Thesis Proposal

Efficient and Sustainable Data Retrieval at Scale

Sara McAllister

February 2024

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Nathan Beckmann, Co-Chair
Gregory R. Ganger, Co-Chair
George Amvrosiadis
Daniel S. Berger, Microsoft Azure & University of Washington
Margo Seltzer, University of British Columbia

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Abstract

Datacenters are projected to account for 33% of the global carbon emissions by 2050. As datacenters increasingly rely on renewable energy for power, the majority of datacenter emissions will be embodied — emissions from lifecycle stages including acquiring raw materials, manufacturing, transportation, and disposal. To reach the ambitious emission reduction goals set by both companies and governments, datacenters need to reduce emissions throughout their operations, including (and particularly relevant for this thesis) the storage system. Unfortunately, while data storage and retrieval systems are large contributors to embodied emissions, reducing their embodied emissions have largely been overlooked.

This thesis aims to address reducing emissions in data retrieval for large-scale storage systems. These storage systems can reduce their carbon footprint by enabling storage devices to have longer lifetimes and use denser media. However, storage hardware's IO limits combined with software's unnecessary additional IO often severely restrict emission reductions, or at worse cause increased emissions. Thus, this thesis focuses on reducing IO in several parts of the storage stack to enable efficient and sustainable data retrieval.

First, this proposal addresses the efficiency and sustainability of flash caching, a critical layer in datacenter storage systems that is limited by flash write endurance. This improvement will be realized in two caching systems: Kangaroo and FairyWREN. Together, these caches dramatically reduce writes by over 28x, allowing flash devices to use denser flash for longer lifetimes, ultimately reducing emissions. Then, this thesis will discuss proposed work to enable more sustainable bulk storage, where bandwidth limitations prevent deployment of denser HDDs. I propose a new IO interface, Declarative IO, that empowers the storage system to eliminate duplicate IO accesses through exposing the time- and order-flexibility in maintenance tasks. This work will enable using larger HDDs, further reducing emissions from storage systems.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Today, datacenter emissions are on par with the aviation industry [56]. In the next decade, datacenters will account over 20% of the world's carbon emissions [48]. By 2050 without targeted intervention, datacenters will account for up to 33% of the world's carbon emissions [56]. To avoid this catastrophic outcome, companies and regulators [75] are increasingly focusing on making datacenters more sustainable. For example, in the next few decades, many companies — including Amazon [1], Google [2], Meta [9], Microsoft [74] — are looking to achieve Net Zero, i.e., greenhouse gas emissions close to zero.

To reduce emissions, many datacenters are adopting renewable energy sources such as solar and wind [9, 38, 66, 74]. Google, AWS, and Microsoft are expected to complete their renewable-energy transition by 2030 [29, 49, 61]. However, this switch in energy source does not reduce datacenters' *embodied emissions*, the emissions produced by the manufacture, transport, and disposal of datacenter components. Embodied emissions will account for more than 80% of datacenter emissions once datacenters move to renewable energy [38]. To continue reducing datacenters' carbon footprint, their embodied emissions need to be reduced.

Unfortunately, prior work has focused on compute's impact on carbon emissions [12, 39, 94], which makes sense for datacenter emissions dominated by operational emissions. However, even on compute servers, 46-80% of the server's embodied carbon comes from storage [66, 93]. This figure ignores storage servers, which hold the bulk of data. By 2025, 175 zettabytes of data will be generated annually and the vast majority will be stored in datacenters [83]. Even today, thousands of servers and 100Ks of devices are needed to cache and store this data, which is essential to applications from machine learning to social media to communication. To achieve the goal of dramatically reducing future emissions, we need to expand sustainability research to also focus on storage.

Thus, this thesis will address enabling more efficient and sustainable storage and retrieval of data in datacenters. Without changing how devices are manufactured, storage systems have two main strategies to reduce embodied emissions: (1) extend the lifetime of devices and (2) reduce the quantity of hardware, often through using more dense storage technology. However, these solutions can be challenging to implement without reducing performance or increasing the likelihood of data loss, a non-starter for many workloads.

Fortunately, many of these strategies are limited due to software-induced IO bottlenecks. Workloads often perform extraneous IO because today's storage interfaces do not allow for consolidation of IO. Thus, the goal of this work is to eliminate IO, without reducing other performance metrics, in order to increase measurably cutting carbon emissions. This thesis specifically focuses on two IO-bottlenecks: large-scale flash caches and HDDs underlying the distributed storage system. Together this work will support the thesis that:

> **Reducing IO through adopting more expressive and symbiotic interfaces and eliminating overlapping IO allows datacenters to achieve more efficient and sustainable data-retrieval systems.**

This thesis will support this proposition through three projects:

Kangaroo: Efficient caching of tiny objects in flash [70, 71]: Tiny objects (≈100 bytes) are prevalent in many workloads including social graphs and sensor data, but their size creates challenges: previously requiring either massive amounts of DRAM or generating many extra writes that lead to flash wear out. Kangaroo is a flash cache tailored for applications with tiny objects, enabling reductions in either writes or misses. These write reductions enable flash caching without massively over-provisioning on flash devices currently deployed in data centers.

FairyWREN: Emerging interfaces allow for sustainable flash caching: To truly be sustainable, flash caches cannot just efficiently use current flash devices, but they also enable longer lifetimes on more dense flash. Both of these goals will result in caches having fewer writes per year. Unfortunately, even highly optimized caches like Kangaroo cannot accommodate fewer writes because most remaining writes in these designs come from the device as a result of flash's traditional interface. Thus, I propose a class of interfaces, *Write-Read-Erase iNterfaces (WREN)*, that expose fundamental flash properties to the application, allowing caches to control *all* writes. Then, I present a flash cache, *FairyWREN*, that leverages WREN to reduce writes by 12.5x and emissions by 33% over Kangaroo.

Declarative IO: Exposing flexibility in distributed storage IO (Proposed): Each generation of more-dense HDDs accommodate less bandwidth (MB/s) per terabyte (TB) of capacity. Unfortunately, current storage system designs necessitate inefficiency due to their historic Imperative IO interface, where applications request reads and writes for blocks as the application needs them. This interface increasingly forces the deployment of more devices than capacity would dictate and threatening the adoption of carbon-saving new storage technologies. With this project, I introduce the *Declarative IO Interface*, which re-imagines storage interfaces to expose order- and time-flexibility in IO, allowing the combination of independent IO activities. Ultimately, this interface, Declarative IO, will lower the bandwidth-per-TB requirements of distributed storage systems enabling a reduction in the number of storage devices required. Additionally, it will enable deployment of new storage technologies that are more carbon-efficient per-TB.

Outline for remainder of proposal: The rest of this proposal is split into 4 sections: chapter 2 provides necessary background and motivation for the three projects in this thesis; chapter 3 discusses the completed work on flash caching (including both Kangaroo, Sec. 3.1, and FairyWREN, Sec. 3.2); chapter 4 presents my proposed work on Declarative IO; and chapter 5 proposes a timeline for completing the thesis.

2 Background and Motivation

This chapter describes the background and motivation for the projects in this proposal, leading into a discussion of the opportunities and challenges of increasing efficiency and sustainability of data retrieval in data centers.

2.1 Data retrieval in the datacenter

Data centers deploy a complex, interconnected collection of services to manage its data, including caching, bulk storage, and data management services. Generally, requests from applications will first encounter layers of caches, then encounter a data management service (ie. data lakehouse or database), which will then lead to more caches and eventually bulk storage.

Caching layers. Caches exist throughout the datacenter to minimize the latency of services and reduce the load on backend services, including data management services and bulk storage. Fig. 2.1 shows some of the services at Meta and the location of caches in their datacenters. Caches farther from user requests need to be large in order to effectively cache requests since much of the locality in requests is removed by earlier caching stages. To accommodate these large caches, many datacenters deploy flash caching as their last layer of caching before going to storage [16, 21, 22, 33, 102].

Data Management Services. Data management services exist to organize data and make it accessible to applications. These services include table stores, data warehouses, and data lakes. In addition to sending IO to bulk storage based on application requests, these services also send IO to bulk storage to manage their data. For example, many data management services deploy log-structured merge trees [6, 8, 80, 81, 97], which require compacting their data repeatedly to ensure their performance and to achieve reasonable space utilization.

Bulk Storage. Data is stored for long-term retention in bulk storage. Bulk storage consists primarily of 100Ks of HDDs [35, 78, 88, 98], which are together combined into a single storage system. The bulk storage system generally stores large blocks of data spread across many devices and servers, using a metadata service to track where data resides and any other metadata about the blocks. Since bulk storage needs to retain data until it is deleted (usually years later if it is ever deleted), it also runs many maintenance services, from scrubbing [45, 76, 82] to reconstruction [27, 37] to load balancing to transcoding [52, 53]. These maintenance tasks result in IO to the hard drives.

2.2 Opportunity for sustainable storage

This section will discuss the opportunities to achieve more sustainable storage: from using less carbon-intensive hardware to deploying denser flash for longer, to using the denser hard drives.

2.2.1 Flash and hard-drives are less carbon-intense than DRAM

DRAM often makes up 40% to 50% of server cost [60, 87, 91]. DRAM also has a large embodied carbon footprint (46% of a server in Azure [66]) and has large operational emissions due to requiring up to half of system power [39].

Flash is cheaper per-bit than DRAM, embodies 12× less carbon, and requires less power per-bit [39]. Thus, datacenters should use flash over DRAM whenever possible [36], even for

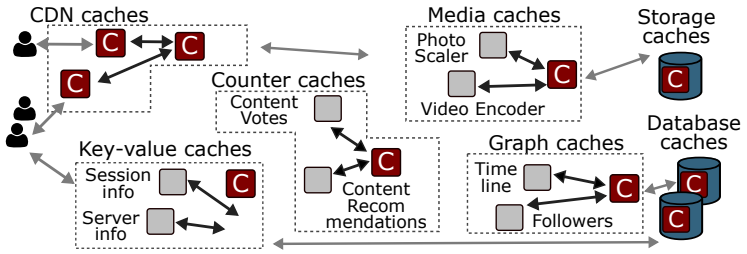


Figure 2.1: Overview of caching at Meta. Caching is deployed in many different services across Meta from the CDN all the way back to storage. [16]

traditionally DRAM workloads, such as caching [16, 33, 70, 71] or machine learning [101].

