure between specific code sections. All benchmarks were executed with 64 reader and writer threads, and unbounded user-threads. Clients send requests to the cache using 64, 256, and 1024 client threads for 1077 MiB, 256 MiB and 64 KiB chunk sizes respectively to saturate the device. For this experiment we only issue requests at the same granularity as chunk size. We interface with the raw device directly using NVME commands, bypassing the OS buffer cache thus avoiding the additional variable of another layer of caching.

Prior to all experiments we issue a preconditioning phase consisting of filling the disk to capacity with a random write 64KiB block size workload to ensure a baseline for each disk type. The preconditioning phase helps simulate a fully utilized disk. For the 64 KiB, 256 MiB and 1077 MiB chunk benchmarks, the total disk I/O performed is fixed at 12 TiB, including both disk reads (cache hits) and writes (cache misses).

Fig. 4.1 demonstrates an example of a complete workload, along with the "cliff" that occurs on the block-interface device during the fill phase (close to minute 1), and the subsequent recovery (concluding close to minute 40), followed by a steady state. We attribute this known performance pattern [15, 16] to foreground GC triggered under sustained high write pressure. The initial throughput collapse occurs when the device enters an emergency GC regime, which is exacerbated by the highly fragmented state induced

during the preconditioning phase. As the workload progresses, throughput temporarily improves as writes exhibit a more sequential allocation pattern, allowing the device to rebuild free block headroom and reduce GC interference.

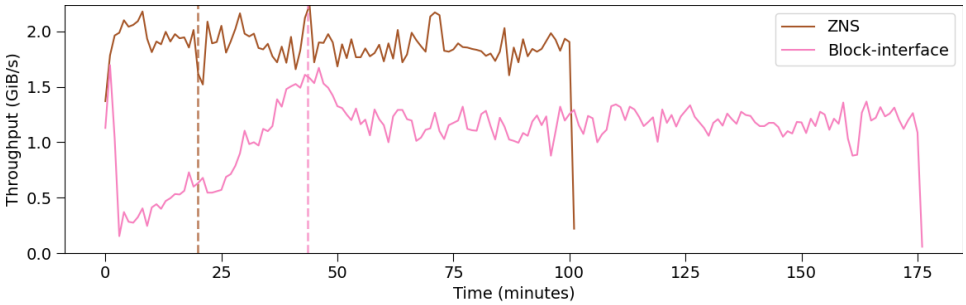


Figure 4.1: Cache throughput (256 MiB chunk size, Zipfian distribution, cache size to workload ratio 1 to 10, Zone LRU eviction. Vertical lines show the location where the cache has first filled, and eviction begins.

Fig. 4.2 shows the same workload under Chunk LRU. Under this configuration, the ZNS device also exhibits a pronounced performance cliff beginning around minute 20, which coincides with the onset of eviction and the associated host-side GC. Together, these results indicate that performance during the initial filling phase does not reflect the steady-state behavior of the system. Consequently, we base our analysis primarily on measurements collected after eviction begins, when the system has reached steady state.

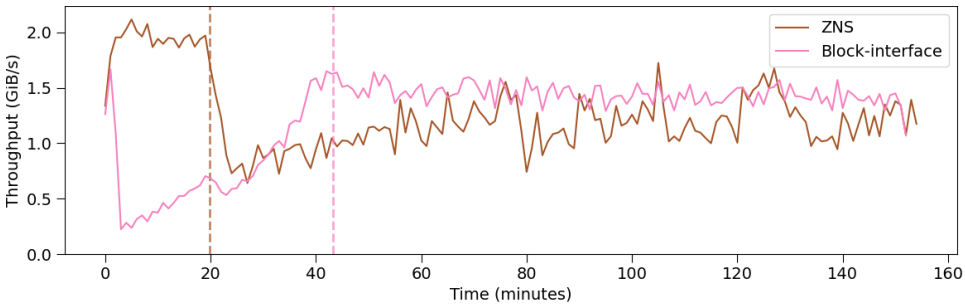


Figure 4.2: Cache throughput (256 MiB chunk size, Zipfian distribution, cache size to workload ratio 1 to 10, Chunk LRU eviction. Vertical lines show the location where the cache has first filled, and eviction begins.

For these synthetic workloads, we observe comparable hit ratios across eviction policies on ZNS devices, with differences that are effectively negligible (Fig. 4.3). This behavior arises from temporal clustering induced by the cache write path rather than the workload alone. Writes are appended sequentially within a zone, and early in execution, frequently accessed (hot) chunks are written in close temporal proximity. As a result, hot chunks tend to be co-located within the same or adjacent zones.

Once the cache reaches steady state, these zones remain hot and are preferentially retained under Zone LRU, while subsequently populated zones primarily contain colder data. Consequently, zone-level eviction is sufficient to preserve hot data, yielding hit ratios comparable to those achieved with Chunk LRU. We further examine hit ratio behavior under more realistic access patterns in §4.4.

In addition to eviction behavior, we observe secondary effects related to cache capacity and experimental configuration. We typically observe slightly higher hit ratios on ZNS devices due to the larger capacity. Smaller chunks exhibit a higher hit ratio under the Zipfian distribution simply due to an artifact of how we have executed the experiment. Because we have maintained a constant amount of I/O to the disk, this translates to more requests with smaller chunk sizes meaning the cache essentially has longer to improve on its ratio.

