Building and evaluating disk cache, a comparison of Rust and C

John Ramsden
University of British Columbia
Vancouver, Canada

Sam Cheng
University of British Columbia
Vancouver, Canada

ABSTRACT

The C programming language has long been the dominant tool for system programming, prized for its low overhead and fine-grained control. However, C also demands that programmers manage memory and concurrency manually - tasks that are notoriously error-prone and difficult to scale safely. Even experienced developers often fall victim to subtle bugs that compromise correctness, stability, and maintainability.

To address these challenges, the Rust programming language introduces compile-time memory safety and ownership semantics, offering a modern alternative for building reliable systems software. In this project, we revisit a previously implemented concurrent cache written in C and redesign it in Rust - not as a direct port, but as an exploration of how Rust's guarantees enable cleaner abstractions and safer concurrency.

This case study highlights the practical trade-offs of adopting a safety-oriented language, including changes in design structure, development effort, and runtime behavior. This work presents a concurrent, disk-based cache as a concrete example of how modern language features can support the development of more robust systems software, suggesting that rethinking core components with safety in mind may offer lasting benefits.

1 INTRODUCTION

The C programming language provides a low-level interface to hardware and has served the programming community well for decades. C provides a tremendous amount of power to software designers, allowing developers complete control over the underlying hardware. Unfortunately, programmers are human and make mistakes. As a result, memory safety bugs are still considered one of the top three software errors [20, 24].

The Rust programming language provides strong guarantees of memory and concurrency safety at compile time by introducing a richer type system and the concept of explicit lifetimes and ownership (Listing. 1). These guarantees eliminate many common classes of bugs found in systems code, including data races and use-after-free errors. Compile-time enforced ownership is particularly useful in the context of concurrent programming, where explicit control over data access across multiple threads can prevent race conditions caused by concurrent modification. While the benefits that Rust brings come with an overhead (1.77x performance overhead on average vs C [35]) in the form of runtime checks, given the powerful safety guarantees, we argue this overhead is justifiable in many situations.

```rust
// Types that do not implement the 'Copy' trait
// cannot be used after move
let a = String::from("Hello world");
let b = a; // Move a to b

// The following fails to compile!!
println!("value of a: {}", a);
// The value of a was moved to b so it can
// no longer be used.
```

Listing 1: An example of Rust's ownership model. Since values can only associated be with one owner in Rust, the destructor for this code will only run once, preventing double-free errors.

We revisit our earlier implementation [23] of a disk-based caching system designed specifically for both Zoned Namespaces (ZNS) SSDs [7] as well as conventional SSDs. We re-implement it from the ground up in Rust. By leveraging Rust's language features and stricter compile-time guarantees, we aim to improve system reliability, simplify concurrency management, and reduce the likelihood of memory-related bugs.

In our previous implementation, we encountered several challenges inherent to the C programming language. C lacks type safety, especially in terms of generic functions, which lead to many subtle bugs at runtime. Rust offers modern tools for safely composing modular systems, and we aim to leverage these features to build a more reliable and maintainable implementation that is easier to debug.

In this project, we build a fully functional disk-based concurrent cache, leveraging Rust's features to improve modularity, correctness, and developer productivity. We design a modular system that allows us to swap out components safely, such as the block-device type.

Ultimately, we explore the benefits that Rust offers, as well as any potential drawbacks compared to the C programming language, which remains the de facto standard in systems

software development. In addition to differences between Rust and C, we make new findings related to ZNS and block-interface devices that build upon those that we made in our previous implementation.

This paper makes four contributions. (1) We present *OxCache*, a Rust re-implementation of our disk-backed, concurrent key–value cache, with a modular architecture that cleanly separates server, device, eviction, and I/O pools (§3). (2) We empirically compare *OxCache* to our prior C-based *ZNCache* under identical workloads across ZNS and block-interface SSDs, reporting throughput, latency, CPU, and memory (§5). (3) We analyze ZNS-specific mechanisms - most notably Zone Append - and their interaction with device-side garbage collection, showing how offloading write-pointer management removes per-zone lock contention. (4) We find that *OxCache* markedly improves small-chunk performance and CPU efficiency, that ZNS sustains higher throughput when GC is active while offering little advantage in low-throughput workloads, and that architectural choices - rather than language alone - explain most performance deltas.

2 BACKGROUND AND RELATED WORK

Rust is a modern systems programming language that emphasizes memory and type safety. Rust's borrow checker, a compile-time validation mechanism, prevents the occurrence of use-after-free, double-free errors, as well as data race bugs caused by simultaneous reads. As a result of Rust's numerous positive aspects, many new and existing software projects are adopting it as the primary language of choice [1, 2]. Large systems such as the Linux kernel [3] or Firefox [8] have also begun incrementally adding Rust support due to the strong guarantees the language gives. Rust's growing adoption in high-performance infrastructure projects makes it a natural candidate for ZNS-aware cache designs.

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 [32], highlight the importance of reducing access costs in production environments [34]. While slower than DRAM, flash provides lower cost per byte and is non-volatile [25].

Zoned Namespace (ZNS) SSDs improve efficiency by reducing the overhead of the traditional *block-interface* [7]. 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 garbage collection (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.

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. Existing cache systems such as CacheLib [9] often adopt a region-based design in which data is first written to memory regions and later flushed to disk in bulk [33]. This approach reduces garbage collection (GC) overhead on the underlying flash devices and aligns naturally with the characteristics of ZNS SSDs. Yang et al. [33] explored this design by extending CacheLib with a ZNS-based backend. Their work evaluated multiple eviction policies, including ones that eliminate the need for software-managed GC by mapping regions directly to zones in a one-to-one manner, as well as policies that map multiple regions to a single zone. The latter requires software-level GC but can offer lower latency. Similarly, Lv et al.[18] proposed a ZNS-based caching design in their system, "ZonedStore". Other caching systems do exist in the Rust ecosystem with similarities to CacheLib. Foyer [10] is a hybrid cache library explicitly inspired by CacheLib. While it provides support for block-interface backends, it does not currently support ZNS. Our work complements such efforts by exploring ZNS-aware caching in Rust, bridging the gap between existing designs and the needs of zoned storage. Beyond Foyer, in-memory Rust caches such as Moka [2] demonstrate the language's growing adoption for high-performance caching, though they do not target hybrid or ZNS-aware designs.

In our previous work, we developed a C-based concurrent cache - *ZNCache* [23] - to evaluate ZNS SSDs and compare them to block-interface (conventional) SSDs. While prior caching systems explored ZNS-based caches, we chose to build a from-scratch system so we could examine additional eviction and zone management policies. ZNCache is a multi-threaded caching system that supports two types of SSD backends: zoned and block-interface. It employs an abstraction layer that simulates zones on block-interface SSDs, enabling a unified logic path across both backends. We handle eviction using a single policy, *promotional eviction*, which applies an LRU strategy at the zone level. When a write occurs in a zone - which may contain multiple cacheable objects (*chunks*) - the entire zone moves up within the LRU structure. This simple yet effective eviction mechanism aligns with the constraints of zoned devices, where erasures occur only at zone granularity. Our redesign in Rust (described in §3) retains some of the design decisions from ZNCache - particularly eviction policies and the abstraction layer over

zones - but ultimately diverges significantly in its overall architecture.

3 METHODOLOGY

We designed and implemented *OxCache*, a concurrent key–value cache that leverages a remote store. Similar to ZNCache, we introduced a simplification to reduce system complexity: although the cache operates on top of a non-volatile layer, it does not support persistence across server restarts.

Following the design of *ZNCache*, we define the unit of caching as a *"chunk"*. Larger chunks typically improve overall throughput and reduce the number of requests issued to the remote store. However, they may lead to performance degradation when eviction rates are high. To study this trade-off, *OxCache* allows configurable chunk sizes at initialization, enabling evaluation of how varying chunk sizes affect performance.

Some of the problems we faced in *ZNCache* involved a lack of modularity in some of our subsystems. We struggled with developing truly generic systems in C, leading to difficulties when we expanded the code base and swapped out components such as eviction policies. While we did achieve a degree of modularity in the C system, much of it was achieved with unsafe mechanisms or function pointer manipulation. With *OxCache* we have developed a more modular system with truly swappable components, using a more modern and safe design taking advantage of Rust features.

Like *ZNCache*, *OxCache* supports two backends which we evaluate: a block-interface backend, and a ZNS backend. Unlike our previous implementation which used libc write and read calls, we interface directly with the nvme devices using libnvme [17]. This lower level library allows us to take advantage of more advanced ZNS features, and we discuss it further in §3.0.4. Our two backends operate on zones. While block-interface SSDs do not have a concept of zones, we use this abstraction to allow for shared logic between the two backends. On block-interface SSDs we logically break up data into regions, resulting in an interface resembling a ZNS device. This is the same method we used in *ZNCache*, and this approach has also been used in prior caching systems [9].

The high-level design of our cache is depicted in Fig. 1. 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 the 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. 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. The detailed design and operation of each component will be discussed in the following sections.

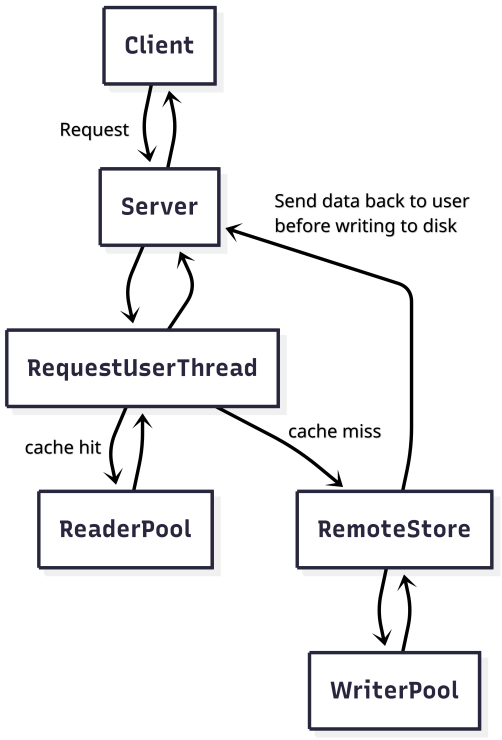

Figure 1: High level OxCache design.

We had two choices for implementing concurrency in Rust. One option was to spawn a kernel thread per connection, which was used in our previous implementation. Kernel threads are managed by the operating system, which is aware of I/O events and can preempt threads when they are blocked by I/O. They are also well supported in C and Rust. However, it is expensive to spawn one thread to service a connection, and context switches between different threads are expensive to perform. In *ZNCache*, we artificially limited the number of kernel threads in order to avoid excessive resource overhead.

The second option is to utilize Rust's stackless coroutines (also known as user threads). These require the programmer to annotate functions with `async`. This transforms the function to return a `Future` object which encodes and stores the *state* of the computation within the function, allowing the function to be paused and resumed. Distinct states within the `Future` represent different points of computation, and state transitions occur at user-defined *yield points*, where computation can be paused and resumed. These yield points

are annotated with `await`. An executor (like a thread pool) drives these user threads to completion, pausing at yield points and resuming other paused user threads.

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. The downsides of user threads are the additional complexity. User threads cannot be used by regular functions as they must be spawned and run by the executor. This results in "*async contagion*", where functions calling `async` functions must also be annotated with `async`. User threads in Rust are also *cooperatively scheduled*. Threads only yield at user-defined yield points, so computations should be short to avoid starving other threads. Additionally, there are restrictions for async functions appearing in traits.

Throughout our design we made heavy use user threads. We used the popular Rust library Tokio [6] to facilitate this. Tokio provides an easy-to-use runtime that is the de facto standard in Rust for developing async and user thread-based code.

OxCache is composed of several key modules (Fig. 2), whose interfaces are defined by traits. This allows easily swapping the internal implementation at compile time (with generics) or runtime (with trait objects). The *server* module (§3.0.1) acts as the entry point to the cache and spawns user threads to service client requests. The *RemoteStore* module (§3.0.2) provides a interface for interacting with remote data stores. The *device* module (§3.0.4) abstracts underlying storage devices, exposing a unified interface for both Zoned and block-interface devices. The *eviction* module (§3.0.5) manages cache eviction and invalidation through both periodic and foreground eviction. The *readerpool* (§3.0.6) and *writerpool* (§3.0.6) modules handle read and write requests, respectively. The *ZoneList* module (§3.0.3) manages zone state. Finally, the *metrics* module (§3.0.7) provides system observability through logging and a Prometheus [5] exporter.

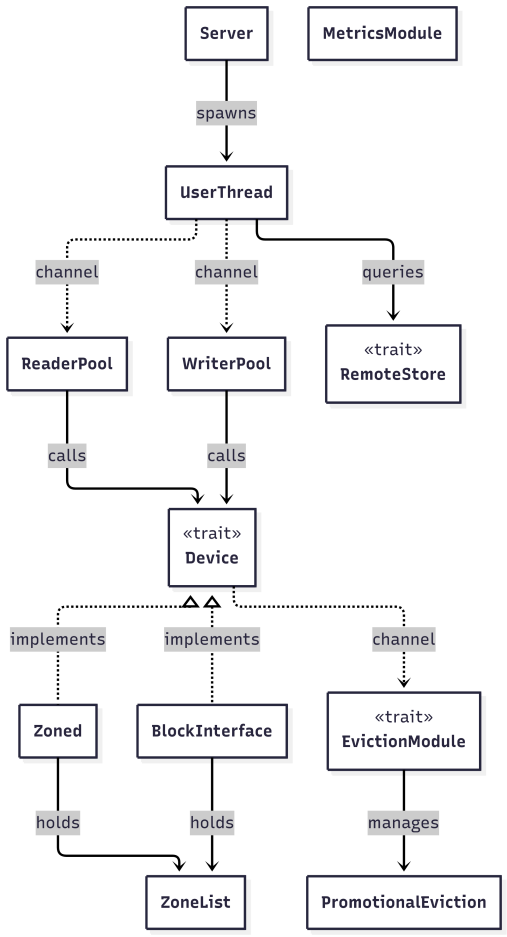

Figure 2: OxCache module interactions.