HDDs are even less carbon-intensive, ≈3-10x less embodied carbon per bit than flash [39, 93]. This means that hard drives are going to be an essential part of a sustainable storage stack, particularly more dense drives that have similar hardware manufacturing requirements.

2.2.2 Denser and longer lifetimes for Flash SSDs

Flash devices have two potential ways to reduce emissions: deploying denser flash and keeping that flash in datacenters for longer.

Flash caches should use denser flash where possible to reduce emissions. Flash is becoming more dense, moving from *single-level cells (SLC)*, which store 1 bit/cell, to *tri-level cells (TLC)*, which store 3 bits/cell. Flash SSDs will soon use *quad-level cells (QLC)* (4 bits/cell) and *penta-level cells (PLC)* (5 bits/cell) [79]. The denser the flash, the more it reduces embodied emissions, since more bits are packed onto roughly the same silicon.

Lengthening device lifetime is an effective way to significantly improve datacenter sustainability. Traditionally, datacenter hardware replacement cycles have been around 3 years [66] due to the rate of improvement in hardware performance and power-efficiency. Today, datacenters deploy devices for longer. Longer replacement cycles have become common due to their cost advantages and the slowing of Moore's Law. For example, Microsoft Azure increased the depreciable lifetime of servers from 4 to 6 years [43, 67], and Meta recently started planning for servers to last 5.5 years [12]. Additionally, hyperscalers are finding that servers do not fail quickly: failure rates at Azure have little evidence of increasing before 8 years [18, 66]. Moving to longer lifetimes amortizes embodied carbon.

2.2.3 Denser and fewer HDDs

Technology advancements in hard drives are continuing to increase their density. Shingled magnetic recording (SMR) HDDs have reached over 20 TB [42] and the first heat assisted magnetic recording (HAMR) HDDs are just starting to reach the market with capacities over 30 TB [86]. These denser HDDs will amortize their embodied emissions over more bits, decreasing bulk storage's carbon footprint. Additionally, fewer hard drives mean fewer servers, fewer racks, and less networking needed to store the same amount of data, further lowering datacenter embodied carbon.

2.3 IO limites storage's sustainability

Unfortunately, these improvements in flash and hard drive emissions-per-bit come with drawbacks in the form of decreased IO. Deploying denser flash and increasing its lifetime both lower the

number of writes that flash can withstand without wearing out. Denser hard drives cannot increase their bandwidth to keep up with their capacity. Without addressing these IO limitations, at best, these technologies cannot be deployed, at worse, they can increase emissions.

2.3.1 Write endurance limits flash caches

Flash SSDs have limited write endurance and are warrantied only for a stated write budget [10]. Exceeding this write budget can cause the device to become read-only or fail. Flash caching, in particular, can easily overcome this write budget. Hence, while flash caching presents carbon-saving opportunities from minimizing DRAM and using dense flash for longer, caches must find a way to severely limit the amount they write.

Prior flash caching architectures require carbon-intensive tradeoffs. One prevalent workload in flash caching is tiny objects (O(100) bytes). These objects are uniquely challenging for flash caches because the objects are orders of magnitude smaller than the write size of flash, causing either write endurance problems or requiring unreasonable memory overheads. While many flash caches avoid small objects [84, 92], there are two main, prior approaches log-structured caches [25, 33, 57, 81] and set-associative cache [16].

Log-structured caches are the default choice for many flash-based designs because they minimize writes. These caches write all objects sequentially, buffering objects in memory before writing them and thus achieving close to the minimum writes. Unfortunately, these caches need a full-index to quickly find objects which for a 2 TB cache with 100 bytes objects, with the lowest published overhead at 30 bits/obj [33], still requires more than 70 GB of DRAM just for the index. Using lots of memory defeats the sustainability benefits of using flash.

To decrease memory, tiny object caches often are set-associative. Set-associative caches, similar to those in CPU caching, map objects to fixed size locations, *sets*, on flash using a hash of the object's key. These caches require very little metadata overhead which makes them appealing where DRAM capacity is limited. However, they suffer from high *application-level write amplification* (ALWA): flash requires caches to write at least 4 KB at one time so for a 100 byte object, the cache needs to write at least 40x more data than needed. These additional write tear through flash's limited write endurance, leading to wearout. Thus, no existing cache designs either require too many writes or too much memory, reducing the efficiency of flash caching.

Flash device architecture causes extra writes. To further complicate sustainable caching, flash devices also add writes. Flash devices cannot write new values without first erasing a large region of the device. To support random writes, devices must read all live data in a region, erase the region, and then write the live data back to the drive along with any new data. As a result, flash SSDs perform more writes than requested by the application. This is captured by the *device-level write amplification* (DLWA) factor [23, 33, 41, 57, 59, 64, 92], the relative increase in bytes actually written to flash vs. bytes written by an application. (If an SSD writes 3GB to serve 1GB of application writes, then DLWA is 3×.) DLWA can be large: a factor of 2 to 10× is common [70]. DLWA causes write-intensive applications to quickly wear out flash devices, increasing their replacement frequency and increasing embodied emissions.

DLWA is primarily caused by the physical limitations of flash storage. Flash devices are organized in a physical hierarchy (Fig. 2.2). The smallest unit is the *page*, usually 4 KB. Flash can be written at page granularity, but a page must be erased before it can be rewritten. To avoid electrical interference during erasure, pages are grouped into *flash blocks* [13, 19, 20, 41, 65]. A flash block is the minimum erase size. In practice, however, flash drives stripe writes across blocks to improve bandwidth and error correction. Striping increases the effective *erase unit (EU)* size

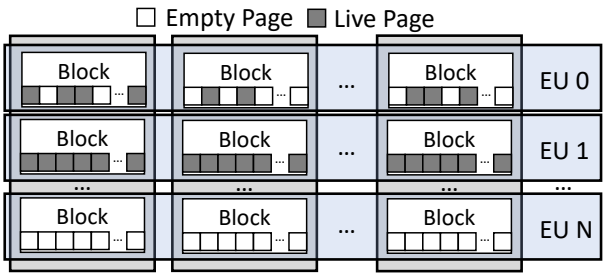


Figure 2.2: Flash architecture. The internal arrangement of flash devices into planes, blocks, pages, and EUs. Each EU has blocks in multiple pages. EU 0 is a partially full, EU 1 is entirely full, and EU N has just been erased.

to gigabytes [20].

The mismatch between the granularity of writes and erases is the root cause of DLWA. To maintain the 4 KB read/write block interface, flash devices garbage collect (GC), moving live pages from partially empty EUs (such as EU 0 in Fig. 2.2) to a writable EU (such as EU N) before erasing the EU and freeing dead pages. The less the available capacity on the device, the more frequently it has to GC, introducing a tradeoff between flash utilization and flash writes.

One might hope that, over time, technological advances would allow for smaller EU sizes, closing the gap between write and erase granularities. However, *flash EU sizes have actually gotten larger over time as flash has gotten denser* because denser flash is more susceptible to electrical disturbances during erase. Effective block sizes on an SLC flash device were 128 KB[95], MLC and TLC flash devices had block sizes of around 20 MB [89], and QLC devices will have 48 MB blocks [90]. Striping these blocks with hundreds of 3D-stacked layers [90] quickly results EUs in the gigabyte range as seen in practice [20, 72].

Although applications cannot traditionally control these device writes, caches need to control these writes in order to achieve sustainable flash caches. Furthermore, the *write amplification* of the entire cache is the multiple of ALWA and DLWA. Flash caches need to address both ALWA and DLWA to maximize sustainability.

Lowering carbon-intensity lowers write rate. To make the write problem worse, as flash becomes denser and its lifetime increases, its write endurance drops significantly.

Denser flash has less write endurance. For example, while PLC flash is up to 40% more dense than TLC, PLC is forecast to have only *16%* of TLC's writes [7]. Additionally, because denser flash has to differentiate between more voltage levels, even small voltage changes can make data unreadable. TLC uses two-phase writes and more frequent refresh to prevent data loss [73]. Two-phase writes require the device to have enough RAM and capacitance to remember all in-flight writes, limiting the number of EUs that can be "active" (i.e., writable) at any point in time, often to less than ten. Writing to more EUs than this limit requires closing an active EUs, causing internal device writes, creating a further restriction for flash caching.

As flash has longer expected lifetimes, its write budget is spread over a longer period of time. This reduces the write rate that an application can sustain without wearing out the flash before the desired lifetime.

Fig. 2.3 models how write rate affects both emissions when varying lifetimes and flash density. Each line shows a device of a different lifetime, and shaded regions show which flash density is best for a given write rate. The model calculates how much capacity must be provisioned

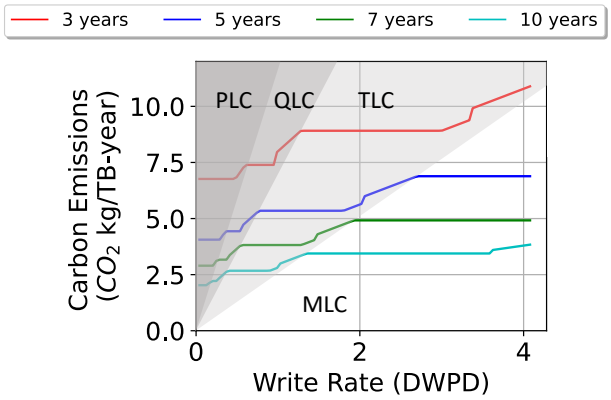


Figure 2.3: Flash carbon emissions vs write rate. The annual carbon emissions of flash depending on the required average write rate and desired lifetime. Lifetime has a much larger impact on emissions than density, but both are important to lower emissions.

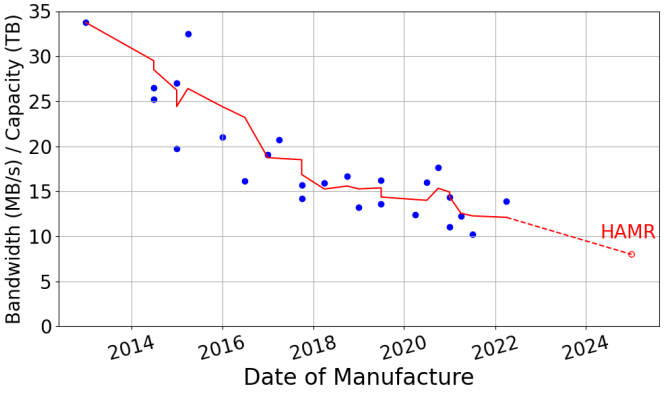


Figure 2.4: HDDs bandwidth-per-TB is dropping. HDD Sustained Bandwidth-per-TB trend over years [11]. The red empty circle denotes the speculated bandwidth/TB cost after the introduction of future disk technologies like HAMR [3, 4, 5].

for each technology to achieve the desired lifetime at a given write rate. Device lifetime is the most important factor in reducing carbon emissions. Moreover, denser flash has the potential to improve sustainability, but only if flash write rate is very small — much less than one device-write per day. However, if flash applications use dense flash, but cannot reduce their write rate enough for a long lifetime, then their carbon emissions increase over using a less dense flash device for a longer lifetime.