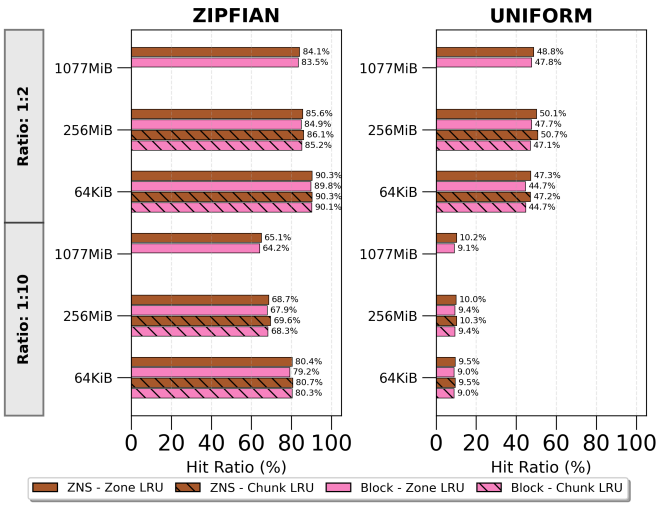


Figure 4.3: Hit ratios for the uniform and Zipfian distributions

Figs. 4.4 and 4.5 demonstrate the throughput results for uniform and Zipfian distributions respectively. Across all workloads and fill conditions, ZNS devices achieve their highest throughput under Zone LRU and either outperform or closely match block-interface SSDs. In contrast, ZNS performance under Chunk LRU degrades substantially due to host-side GC, which requires relocating valid data during cleaning. Zone LRU avoids this overhead by aligning eviction with zone reset boundaries, thereby eliminating write amplification.

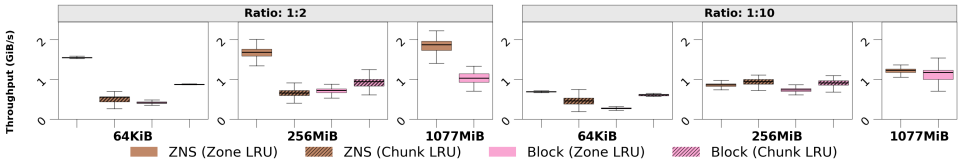


Figure 4.4: Cache throughput for the uniform distribution. The boxplots show the median (center line), interquartile range (box), and whiskers extending to 1.5x the IQR.

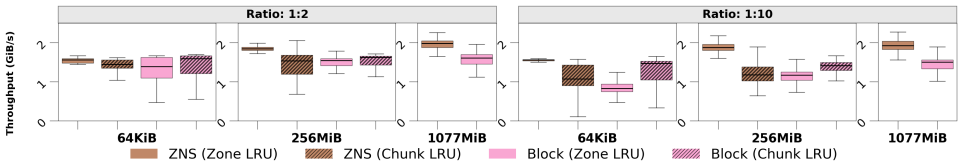


Figure 4.5: Cache throughput for the Zipfian distribution. The boxplots show the median (center line), interquartile range (box), and whiskers extending to 1.5x the IQR.

Block-interface SSDs exhibit the opposite behavior: their best performance is achieved under Chunk LRU. Under both eviction policies, block-interface SSDs incur device-side GC, whereas ZNS devices incur host-side GC overhead only when using Chunk LRU. Notably, simply switching the eviction policy from Zone LRU to Chunk LRU substantially reduces the impact of GC on block-interface SSDs. This behavior arises from how Chunk LRU separates hot and cold data under workloads with temporal locality. Frequently accessed (hot) chunks are promoted and retained, while colder chunks are repeatedly invalidated and rewritten in place as new data is admitted. As a

result, churn is concentrated within a relatively small logical address range corresponding to these cold chunks. On block-interface SSDs, this concentration causes a limited number of erase blocks to accumulate a high fraction of invalid pages, making them inexpensive to reclaim during GC. In contrast, Zone LRU operates at a coarser granularity and therefore disrupts overwrite locality. Because recency is tracked at the zone level, accesses to a small number of hot chunks repeatedly promote entire zones, even when most of their contents are cold. As a result, zones remain resident longer and evictions occur at the level of whole regions rather than individual chunks. New data is therefore written across many zones over time, distributing invalidations across a wide logical address range. On block-interface SSDs, this distributed overwrite pattern leaves many erase blocks partially live, increasing GC cost and amplifying performance variability.

Under Zipfian access, particularly at a 1:10 cache-to-workload ratio and with larger chunk sizes, Zone LRU on ZNS outperforms Chunk LRU on block-interface SSDs by a noticeable margin. Zipfian workloads retain a persistent hot set that continues to generate a substantial fraction of requests, requiring the cache to serve frequent reads while evicting colder data. On block-interface SSDs, this sustained eviction activity triggers device-side GC that competes with read servicing and degrades throughput. Zone LRU on ZNS avoids this interference by evicting data at zone granularity without device-side GC, allowing hot reads to be served more consistently. In contrast, the performance gap between ZNS and block-interface SSDs is smaller at a 1:2 cache-to-workload ratio on Zipfian because the workload becomes largely read-dominated once the cache reaches steady state. In contrast to uniform 1:2, a small hot set accounts for most requests, reducing cache admissions and eviction activity and thereby limiting GC on block-interface SSDs. In addition, the strong locality of Zipfian access causes evictions to occur primarily from colder regions of the cache, allowing block-interface SSDs to reclaim space at the device-level with relatively low overhead when GC does occur.

When examining the effect of cache-to-workload ratio under a uniform distribution, we observe a somewhat unintuitive trend, particularly with 256 MiB chunks. Intuitively, one might expect a lower ratio (1:10), which induces more cache misses and more frequent eviction, to amplify performance differences between eviction policies and device types, particularly favoring ZNS with Zone LRU. Instead, we observe that performance largely converges across eviction policies and devices at a 1:10 ratio.