3.0.1 Server Module. The server module functions as the core of the cache. It listens on a Unix socket for client-issued requests, which are sent as serialized Rust structures. Rust made this straightforward through the use of Serde [30] for serialization, combined with built-in networking support that provides convenient access to Unix sockets. By contrast, our *ZNCache* implementation executed queries through direct function calls and did not expose an external client. The ease with which we were able to establish this robust functionality stood out to us. In comparison, implementing the same functionality in C, even with libraries is a non-trivial effort. This isn't to say it can't be done in C - memcached's [14] server code parses and frames incoming commands with a state machine and buffer management. It builds outgoing replies by hand using iovec arrays. These routines span thousands of lines of C, with careful error handling and memory management, underscoring that even with good libraries the effort to match Serde and Rust's ergonomic network stack is far from trivial.

Once a request is received by the server, a corresponding user thread is spawned to service the request. Within this thread we communicate with the various components using Rust channels [27] for message passing. This also serves as a form of synchronization between threads.

The flow of a service request begins with deserializing and validating the message from the client. If the requested data is not present in our "state map" (a mapping from keys to on-disk locations), a request is issued to the remote store, and the corresponding data is immediately returned to the client through the initial Unix socket. Since the data has not yet been persisted, the state is marked as "in progress". In this way, if a subsequent request is made for the same data,

it waits for a notification that the write has completed before proceeding to read[1]. Data is then sent to the writerpool through a dedicated channel. While the write is proceeding the user thread awaits a response from the writerpool, allowing other user threads to service requests. Once the data is persisted to disk, the entry is updated in the map as complete, and all waiters are notified. Finally, the user-level thread is terminated.

3.0.2 RemoteStore Module. The `RemoteStore` module defines a trait 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 to mimic response times from an object store. It generates data 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.

3.0.3 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 zones and provides an interface for managing their availability. The module allows a zone to be allocated if one is available, or notifies the caller when the device is full and eviction is required. It also supports returning zones after use and updating the zone state appropriately when eviction occurs.

3.0.4 Device Module. The device module provides an abstraction layer over both ZNS and block-interface devices. Rust trait objects allowed us to define a common interface and access the relevant device implementations at runtime through dynamic dispatch [15]. This enables modularity in a safe manner, avoiding the need for unsafe constructs that are often required in C [21], where function pointers and manual vtable management are commonly used. As a result, implementing generic device structures became significantly easier.

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 access to NVMe devices (both ZNS and block-interface) via the kernel's NVMe stack. Since all of our evaluation platforms (§5.1) are NVMe-based, this choice was natural. To integrate the library, we developed Rust bindings to the underlying C API and implemented a Rust wrapper that exposes a high-level, safe interface.

`libnvme` also exposes support for Zone Append [37], which we did not exploit in *ZNCache*. Ordinarily, when using a ZNS device, writes within a zone are constrained by the 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. Instead of specifying the write pointer explicitly, the host issues a Zone Append with the data, and upon completion, the device returns the physical address where the data was placed (i.e., the prior write pointer location). As illustrated in Fig. 3, 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.

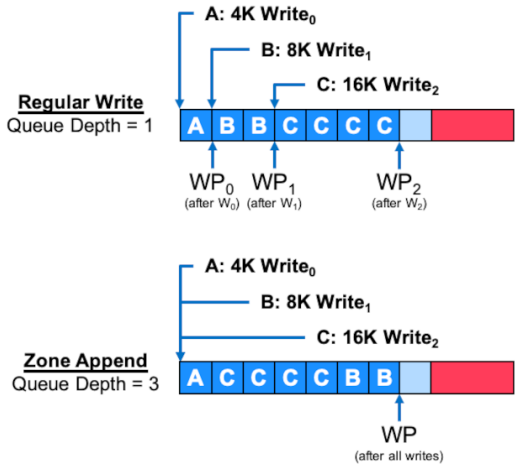

Figure 3: Regular Writes and Zone Append Writes [37].

Both device types implement the common Device trait, providing functions to append, read, and evict - key functionality that must interface with the disk directly.

The `Zoned` device implementation additionally 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 provides an abstraction over a traditional block device, making it appear similar to a ZNS SSD. We achieve this by maintaining an internal

[1]An implication of immediately returning data to the client while keeping the user-level thread open is the potential accumulation of a large number of active threads. This behavior may lead to increased resource usage if many threads remain in progress concurrently. In our experiments, we did not observe resource consumption reaching problematic levels. Nevertheless, a possible mitigation strategy is to introduce a user-thread limit determined by available system resources. If this limit is reached, new requests would be deferred until existing threads complete, thereby bounding resource utilization.

`ZoneList` that simulates zones through logical segmentation of the block device.

3.0.5 Eviction Module. The *eviction module* hosts an OS thread which periodically (based on a predefined interval) wakes up and performs eviction if needed. The module also listens on a channel for foreground eviction events in case the system completely runs out of capacity before the periodic eviction can sufficiently clean. Like the device module, the eviction module is also defined to abstract over multiple eviction policies.

For our eviction policy we used same policy we used in *ZNCache* - a simple "promotional eviction" policy. This policy was also managed in the eviction module. Promotional eviction involves updating a zone's position in an LRU whenever any chunk within it is read from or written to. This causes the entire Zone to be "promoted". When eviction is required, the least recently used zone is popped off the LRU, and all of the data within it is evicted.

3.0.6 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 waiting on the channel. This design simplifies device access by abstracting away concurrency management and delegating parallelism to the pools.

3.0.7 Metrics Module. The metrics module provides mechanisms for exporting and recording key cache statistics such as latency, throughput, hit ratio, and fill percentage. Using the metrics[16] library in combination with `metrics exporter prometheus` [19], we can export metrics in real time to Prometheus. This integration greatly improved our development workflow by allowing us to visualize results immediately and receive rapid feedback. In addition, we employed the tracing [22] library to log fine-grained metrics to JSON files, enabling detailed post-run analysis.

4 LANGUAGE EVALUATION

Rust, being a modern systems language, offered a number of conveniences compared to C. When implementing *OxCache*, we found Rust's compile-time guarantees, extensive ecosystem, and excellent tooling helpful to the development process. At the same time, the additional complexity from the borrow checker and type system caused some headaches.

We compared the development experience of *ZNCache* and *OxCache* to contrast Rust and C in the context of cache development. Overall, the experience of writing *OxCache* in Rust was positive, and the negatives mostly stemmed from the same strict guarantees that make Rust advantageous.

4.1 Advantages

4.1.1 Compile-time and run-time safety. Rust provides memory safety against use-after-frees and double-frees with the borrow checker that runs at compile time. Compared to *ZNCache*, we did not experience any segmentation faults or memory-related bugs when running *OxCache*. For example, during development, the ownership model prevented access-related bugs, such as modifying a collection in an iterator for-loop 2. This is impossible in Rust as values cannot be mutated if there are immutable references to the container.

```rust
fn main() {
    let mut v = vec![1, 2, 3];
    for x in &v {
        v.push(*x); // error: cannot borrow `v` as
        ↪  mutable because it is also borrowed as
        ↪  immutable by the iterator
    }
}
```

Listing 2: Naive attempt at modifying the container while also holding a reference to the container.

Subtle concurrency bugs were also mitigated by Rust's trait system. Through the Send and Sync trait [28], we were able to ensure data races between threads did not occur, ruling out a class of errors we previously had to consider in *ZNCache*.

Rust also provides runtime safety that prevents out-of-bounds access. If the program accesses memory out of bounds, it panics and prints a stack trace instead of invoking undefined behaviour, which is useful for debugging.

4.1.2 Type System. Rust's type system prevents errors, supports generic parameters, and enables OOP-style programming. By being strongly typed with explicit conversions, we were able to avoid errors caused by casting to wrong types. As part of an effort to introduce uniform interfaces between block-interface and zoned namespace devices in *ZNCache*, we used void pointers to implement type-erased generic data structures and algorithms. This sacrificed type safety and led to subtle bugs at runtime. Additionally, void pointers require a dereference to access the data, even if the generic property was desired at compile time and the type is known.

The generic system in Rust allowed us to create type-safe generic interfaces that were also efficient through monomorphization, which instantiates a unique function or struct for each generic.

The trait system additionally supported the style of object-oriented programming, which was a natural way to organize structures in our codebase.

4.1.3 Libraries and Tooling. Cargo is the main tool for managing Rust projects. Its ease of use and reasonable defaults allowed us to get started quickly with *OxCache*, even despite the relative lack of experience in Rust. Adding libraries (known as crates) to the project was simple as well. We did not have an equivalent when building *ZNCache*. There is no official repository for packages, so dependencies were sourced, downloaded, and built manually. This introduced some complexity.

Although Rust is a relatively new language compared to C, its ecosystem contains a large number of high-quality and mature libraries. Notably, we utilized Tokio [6] for the asynchronous runtime, and Bindgen [4] for interfacing with the libnvme library. While these libraries had some amount of learning curve, they significantly reduced the amount of work required to develop *OxCache*.

4.2 Disadvantages

Although Rust does help eliminate certain classes of bugs, reducing debugging time, it does not mean code "just works". On many occasions, we spent hours or even days debugging race conditions and unexpected behavior.

4.2.1 Ergonomics. In certain cases, what may have been intuitive and simple to write in C requires significant thought, and potentially unsafe code, in Rust. For example, writing a linked list in Rust is a non-trivial endeavor due to ownership and borrowing rules that forbid multiple mutable references to the same node. A naive implementation that would be straightforward in C will fail to compile in Rust (Listing 3).

This behavior reflects Rust's philosophy of protecting the programmer from common classes of errors. However, it underscores that certain tasks that are straightforward in C can become significantly more complex in Rust. Solutions exist, but they often require more sophisticated abstractions or patterns.

At compile time, Rust must determine the lifetime of each variable. This requirement can complicate scenarios where data is shared across multiple threads, since each thread needs to know when it is safe to drop the shared value once no references remain. The standard solution in Rust is to use an atomic reference counter (`Arc<T>`). Wrapping a value in an Arc allows it to be cloned, incrementing the reference count each time, and then moved into additional threads. Once the final reference is dropped, the underlying value is deallocated. This approach is explicit and robust, but it often results in many values being wrapped in `Arc`, introducing repetitive boilerplate.

Often, our implementation needed to move a value (e.g. within an async move block or closure) within a loop. The natural approach is to clone the value at the point where the value is used. However, we could not simply clone inside

```rust
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}

impl Node {
    fn push_front(self, value: i32) -> Node {
        Node {
            value,
            next: Some(Box::new(self)), // ERROR:
            ↪  ownership move issues
        }
    }
}

fn main() {
    let list = Node { value: 1, next: None };
    let _list2 = list.push_front(2);
    // This fails because `list` is moved and cannot
    ↪  be reused.
}
```

Listing 3: Naive attempt at implementing a linked list in Rust, which which does not compile because of ownership and borrowing restrictions.

of the closure, as doing so referenced the original variable. Instead, we needed to create a clone outside of the closure, clone the value into the scope, and then move the cloned value inside. This often reduced the readability of the code.

```rust
for i in 0..10  t {
    let cloned_arc = arc_object.clone();
    some_function(
        async move {
            // Compiles
            cloned_arc.use();
            // Does not compile
            arc_object.use();
        }
    )
}
```

Listing 4: Moving a value without cloning it explicitly fails to compile.

Rust does not support async trait objects [29]. This ergonomically feels awkward, since something that is a typical behavior is unsupported. To get around this limitation we took advantage of *async-trait* [13], a third-party library that allows you to annotate a trait, enabling it to be used as an async trait object.

4.2.2 Lack of multiple dispatch. One source of frustration that we encountered was the lack of multiple dispatch, or dispatch based on the dynamic type of multiple objects. Specifically, our original interface of the device module exposed a `evict_chunk` and `evict_zone` method, and we intended the eviction policy to pass a zone-or-chunk value to the device, which would have been a dynamic type. However, we could not do this because Rust does not support pattern matching on the dynamic types of both the evict target and device. As a workaround, we wrapped the eviction policy and its output as typed unions, and pattern matched inside the device function. While this method worked, it required some boilerplate.

```
// Intended behaviour which fails to compile:
trait Zoned {
    fn evict(&self, evict_target: Chunk, ...);
    fn evict(&self, evict_target: Zone, ...);
}

trait BlockInterface {
    fn evict(&self, evict_target: Chunk, ...);
    fn evict(&self, evict_target: Zone, ...);
}

// Callsite
let evict_target: dyn EvictTarget =
↪  evict_policy.get_evict_target();
device.evict(evict_target);
```

Listing 5: The intended code that we wanted to write if Rust had multiple dispatch.

5 PERFORMANCE EVALUATION

The goal of our performance evaluation is to compare results with those from *ZNCache*. We repeat the same experiments and examine the differences, highlighting any new findings. Unlike our first paper, we also evaluate how our design choices, re-architected implementation, and the language switch influence performance. As with *ZNCache*, we run the experiments:

(1) **Parameter Eval (§5.2.1)**: Evaluate how the different SSDs compare under workloads with varying parameters, such as data distribution and chunk size.
(2) **GC Eval (§5.2.2)**: Determine the effects of device-side GC on block-interface SSDs for cache workloads, and how device-side GC impacts throughput and tail-latency.