2.3.2 Bandwidth-per-capacity wall limits HDDs

Although hard drives are not currently write-limited like flash, they are quickly running into their own IO limitation. For the past 10 years as hard drives have increased their density, their bandwidth has not increased at the same pace. Fig. 2.4 shows the decrease in bandwidth-per-TB, a ∼8.5% reduction per year. HDDs are already running into bandwidth limitations in bulk storage. Therefore, deploying HDDs that have lower bandwidth is not possible in many datacenters today, preventing datacenters from realizing the carbon-savings of these denser drives unless bulk storage can decrease its IO requirements.

3 Completed Work

This chapter discusses the two completed projects, both of which address enabling sustainable flash caching. Sec. 3.1 discusses Kangaroo, a system that removes extraneous application-level writes in the caching architecture of flash caches for tiny objects. Sec. 3.2 introduces FairyWREN, a flash caching system that leverages emerging flash interfaces to remove previously unreachable device-level writes. Ultimately, these systems together enable flash caching to use denser flash devices for longer, lowering emissions.

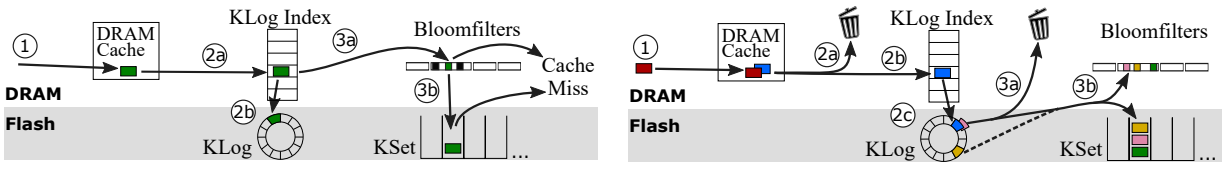
3.1 Kangaroo: Efficiently caching tiny objects in flash

Many social-media and IoT services have very large working sets consisting of billions of tiny (≈100 B) objects. Large, flash-based caches are important to serving these working sets at acceptable cost. However, caching tiny objects on flash is challenging for two reasons: *(i)* SSDs can read/write data only in multi-KB "pages" that are much larger than a single object, stressing the limited number of times flash can be written; and *(ii)* very few bits per cached object can be kept in DRAM without losing flash's cost advantage. Unfortunately, existing flash-cache designs fall short of addressing these challenges: write-optimized designs require too much DRAM, and DRAM-optimized designs require too many flash writes.

Kangaroo is a new flash-cache design that aims to achieve maximum efficiency — optimizing both DRAM usage and flash writes to maximize cache performance while minimizing cost. Kangaroo combines a large, set-associative cache with a small, log-structured cache. The set-associative cache requires minimal DRAM, while the log-structured cache minimizes Kangaroo's flash writes. Experiments using traces from Facebook and Twitter show that Kangaroo achieves DRAM usage close to the best prior DRAM-optimized design, flash writes close to the best prior write-optimized design, and miss ratios better than both. These results are corroborated with a test deployment of Kangaroo in a production flash cache at Facebook.

3.1.1 Kangaroo design overview

Kangaroo is a new flash-cache design optimized for billions of tiny objects. Kangaroo aims to maximize hit ratio while minimizing DRAM usage and flash writes. Like some key-value stores [26, 62, 68], Kangaroo adopts a hierarchical design, split across memory and flash. Fig. 3.1 depicts the two layers in Kangaroo's design: *(i)* KLog, a log-structured flash cache and *(ii)* KSet, a set-associative flash cache; as well as a DRAM cache that sits in front of Kangaroo.



(a) A lookup in Kangaroo.

(b) Insertion and eviction in Kangaroo.

Figure 3.1: Overview of Kangaroo. Objects first go to a tiny DRAM cache; then KLog, a small on-flash log-structured cache with an in-DRAM index; and finally KSet, a large on-flash set-associative cache. KLog minimizes flash writes, and KSet minimizes DRAM usage.

Basic operation. Kangaroo is split across DRAM and flash. As shown in Fig. 3.1a, ① lookups first check the DRAM cache, which is very small (<1% of capacity). ② If the requested key is not found, requests next check KLog (≈5% of capacity). KLog maintains a DRAM index to track objects stored in a circular log on flash. ③ If the key is not found in KLog's index, requests check KSet (≈95% of capacity). KSet has no DRAM index; instead, Kangaroo hashes the requested key to find the set (i.e., the LBA(s) on flash) that might hold the object. ③a If the requested key is not in the small, per-set Bloom filter, the request is a miss. Otherwise, the object is probably on flash, so ③b the request reads the LBA(s) for the given set and scans for the requested key.

Insertions follow a similar procedure to reads, as shown in Fig. 3.1b. ① Newly inserted items are first written to the DRAM cache. This likely pushes some objects out of the DRAM cache, where they are either ②a dropped by KLog's pre-flash admission policy or ②b added to KLog's DRAM index and ②c appended to KLog's flash log (after buffering in DRAM to batch many insertions into a single flash write). Likewise, inserting objects to KLog will push other objects out of KLog, which are either ③a dropped by another admission policy or ③b inserted into KSet. Insertions to KSet operate somewhat differently than in a conventional cache. For any object moved from KLog to KSet, Kangaroo moves *all objects in KLog that map to the same set* to KSet, no matter where they are in the log. Doing this amortizes flash writes in KSet, significantly reducing Kangaroo's ALWA.

Design rationale. Kangaroo relies on its complementary layers for its efficiency and performance. At a high level, *KSet minimizes DRAM usage* and *KLog minimizes flash writes*. Like prior set-associative caches, KSet eliminates the DRAM index by hashing objects' keys to restrict their possible locations on flash. But KSet alone suffers too much write amplification, as every tiny object writes a full 4 KB page when admitted. KLog comes to the rescue, serving as a write-efficient staging area in front of KSet, which Kangaroo uses to amortize KSet's writes.

On top of this basic design, Kangaroo introduces three techniques to minimize DRAM usage, minimize flash writes, and reduce cache misses. *(i)* Kangaroo's *partitioned index* for KLog can efficiently find all objects in KLog mapping to the same set in KSet, and is split into many independent partitions to minimize DRAM usage. *(ii)* Kangaroo's *threshold admission* policy between KLog and KSet only admits objects to KSet when at least n objects in KLog map to the same set, reducing ALWA by $\geq n\times$. *(iii)* Kangaroo's *"RRIParoo" eviction* improves hit ratio in KSet by approximating RRIP [46], a state-of-the-art eviction policy, while only using a single bit of DRAM per object.

Theoretical foundations. We develop a Markov model of Kangaroo's basic design, including threshold admission. This model rigorously demonstrates that Kangaroo can greatly reduce ALWA compared to a set-only design, without any increase in miss ratio. (In fact, RRIParoo, which is not modeled, significantly improves miss ratio with negligible impact on ALWA.)

Formally, we assume the commonly used *independent reference model* [14, 17, 28, 30, 34, 51], in which objects are referenced independently with fixed probability per object. However, we make no assumptions about the object popularity distribution, so Theorem 1 holds across any popularity distribution (uniform, Zipfian, etc.). Suppose that KLog contains q objects; KSet contains s sets with w objects each; objects are admitted to flash with a p probability; and objects are only admitted to KSet if at least n new objects are being inserted.

Theorem 1. *Kangaroo' app-level write amplification is*

$$\text{ALWA}_{Kangaroo} = p\left(1 + \frac{w\,\overline{F}_X(n)}{\overline{F}_X(1)\mathbb{E}\left[X|X \geq n\right]}\right), \tag{3.1}$$

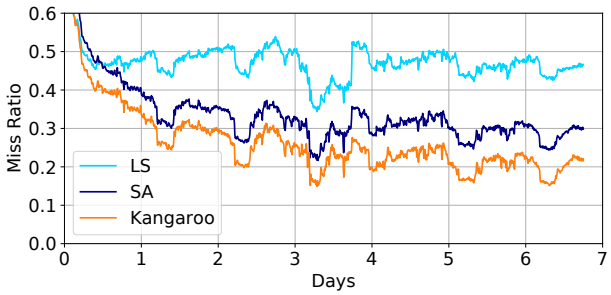


Figure 3.2: Kangaroo reduces misses. Miss ratio for all three systems over a 7-day Facebook trace. All systems are run with 16 GB DRAM, a 1.9 TB drive, and with write rates less than 62.5 MB/s.

where $X \sim Binomial(q, 1/s)$ *and* $\overline{F}_X(n) = \sum_{i=n}^{\infty} \mathbb{P}[X = i]$ *is the probability of a set being rewritten. Furthermore, the probability of admitting an object to KSet is* $\mathbb{P}[X \geq n | X \geq 1]$.

For example, a reasonable parameterization of Kangaroo on a 2 TB drive with 5% of flash dedicated to KLog is $q = 5 \cdot 10^8$, $s = 4.6 \cdot 10^8$, $w = 40$, $p = 1$, and $n = 2$, which results in $\text{ALWA}_{\text{Kangaroo}} \approx 5.8$. In contrast, a set-associative cache of the same size and admission probability, $\mathbb{P}[X \geq n | X \geq 1] \approx 0.45$, gets $\text{ALWA}_{\text{Sets}} = w \cdot 0.45 = 17.9\times$. That is, Kangaroo improves ALWA by $\approx 3.08\times$, a large decrease in ALWA with only a small percentage of flash dedicated to KLog.