This convergence arises from how the cache-to-workload ratio influences the accumulation of invalid data within zones. At a 1:10 ratio, the hit rate is low, and most chunks are written once and rarely re-accessed. As a result, chunks within a zone tend to age out together, producing zones that are already largely invalid by the time they are selected for eviction. Under these conditions, host-side cleaning on ZNS incurs relatively low overhead, reducing the performance gap between Zone LRU and Chunk LRU. Similarly, on block-interface SSDs, GC remains continuously active but does not disproportionately interfere with read performance due to the absence of a persistent hot set. Consequently, both devices operate in a steady, churn-dominated regime where performance differences between eviction policies and device types are largely masked.

In contrast, at a 1:2 cache-to-workload ratio, the higher hit rate causes chunks to be repeatedly promoted in the LRU across many zones, keeping data warm and more evenly distributed throughout the cache. This delays the formation of largely invalid zones and increases the amount of live data that must be migrated during eviction, amplifying the cost of host-side GC. As a result, performance differences between eviction policies and device types become more pronounced at higher cache-to-workload ratios, despite the lower overall eviction rate.

This additionally corroborates the observed performance of block-interface SSDs with uniform access and a 1:2 cache-to-workload ratio. ZNS devices show good performance with Zone LRU because they are unaffected by device-side GC. In contrast, block-interface SSDs exhibit performance similar to the uniform 1:10 case despite the greater read load, indicating that read performance remains strongly affected by write-driven churn. Additionally, uniform access spreads activity more evenly across the cache, sustaining fragmentation and exposing block-interface SSDs to higher GC costs even at the same cache-to-workload ratio, contributing to the larger performance gap relative to ZNS.

Beyond throughput, we examine latency variation to capture the impact of GC and eviction on individual requests. We focus on *end-to-end latency*, defined as the time from when a request is issued until the corresponding data is persisted to disk or retrieved from the cache. This metric reflects the full cost of request processing and exposes the effects of both host-side and device-side GC.

We intentionally do not report client-visible response latency, as response time may differ due to buffering and asynchronous completion and can

obscure the underlying behavior of the storage device and eviction algorithms. End-to-end latency therefore provides a more faithful representation of device-level performance under sustained eviction.

Figs. 4.6 and 4.7, show the end-to-end latency distributions for uniform and Zipfian workloads, respectively. These results demonstrate that under Zone LRU, ZNS devices achieve substantially lower tail latency, reflecting the absence of device-side GC and the benefits of zone-aligned eviction.

The empirical cumulative distribution functions (ECDFs), showing the cumulative fraction of operations below each latency threshold, exhibit a characteristic step pattern reflecting the bimodal latency distribution of cache hits and misses. The initial step corresponds to cache hits, with its horizontal position indicating hit latency and its vertical height indicating the fraction of requests served from the cache. On ZNS devices, this step is typically sharp, indicating low variance in hit latency. In contrast, block-interface SSDs do not always exhibit a similarly distinct step, with the distribution often appearing more gradual. We attribute this smoothing primarily to increased latency variance introduced by device-side GC, which can intermittently delay otherwise low-latency cache hits.

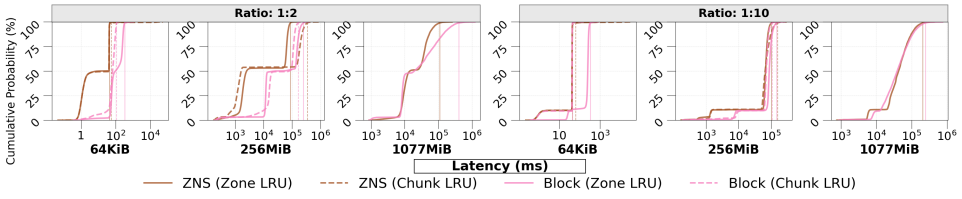


Figure 4.6: End to end latency for the Uniform distribution. Vertical lines represent P99 latency.

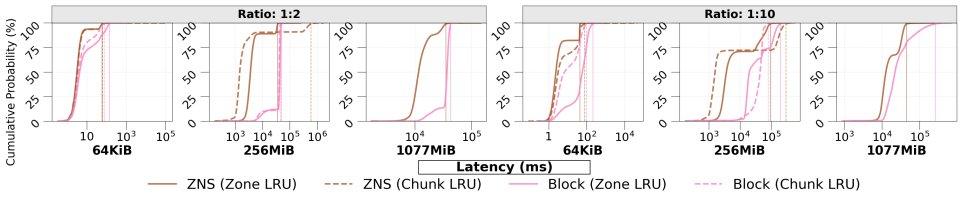


Figure 4.7: End to end latency for the Zipfian distribution. Vertical lines represent P99 latency.

Table 4.2 demonstrates the P99 tail latency comparison between the best throughput result on ZNS devices to the best result on block-interface devices. Overall, ZNS devices outperform or closely match block-interface SSDs across most configurations, with the largest advantages appearing when GC is active. When eviction pressure is low or churn causes regions to age out together, GC overhead becomes small and performance differences between devices narrow.