Alongside these experiments, we also compare CPU and RAM overheads between the two implementations.

Our goal is not to claim that changing languages directly improves or degrades performance. The two implementations are not directly comparable, as they differ significantly in design. In *OxCache*, we applied lessons learned from *ZNCache* and introduced several improvements. Thus, our evaluation is intended to show how the new implementation performs relative to the earlier *ZNCache* design. In addition, it provides an opportunity to uncover new insights into the behavior of ZNS and block-interface SSDs.

5.1 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 24.04 with Linux 6.14.0. Importantly this differs from our previous experiments in that we transitioned from Ubuntu 22.04 with Linux 6.8.0. While the minor kernel change is not optimal, this is the only platform difference, and we expect the variance to be minimal.

We evaluate a ZNS SSD and a block-interface SSD. The underlying hardware of the two devices is identical, with the distinction lying solely in the firmware, as confirmed by the vendor.

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

Their capacities differ because the block-interface SSD must reserve space for device-side garbage collection, a requirement not present on ZNS SSDs. As a result, the ZNS configuration exposes more usable capacity to the system.

We conducted all experiments using the *mq-deadline* scheduler on the ZNS SSD and the *none* scheduler on the block-interface SSD.[2]. This configuration mirrors our prior experiments: *mq-deadline* was required because it enforces strict I/O ordering [26], which is necessary to maintain write pointer consistency on ZNS devices under Linux 6.8.0. Conversely, the *none* scheduler is preferred for block-interface SSDs, and is the default in Ubuntu 22.04 and 24.04, due to its lower latency and higher IOPS under a simple FIFO model where strict ordering is unnecessary [31]. As of Linux 6.10.0, however, the introduction of Zone Write Plugging [36] has eliminated the need for a scheduler to enforce write ordering on ZNS devices.

5.2 Benchmarks

We use the same benchmarks from *ZNCache*, executing workloads with Zipfian and uniform random distributions for cache access patterns through a YCSB-based generator [11].

[2]While we also tested the *none* scheduler on ZNS devices, performance differences were negligible

We use the same Zipfian parameter of 0.99, which is considered reasonable for database and cache workloads [12].

On both caches, workloads consist of executing a sequence of cache calls defined by experimental parameters. We pregenerate workloads representing chunk accesses and execute them from start to finish while recording relevant metrics. We bin results over 60-second intervals to improve readability in graphs. For raw calculations (Appendix B) no binning is applied. We vary three parameters: *chunk size*, *distribution*, and *ratio*, where ratio denotes the cache-to-workload size ratio. All benchmarks are executed with 64 threads (matching the number of available hyperthreads) to maximize achievable bandwidth.

For all experiments, we use our emulated backend (§3.0.2) as the remote store to eliminate variability in results, following the same method as in *ZNCache*. We derive artificial latencies from real-world measurements (Appendix A), which provide expected average request times based on chunk size.

The method of executing queries differs substantially between the two systems. In *ZNCache*, there is no dedicated client; queries are issued directly through function calls within the main server. By contrast, *OxCache* has a client–server architecture, with requests sent over a Unix socket and responses returned through the same channel. To approximate *ZNCache*'s concurrency, we use a multi-threaded client that opens 64 simultaneous connections, analogous to the 64 threads issuing function calls in *ZNCache*. Given the architectural differences, the two systems are not directly comparable. Nevertheless, we minimize effects of any differences wherever possible.

5.2.1 Parameter Evaluation. Our parameter evaluation executes workloads on the cache while varying a broad set of parameters. To enable direct comparison with *ZNCache*, we use the same parameter settings, evaluate the resulting differences, and highlight new findings. We test two extremes of chunk size - 64KiB and 512MiB. As in the original *ZNCache* evaluation, we artificially restrict cache size and total data transfer to ensure one-to-one comparability. For 512MiB chunks, we use 200 zones; for 64KiB workloads, where throughput is lower and experiments run significantly longer, we reduce the cache size to 40 zones. Each workload executes a fixed amount of I/O corresponding to cache requests: 3TiB for 512MiB chunks and 600GiB for 64KiB chunks. We scale the two I/O volumes to maintain consistency relative to cache size: the 64KiB configuration uses 40 zones (20% of the 200 zones used for 512MiB), and therefore performs 20% of the total I/O (600GiB). We evaluate two cache-to-workload ratios, 1:2 and 1:10, to capture scenarios of low and high eviction pressure, respectively. The low-eviction case reflects a common deployment where a large cache is expected to yield a high hit ratio, while the high-eviction case stresses the eviction policy. In the latter scenario, the increased write volume is also more likely to trigger SSD garbage collection. Prior to each experiment, we run a conditioning phase to return devices to a baseline state, following the approach of Björling et al. [7]. This preconditioning simulates a disk under sustained use, and produces internal fragmentation.

Our findings broadly mirror those of our initial experiments, but with key differences, particularly for small chunk sizes and resource usage. In *ZNCache*, 64KiB workloads exhibited significantly lower throughput compared to 512MiB workloads. As noted in our previous paper, this was an issue warranting further investigation and could likely be addressed through improved design. In *OxCache*, we achieve markedly better throughput for small chunks. For example, as shown in Figs. 4 and 5, our best-performing workload - a Zipfian distribution with a 1:2 ratio - reaches average throughputs of 0.825 GiB/s (ZNS) and 0.789 GiB/s (block) in *OxCache*, compared to only 0.173 GiB/s (ZNS) and 0.178 GiB/s (block) in *ZNCache*. We attribute these improvements to several design changes.

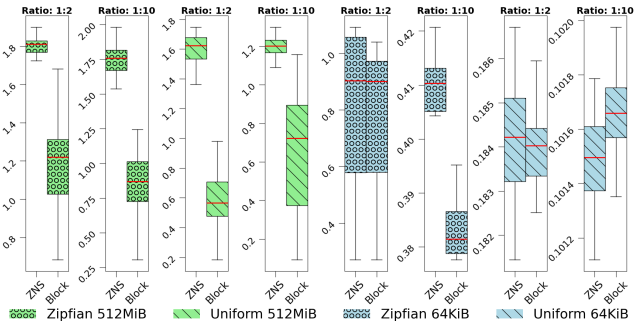

Figure 4: OxCache throughput (GiB/s) comparing ZNS and block-interface (Block) SSDs

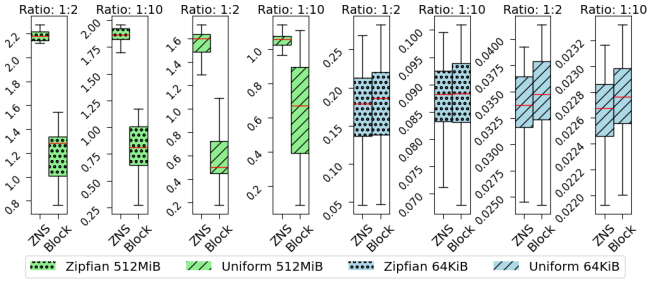

Figure 5: ZNCache throughput (GiB/s) comparing ZNS and block-interface (Block) SSDs

In *ZNCache*, writes required locking at the zone level to maintain write pointer consistency. This was especially problematic for small chunk sizes, where frequent, successive

writes caused lock contention and reduced parallelism. Maintaining correct and consistent write pointer tracking also introduced significant CPU overhead. In contrast, *OxCache* leverages Zone Append, which allows safe writes to ZNS devices without explicit locking, since the device internally manages the write pointer. This eliminates much of the overhead of manual write pointer tracking and reduces the work required to maintain state. The effect is evident in CPU utilization (Appendix D): for example, *OxCache* averages 273% CPU, and 710% CPU utilization for 64KiB and 512MiB respectively, while *ZNCache* averages 4897% CPU, and 3988% CPU for 64KiB and 512MiB. This stark difference highlights the high internal bookkeeping costs in *ZNCache* compared to the lighter-weight design of *OxCache*.

As discussed earlier (§3.0.1), our use of user-level threads lets us service many concurrent requests. Because each request requires a buffer in RAM to hold the requested data, this design can result in high memory usage[3] (Appendix D). Across workloads, *OxCache* averages 429MiB and 50GiB of RAM for 64KiB and 512MiB chunks, respectively, while *ZNCache* averages 254MiB and 7.5GiB for the same workloads (Appendix D). By contrast, *ZNCache* imposes a fixed buffer cap equal to the number of threads specified at startup, which naturally bounds the number of active buffers.

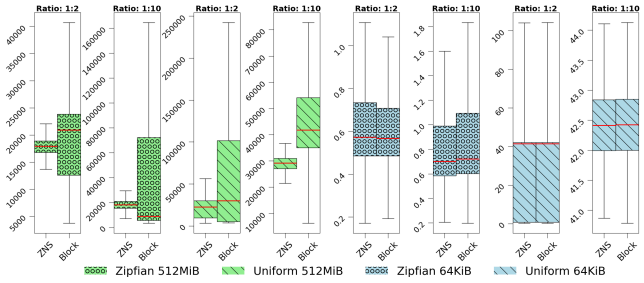

Figure 6: OxCache Get Latency (ms) comparing ZNS and block-interface (Block) SSDs

As demonstrated in Figs. 6 and 7, *OxCache* exhibits latency and throughput trends similar to *ZNCache*. For large chunk sizes, ZNS shows more consistent performance, with average latency 43.00% lower than the block-interface device (compared to *ZNCache*'s 50.58% reduction), and significantly reduced tail latency (on average 84.78% lower, versus *ZNCache*'s 55.87%).

Throughput results follow the same pattern, as shown in Figs. 4 and 5. ZNS consistently achieves higher throughput than block devices and sustains its peak throughput after the initial warmup phase. Compared to *ZNCache*, *OxCache* achieves comparable throughput at large chunk sizes (on average 2.7% lower).

[3]We could reduce peak memory usage by enforcing a limit on the number of active user threads, thereby restricting the number of simultaneously allocated buffers.

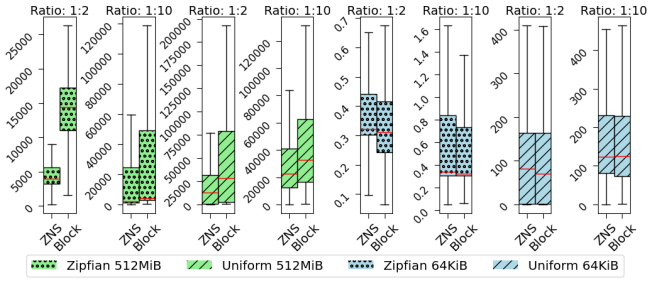

Figure 7: ZNCache Get Latency (ms), end to end latency including hits and misses comparing ZNS and block-interface (Block) SSDs

As shown in Fig. 8, our overall conclusions regarding ZNS versus block remain unchanged. ZNS again delivers consistently high throughput, whereas the block-interface exhibits an initial ramp-up to the hardware's expected maximum, followed by a sharp decline attributable to GC. A similar pattern appears in latency, with an initial spike coinciding with GC activity (Fig. 9).

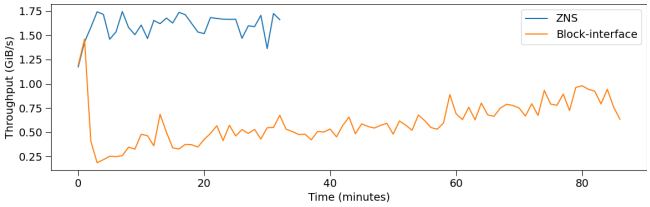

Figure 8: OxCache Get Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:2 ratio.

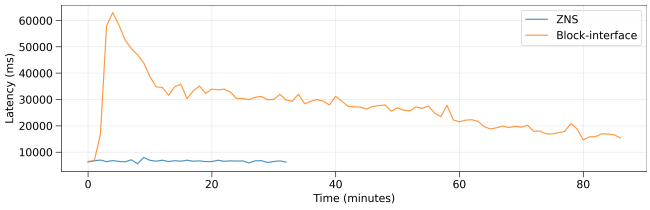

Figure 9: OxCache disk Write latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

As with *ZNCache*, throughput gradually increases even after the hit ratio plateaus. We attribute this effect to the device defragmenting itself during the run due to preconditioning, and investigate it further in §5.2.2.

Small chunk performance highlights the most pronounced differences between the two implementations. With 64KiB

chunks on *ZNCache*, throughput was substantially lower than with large chunks, with the best-performing configuration (Zipfian, ratio 1:2) reaching averages of 0.173GiB/s and 0.178GiB/s for ZNS and block-interface devices, respectively. By contrast, *OxCache* achieves averages of 0.825GiB/s and 0.790GiB/s for ZNS and block, representing increases of 376% and 344%.

As in our previous experiments, we observe far less variance between device types when using small chunks, with ZNS and block-interface exhibiting nearly identical behavior. This reinforces the observation that for workloads with inherently low throughput, such as those using small chunk sizes, ZNS does not necessarily provide a performance advantage over block-interface SSDs. Even with this increased throughput, we did not observe GC; if present, results might differ. We examine this in more detail in §5.2.2.

5.2.2 GC Eval. As with *ZNCache*, to evaluate the effects of GC we execute a uniform random 1:10 workload with a chunk size of 256MiB rather than 512MiB, chosen to introduce greater internal fragmentation. Based on our previous work, this slightly smaller chunk size increases the GC rate on block-interface devices. This workload is most likely to trigger GC, due to both the high eviction rate and uniform access pattern.