3.1.2 Evaluation

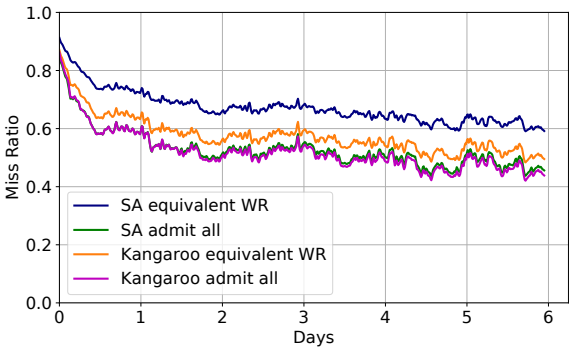
This section presents a selection of experimental results from Kangaroo and prior systems. In the full results, we find that: *(i)* Kangaroo reduces misses by 29% under realistic system constraints. *(ii)* Kangaroo improves the Pareto frontier when varying constraints. *(iii)* In a production deployment, Kangaroo reduces flash-cache misses by 18% at equal write rate and reduces write rate by 38% at equal miss ratios.

Experimental Setup We run experiments on two 16-core Intel Xeon CPU E5-2698 servers running Ubuntu 18.04, one with 64 GB of DRAM and one with 128 GB of DRAM. We use Western Digital SN840 drives with 1.92 TB rated at three device-writes per day. This gives a sustained write budget of 62.5 MB/s. We chose these configurations to be similar to those deployed in the large-scale production clusters that contributed traces to this work, but with extra DRAM to let us explore large log-structured caches.

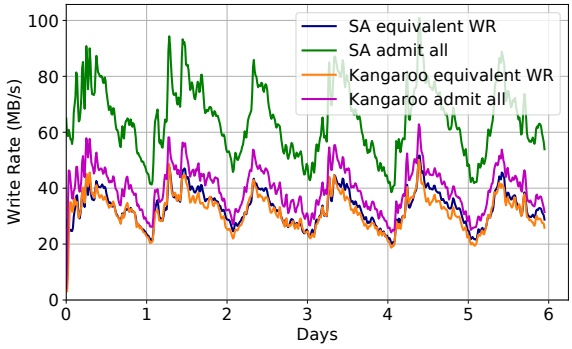
We compare Kangaroo, implemented in Cachelib [16], to *(i)* CacheLib's small object cache (SA), a set-associative design that currently serves the Facebook Social Graph [24] in production; and *(ii)* an optimistic version of a log-structured cache (LS) with a full DRAM index. For LS, we configure KLog to index the entire flash device and use FIFO eviction. Our experiments use sampled 7-day traces from Facebook [16].

Kangaroo significantly reduces misses vs. prior cache designs under realistic constraints. Kangaroo aims to achieve low miss ratios for tiny objects within constraints on flash-device write rate, DRAM capacity, and request throughput.

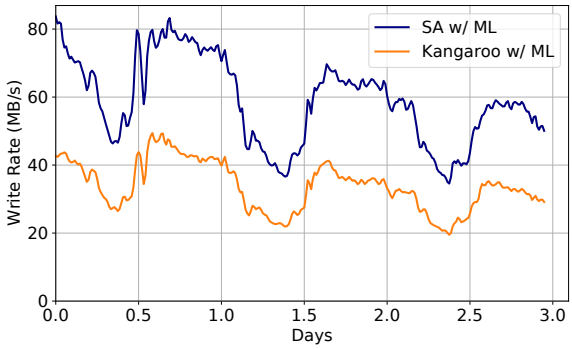
Fig. 3.2 shows that Kangaroo reduces cache misses by 29% vs. SA and by 56% vs. LS. This is because *Kangaroo makes efficient use of both limited DRAM and flash writes,* whereas prior designs are hampered by one or the other. Specifically, SA is limited primarily by its high write rate, which forces it to admit a lower percentage of objects to flash and to over-provision flash to reduce device-level write amplification. SA uses only 81% of flash capacity. Similarly, LS is



(a) Flash miss ratio.



(b) Application flash write rate.



(c) ML admission.

Figure 3.3: Results from production deployment of Kangaroo. Kangaroo had two test deployments to compare it against SA. These results show the (a) flash miss ratio and (b) application flash write rate over time using pre-flash random admission and (c) application flash write rate over time using ML admission. With random admission at equivalent write-rate, Kangaroo reduces misses by 18% over SA. When Kangaroo and SA admit all objects, Kangaroo reduces write rate by 38%. With ML admission, Kangaroo reduces the write rate by 42.5%.

limited by the reach of its DRAM index. LS warms up as quickly as Kangaroo until it runs out of indexable flash capacity at 61% of device capacity, even though we provision LS extra DRAM for both an index and DRAM cache. By contrast, Kangaroo uses 93% of flash capacity, increasing cache size by 15% vs. SA and by 52% vs. LS. On top of its larger cache size, Kangaroo's has lower ALWA than SA and its RRIParoo policy makes better use of cache space [70, 71]. Kangaroo also does not significantly impact the flash cache's maximum throughput or tail latency.

Production shadow deployment. Kangaroo was deployed for two production tests on a small-object workload at Facebook, comparing Kangaroo to their production tiny-object flash cache, SA. Each cache receives the same request stream as production servers but does not respond to users. Due to limitations in the production setup, we can only present application-level write rate (i.e., not device-level) and flash miss ratio (i.e., for requests that miss in the DRAM cache). In addition, both systems use the same cache size (i.e., Kangaroo does not benefit from reduced over-provisioning).

To find appropriate production configurations, we chose seven configurations for each system that performed well in simulation: four with probabilistic pre-flash admission and three with a machine-learning (ML) pre-flash admission policy. The first production deployment ran all

probabilistic admission configurations and the second ran all ML admission configurations. Since these configurations ran under different request streams, their results are not directly comparable.

Fig. 3.3a and Fig. 3.3b present results over a six-day request stream for configurations with similar write rates ("equivalent WA") as well as configurations that admit all objects to the flash cache ("admit-all"). Kangaroo reduces misses by 18% vs. SOC in the equivalent-WA configurations, which both have similar write rates at ≈33 MB/s. The admit-all configurations achieve the best miss ratio for each system at the cost of additional flash writes. Here, Kangaroo reduces flash misses by 3% vs. SA while writing 38% less.

We also tested both systems with the ML pre-flash admission policy that Facebook uses in production [16]. Fig. 3.3c presents results over a three-day request stream. The trends are the same: Kangaroo reduces application flash writes by 42.5% while achieving a similar miss ratio to SA. Kangaroo thus significantly outperforms SA, independent of pre-flash admission policy. This write result is particularly important for sustainability: *even under the same deployment capacity, Kangaroo reduces writes while slightly improving its miss ratio over the prior production state-of-the-art tiny-object flash cache.*

3.2 FairyWren: Sustainable caching using emerging flash interfaces

To achieve more sustainable flash caching, flash caches need to be able to adopt more dense flash and longer lifetimes. Even highly-optimized caches like Kangaroo struggle to handle the resulting write rate decrease that both of these trends will cause. To overcome these write limitations, caches need to lower writes without resorting to DRAM overheads or performance drops.

Unfortunately, a large percentage of writes comes from prior caches inability to directly manage DLWA, a consequence of flash's current interface, *Logical-Block-Addressable Devices (LBAD)*. This interface only exposes an interface of page-sized reads and writes, leaving the device to manage the garbage collection caused from the granularity difference between pages and erase units. This garbage collection leads to uncontrollable writes, particularly bad in caches with their high write rates and ability to choose which objects to keep.

Thus, to overcome these write limitations to achieve more sustainable caches, flash caches need a different interface. FairyWren introduces the classification of Write-Read-Erase iNterface (WREN), to captures the necessary and sufficient operations for flash caches. However, WREN does not immediately ameliorate the write problems with flash caches, it mainly gives the cache control of all writes. FairyWREN, a flash cache built for WREN, harnesses this control to reduce writes. Building off Kangaroo, FairyWREN combines previous device-controlled garbage collection and the flush-eviction decisions of a hybrid flash cache into one operation called *nesting*, removing what used to be duplicate writes between the cache and device layers. In addition, its physical separation between data lifetimes of KLog's and KSets' components allow it to completely remove interference that causes extra writes between the two. With several other optimizations, FairyWREN is able to decrease writes 12.5x over Kangaroo, reducing carbon emissions by 33%.

3.2.1 Write-Read-Erase iNterfaces (WREN)

Prior flash caches incur excessive DLWA from the mismatch between write and erase granularities and a legacy LBAD interface that hides this from software. Although there has been work to decrease DLWA in the context of LBAD [40, 41, 44, 58, 97, 99], LBAD devices fundamentally hide erase units and garbage collection (GC) from applications, preventing co-optimization to minimize overall flash writes. Recent *Write-Read-Erase iNterfaces (WREN)*, such as ZNS [20] and FDP[69],

that include `Erase` as a first-order operation and let software control GC.

Challenges of new interface design. While a variety of flash interfaces have been proposed [19, 44, 54, 55, 77, 85, 96, 103], none have gained widespread adoption. Two proposals, Multi-streamed SSDs and Open-Channel SSDs, illustrate the pitfalls of designing a new flash interface.

Multi-streamed SSDs [54, 55] allow users to direct writes to different *streams*. Streams provide isolation between workloads: different streams write to different EUs. When objects with similar lifetimes are grouped into the same stream, GC is more efficient. However, because the application does not control GC directly, DLWA remains a significant issue on these devices. Open-Channel SSDs [19] remove all flash-device logic and force applications to handle *all* of flash's complexities. While the hope was to develop layers of abstraction in software to hide some of this complexity, this software was never widely deployed.

An ideal flash interface for caching would allow the cache to control all *writes, including GC, but still present a simple abstraction to application developers.*

What makes an interface WREN? We call interfaces that delegate `Erase` commands and garbage collection to the host *Write-Read-Erase iNterfaces (WREN)*. WREN is defined by three main features:

1) WREN operations. WREN devices must let applications control which EU their data is placed in, and when that EU is erased. Specifically, WREN devices must, at least, have `Write`, `Read`, and `Erase` operations.

These operations can be implemented differently. For example, *Zoned Namespaces* (ZNS)[20] and *Flexible Data Placement* (FDP)[69] both are WREN. Both interfaces are NVMe standards with strong support from industry and provide an abstraction for writing to an EU[1]. However, they have different philosophies, which can be seen, for instance, in their `Write` operations. ZNS provides either sequential writes to an EU or nameless writes[103] through Zone Append. FDP provides random writes within an EU as long as the application tracks that the number of pages written is less than the EU size. Despite these differences, both provide the control over data placement into EUs required by WREN.

Moreover, the aforementioned Open-Channel interface is also WREN. But Open-Channel SSDs expose the full complexity of the device to the host, which is additional complexity *not* required to reduce a cache's DLWA.