Table 4.2: Consolidated P99 latency comparison. Percentages show relative latency change of best throughput on ZNS vs best on block-interface device. (ZNS / Block, ms)

Chunk Size	ZIPFIAN		UNIFORM	
	R=2	R=10	R=2	R=10
64K	59.43 / 77.76 (+30.8%)	44.48 / 96.95 (+118.0%)	45.72 / 114.90 (+151.3%)	44.57 / 45.24 (+1.5%)
256M	36159.49 / 45553.03 (+26.0%)	96489.96 / 81467.13 (-15.6%)	88219.48 / 166099.73 (+88.3%)	102770.33 / 108685.94 (+5.8%)
1077M	35822.14 / 43191.80 (+20.6%)	45084.67 / 258944.97 (+474.4%)	110514.07 / 411037.01 (+271.9%)	206274.43 / 248436.99 (+20.4%)

Figs. 4.8 and 4.9 present the disk read latency at chunk granularity under uniform and Zipfian access patterns, respectively. Corresponding write latencies are shown in Figs. 4.10 and 4.11. Unlike the end-to-end latency results in Figs. 4.6 and 4.7, which measure the time from request arrival to completion, these figures isolate disk-level latency from various cache management overheads on the host. As a result, they reflect the latency of a single read or write operation in isolation. These isolated measurements demonstrate that ZNS devices exhibit lower intrinsic disk-level latency. The contrast between disk-level latency measurements and end-to-end latency results reveals the impact of host-side GC and cache management overheads. While disk-level latency ECDFs show a clear separation between ZNS and

block-interface devices, this separation is typically less pronounced in end-to-end latency measurements. This discrepancy indicates that the higher latencies observed in end-to-end measurements are primarily driven by host-side effects rather than intrinsic device latency, suggesting that improved host-side designs could better exploit the low-latency characteristics of ZNS devices.

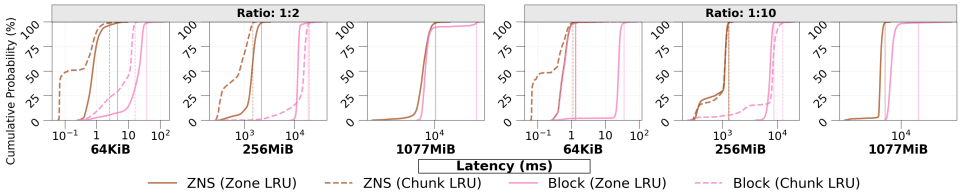


Figure 4.8: Read latency for the Uniform distribution at chunk granularity. Vertical lines represent P99 latency.

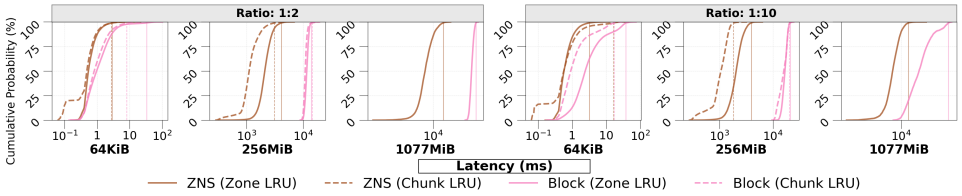


Figure 4.9: Read latency for the Zipfian distribution at chunk granularity. Vertical lines represent P99 latency.

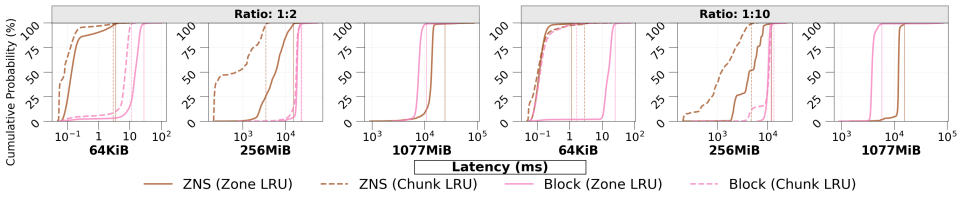


Figure 4.10: Write latency for the Uniform distribution at chunk granularity. Vertical lines represent P99 latency.

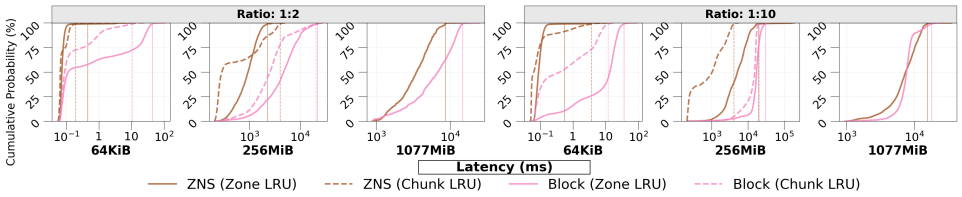


Figure 4.11: Write latency for the Zipfian distribution at chunk granularity. Vertical lines represent P99 latency.

Latency behavior is strongly influenced by the interaction between device interface semantics and eviction policy. ZNS devices achieve the lowest and most predictable latency under Zone LRU, where eviction aligns with zone reset boundaries and avoids both device-side and host-side GC. In contrast, applying fine-grained eviction on ZNS via Chunk LRU incurs substantial host-side GC overhead, resulting in the highest latency and variance observed across all configurations.

Block-interface SSDs exhibit the opposite trend: while they incur device-side GC under both eviction policies, Chunk LRU concentrates overwrites and mitigates GC-induced variance, often yielding lower latency than ZNS Chunk LRU. These results demonstrate that no single eviction policy minimizes latency across all device types; instead, optimal latency behavior requires eviction strategies that are tailored to the underlying device interface.

Across workloads, host-side GC on ZNS under Chunk LRU introduces both substantial throughput penalties and elevated tail latency relative to Zone LRU, indicating that fine-grained eviction without ZNS-specific optimizations can negate many of the performance benefits of ZNS devices. This behavior arises in part from our use of a chunk-level eviction policy designed to mirror traditional LRU semantics rather than being specialized for ZNS constraints. While more ZNS-aware fine-grained eviction strategies may mitigate these overheads, exploring such designs is outside the scope of this work.