To place additional stress on the devices, we remove[4] the artificial latency associated with accessing the remote data store, thereby increasing the read and write rate. We also issue a TRIM command prior to the workload, informing the device that all blocks are no longer in use and may be erased. This places the device in a clean state where no GC should initially be present. Once the disk fills to capacity, we expect device-side GC to begin. This setup contrasts with the preconditioned experiments, where the device starts in a utilized state and we expect GC almost immediately.

As in *ZNCache*, we observe clear signs of garbage collection (GC): once the block-interface device first reaches full capacity, throughput drops sharply and remains below the pre-GC plateau (Fig. 10). Removing the artificial latency cap does not improve performance on the ZNS device, suggesting that the workload was not latency-bound under the cap. In contrast, in *ZNCache* (Fig. 11), removing the cap allowed throughput to increase. This suggests *OxCache* is more efficient, reaching higher throughput. *ZNCache* consumes substantially more CPU resources (Appendix D), suggesting that its performance difference may stem from being CPU-bound.

[4]We discovered unexpected behavior where eliminating the artificial latency reduced throughput. We suspect this is related to contention effects. For our experiments, we set the latency to 1000 microseconds (effectively 0s). We plan to explore this phenomenon in future work.

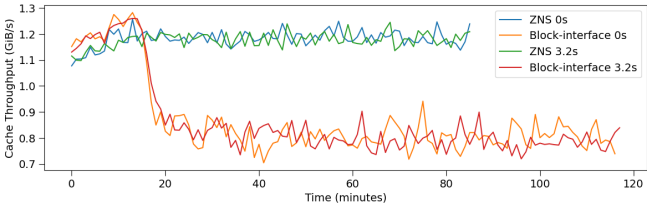

Figure 10: OxCache throughput with GC (1:10 ratio, uniform random, 256MiB chunk, 6TiB of I/O workload) comparing artificial latency (3.2s) with no artificial latency (0s)

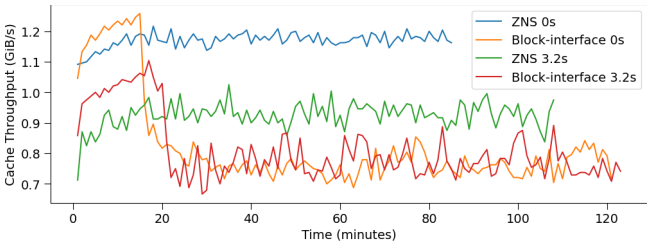

Figure 11: ZNCache throughput with GC (1:10 ratio, uniform random, 256MiB chunk, 6TiB of I/O workload) comparing artificial latency (3.2s) with no artificial latency (0s)

6 FUTURE WORK

We identify several directions for future work.

(1) Eviction policies. We will implement additional eviction policies to measure the impact of software-managed GC on ZNS and compare it directly with hardware-managed GC on block-interface SSDs, clarifying trade-offs across device types. (2) Thread bounding. Because many user-level threads can be active concurrently, we will introduce a bound on active threads. (3) Latency-cap anomaly. In our GC evaluation (§5.2.2), removing the artificial latency cap unexpectedly reduced throughput. We will investigate the root cause and evaluate mitigation strategies.

7 CONCLUSION

We confirm our prior finding that ZNS devices outperform block-interface SSDs when GC is present. Our Rust reimplementation improved small-chunk performance and yielded new insight into GC behavior, while reaffirming that when workloads do not saturate disk capacity, ZNS does not offer clear advantages. Although the transition from C to Rust had a steep learning curve, Rust's type system and safety guarantees helped us produce a more robust, maintainable cache that ultimately outperformed our earlier implementation.

This comparison is not one-to-one, since OxCache incorporates architectural improvements absent from ZNCache,

and results may vary across platforms. Nevertheless, our experience indicates that Rust provides tangible benefits for building reliable storage systems without prohibitive performance costs.

AVAILABILITY

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

8 FOOTNOTES

Generative AI was used to assist with text restructuring and limited code synthesis.

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] [n. d.]. *Fish Shell.* https://fishshell.com Accessed: 2025-09-03.
[2] [n. d.]. *Moka.* https://github.com/moka-rs/moka Accessed: 2025-09-03.
[3] [n. d.]. *Rust for Linux.* https://rust-for-linux.com Accessed: 2025-09-03.
[4] 2025. *Bindgen.* https://github.com/rust-lang/rust-bindgen Accessed: September 3, 2025.
[5] 2025. *Prometheus Monitoring System.* https://prometheus.io/ Accessed: August 27, 2025.
[6] 2025. *Tokio: An Asynchronous Runtime for the Rust Programming Language.* https://tokio.rs/ Accessed: August 27, 2025.
[7] 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).*
[8] Dmitry Bushev. [n. d.]. *How much Rust in Firefox?* https://4e6.github.io/firefox-lang-stats Accessed: 2025-09-03.
[9] CacheLib. 2025. CacheLib – Pluggable caching engine to build and scale high performance cache services. https://cachelib.org. Accessed: 2025-02-07.
[10] Foyer Contributors. 2025. *Foyer: Hybrid cache library for Rust.* https://github.com/foyer-rs/foyer Accessed: 2025-09-03.
[11] Brian Cooper. 2019. *YCSB.* https://github.com/brianfrankcooper/YCSB/blob/ce3eb9ce51c84ee9e236998cdd2cefaeb96798a8/core/src/main/java/site/ycsb/generator/ZipfianGenerator.java Accessed: February 10, 2025.
[12] 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.*
[13] David Tolnay. 2025. *async-trait: Define async functions in traits.* https://crates.io/crates/async-trait Accessed: 2025-09-01.
[14] Brad Fitzpatrick and contributors. 2003. *memcached: a distributed memory object caching system.* https://github.com/memcached/memcached Accessed: 2025-08-29.
[15] Steve Klabnik and Carol Nichols. 2025. *The Rust Programming Language.* No Starch Press. https://doc.rust-lang.org/book/ch17-02-trait-objects.html Accessed: 2025-09-01.
[16] Toby Lawrence and contributors. 2025. *metrics: a metrics instrumentation library for Rust.* https://github.com/metrics-rs/metrics Accessed: 2025-08-29.
[17] linux nvme. 2025. *libnvme: C Library for NVM Express on Linux.* https://github.com/linux-nvme/libnvme Accessed: August 27, 2025.
[18] 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.
[19] metrics-rs contributors. 2025. *metrics-exporter-prometheus: Prometheus exporter for the Rust metrics library.* https://github.com/metrics-rs/metrics/tree/main/metrics-exporter-prometheus Accessed: 2025-08-29.
[20] MITRE Corporation. 2024. CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/ Accessed: 2025-04-30.
[21] Neil Brown. 2011. *The Linux Kernel: Object-Oriented Design Patterns in the Kernel.* https://lwn.net/Articles/444910/
[22] Tokio project contributors. 2025. *tracing: Application-level tracing for Rust.* https://github.com/tokio-rs/tracing Accessed: 2025-08-29.
[23] John Ramsden and Sam Cheng. 2025. ZNCache - ZNS Workload Analysis. https://github.com/johnramsden/ZNCache. University of British Columbia.
[24] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy.* IEEE.
[25] 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).*
[26] Nick Tehrany and Animesh Trivedi. 2022. Understanding nvme zoned namespace (zns) flash ssd storage devices. (2022).
[27] The Rust Programming Language Team. 2025. *Message Passing in Rust.* https://doc.rust-lang.org/book/ch16-02-message-passing.html Accessed: 2025-08-28.
[28] The Rust Project Developers. 2025. *Rust Reference: Send and Sync.* https://doc.rust-lang.org/reference/special-types-and-traits.html#send-and-sync Accessed: 2025-09-01.
[29] The Rust Project Developers. 2025. *Rust Reference: Traits and async functions.* https://doc.rust-lang.org/reference/items/traits.html#async-functions-in-traits Accessed: 2025-09-01.
[30] Erick Tryzelaar, David Tolnay, and Contributors. 2025. *Serde: Serialization Framework for Rust.* https://serde.rs/ Accessed: August 27, 2025.
[31] 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.*
[32] WiredTiger Project. 2025. Chunk Cache in WiredTiger. https://github.com/wiredtiger/wiredtiger.github.com/blob/062e0eb42ed1dc8777f8cf1b8651ca9eb6ac33ce/develop/chunkcache.html Accessed: February 10, 2025.
[33] Chongzhuo Yang, Zhang Cao, Chang Guo, Ming Zhao, and Zhichao Cao. 2024. Can ZNS SSDs be Better Storage Devices for Persistent Cache? *(HotStorage '24).*
[34] 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).
[35] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. [n. d.]. Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.*
[36] Zoned Storage Project. 2025. Write Ordering Control. https://zonedstorage.io/docs/linux/sched#zone-write-plugging. Accessed: 2025-04-17.

[37] Zoned Storage Project. 2025. Zoned Storage Devices. https://zonedstorage.io/docs/introduction/zoned-storage#zone-append. Accessed: 2025-08-28.

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

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 LATENCY AND THROUGHPUT TABLES

The following tables show distributions for latency and throughput throughout workloads.

Name	Mean (ms)	P99 (ms)
ZNS-512M-UNIF-2	20548.26 (+62.50%)	43584.92 (+78.91%)
Block-512M-UNIF-2	54793.77	206632.99
ZNS-512M-UNIF-10	27342.11 (+48.56%)	37412.80 (+84.82%)
Block-512M-UNIF-10	53156.16	246416.68
ZNS-512M-ZIPF-2	17239.12 (+36.66%)	25178.34 (+89.37%)
Block-512M-ZIPF-2	27218.81	236752.36
ZNS-512M-ZIPF-10	18509.96 (+52.31%)	35350.96 (+86.03%)
Block-512M-ZIPF-10	38810.68	252975.99
ZNS-64K-UNIF-2	22.02 (+0.09%)	43.22 (-0.02%)
Block-64K-UNIF-2	22.04	43.21
ZNS-64K-UNIF-10	38.59 (-0.14%)	44.22 (+0.05%)
Block-64K-UNIF-10	38.53	44.24
ZNS-64K-ZIPF-2	4.84 (+3.33%)	42.16 (+0.07%)
Block-64K-ZIPF-2	5.00	42.19
ZNS-64K-ZIPF-10	9.98 (+4.81%)	42.49 (+0.10%)
Block-64K-ZIPF-10	10.49	42.53

Table 5: Latency comparison results for Gets

Name	Mean	P99
ZNS-512M-UNIF-2	1.60 GiB/s (+167.58%)	1.74 GiB/s (+41.83%)
Block-512M-UNIF-2	611.46 MiB/s	1.23 GiB/s
ZNS-512M-UNIF-10	1.20 GiB/s (+87.37%)	1.30 GiB/s (+12.30%)
Block-512M-UNIF-10	655.61 MiB/s	1.16 GiB/s
ZNS-512M-ZIPF-2	1.80 GiB/s (+53.51%)	1.89 GiB/s (+8.15%)
Block-512M-ZIPF-2	1.17 GiB/s	1.75 GiB/s
ZNS-512M-ZIPF-10	1.74 GiB/s (+105.13%)	1.98 GiB/s (+16.63%)
Block-512M-ZIPF-10	868.63 MiB/s	1.70 GiB/s
ZNS-64K-UNIF-2	181.22 MiB/s (+0.10%)	191.06 MiB/s (+0.14%)
Block-64K-UNIF-2	181.04 MiB/s	190.80 MiB/s
ZNS-64K-UNIF-10	103.56 MiB/s (-0.15%)	104.22 MiB/s (-0.18%)
Block-64K-UNIF-10	103.72 MiB/s	104.42 MiB/s
ZNS-64K-ZIPF-2	824.99 MiB/s (+4.47%)	1.09 GiB/s (+4.91%)
Block-64K-ZIPF-2	789.73 MiB/s	1.04 GiB/s
ZNS-64K-ZIPF-10	398.29 MiB/s (+4.99%)	435.08 MiB/s (+2.37%)
Block-64K-ZIPF-10	379.36 MiB/s	425.00 MiB/s

Table 6: Throughput comparison results for Gets

C GRAPHS

The following section has detailed graphs for various metrics:

(1) Get latency: End-to-end latency including both hits and misses for the entire path required to “get” an object from the cache
(2) Hit latency: The entire code path executed when a cache hit occurs (includes read latency)

(3) Miss latency: The entire code path executed when a cache miss occurs (includes write latency)
(4) Read latency: Disk IO read latency
(5) Write latency: Disk IO write latency
(6) Get throughput: Complete cache throughput (eg. user requests a 512MiB chunk, this is 512MiB of data contributing to throughput)
(7) Read throughput: Throughput contributions from only disk reads
(8) Write throughput: Throughput contributions from only disk writes
(9) Hit ratio: Cache hit ratio throughout a run