2) The `Erase` requirement. Unlike LBAD, WREN devices do not move live data from an EU before erasing it. Applications are responsible for implementing GC to track and move live data before calling `Erase`. `Erase` is different from a traditional `trim` because `Erase` targets an entire EU rather than individual pages. Failure to perform correct and timely GC is subject to implementation-specific error handling by the device. A major difference between FDP and ZNS is how they treat violations of `Erase` semantics, but this error behavior is inessential to reducing DLWA and thus outside of WREN.

3) Multiple, but limited active EUs. An *active EU* is one that can be written to without being erased. WREN devices support multiple, but limited active EUs at one time. Since an active EU often is a device buffer storing the active EU's data, the maximum number of active EUs is implementation-specific. FairyWREN requires four simultaneous active EUs, which we expect will be supported in the vast majority of WREN devices.

[1]This abstraction is called a *zone* in ZNS and a *reclaim unit* in FDP.

WREN alone is not a cure for WA. WREN devices make it easy to perform large, sequential writes with no WA. If all writes are large and sequential, it is generally easy to find an EU consisting of invalid data when GC is required, resulting in low WA. For sustainable caching, we also need low WA for set-associative caches' small random writes, which incur high DLWA on LBAD devices. One might hope that WREN devices can achieve lower WA. A reasonable first attempt at implementing a set-associative cache on WREN is to treat each set as an object in a log-structured store, allowing the cache to write set updates sequentially to a single active EU. We find that this naive approach does not significantly reduce WA because it just moves the GC from within the device to within the cache.

The impact of smaller EUs. One idea for mitigating WA under small, random writes is to reduce the EU size from a GB to tens of MB by removing error correction between flash blocks, a tolerable loss for caches. Prior literature uses smaller EUs to minimize GC [15, 72] because, intuitively, lowering the number of sets per EU creates more EUs that are either mostly invalid (good candidates for GC) or mostly valid (bad candidates for GC that are skipped).

However, prior work analyzing the WA of FIFO GC policies[32, 47] has largely ignored the effect of EU size. In fact, this modeling work assumes that changing the EU size will not change the WA from GC. To remedy this discrepancy, we model the WA of a FIFO GC policy for a set-associative cache, capturing the effect of EU size.

Following the approach of [47], we approximate the distribution of the number of live pages in the EU at the tail of a log-structured store. Our approximation shows that when EU sizes are smaller, FIFO is more likely to find EUs that are mostly invalid or completely valid. To quantify this effect, we approximate the long-run average WA under FIFO. Our approximation (Fig. 3.4) matches simulation results, with a R^2 value of 0.9996.

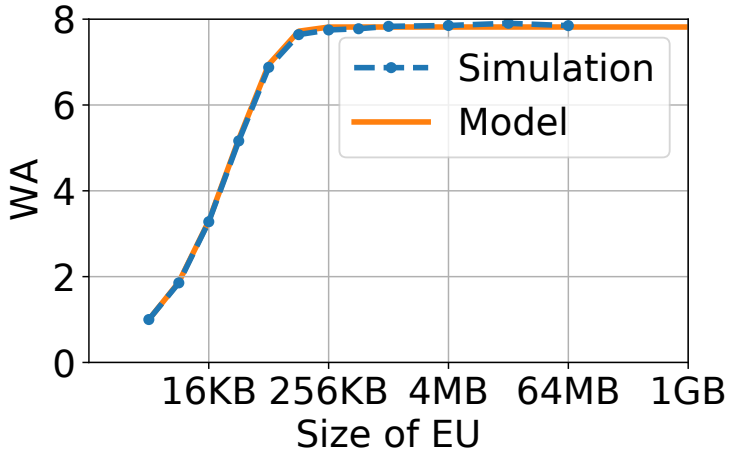


Figure 3.4: DLWA for random write workloads. The DLWA for a set-associative cache running on WREN with 7% overprovisioning. EUs have to be less than 128 KB to significantly reduce DLWA.

We find that *reducing EU size only improves* WA *for very small EU sizes.* To realize a significant reduction in WA, the EU size must be tens of KBs, but that is unachievable in current devices. Hence, we conclude that WREN alone does not reduce WA for caches. To reduce WA, we must also re-design the cache.

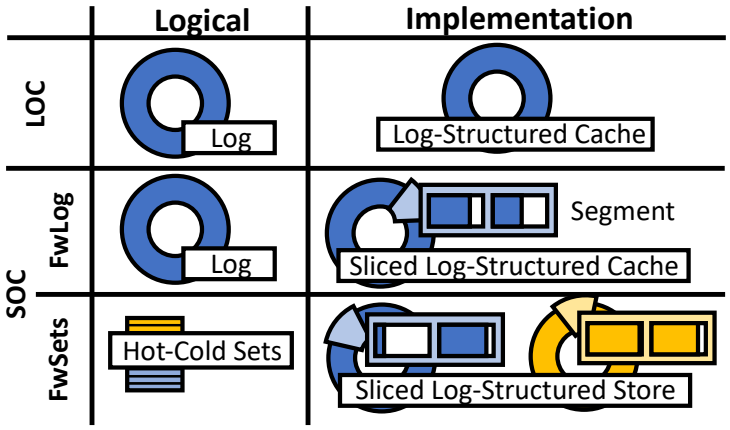


Figure 3.5: Components of FairyWren. The pieces of FairyWren logically and with implementation details.

3.2.2 FairyWREN design overview

FairyWren uses WREN to nearly substantially reduce WA by unifying cache admission with garbage collection. The resulting large reduction in overall flash writes lets FairyWren use next-generation flash (QLC, PLC) while extending device lifetime to improve sustainability.

How FairyWren reduces writes. FairyWren uses WREN's control over data placement and garbage collection to reduce writes in two main ways. First, FairyWren groups data with similar lifetimes into the same EU, even data that in prior caching systems would have been in the same page. If all of the data in each EU has roughly the same lifetime, EUs will either consist mostly of live data or mostly of dead data. FairyWren can then GC the mostly dead EUs with few additional writes. Second, FairyWren piggybacks on GC operations to opportunistically perform cache admission and eviction. When live data is rewritten during GC, FairyWren has an opportunity to evict unpopular objects and admit new objects in their place. In LBAD, these objects would have had to be rewritten separately for GC and admission/eviction.

Architecture of FairyWren. FairyWren partitions its capacity into a large-object cache (LOC) and a small-object cache (SOC), as seen in Fig. 3.5. Incoming requests first check the LOC and then check the SOC. Each cache uses an organization that minimizes its DRAM overhead.

The *large-object cache* stores objects larger than 2 KB and uses a simple log-structured design, since it can tolerate higher per-object DRAM overhead.

The *small-object cache* uses a hierarchical design based on Kangaroo [70]. The SOC contains two levels: FwLog and FwSets. FwLog is a log-structured cache with a relatively high per-object DRAM overhead. The main function of FwLog is to buffer objects so they can be written efficiently to FwSets. FwLog contains about 5% of the SOC's capacity, with the remaining 95% for FwSets.

SOC details. At a high level, FwLog is very similar to KLog, however, many DRAM optimizations that Kangaroo employed do not work on WREN. Thus, FwLog relies on a sliced log-structured cache to minimize DRAM and a double-buffering technique over slices to ensure a low DRAM overhead.

FwSets is also similar to KSets. However, overwriting is impossible in WREN, so FwSets stores *the sets themselves* as objects in a log-structured store. FwSets uses an in-memory index

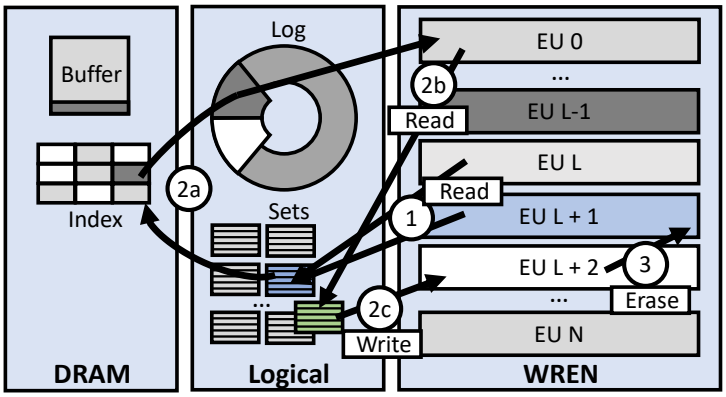


Figure 3.6: Eviction in SOC. Nest packing in FairyWREN small-object cache on WREN, which combines garbage collection and caching logic.

to track the location of each set on flash, but does not track individual objects unlike prior work [58, 63, 85] since this would have too high of a DRAM overhead. The index's DRAM overhead is low because sets are at least 4 KB, whereas objects can be just 10s of bytes. Larger sets (8-64 KB) reduce the size of store's in-memory index, but increase average read latency. To further reduce DRAM, FwSets again employs the slicing and double-buffing scheme from FwLog.

When FwSets's log-structured store is close to full, it must garbage collect in order to admit new objects to the cache. The simplest scheme would be to erase the EU at the tail of the log, evicting all sets — and thus their objects — mapped to this segment[2]. However, since each set contains a mixture of popular and unpopular objects, throwing away entire sets would significantly increase the miss ratio. Instead, FwSets rewrites live sets during GC before erasing the EU.

SOC Eviction (nest packing). If either FwLog or FwSets is running out of space, FairyWren needs to perform *nest packing* (Fig. 3.6). FairyWren's SOC chooses an EU for eviction from FwLog or FwSets depending on which is full. If both logs are full, FwSets is chosen. Since FwLog evicts objects into FwSets, FwSets must have space to evict from FwLog.

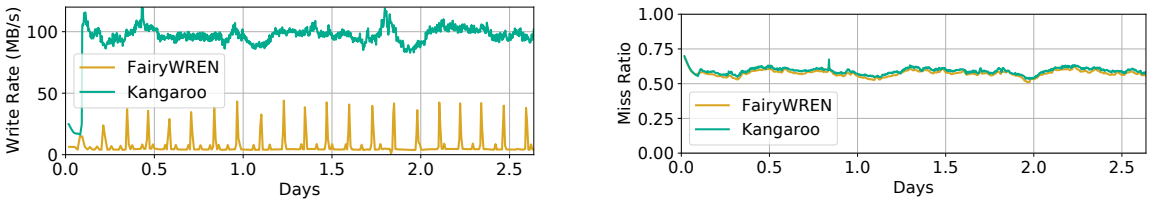
The victim EU is ① read into memory. If evicting in FwLog, each object in the EU corresponds to a *victim set*. In FwSets, each set in the EU *is* a victim set. Then, for each victim set, FairyWren rewrites that set in FwSets by: ②a finding all objects in FwLog that map to this set, ②b forming a new set containing these objects (evicting objects if necessary), and ②c rewriting the set by appending it to FwSets's log. Finally, ③ FairyWren erases the victim EU.