4.4 Real World Case Study

This case study is not intended as a comprehensive evaluation of real-world cache workloads. Instead, it serves to illustrate how conclusions drawn from controlled synthetic experiments manifest under a realistic database-driven access pattern with strong spatial locality. We focus on a single representative workload to examine whether the qualitative behaviors observed earlier - particularly the interaction between eviction granularity (Zone LRU vs Chunk LRU) and GC - persist when access pattern is shaped by a real storage engine.

Artificial workloads based on predefined statistical distributions allow us to study how individual parameters influence performance; however, such traces are not representative of real-world workloads, significantly impacting hit ratio and number of bytes read. We complement synthetic workloads with traces derived from real-world experiments.

Our earlier evaluations used distributions such as Zipfian, applied across the cache key space. In practice, however, the access pattern seen by a cache backing a database differs substantially from the distribution used at the application level. For example, even if a Zipfian distribution drives queries in a MongoDB workload, the resulting I/O pattern on disk depends on the database's internal data structures (e.g., B+ trees), producing a very different spatial access pattern. When a cache operates beneath the database, it therefore observes this projected pattern rather than the synthetic one. In particular, a Zipfian distribution to keys will exhibit much stronger spatial locality at the level of B+-tree pages: nearby keys tend to be accessed together and reside on the same pages.

Furthermore, because databases cache data in memory at page granularity, disk accesses reflect page-level rather than key-level behavior. Multiple hot keys therefore map to the same cache object, increasing locality and reuse compared to a synthetic Zipfian model and reducing the number of distinct objects accessed.

Our real-world evaluation consists of a YCSB-C workload on WiredTiger, the MongoDB storage engine, as it is documented to be one of the industry standards for benchmarking MongoDB [17].

YCSB-C on WiredTiger: We run the YCSB-C workload on the WiredTiger storage engine (release `11.3.1`) using the MongoDB wtperf testing tool. We use a WiredTiger database consisting of 4,294,967,295 8 KiB values resulting

in a roughly 32 TiB database and a 1 GiB in-memory cache. Any requests that hit this in memory cache are not recorded since they would not consist of a read or write from the actual disk. This scenario stresses the on-disk cache and puts the spotlight on any differences between devices and eviction policies.

We record the file-level accesses made to the underlying database files and quantize each I/O request by mapping its file offset to one or more cache object identifiers, depending on whether the request spans multiple objects. From this, we generate a trace consisting of `(key, offset, size)` tuples, which we subsequently replay against our cache.

We evaluate with a 64 KiB chunk size due to it being relatively close to the most frequent request size (4 KiB). Using a much larger size such as the large chunks evaluated in previous experiments lead to a high degree of over-fetching, and therefore reduced performance.

Unlike the synthetic workloads, where all requests are issued at exactly the chunk granularity, the WiredTiger traces contain I/O requests at varying sizes, with many accesses occurring at granularities as small as 4 KiB. As a result, the observed throughput in this case study is lower than in the synthetic experiments. This difference arises from the smaller average request size rather than from any device-level limitation: smaller I/O operations achieve lower per-request bandwidth, reducing aggregate throughput.

Fig. 4.12 demonstrates throughput comparisons for ZNS and block-interface devices. The two different device types exhibit very similar maximum throughput when comparing ZNS (Zone LRU) and Block (Chunk LRU). With the block-interface device exhibiting a higher amount of variance.

All workloads show the comparable hit ratios of about 78%. For ZNS devices we observe 78.7% and 78.8%, and for block-interface devices we see 78.1% and 78.4% (Zone LRU, Chunk LRU). Zone LRU exhibits the same temporal clustering observed in the synthetic workloads, leading to the similar hit ratios.

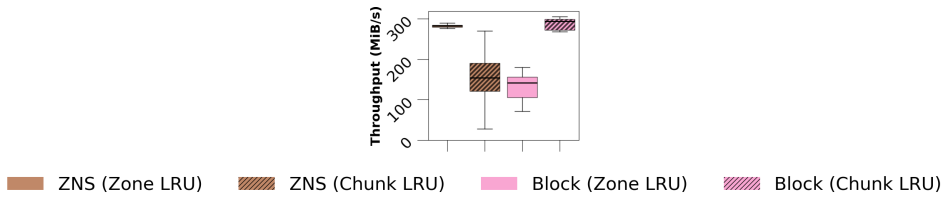


Figure 4.12: End-to-end throughput boxplots for WiredTiger workload. The boxplots show the median (center line), interquartile range (box), and whiskers extending to 1.5x the IQR.

We observe the throughput of the workloads in Figs. 4.13, 4.14 for Chunk LRU and Zone LRU respectively. Under Chunk LRU, the block-interface SSD reaches peak throughput and sustains this throughput throughout steady state. In contrast, under Zone LRU, the block-interface SSD exhibits heavy GC once eviction begins and throughput degrades substantially. Consistent with our observations in the synthetic workloads, we attribute the performance degradation under Zone LRU to its distributed overwrite pattern, in contrast to the concentrated overwrite behavior of Chunk LRU. This difference in write locality substantially increases GC cost on block-interface SSDs, leading to reduced throughput and higher variability. We observe the opposite results on ZNS SSDs. Under Zone LRU, consistently high performance with low variance is maintained throughout the workload. In contrast, under Chunk LRU performance degrades significantly once eviction begins, leading to high variance and degraded throughput.