C.1 Get Latency

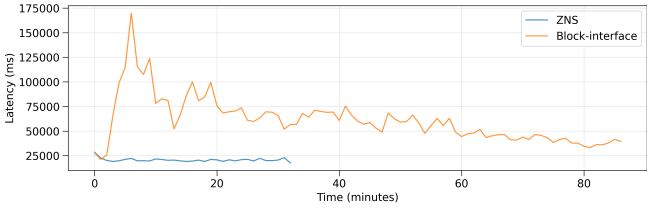

Figure 12: Cache Get latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

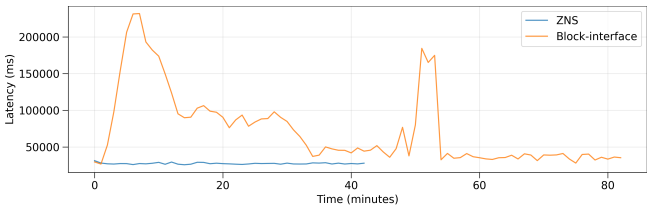

Figure 13: Cache Get latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

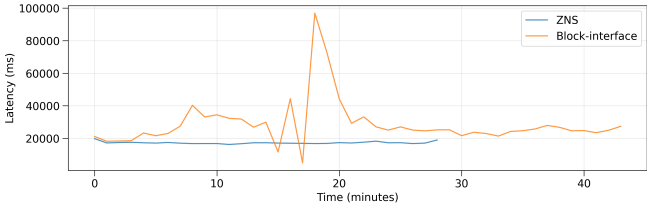

Figure 14: Cache Get latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

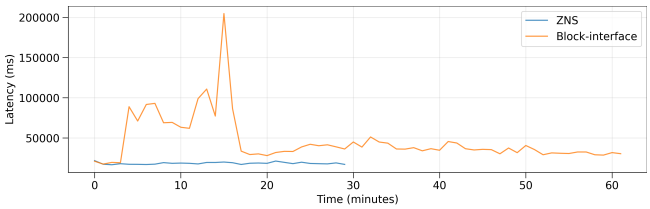

Figure 15: Cache Get latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

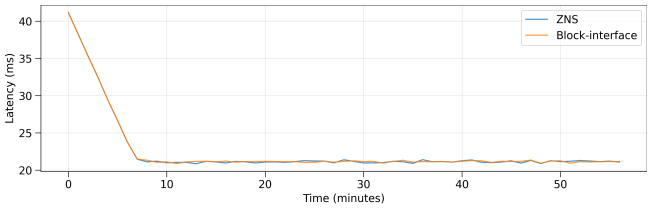

Figure 16: Cache Get latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

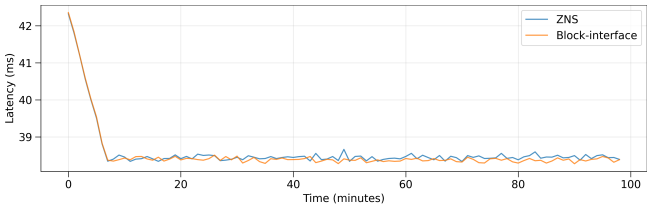

Figure 17: Cache Get latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

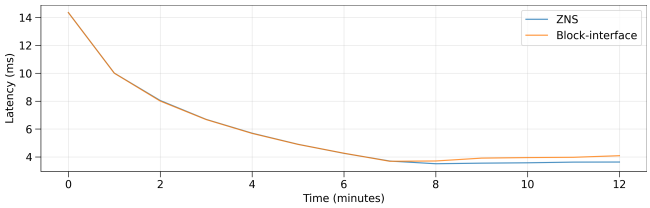

Figure 18: Cache Get latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

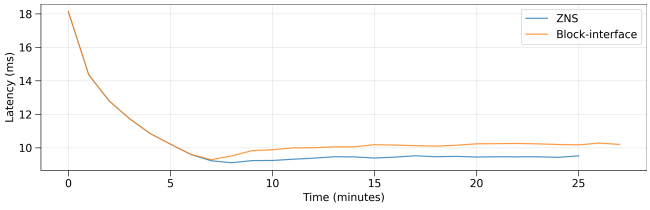

Figure 19: Cache Get latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

C.2 Hit Latency

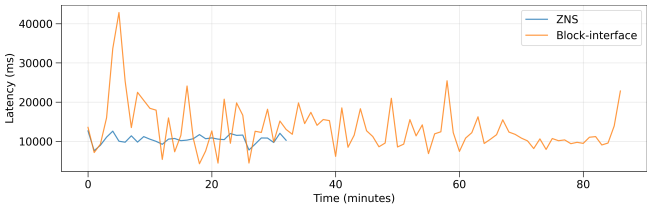

Figure 20: Cache Hit latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

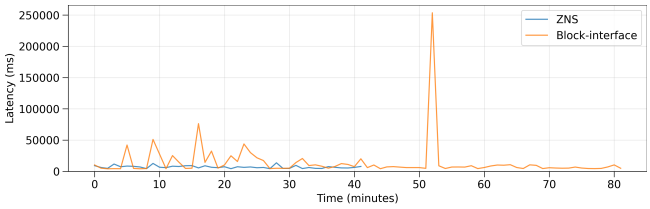

Figure 21: Cache Hit latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

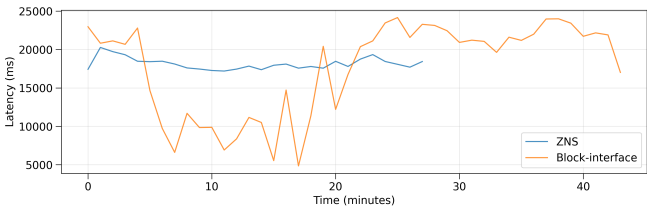

Figure 22: Cache Hit latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

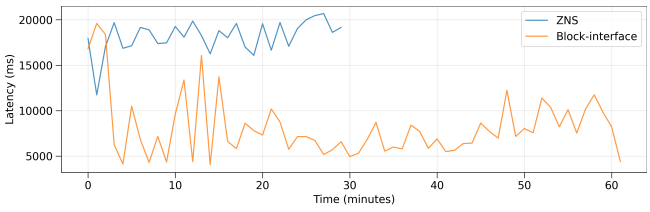

Figure 23: Cache Hit latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

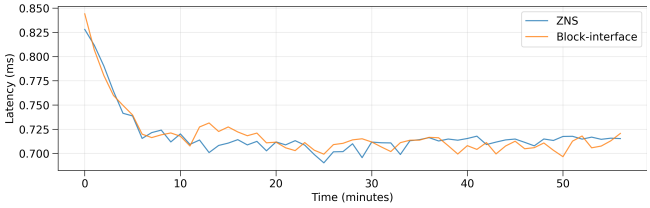

Figure 24: Cache Hit latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

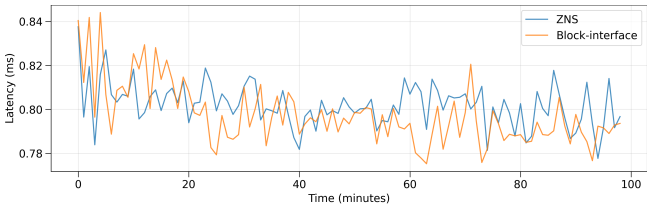

Figure 25: Cache Hit latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

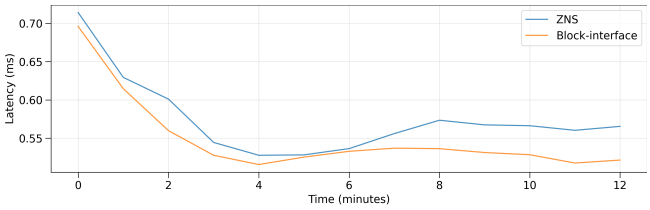

Figure 26: Cache Hit latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

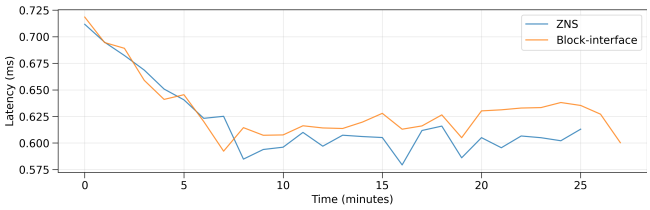

Figure 27: Cache Hit latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

C.3 Miss Latency

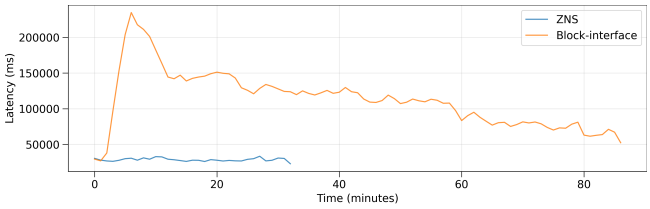

Figure 28: Cache Miss latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

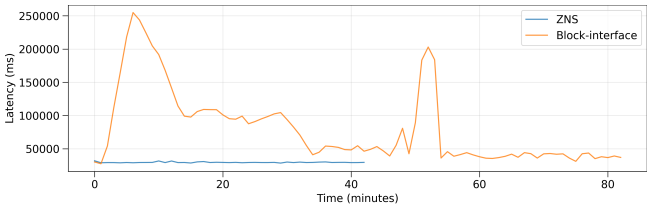

Figure 29: Cache Miss latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

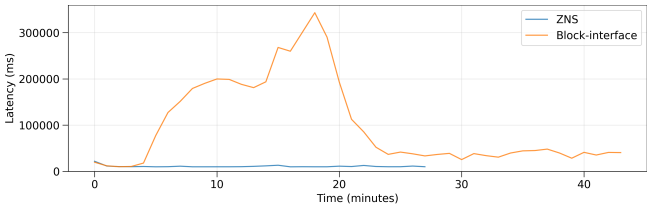

Figure 30: Cache Miss latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

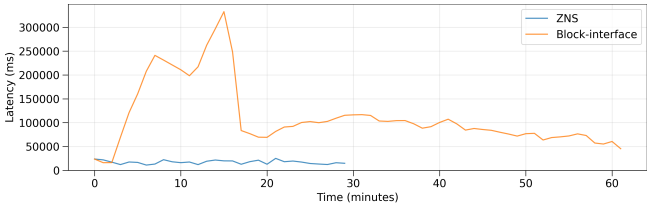

Figure 31: Cache Miss latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

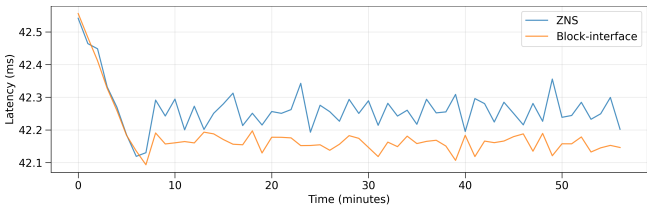

Figure 32: Cache Miss latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

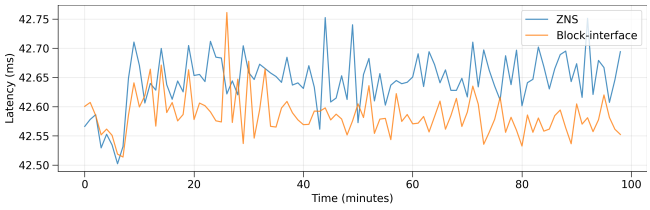

Figure 33: Cache Miss latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

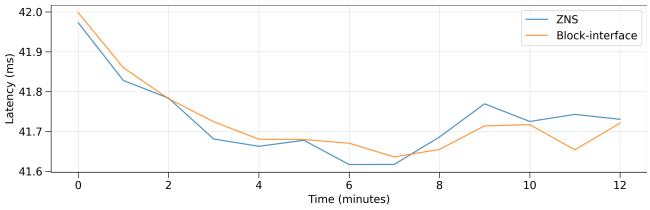

Figure 34: Cache Miss latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

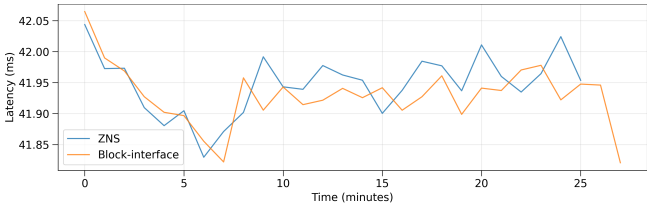

Figure 35: Cache Miss latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

C.4 Read Latency

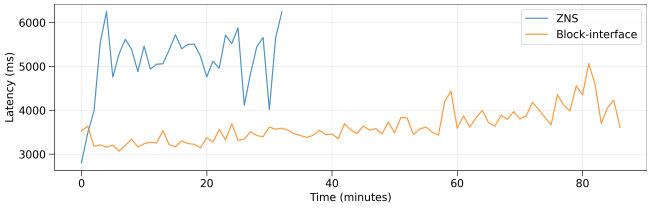

Figure 36: Disk Read latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

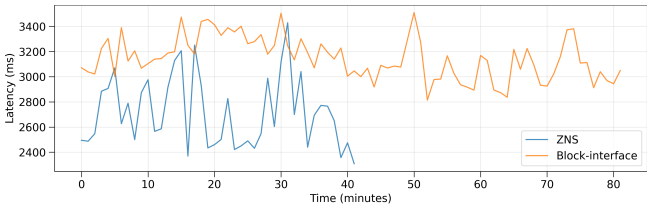

Figure 37: Disk Read latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

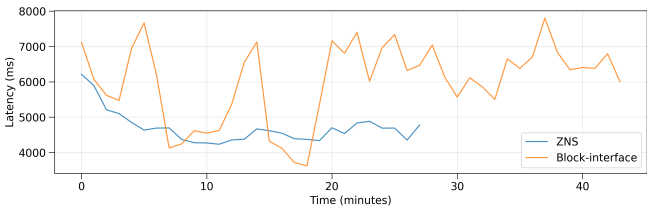

Figure 38: Disk Read latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

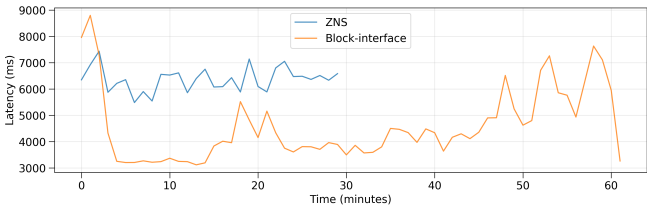