SOC design rationale. Prior flash caches relied on LBAD GC to reclaim flash space from evicted sets, causing DLWA. *The key difference of Fairy*Wren *from prior flash caches is its coordination of cache insertion and eviction with flash GC.*

FairyWren's nest packing algorithm combines previously distinct processes. LBAD caches pay for eviction as ALWA and for garbage collection as DLWA. In the worst case, a set is copied by garbage collection and then immediately overwritten to admit objects from FwLog. Whereas it is impossible to merge these flash writes in LBAD, FairyWren leverages WREN to eliminate unnecessary writes by aligning the eviction and garbage collection cadences of FwLog and FwSets.

Parameter	FairyWren	Kangaroo
Interface	WREN (ZNS)	LBAD
Flash capacity	400 GB	400 GB
Usable flash capacity	383 GB	376 GB
LOC size	10% of flash	10% of flash
SOC Log size	5% of SOC	5% of SOC
SOC Set size	4 KB hot, 4 KB cold	4 KB
Hot set write frequency	every 5 cold set writes	
Set over-provisioning	5%	

Table 3.1: FairyWren and Kangaroo experiment parameters



(a) Write rate

(b) Miss ratio

Figure 3.7: FairyWren reduces writes. The miss ratio and write rate for Kangaroo and FairyWren.

3.2.3 On-flash Experiments

We implement FairyWren in C++ as a module in CacheLib [16]. All experiments were run on two 16-core Intel Xeon CPU E5-2698 servers running Ubuntu 18.04 with 64 GB of DRAM, using Linux kernel 5.15. For WREN experiments, we use a Western Digital Ultrastar DC ZNS540 1 TB ZNS SSD, using the LOC and ZNS library written by Western Digital [50]. The ZNS SSD has a zone (EU) capacity of 1077 MiB. The devices support 3.5 device writes per day for an expected 5-year lifetime.

We compare to Kangaroo [70] over the first ≈2.5 days of a production trace from Meta. FairyWren uses a ZNS SSD and Kangaroo uses an equivalent LBAD SSD with similar parameters (Table 3.1). Both caches use 400 GB of flash capacity, which achieves similar miss ratios as in Kangaroo's production experiments [70]. We overprovision FwSets by 5% to ensure forward progress during nest packing by giving several free EUs to the set log-structured stores. Thus, FairyWren effectively uses 383 GB which could be lowered further at larger capacities. Kangaroo can only use 376 GB of capacity due to device-level overprovisioning. We use the DLWA curve for Kangaroo from its paper [70].

FairyWren greatly reduces writes over Kangaroo. FairyWren aims to achieve a lower write rate to enable caching on more write-intensive caches. In Fig. 3.7a, FairyWren reduces the writes by 12.5× over Kangaroo, from 97 MB/s to 7.8 MB/s. FairyWren achieves this by leveraging WREN to separate writes of different lifetimes, combining cache logic and GC into nest packing, and hot-cold object separation.

The figure shows small write rate spikes in FairyWren. This is because FairyWren performs nest packing at the granularity of an EU, ∼1 GB. Kangaroo's write rate appears smooth as it flushes logs at 256 KB granularity and we estimate its average DLWA.

[2]In this scenario, FwSets would be on a log-structured cache.

	SLC	MLC	TLC	QLC	PLC
Write Multiple	4.4	4	1	.32	.16
Capacity Discount	3	1.5	1	.75	.6

Table 3.2: Emissions for different flash densities. Model parameters for different flash densities. Carbon emissions both optimistically assume that, for the same emissions, capacity increases by the increase in bits per cell.

In the effort to reduce writes, we do not want to sacrifice miss ratio since that will require increasing the resources for backing store and potentially increase end-to-end latency. Fig. 3.7b shows that FairyWren and Kangaroo have very similar miss ratios at 0.575 vs 0.594 on average — 3% lower for FairyWren. FairyWren's advantage comes from using more capacity without incurring too many writes.

While the primary performance metric for caches is miss ratio, FairyWren has better throughput than Kangaroo and a better tail latency.

3.2.4 FairyWren reduces flash cache carbon emisssions

In addition to flash experiments, we implemented a simulator to compare a much wider range of possible configurations for FairyWren to explore its ability to extend lifetime and use denser flash. The simulator replays a scaled-down trace to measure writes and misses from each level of the cache, from the LOC to FwLog to FwSets. We evaluate our cache in simulation on a 21-day trace from Meta [16] and a 7-day trace from Twitter [100]. Both of these traces come have a combination of large and small objects. We present results for the last 2 days of the trace.

Carbon emissions model. We model the carbon emissions that different simulated cache configurations on different densities of flash for different lifetimes require. We assume that a flash device will have the same caching workload for its entire lifetime and that flash write endurance is the main lifetime constraint. While we normalize all results based on the lifetime and we assume that all required flash is purchased at the beginning of deployment.

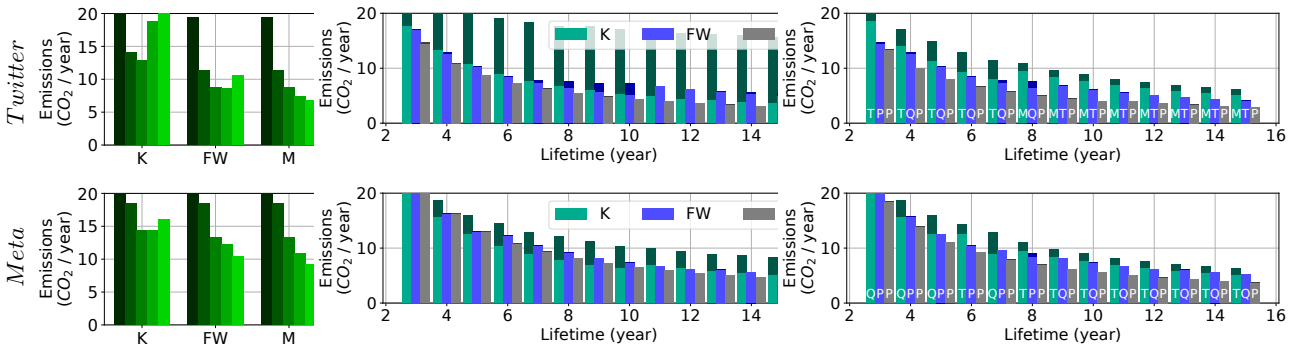
The model bases its estimates on the total flash capacity needed for a simulation to handle both the cache's size and write rate. For instance, a 2 TB cache will require at least 2TBs of cache, but may require 2.5 TB capacity to accommodate the cache's write rate. In addition, LBAD devices need 7% overprovisioning, the standard on datacenter drives.

We base our numbers for write endurance and cost on Micron 7300 NVMe U.2 TLC SSDs. For different densities, we multiply the TLC write endurance by the write multiple in Table 3.2, which are based on [7]. We optimistically assume that more dense flash will have the same emissions for the same number of cells, resulting in a discount for denser flash — i.e. since PLC stores 5 bits compared to TLC's 3 bits, the 1 TB of PLC will release the same emissions as 600 GB of TLC. Our model can be adapted to use more data becomes available. We use the ACT model [39] to estimate operational and embodied emissions. We assume the grid is a 50-50 mix of wind and solar — a common, renewable-energy mix [12].

FairyWren extends flash's lifetime. We evaluate FairyWren on a simulated flash deployment that is constrained to specific devices, ie with a pre-determined capacity and density. For these situations, emissions reductions come from lifetime increases. In Fig. 3.8, we evaluate how long we can use a 2 TB drive for the twitter trace and a 3.5 TB drive for the Meta trace depending on the desired miss ratio. For both traces, FairyWren is able to extend the device's lifetime by

(a) Twitter **(b)** Meta

Figure 3.8: Lifetime improvements from FairyWren. The carbon emissions for a given flash capacity for Kangaroo (K) and FairyWren (FW).



(a) Different densities **(b)** Different lifetimes on QLC **(c)** Different lifetimes on best density

Figure 3.9: FairyWren emissions for different desired lifetimes. The carbon emissions to achieve a target 30% miss ratio (Twitter) or a 20% miss ratio (Meta) for (a) 6-years on SLC (darkest) to PLC (lightest), (b) different lifetimes with QLC flash, and (c) different lifetimes with any flash density.

at least 2 years for all miss ratios and to over an additional 5 years for the lower miss ratios in the Meta trace.

FairyWren is able to use denser flash. We also evaluate FairyWren based on emissions for a target lifetime, exploring the carbon reductions that translating writes to using denser flash can have on flash caching.

FairyWren is the first flash cache to benefit from QLC or PLC for write-intense workloads. A major trend is flash devices becoming denser. We find carbon-optimal configurations for the caches using different densities in Fig. 3.9a. The target miss ratio is 30% for Twitter and 20% for Meta over a 6-year lifetime. Kangaroo performs best with TLC on the Twitter trace and QLC on the Meta one. Using PLC is increases emissions due to excessive overprovisioning needed to account for PLC's fewer writes. Meanwhile, FairyWren's lower write rate enables it to use QLC and PLC drives for Twitter and Meta, respectively. Since Twitter's trace causes more writes, FairyWren on PLC is 24% more carbon-intense than QLC due to overprovisioning. For Minimum Writes, carbon emissions decreases from TLC to QLC by 17%/18% and from QLC to PLC by 8%/15% for Twitter and Meta, respectively. While these numbers show denser flash reduces emissions, they suggest diminishing returns for increasing flash density. For further benefits, we need ways of reducing emissions other than adding more bits per cell.

FairyWren's low WA allows it to avoid massive overprovisioning on dense flash as lifetime is increased. Another trend is increasing device lifetime. Fig. 3.9b explores the impact of longer

lifetimes on carbon emissions for QLC devices. These graphs uses a shaded color to represent each system's overprovisioning. For 6 years, Kangaroo requires 2.2× and 1.17× the emissions of FairyWren, on Twitter and Meta traces, respectively. At 12 years, this increases to 2.6× and 1.54×. Kangaroo has lower emissions when it has some amount of overprovisioning due to DLWA in LBAD. Whereas, FairyWren does not need this overprovisioning and can usually utilize most of the flash capacity. For the Meta trace, FairyWren also does not need much overprovisioning due to its low write overhead.