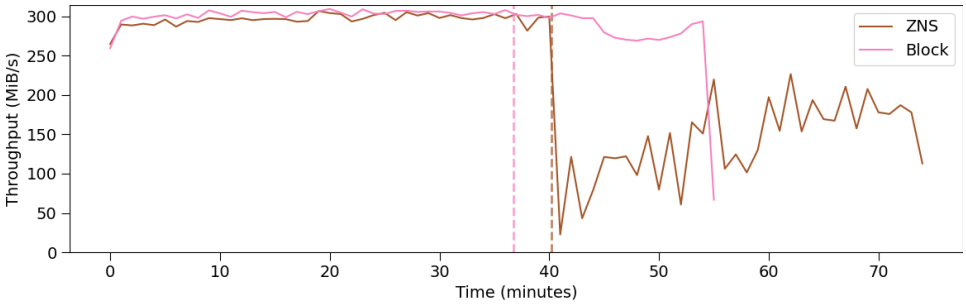


Figure 4.13: End-to-end throughput for complete WiredTiger workload (Chunk LRU eviction). Vertical lines show the first eviction.

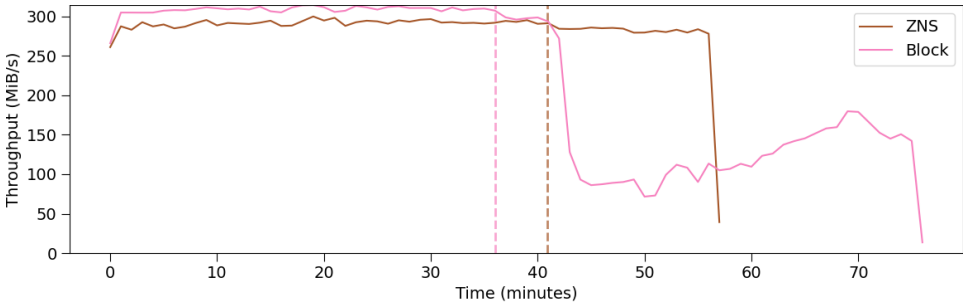


Figure 4.14: End-to-end throughput for complete WiredTiger workload (Zone LRU eviction). Vertical lines show the first eviction.

With respect to latency, however, we see clear advantages to using the ZNS SSD. End-to-end request latency is maintained at a much more consistent low value with a short tail on ZNS devices under Zone LRU. Chunk LRU on ZNS exhibits a high tail latency due to the write amplification effects of the eviction algorithm.

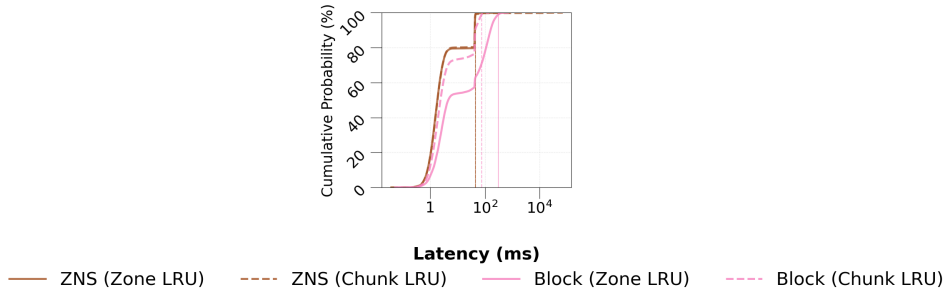


Figure 4.15: End-to-end request latency for complete WiredTiger workload. Vertical lines represent P99 latency.

Table 4.3 demonstrates the raw latency values. Comparing Zone LRU on the ZNS device against Chunk LRU on the block-interface SSD (the better result from each device type), we observe that the block-interface device exhibits higher median (+36.8%), mean (+37.5%), and P99 latency (+77.3%). We report maximum latency for completeness; and in this scenario the ZNS device exhibits higher latency, however, our analysis focuses on P99 as it better reflects typical behaviour, rather than outliers. What we observe here

is that ZNS retains the characteristics noticed in the synthetic workloads where overall tail (P99) latency is significantly lower than on the block-interface device.

Table 4.3: Latency statistics for Zone LRU and Chunk LRU eviction policies. Each entry reports ZNS / block-interface latency, with the percentage indicating the relative change when using the block-interface device. (ms)

Statistic	**Zone LRU**	**Chunk LRU**
P99	44.61 / 317.31 (+611.3%)	45.88 / 79.08 (+72.4%)
Median	1.82 / 4.27 (+134.1%)	1.85 / 2.49 (+35.0%)
Mean	10.89 / 53.10 (+387.7%)	53.54 / 14.97 (-72.0%)
Maximum	863.58 / 742.70 (-14.0%)	70913.68 / 216.63 (-99.7%)

These results address RQ4 by illustrating that several conclusions drawn from synthetic workloads also appear in a realistic database-driven workload with strong spatial locality. Block-interface SSDs can achieve performance comparable to ZNS devices, although ZNS continues to offer lower variance and more predictable latency.

4.5 Summary of Findings

If throughput were the only metric that mattered, block-interface SSDs would appear to perform just fine. Across many of our experiments, block-interface SSDs match or even slightly exceed the steady-state throughput of ZNS SSDs under certain eviction policies. However, as emphasized by Dean and Barroso [18], average performance is often a poor predictor of system behavior in large-scale services. Tail latency, not throughput, is frequently the dominant factor determining user experience and system reliability in cloud environments.

Viewed through this lens, our results indicate that ZNS SSDs can provide meaningful advantages in reducing latency variability and improving predictability, depending on workload characteristics and eviction strategy.

Across synthetic workloads, ZNS devices operating with zone-granular eviction (Zone LRU) exhibit more stable latency behavior under sustained eviction pressure, particularly when device-side GC would otherwise be active. In our database-driven case study, latency behavior across ZNS and block-interface SSDs is broadly comparable under high hit ratio conditions, with neither device demonstrating a consistent advantage across median, tail, or

variability metrics. These results indicate that when GC activity is largely suppressed, the choice of device interface has limited impact on latency behavior, and simpler block-interface SSDs can perform competitively.