Figure 39: Disk Read latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

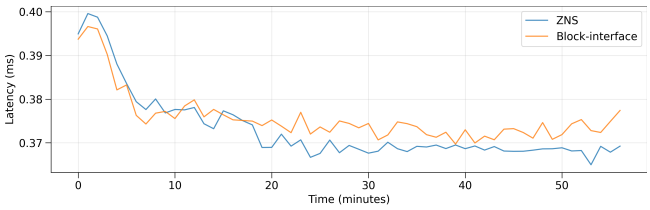

Figure 40: Disk Read latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

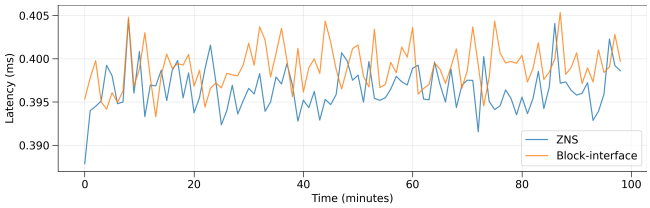

Figure 41: Disk Read latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

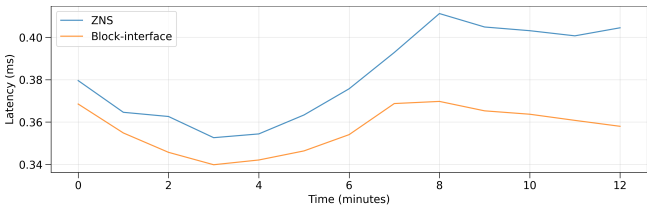

Figure 42: Disk Read latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

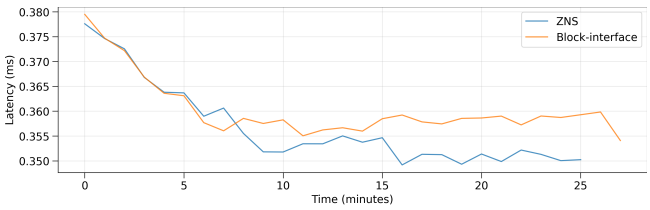

Figure 43: Disk Read latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

C.5 Write Latency

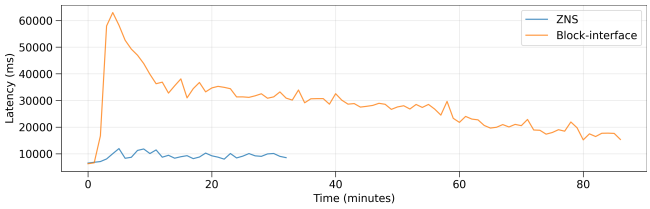

Figure 44: Disk Write latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

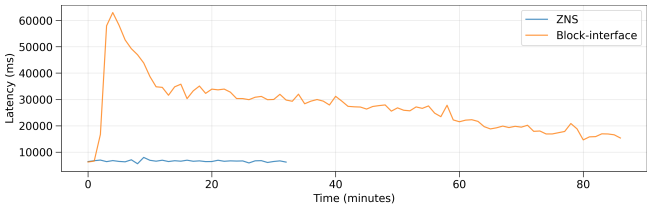

Figure 45: Disk Write latency (ms) for 512M chunk size, Uniform distribution, and 1:2 ratio.

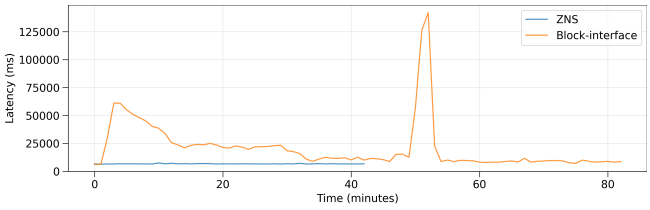

Figure 46: Disk Write latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

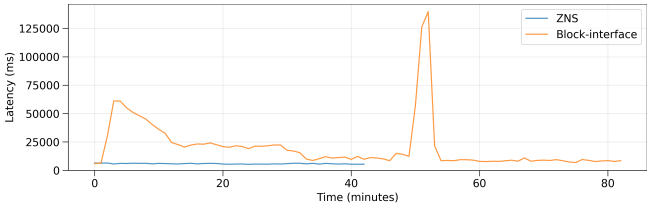

Figure 47: Disk Write latency (ms) for 512M chunk size, Uniform distribution, and 1:10 ratio.

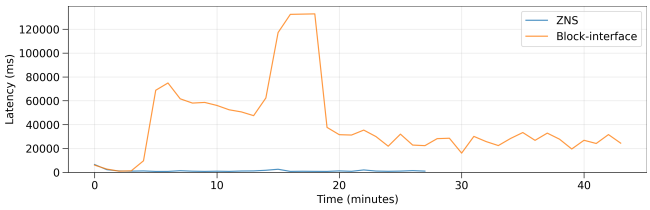

Figure 48: Disk Write latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

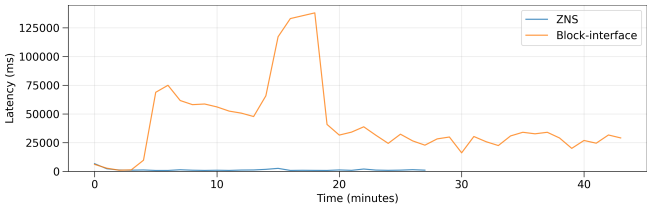

Figure 49: Disk Write latency (ms) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

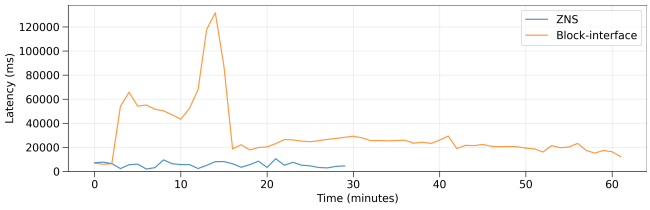

Figure 50: Disk Write latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

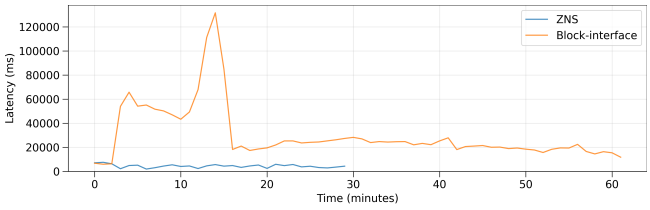

Figure 51: Disk Write latency (ms) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

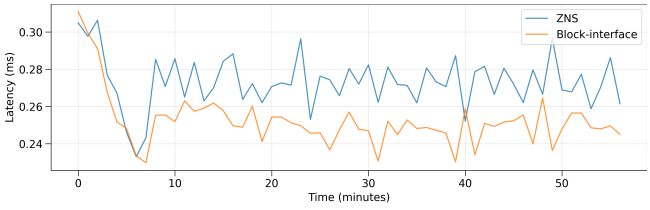

Figure 52: Disk Write latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

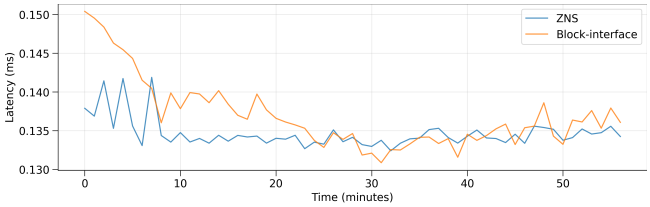

Figure 53: Disk Write latency (ms) for 64K chunk size, Uniform distribution, and 1:2 ratio.

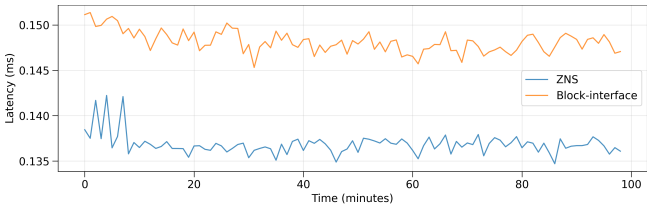

Figure 54: Disk Write latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

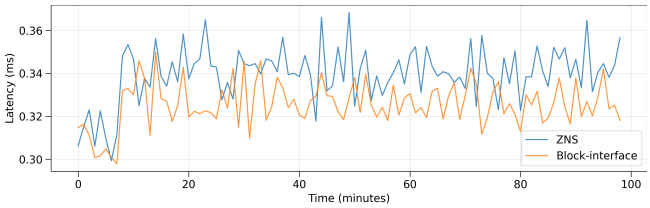

Figure 55: Disk Write latency (ms) for 64K chunk size, Uniform distribution, and 1:10 ratio.

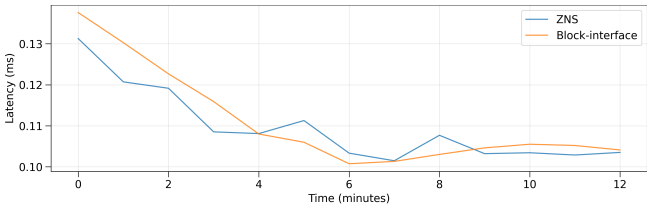

Figure 56: Disk Write latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

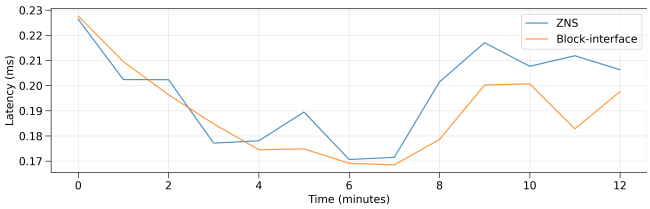

Figure 57: Disk Write latency (ms) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

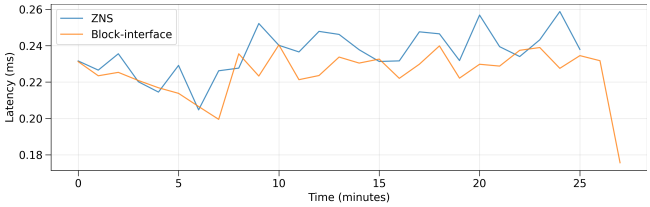

Figure 58: Disk Write latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

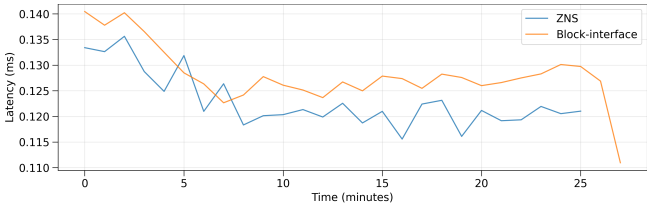

Figure 59: Disk Write latency (ms) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

C.6 Get Throughput

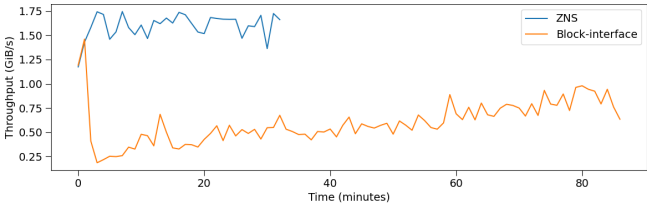

Figure 60: Cache Get Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:2 ratio.

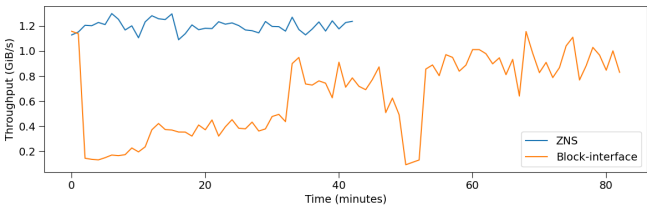

Figure 61: Cache Get Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:10 ratio.

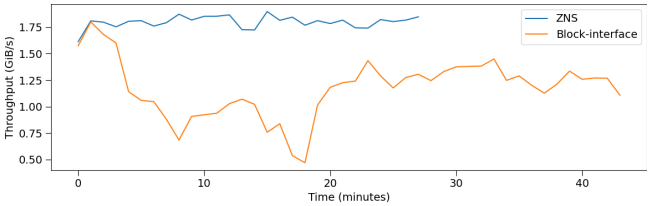

Figure 62: Cache Get Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

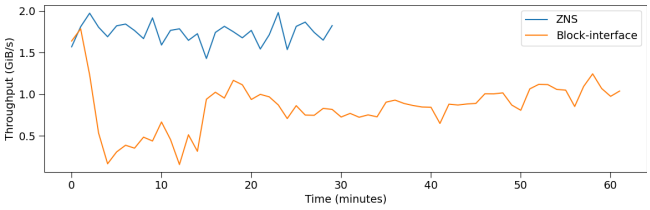

Figure 63: Cache Get Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

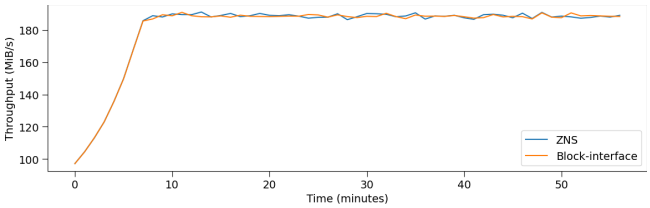

Figure 64: Cache Get Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:2 ratio.

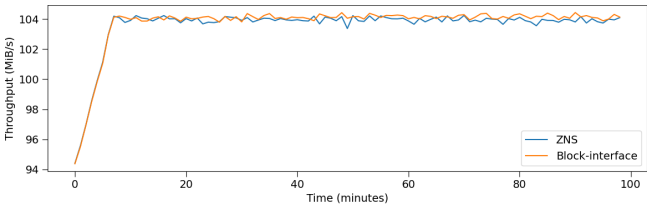

Figure 65: Cache Get Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:10 ratio.