Caches use less dense flash for longer lifetimes showing that dense flash's write decrease eventually becomes more carbon intense than its emissions-per-bit improvement. To fully minimize emissions, we need to vary both lifetime and flash density. Fig. 3.9c shows each system's emissions for each lifetime with the best density for that lifetime displayed above the bar. Kangaroo is more competitive when allowed to use MLC and TLC, because increasing writes through decreasing density is cheaper than buying more high-density capacity. FairyWren still maintains an advantage for all lifetimes and stays within 30% of Minimum Writes.

In addition, the desired density decreases for longer lifetimes. For example, Kangaroo prefers MLC for the Twitter trace and TLC for the Meta trace at longer lifetimes. FairyWREN's preference is for TLC and QLC, respectively. For flash caching, having enough writes for extending lifetime is more effective at reducing carbon emissions than pursuing efficiency through density.

4 Proposed Work: Declarative IO

Hard drives are quickly running into a bandwidth-capacity wall. To avoid this wall and to allow datacenters to deploy denser, carbon-efficient drives, storage systems need to reduce their bandwidth requirements. Fortunately, bulk storage has a lot of IO overlap between maintenance tasks in the data center that, if eliminated, could significantly reduce the IO. However, the order- and time-flexibility needed to eliminate this IO is hidden behind bulk storage's imperative interface. My proposed project, *Declarative IO*, will enable the storage system to remove this unnecessary IO.

4.1 IO overlap between maintenance tasks

Surprisingly, most IO in distributed, large-scale storage systems is from maintenance tasks rather than application-driven tasks. Caches, particularly the last layer of flash caches, respond to most application IO. However, they are not effective for often low-locality maintenance tasks that have low locality. If there were only a few maintenance tasks, this IO would be hard to reduce, but there are many of these tasks, including:

- *Scrubbing:* To ensure data corruption is detected, every piece of data in the storage system needs to be read, often every month.
- *Compaction:* Many data management services are built on log-structured merge trees (LSMs). This data structure requires frequently compaction, where large files are read, merged together, and rewritten, to maintain performance and regain space.
- *Rebalancing:* Data needs to be distributed among drives, servers, racks, power domains, and regions. Often the data needs to be rebalanced as conditions change such as more disks coming online. Thus, rebalancing requires large amounts of IO to move data.
- *Reconstruction:* When data blocks are lost, they must be reconstructed from erasure codes to ensure that the data is not lost if more blocks are lost.
- *Encoding transitions:* Over time, erasure codes need to change to ensure that data is not lost without wasting space. To accomplish these transitions, data is read and rewritten across the storage system.

Many if not most of these tasks do not have an inherent order or tight deadline. Each piece of the task needs only to eventually be completed. For instance, scrubbing does not care if Block A or Block B is read, as long as both blocks are within the next month. Unfortunately, there is no way to express this flexibility in today's *imperative IO interface*. Each task needs to add both order and time to requests since the interface only allows tasks to specify that they need a specific block now. For the scrubbing example, the task needs to say that it needs Block A now, and then request Block B after it receives Block A.

For just one maintenance task, having it order and time its own IO requests may be fine. However, in aggregate, this imperative paradigm leads to large amounts of unnecessary IO. Fig. 4.1 is an example of just three maintenance tasks. All three maintenance tasks have overlap, but bulk storage using imperative IO cannot detect this overlap since the system receives sequentialized read requests, leading to 2x more IO than necessary. Thus, storage systems need a different interface to reduce IO and avoid the bandwidth-capacity wall.

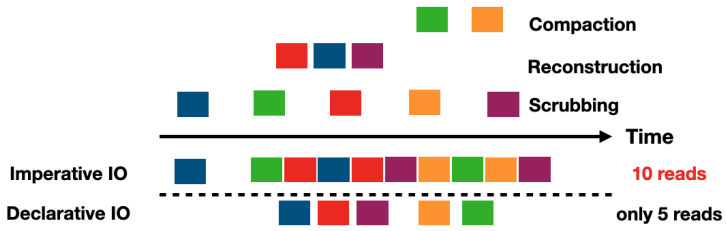


Figure 4.1: Declarative IO reduces IO. In this demostrative example, there are three maintenance tasks fetching IO independently. Each square block represents a request for a piece of data based on color. Moving from imperative IO to declarative IO reduces IO by 50%.

4.2 Declarative IO: Exposing IO flexibility to eradicate duplicate work

Declarative IO allows the storage system to exploit order- and time-flexibility of maintenance tasks to achieve a large IO reduction, scaling the bandwidth-per-TB wall. In order to realize this vision, changes are needed throughout the data management, caching, and bulk storage tiers (Fig. 4.2). As part of this project three main components need to be designed to support declarative IO: a fully specified interface for maintenance tasks, an IO planner, and a combined cache-staging area.

Declarative IO interface: Our proposed interface needs to support a wide variety of maintenance tasks. Based on the list of maintenance tasks in Sec. 4.1, our proposed interface is:

```
declare(IO_TYPE, DATA_PREDICATE, NOTIFY_OPTION, FLAGS, DEADLINE)
```

Listing 4.1: **Proposed Declarative IO interface.** This proposed interface covers the requirements for the described maintenance tasks.

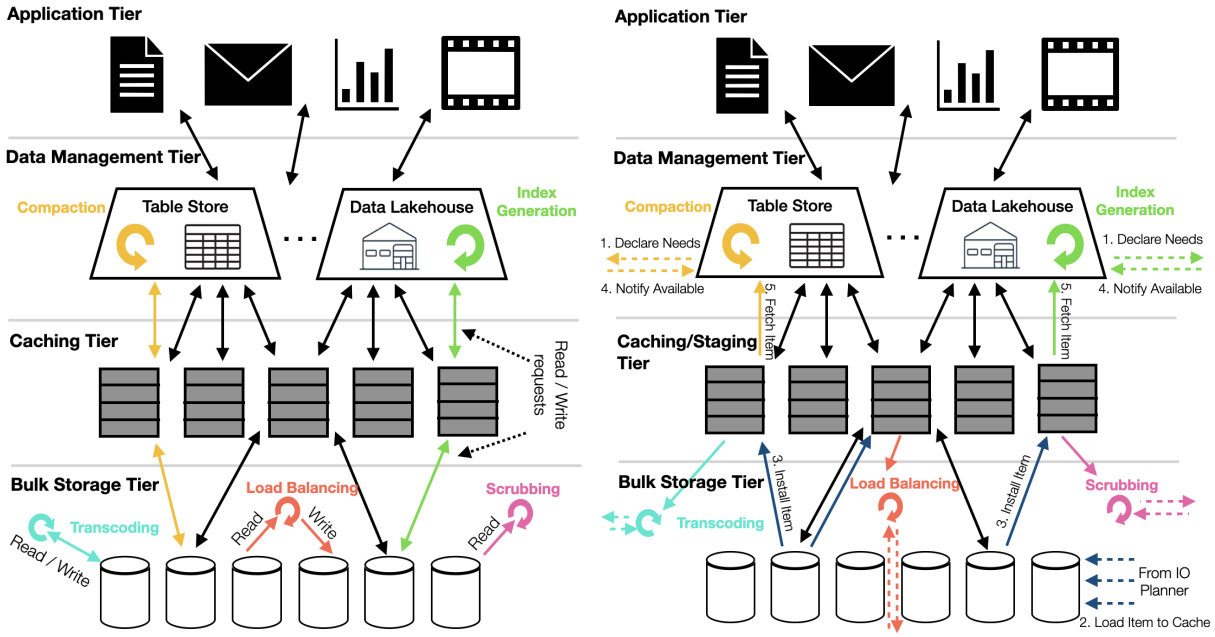
Here, the `IO_TYPE` is either `READ` or `WRITE`. The `DATA_PREDICATE` is to specify data units such as all blocks, a set of stripes, or a set of files. The `NOTIFY_PREDICATE` specifies what component of the data is needed before the tasks should be notified. For instance, a reconstruction job for a stripe of n blocks that has k data blocks only needs k blocks to complete whereas scrubbing just needs a single block. `FLAGS` specifies any additional flags such as skipping the cache. `DEADLINE` specifies the time all data needs to be completed, such as in a week or two hours. An example for scrubbing is:

```
declare(READ, ALL_BLOCKS, EVERY_BLOCK, NO_CACHE, ONE_MONTH)
```

Listing 4.2: **Scrubbing Declarative Interface Example.** For scrubbing, we need to read all of the data from the hard drive directly, but only need to know about one block at a time.

To further refine this interface, it needs to be tested in practice for all of our target maintenance tasks. As maintenance tasks are implemented using the declarative IO paradigm, we will refine this interface and investigate if there are any maintenance tasks that cannot be represented in this paradigm. These tasks would have to default back to the imperative interface, which storage systems will still need to support for application IO. Additionally, while we write-optimizations are possible, we will be focusing on reducing the number of reads.

IO planner: To support this new interface, storage systems will need a new scheduling system to take all of the declarations and execute them *without violating constraints*. This is the responsibility of Declarative IO's *IO Planner*. This planner needs to meet all IO request deadlines *without affecting application IO*, while minimizing the number of re-reads.



(a) Imperative IO (current) **(b)** + Declarative IO (proposed)

Figure 4.2: Imperative vs Declarative IO. Comparison of data center architecture and inter-tier communication with imperative and declarative IO.

More formally, our IO Planner receives a sequence of jobs over time, $(j_1, t_1), \ldots, (j_n, t_s)$. Each job has a set of IO blocks requested, $b_1 \ldots b_m$, the number of IO that need to be fulfilled to be notified, c, and a deadline d. The IO planner needs to schedule all jobs to minimize IO to bulk storage while ensuring that for any job j_i, c_i of the blocks $b_{i1} \ldots b_{im}$ have been fetched by d_i and that the number of blocks fetched during any timestep is less than a pre-specified limit, l. Application requests in this formalization can have a deadline of the next timestep.

One question that comes out of this formation is how and when to convert between different granularities of data. This formulation assumes blocks, but we also want to target files such as from compaction, but the mapping from files to blocks changes over time so we may need to update our tracking information. The granularity is also important to minimize memory overheads of the IO Planner.

Another question from this formulation is that it is not possible to guarantee that all deadlines are accomplished with an online scheduler. Thus, in practice, we will need a priority system between IO requests, likely taking advantage of priority systems that already exist in large distributed storage systems.