The benefit is most pronounced under realistic workloads where cache access patterns consist of a mixture of hits and misses and exhibit skew toward frequently accessed regions, as observed under Zipfian distributions. When the cache is full and continuously admitting new data, block-interface SSDs must relocate live data internally, competing with foreground reads and writes. This competition manifests as elevated tail latency. ZNS SSDs avoid this behavior when eviction occurs at zone granularity, resulting in lower tail latency.

Importantly, these tail-latency improvements often persist even when throughput differences between devices are modest. In several workloads, block-interface SSDs achieve throughput comparable to ZNS devices while still exhibiting higher P99 latency. This highlights a critical point for system designers: throughput alone can mask the performance impact of GC. For latency-sensitive cloud services, tail behavior frequently dominates user-visible performance and operational planning.

Our results also clarify when ZNS does *not* provide an advantage. When fine-grained eviction is imposed on ZNS devices using eviction policies not specifically optimized for zone semantics (e.g., Chunk LRU), GC responsibility shifts to the host. In this configuration, ZNS devices lose their latency advantage, as host-side data relocation introduces both throughput degradation and elevated tail latency. This outcome underscores that ZNS benefits are not inherent to the hardware alone, but depend critically on eviction strategies that respect zone constraints.

Finally, our WiredTiger case study reinforces these observations under a realistic databasedriven workload. Even when block-interface SSDs achieve competitive throughput under optimized eviction, ZNS devices typically exhibit comparable or lower latency variance and more predictable tail behavior. For large-scale cache deployments where tail latency directly impacts service quality, this increased predictability represents an important factor in evaluating ZNS SSDs.

In summary, ZNS SSDs do not universally outperform block-interface SSDs in terms of average throughput or latency. Their primary advantage emerges under workloads that induce sustained eviction pressure and GC activity, where zone-granular eviction can eliminate device-side GC from the critical

path. In such settings, reductions in tail latency and performance variability can be as important as, or more important than, improvements in average throughput.

Taken together, these findings address our evaluation goals. We show that ZNS SSDs provide the greatest benefit under sustained eviction pressure when eviction operates at zone granularity, while block-interface SSDs can achieve comparable performance under eviction policies such as Chunk LRU that concentrate overwrites (RQ1). We demonstrate that shifting GC to the host under fine-grained eviction introduces significant overhead on ZNS devices, often negating their advantages (RQ2). We further identify access distribution and eviction pressure as key workload characteristics that amplify GC costs and tail latency (RQ3). Finally, our WiredTiger case study confirms that these behaviors persist under realistic database-driven workloads, particularly with respect to latency variability (RQ4).

Chapter 5

Conclusion

ZNS SSDs are not a universal replacement for block-interface SSDs, but they offer clear benefits in the right setting. When evaluated by average throughput alone, block-interface SSDs often appear competitive and, in some cases, superior. However, this perspective overlooks a key factor for large-scale services: tail latency.

Our results show that ZNS SSDs are most effective in cache workloads that operate under sustained eviction pressure and serve latency-sensitive requests. In these scenarios, ZNS devices provide more predictable performance and significantly reduce high-percentile latency, even when average throughput is similar to that of block-interface SSDs. This reduction in latency variability can be more important than marginal throughput gains for maintaining service-level objectives.

At the same time, ZNS SSDs do not universally outperform block-interface devices. When workloads are dominated by write churn or when eviction strategies are not aligned with ZNS constraints, the benefits of ZNS diminish and block-interface SSDs remain a strong and often simpler choice.

In short, ZNS SSDs matter most not because they maximize throughput, but because they improve predictability. For systems where tail latency defines user experience and reliability, this predictability can justify the additional complexity of adopting ZNS.

Bibliography

[1] Amazon Web Services. Amazon s3 pricing, 2025.

[2] 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 *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[3] WiredTiger Project. Chunk cache in wiredtiger, 2025. Retrieved: February 10, 2025.

[4] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage*, 2021.

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

[6] Western Digital. Zoned storage devices, 2025.

[7] CacheLib. Cachelib – pluggable caching engine to build and scale high performance cache services, 2025.

[8] Chongzhuo Yang, Zhang Cao, Chang Guo, Ming Zhao, and Zhichao Cao. Can zns ssds be better storage devices for persistent cache? In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '24, page 55–62, New York, NY, USA, 2024. Association for Computing Machinery.

[9] Yanqi Lv, Peiquan Jin, Xiaoliang Wang, Ruicheng Liu, Liming Fang, Yuanjin Lin, and Kuankuan Guo. Zonedstore: A concurrent zns-aware cache system for cloud data storage. In *2022 IEEE 42nd International*

Conference on Distributed Computing Systems (ICDCS), pages 1322–1325, 2022.

[10] Michael Allison, Arun George, Javier Gonzalez, Dan Helmick, Vikash Kumar, Roshan R. Nair, and Vivek Shah. Towards efficient flash caches with emerging nvme flexible data placement ssds. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, New York, NY, USA, 2025. Association for Computing Machinery.

[11] Inho Song, Shoaib Asif Qazi, Javier González, Matias Bjørling, Sam H Noh, and Huaicheng Li. Characterizing and emulating {FDP}{SSDs} with {WARP}. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*, 2026.

[12] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 1–15. ACM, 1991.

[13] 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*, 2010.

[14] Brian Cooper. Ycsb, 2019.

[15] Myoungsoo Jung and Mahmut Kandemir. Revisiting widely held ssd expectations and rethinking system-level implications. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):203–216, 2013.