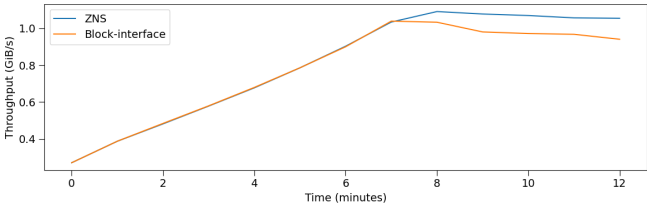

Figure 66: Cache Get Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

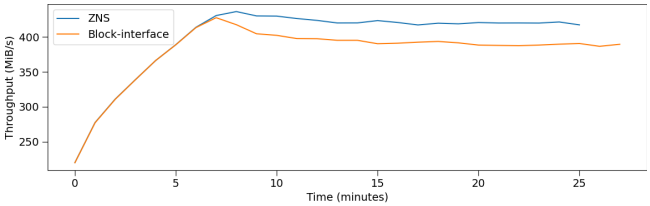

Figure 67: Cache Get Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

C.7 Read Throughput

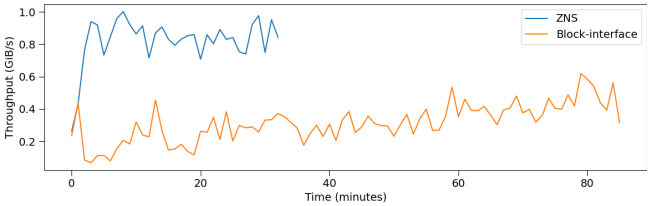

Figure 68: Disk Read Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:2 ratio.

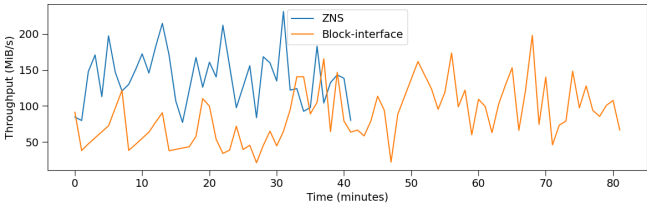

Figure 69: Disk Read Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:10 ratio.

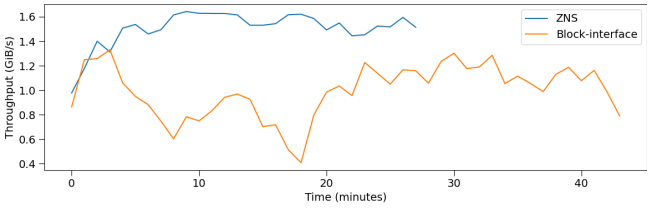

Figure 70: Disk Read Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

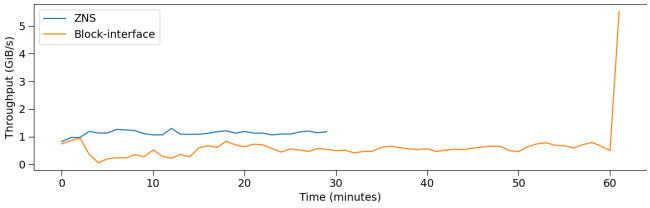

Figure 71: Disk Read Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

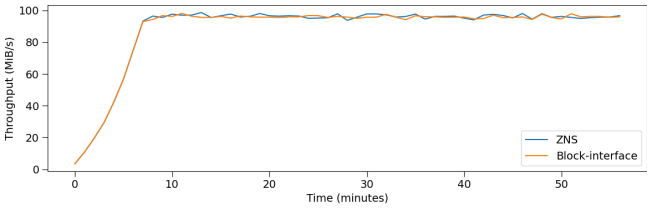

Figure 72: Disk Read Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:2 ratio.

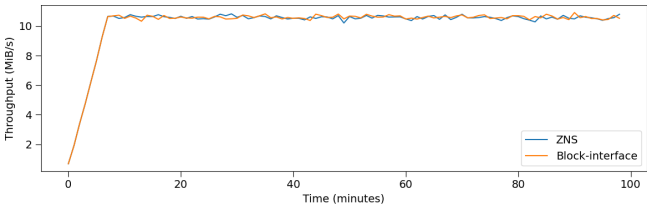

Figure 73: Disk Read Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:10 ratio.

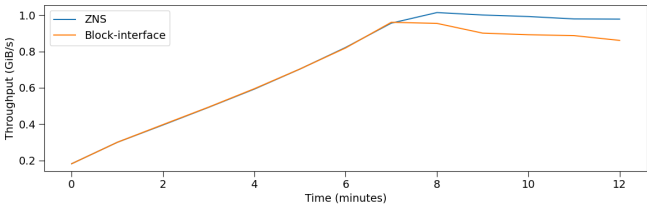

Figure 74: Disk Read Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

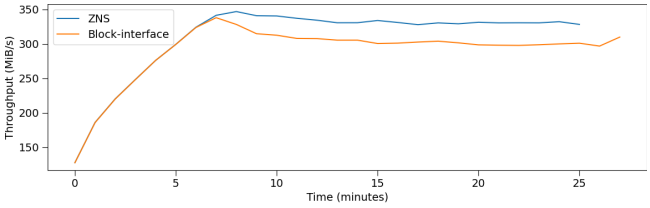

Figure 75: Disk Read Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

C.8 Write Throughput

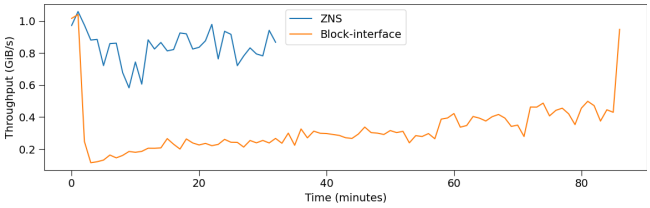

Figure 76: Disk Write Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:2 ratio.

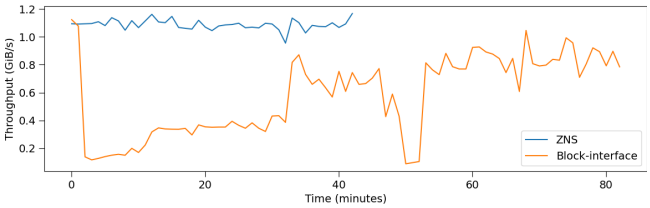

Figure 77: Disk Write Throughput (GiB/s) for 512M chunk size, Uniform distribution, and 1:10 ratio.

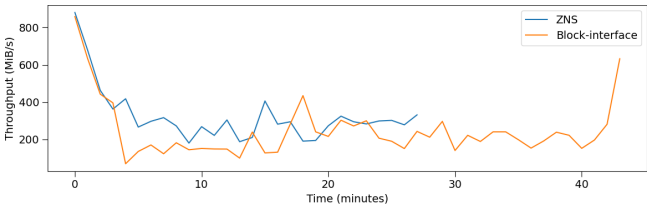

Figure 78: Disk Write Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:2 ratio.

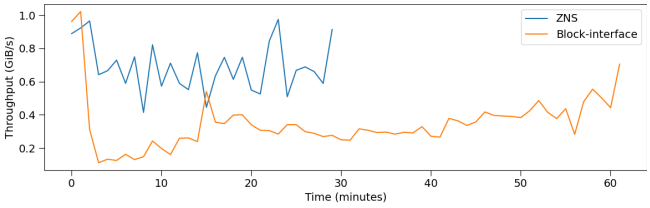

Figure 79: Disk Write Throughput (GiB/s) for 512M chunk size, Zipfian distribution, and 1:10 ratio.

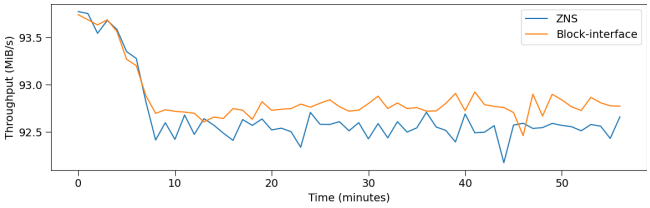

Figure 80: Disk Write Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:2 ratio.

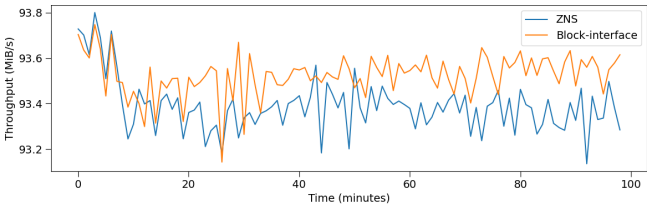

Figure 81: Disk Write Throughput (GiB/s) for 64K chunk size, Uniform distribution, and 1:10 ratio.

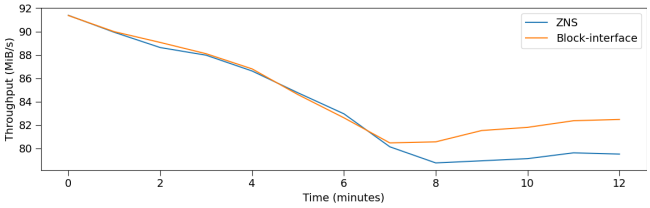

Figure 82: Disk Write Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:2 ratio.

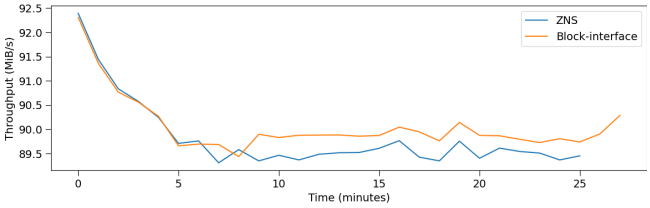

Figure 83: Disk Write Throughput (GiB/s) for 64K chunk size, Zipfian distribution, and 1:10 ratio.

C.9 Hit Ratio

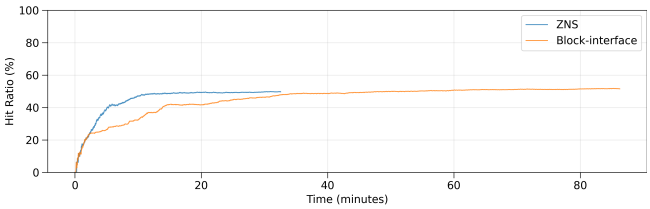

Figure 84: Cache Hit Ratio for 512M chunk size, Uniform distribution, and 1:2 ratio.

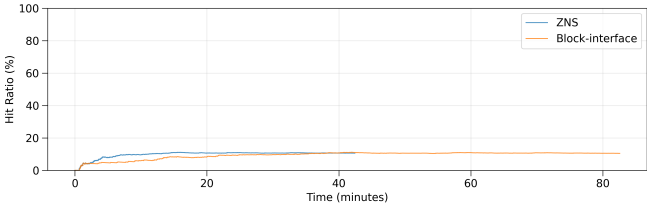

Figure 85: Cache Hit Ratio for 512M chunk size, Uniform distribution, and 1:10 ratio.

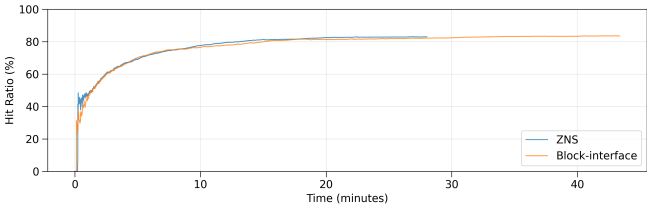

Figure 86: Cache Hit Ratio for 512M chunk size, Zipfian distribution, and 1:2 ratio.

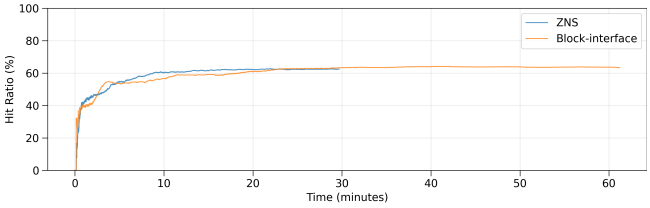

Figure 87: Cache Hit Ratio for 512M chunk size, Zipfian distribution, and 1:10 ratio.

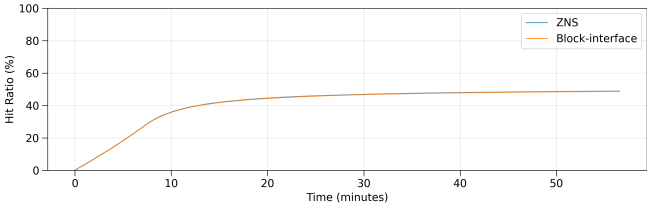

Figure 88: Cache Hit Ratio for 64K chunk size, Uniform distribution, and 1:2 ratio.

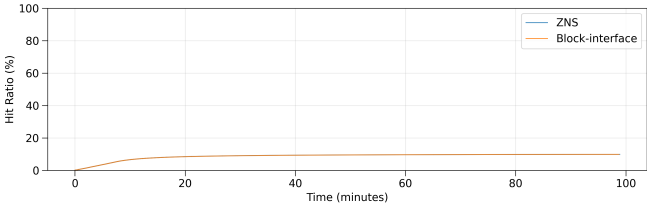

Figure 89: Cache Hit Ratio for 64K chunk size, Uniform distribution, and 1:10 ratio.