[16] Narges Shahidi, Mahmut T Kandemir, Mohammad Arjomand, Chita R Das, Myoungsoo Jung, and Anand Sivasubramaniam. Exploring the potentials of parallel garbage collection in ssds for enterprise storage systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 561–572. IEEE, 2016.

[17] Mark Benvenuto. Mongodb 8.0: Improving performance, avoiding regressions, 2025.

[18] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

Appendix A

Object Storage Latency Evaluation

The following latency evaluation was completed on US West (Oregon) us-west-2, 11ms latency.

Table A.1: Latency evaluation for 64KiB Chunk size making requests to S3

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 A.2: Latency evaluation for 256MiB Chunk size Chunk size making requests to S3

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 A.3: Latency evaluation for 1077 MiB Chunk size Chunk size Chunk size making requests to S3

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

Appendix B

Parameter Evaluation Read and Write Latency

The following tables demonstrate the increase in latency for a cache request.

Table B.1: Total end-to-end latency request latency statistics for Chunk LRU eviction policy. Each entry reports ZNS / block-interface latency, with the percentage indicating the relative change when using the block-interface device. (ms)

Chunk / Stat	ZIPFIAN		UNIFORM	
	R=2	R=10	R=2	R=10
64K				
Max	121708.73 / 288.73 (-99.8%)	66294.19 / 226.22 (-99.7%)	57393.10 / 3305.20 (-94.2%)	40252.67 / 3470.36 (-91.4%)
Median	3.20 / 3.81 (+19.0%)	3.14 / 6.84 (+117.9%)	41.04 / 66.33 (+61.6%)	42.10 / 42.13 (+0.1%)
Mean	56.80 / 10.52 (-81.5%)	48.31 / 22.15 (-54.1%)	126.38 / 66.28 (-47.6%)	90.28 / 38.64 (-57.2%)
P99	63.48 / 77.76 (+22.5%)	78.61 / 96.95 (+23.3%)	54.18 / 114.90 (+112.1%)	64.25 / 45.24 (-29.6%)
256M				
Max	1737718.82 / 63423.00 (-96.4%)	574253.14 / 160239.03 (-72.1%)	886420.43 / 262827.12 (-70.3%)	300385.23 / 178931.01 (-40.4%)
Median	1481.55 / 40727.87 (+2649.0%)	1571.96 / 44448.42 (+2727.6%)	1622.77 / 78770.13 (+4754.0%)	65640.49 / 71256.94 (+8.6%)
Mean	42754.85 / 37407.92 (-12.5%)	53473.63 / 42192.78 (-21.1%)	95801.12 / 65388.89 (-31.7%)	67705.28 / 67810.60 (+0.2%)
P99	596121.27 / 45553.03 (-92.4%)	305907.05 / 81467.13 (-73.4%)	347987.56 / 166099.73 (-52.3%)	159067.01 / 108685.94 (-31.7%)

Table B.2: Total end-to-end latency request latency statistics for Zone LRU eviction policy. Each entry reports ZNS / block-interface latency, with the percentage indicating the relative change when using the block-interface device. (ms)

Chunk / Stat	ZIPFIAN		UNIFORM	
	R=2	R=10	R=2	R=10
64K				
Max	1069.72 / 860.35 (-19.6%)	957.24 / 902.49 (-5.7%)	867.74 / 1642.10 (+89.2%)	954.11 / 906.03 (-5.0%)
Median	3.11 / 4.26 (+36.8%)	2.39 / 59.08 (+2373.9%)	13.22 / 91.44 (+591.7%)	42.08 / 249.76 (+493.5%)
Mean	6.47 / 20.17 (+211.6%)	10.48 / 66.66 (+536.1%)	24.04 / 147.59 (+514.1%)	41.40 / 228.07 (+450.9%)
P99	59.43 / 138.65 (+133.3%)	44.48 / 216.43 (+386.5%)	45.72 / 343.59 (+651.5%)	44.57 / 337.04 (+656.3%)
256M				
Max	241839.24 / 57174.32 (-76.4%)	404565.85 / 524649.62 (+29.7%)	145372.23 / 406942.60 (+179.9%)	151732.94 / 261657.35 (+72.4%)
Median	3652.49 / 43392.55 (+1088.0%)	3648.14 / 20003.39 (+448.3%)	2534.99 / 25552.74 (+908.0%)	67655.94 / 87783.75 (+29.8%)
Mean	6987.54 / 39670.58 (+467.7%)	18887.04 / 52185.35 (+176.3%)	32256.56 / 85995.60 (+166.6%)	63276.17 / 85016.93 (+34.4%)
P99	36159.49 / 50073.77 (+38.5%)	96489.96 / 200377.54 (+107.7%)	88219.48 / 253739.85 (+187.6%)	102770.33 / 150135.23 (+46.1%)
1077M				
Max	146192.43 / 56977.91 (-61.0%)	245519.98 / 691695.03 (+181.7%)	255291.33 / 1222797.62 (+379.0%)	430138.35 / 752815.15 (+75.0%)
Median	11735.35 / 37137.73 (+216.5%)	12745.35 / 22169.57 (+73.9%)	14222.36 / 19269.99 (+35.5%)	42599.63 / 39033.75 (-8.4%)
Mean	13996.84 / 35112.41 (+150.9%)	18968.00 / 39253.50 (+106.9%)	28279.94 / 58442.06 (+106.7%)	53850.41 / 55582.21 (+3.2%)
P99	35822.14 / 43191.80 (+20.6%)	45084.67 / 258944.97 (+474.4%)	110514.07 / 411037.01 (+271.9%)	206274.43 / 248436.99 (+20.4%)