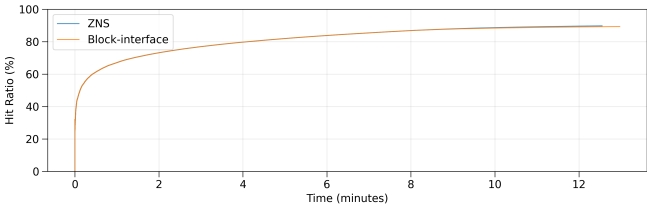

Figure 90: Cache Hit Ratio for 64K chunk size, Zipfian distribution, and 1:2 ratio.

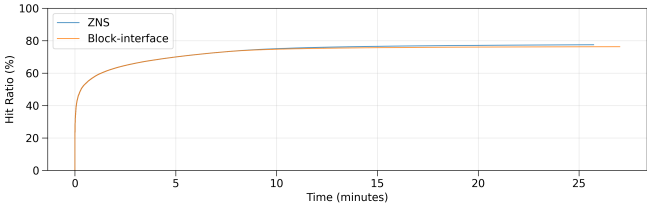

Figure 91: Cache Hit Ratio for 64K chunk size, Zipfian distribution, and 1:10 ratio.

D RESOURCE USE

The following tables show distributions for CPU and RAM usage throughout workloads.

Table 7: OxCache Average Resource Usage - ZNS Chunk Size 536870912

Metric	Value
CPU Usage (%)	709.78
Memory Usage (GiB)	50.166

Table 8: OxCache Average Resource Usage - ZNS Chunk Size 65536

Metric	Value
CPU Usage (%)	272.61
Memory Usage (GiB)	0.419

Table 9: OxCache Average Resource Usage - Block-interface Chunk Size 536870912

Metric	Value
CPU Usage (%)	336.09
Memory Usage (GiB)	57.190

Table 10: OxCache Average Resource Usage - Block-interface Chunk Size 65536

Metric	Value
CPU Usage (%)	269.37
Memory Usage (GiB)	0.520

Table 11: ZNCache Average Resource Usage - ZNS Chunk Size 536870912

Metric	Value
CPU Usage (%)	3988.35
Memory Usage (GiB)	7.472

Table 12: ZNCache Average Resource Usage - ZNS Chunk Size 65536

Metric	Value
CPU Usage (%)	4896.73
Memory Usage (GiB)	0.248

Table 13: ZNCache Average Resource Usage - Block-interface Chunk Size 536870912

Metric	Value
CPU Usage (%)	3723.64
Memory Usage (GiB)	10.256

Table 14: ZNCache Average Resource Usage - Block-interface Chunk Size 65536

Metric	Value
CPU Usage (%)	4890.90
Memory Usage (GiB)	0.235

Table 15: OxCache CPU Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Uniform, Ratio: 10)

Metric	Value (%)
Max	6385.00
Mean	576.98
Median	559.00
Std Dev	377.03
95th Percentile	1046.00
99th Percentile	1238.00

Table 16: OxCache Memory Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Uniform, Ratio: 10)

Metric	Value (GiB)
Max	86.994
Mean	58.059
Median	57.972
Std Dev	3.557
95th Percentile	61.898
99th Percentile	63.426

Table 17: OxCache CPU Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 10)

Metric	Value (%)
Max	4808.00
Mean	821.03
Median	757.50
Std Dev	466.10
95th Percentile	1631.00
99th Percentile	2410.00

Table 18: OxCache Memory Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 10)

Metric	Value (GiB)
Max	62.089
Mean	46.468
Median	46.207
Std Dev	6.101
95th Percentile	56.048
99th Percentile	59.319

Table 19: OxCache CPU Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Uniform, Ratio: 2)

Metric	Value (%)
Max	235.00
Mean	204.43
Median	213.00
Std Dev	23.93
95th Percentile	224.00
99th Percentile	228.00

Table 20: OxCache Memory Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Uniform, Ratio: 2)

Metric	Value (GiB)
Max	0.682
Mean	0.506
Median	0.548
Std Dev	0.152
95th Percentile	0.675
99th Percentile	0.682

Table 21: OxCache CPU Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Uniform, Ratio: 2)

Metric	Value (%)
Max	6201.00
Mean	746.48
Median	676.00
Std Dev	490.69
95th Percentile	1526.00
99th Percentile	2041.00

Table 22: OxCache Memory Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Uniform, Ratio: 2)

Metric	Value (GiB)
Max	84.663
Mean	54.695
Median	54.894
Std Dev	5.699
95th Percentile	61.389
99th Percentile	65.400

Table 23: OxCache CPU Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 2)

Metric	Value (%)
Max	4181.00
Mean	694.64
Median	667.33
Std Dev	287.96
95th Percentile	1114.00
99th Percentile	1347.00

Table 24: OxCache Memory Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 2)

Metric	Value (GiB)
Max	55.104
Mean	41.441
Median	41.514
Std Dev	4.178
95th Percentile	45.980
99th Percentile	49.933

Table 25: OxCache CPU Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Zipfian, Ratio: 10)

Metric	Value (%)
Max	382.00
Mean	321.62
Median	324.00
Std Dev	29.53
95th Percentile	358.00
99th Percentile	371.00

Table 26: OxCache Memory Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Zipfian, Ratio: 10)

Metric	Value (GiB)
Max	0.477
Mean	0.356
Median	0.406
Std Dev	0.118
95th Percentile	0.474
99th Percentile	0.477

Table 27: OxCache CPU Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Uniform, Ratio: 10)

Metric	Value (%)
Max	144.00
Mean	131.18
Median	132.00
Std Dev	5.53
95th Percentile	139.00
99th Percentile	141.00

Table 28: OxCache Memory Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Uniform, Ratio: 10)

Metric	Value (GiB)
Max	0.698
Mean	0.558
Median	0.622
Std Dev	0.139
95th Percentile	0.687
99th Percentile	0.697

Table 29: OxCache CPU Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Zipfian, Ratio: 2)

Metric	Value (%)
Max	606.00
Mean	433.20
Median	447.50
Std Dev	89.56
95th Percentile	545.00
99th Percentile	573.00

Table 30: OxCache Memory Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Zipfian, Ratio: 2)

Metric	Value (GiB)
Max	0.394
Mean	0.255
Median	0.274
Std Dev	0.094
95th Percentile	0.390
99th Percentile	0.394

Table 31: OxCache CPU Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Uniform, Ratio: 10)

Metric	Value (%)
Max	145.00
Mean	128.29
Median	129.00
Std Dev	4.73
95th Percentile	135.00
99th Percentile	137.00

Table 32: OxCache Memory Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Uniform, Ratio: 10)

Metric	Value (GiB)
Max	0.885
Mean	0.790
Median	0.861
Std Dev	0.158
95th Percentile	0.884
99th Percentile	0.884

Table 33: OxCache CPU Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Uniform, Ratio: 10)

Metric	Value (%)
Max	6400.00
Mean	301.69
Median	240.00
Std Dev	321.47
95th Percentile	783.00
99th Percentile	1025.00

Table 34: OxCache Memory Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Uniform, Ratio: 10)

Metric	Value (GiB)
Max	87.597
Mean	60.682
Median	61.020
Std Dev	3.230
95th Percentile	64.023
99th Percentile	65.712

Table 35: OxCache CPU Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Zipfian, Ratio: 2)

Metric	Value (%)
Max	613.00
Mean	430.17
Median	456.00
Std Dev	86.46
95th Percentile	540.00
99th Percentile	579.00

Table 36: OxCache Memory Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Zipfian, Ratio: 2)

Metric	Value (GiB)
Max	0.388
Mean	0.252
Median	0.259
Std Dev	0.095
95th Percentile	0.384
99th Percentile	0.388

Table 37: OxCache CPU Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 2)

Metric	Value (%)
Max	4199.00
Mean	455.93
Median	396.00
Std Dev	302.44
95th Percentile	997.00
99th Percentile	1376.00

Table 38: OxCache Memory Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 2)

Metric	Value (GiB)
Max	62.841
Mean	49.142
Median	46.535
Std Dev	8.140
95th Percentile	59.982
99th Percentile	61.013

Table 39: OxCache CPU Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Zipfian, Ratio: 10)

Metric	Value (%)
Max	362.00
Mean	317.02
Median	321.00
Std Dev	27.30
95th Percentile	348.00
99th Percentile	356.00

Table 40: OxCache Memory Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Zipfian, Ratio: 10)

Metric	Value (GiB)
Max	0.483
Mean	0.365
Median	0.416
Std Dev	0.116
95th Percentile	0.478
99th Percentile	0.482

Table 41: OxCache CPU Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Uniform, Ratio: 2)

Metric	Value (%)
Max	6202.00
Mean	244.93
Median	183.00
Std Dev	269.34
95th Percentile	616.00
99th Percentile	892.00

Table 42: OxCache Memory Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Uniform, Ratio: 2)

Metric	Value (GiB)
Max	80.824
Mean	59.332
Median	59.837
Std Dev	4.172
95th Percentile	62.237
99th Percentile	63.366

Table 43: OxCache CPU Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 10)

Metric	Value (%)
Max	4808.00
Mean	341.80
Median	291.50
Std Dev	281.82
95th Percentile	797.00
99th Percentile	1178.00

Table 44: OxCache Memory Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 10)

Metric	Value (GiB)
Max	70.125
Mean	59.605
Median	60.594
Std Dev	5.103
95th Percentile	63.527
99th Percentile	64.072

Table 45: OxCache CPU Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Uniform, Ratio: 2)

Metric	Value (%)
Max	232.00
Mean	202.01
Median	209.00
Std Dev	22.32
95th Percentile	220.00
99th Percentile	225.00

Table 46: OxCache Memory Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Uniform, Ratio: 2)

Metric	Value (GiB)
Max	0.925
Mean	0.671
Median	0.736
Std Dev	0.207
95th Percentile	0.916
99th Percentile	0.925

Table 47: ZNCache CPU Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 10)

Metric	Value (%)
Max	5097.00
Mean	3833.31
Median	4231.00
Std Dev	1105.56
95th Percentile	4833.00
99th Percentile	4928.00

Table 48: ZNCache Memory Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 10)

Metric	Value (GiB)
Max	11.675
Mean	7.655
Median	7.500
Std Dev	1.169
95th Percentile	9.293
99th Percentile	10.169

Table 49: ZNCache CPU Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Uniform, Ratio: 10)

Metric	Value (%)
Max	4952.53
Mean	4904.34
Median	4907.00
Std Dev	61.98
95th Percentile	4910.00
99th Percentile	4911.00

Table 50: ZNCache Memory Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Uniform, Ratio: 10)

Metric	Value (GiB)
Max	0.552
Mean	0.357
Median	0.365
Std Dev	0.149
95th Percentile	0.543
99th Percentile	0.550

Table 51: ZNCache CPU Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Uniform, Ratio: 10)

Metric	Value (%)
Max	5728.00
Mean	4143.40
Median	4368.00
Std Dev	874.01
95th Percentile	5006.00
99th Percentile	5062.00

Table 52: ZNCache Memory Usage Statistics - ZNS (Chunk Size: 536870912, Distribution: Uniform, Ratio: 10)

Metric	Value (GiB)
Max	9.000
Mean	7.289
Median	7.392
Std Dev	0.506
95th Percentile	7.503
99th Percentile	7.827

Table 53: ZNCache CPU Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Zipfian, Ratio: 10)

Metric	Value (%)
Max	4917.00
Mean	4889.12
Median	4892.00
Std Dev	76.50
95th Percentile	4903.00
99th Percentile	4911.00

Table 54: ZNCache Memory Usage Statistics - ZNS (Chunk Size: 65536, Distribution: Zipfian, Ratio: 10)

Metric	Value (GiB)
Max	0.210
Mean	0.139
Median	0.153
Std Dev	0.052
95th Percentile	0.205
99th Percentile	0.209

Table 55: ZNCache CPU Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 10)

Metric	Value (%)
Max	4723.00
Mean	2762.79
Median	3213.00
Std Dev	1353.20
95th Percentile	4353.00
99th Percentile	4521.00

Table 56: ZNCache Memory Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Zipfian, Ratio: 10)

Metric	Value (GiB)
Max	22.825
Mean	12.780
Median	11.677
Std Dev	4.005
95th Percentile	20.128
99th Percentile	21.741

Table 57: ZNCache CPU Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Uniform, Ratio: 10)

Metric	Value (%)
Max	4915.00
Mean	4900.81
Median	4905.00
Std Dev	11.25
95th Percentile	4911.00
99th Percentile	4912.00

Table 58: ZNCache Memory Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Uniform, Ratio: 10)

Metric	Value (GiB)
Max	0.645
Mean	0.335
Median	0.337
Std Dev	0.179
95th Percentile	0.621
99th Percentile	0.641

Table 59: ZNCache CPU Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Uniform, Ratio: 10)

Metric	Value (%)
Max	5480.00
Mean	4684.49
Median	4759.00
Std Dev	503.40
95th Percentile	4982.00
99th Percentile	5064.00

Table 60: ZNCache Memory Usage Statistics - Block-interface (Chunk Size: 536870912, Distribution: Uniform, Ratio: 10)

Metric	Value (GiB)
Max	9.754
Mean	7.732
Median	7.699
Std Dev	0.603
95th Percentile	8.544
99th Percentile	9.057

Table 61: ZNCache CPU Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Zipfian, Ratio: 10)

Metric	Value (%)
Max	4922.00
Mean	4880.99
Median	4881.00
Std Dev	18.98
95th Percentile	4912.00
99th Percentile	4918.00

Table 62: ZNCache Memory Usage Statistics - Block-interface (Chunk Size: 65536, Distribution: Zipfian, Ratio: 10)

Metric	Value (GiB)
Max	0.232
Mean	0.135
Median	0.130
Std Dev	0.060
95th Percentile	0.225
99th Percentile	0.232