Additionally, we need to determine the main evaluation metrics of Declarative IO. Hard drives have both random IO limitations and as well as their bandwidth limitations. We expect a metric such as disk-head time, which combines hard drives seek overheads and data transfer times, would be a good starting point [31].

Cache-Staging area: The third part of the system to support Declarative IO is the cache. Although most maintenance tasks do not individually have high cacheability, our IO Planner will need to store data temporarily while the system notifies maintenance tasks that the data is available. This function is even more important for tasks that need several pieces of data before

they can execute, since not all of this data will be fetched at the same time. We plan to stage these blocks in the cache since we know that they will be accessed in the near future, a higher likelihood of being useful than many objects currently in the last caching layer. Reusing this cache space also ensures that we do not add additional hardware to this system and its additional carbon emissions. However, with adding this secondary use of the caching space, we need to ensure that application performance does not suffer. Thus, part of this project will be investigating cache strategies to maximize space effectiveness while supporting declarative IO.

Evaluation: We will evaluate declarative IO using two main platforms: a detailed simulation to test IO planner and cach-staging area policies as well as large-scale declarative IO deployments and a HDFS-based testbed to validate the simulation in smaller scale. We plan to use a combination of real-world traces and synthesized traces from high-level workload descriptions, allowing us to analyze declarative IO on several different companies storage systems without requiring traces. We are currently in discussions with several companies to get either traces or these high-level workload descriptions.

5 Timeline

	Plan
Spring 2024	**Propose thesis**
	FairyWREN camera-ready or resubmission
	Design prototyping of Declarative IO system
Summer 2024	Declarative IO system building and experiments
Fall 2024	**Submit Declarative IO Paper**
	Academic job applications
Spring 2025	Job search
Summer 2025	**Finish thesis and defend**

Table 5.1: Timeline. My proposed timeline to finish this thesis.

Bibliography

[1] Amazon sustainability. https://sustainability.aboutamazon.com/climate-solutions.

[2] Climate change is humanity's next big moonshot. https://blog.google/outreach-initiatives/sustainability/dear-earth/.

[3] How Lasers Could Unlock Hard Drives With 10 Times More Data Storage. https://www.popularmechanics.com/technology/a20078/heating-magnets-lasers-could-be-the-key-magnetic-recording/, .

[4] Seagate Reveals HAMR HDD Roadmap: 32TB First, 40TB Follows. https://www.tomshardware.com/news/seagate-reveals-hamr-roadmap-32-tb-comes-first, .

[5] Seagate: HAMR is nailing it – no looming 20TB to 30TB capacity problem. https://blocksandfiles.com/2021/09/24/seagate-hamr-on-course-no-looming-20-to-30tb-capacity-problem/, .

[6] Leveldb. https://github.com/google/leveldb.

[7] Wd and tosh talk up penta-level cell flash. https://blocksandfiles.com/2019/08/07/penta-level-cell-flash/ 5/17/22.

[8] Rocksdb. http://rocksdb.org.

[9] Our path to net zero. https://sustainability.fb.com/wp-content/uploads/2023/07/Meta-2023-Path-to-Net-Zero.pdf, 2023.

[10] Is there a limited warranty for samsung ssds? https://semiconductor.samsung.com/us/consumer-storage/support/faqs/05/, 2023.

[11] HDD User Benchmarks. http://hdd.userbenchmark.com/, (accessed July 5, 2023).

[12] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon Explorer: A Holistic Framework for Designing Carbon Aware Datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 118–132, Vancouver BC Canada, January 2023. ACM. ISBN 978-1-4503-9916-6. doi: 10.1145/3575693.3575754.

[13] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 57–70, USA, 2008. USENIX Association.

[14] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of optimal page replacement. *J. ACM*, 1971.

[15] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. What you can't forget: Exploiting parallelism for zoned namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 79–85, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393997. doi: 10.1145/3538643.3539744. URL https://doi.org/10.1145/3538643.3539744.

[16] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory G.

Ganger. The CacheLib caching engine: Design and experiences at scale. In *USENIX OSDI*, 2020.

[17] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. In *ACM SIGMETRICS*, 2018.

[18] Daniel S. Berger, Fiodar Kazhamiaka, Esha Choukse, Inigo Goiri, Celine Irvene, Pulkit A. Misra, Alok Kumbhare, Rodrigo Fonseca, and Ricardo Bianchini. Research avenues towards net-zero cloud platforms. *Workshop on NetZero Carbon Computing*, 2023.

[19] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel ssd subsystem. In *USENIX Conference on File and Storage Technologies*, pages 359–374. USENIX-The Advanced Computing Systems Association, 2017.

[20] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the block interface tax for flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/bjorling.

[21] Netflix Technology Blog. Application data caching using ssds. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef, 2016.

[22] Netflix Technology Blog. Evolution of application data caching : From ram to ssd. https://bit.ly/3rN73CI, 2018.

[23] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *USENIX FAST*, 2010.

[24] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.

[25] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: an embedded concurrent key-value store for state management. *VLDB*, 2018.

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26 (2):1–26, 2008.

[27] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26 (2):145–185, 1994.

[28] Edward G. Coffman and Peter J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973. ISBN 0136378684.

[29] Amanda Peterson Corio. Five years of 100carbon-free future. https://cloud.google.com/blog/topics/sustainability/5-years-of-100-percent-renewable-energy.

[30] Asit Dan and Don Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *ACM SIGMETRICS.*, 1990.

[31] Carson Molder Sathya Gunasekar Jimmy Lu Snehal Khandkar Abhinav Sharma Daniel

S. Berger Nathan Beckmann Greg Ganger Daniel Lin-Kit Wong, Hao Wu. Baleen: Ml admission and prefetching for flash caches. In *FAST*, 2024.

[32] Peter Desnoyers. Analytic modeling of ssd write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–10, 2012.

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

[34] Massimo Gallo, Bruno Kauffmann, Luca Muscariello, Alain Simonian, and Christian Tanguy. Performance evaluation of the random replacement policy for networks of caches. *SIGMETRICS Perform. Eval. Rev.*, 40(1):395–396, June 2012.

[35] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[36] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O'Connor, and Onur Mutlu. What your dram power models are not telling you: Lessons from a detailed experimental study. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(3), dec 2018. doi: 10.1145/3224419. URL https://doi.org/10.1145/3224419.

[37] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, EECS Department, University of California, Berkeley, 12 1990. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1990/6373.html.

[38] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 854–867. IEEE, 2021.

[39] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. ACT: designing sustainable computer systems with an architectural carbon modeling tool. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, 2022.

[40] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/hao.

[41] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The unwritten contract of solid state drives. In *ACM EuroSys*, 2017.

[42] Anne Herreria. How UltraSMR Makes the Industry's Highest Caapcity Drives. https://blog.westerndigital.com/ultrasmr-the-highest-capacity-drives/, 2023.

[43] Amy Hood, July 2022.

[44] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, Santa Clara, CA, February 2017. USENIX Association. ISBN 978-1-931971-36-2. URL https://www.usenix.org/conference/fast17/

technical-sessions/presentation/huang.

[45] Ilias Iliadis, Robert Haas, Xiao-Yu Hu, and Evangelos Eleftheriou. Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):241–252, 2008.

[46] Aamer Jaleel, Kevin Theobald, Simon Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction. In *ISCA-37*, 2010.

[47] Jaeheon Jeong and Michel Dubois. Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers*, 55(4):353–365, 2006.

[48] Nicola Jones et al. How to stop data centres from gobbling up the world's electricity. *Nature*, 561(7722):163–166, 2018.

[49] Lucas Joppa. Made to measure: Sustainability commitment progress and updates. https://blogs.microsoft.com/blog/2021/07/14/made-to-measure-sustainability-commitment-progress-and-updates/.

[50] Ajay Joshi. Cachelib on zns. https://github.com/ajaysjoshi/CacheLib-zns, 2022.

[51] E. G. Coffman Jr and Predrag Jelenković. Performance of the move-to-front algorithm with markov-modulated request sequences. *Oper. Res. Lett.*, 25(3):109–118, October 1999.

[52] Saurabh Kadekodi, K V Rashmi, and Gregory R Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. 2019.

[53] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, KV Rashmi, and Gregory Ganger. Pacemaker: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

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

[55] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for Multi-Streamed SSDs using program contexts. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 295–308, Boston, MA, February 2019. USENIX Association. ISBN 978-1-939133-09-0. URL https://www.usenix.org/conference/fast19/presentation/kim-taejin.

[56] Bran Knowles. Acm techbrief: Computing and climate change, 2021.

[57] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *USENIX FAST*, 2015.

[58] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A Capacity-Optimized SSD cache for primary storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[59] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage*, 13 (3):24, 2017.

[60] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura,

and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3578835. URL https://doi.org/10.1145/3575693.3578835.

[61] Xin Li, Greg Thompson, and Joseph Beer. How amazon achieves near-real-time renewable energy plant monitoring to optimize performance using aws. https://aws.amazon.com/blogs/industries/amazon-achieves-near-real-time-renewable-energy-plant-monitoring-to-optimize-performance-

[62] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *ACM SOSP*, 2011.

[63] Jian Liu, Kefei Wang, and Feng Chen. Tscache: An efficient flash-based caching scheme for time-series data workloads. 2021.

[64] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *USENIX FAST*, 2016.

[65] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. Improving 3d nand flash memory lifetime by tolerating early retention loss and process variation. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(3), dec 2018. doi: 10.1145/3224432. URL https://doi.org/10.1145/3224432.

[66] Jialun Lyu, Jaylen Wang, Kali Frost, Chaojie Zhang, Celine Irvene, Esha Choukse, Rodrigo Fonseca, Ricardo Bianchini, Fiodar Kazhamiaka, and Daniel S. Berger. Myths and misconceptions around reducing carbon embedded in cloud platforms. In *2nd Workshop on Sustainable Computer Systems (HotCarbon23)*. ACM, July 2023. URL https://www.microsoft.com/en-us/research/publication/myths-and-misconceptions-around-reducing-carbon-embedded-in-cloud-platforms/.

[67] Jialun Lyu, Marisa You, Celine Irvene, Mark Jung, Tyler Narmore, Jacob Shapiro, Luke Marshall, Savyasachi Samal, Ioannis Manousakis, Lisa Hsu, et al. Hyrax:{Fail-in-Place} server operation in cloud platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 287–304, 2023.

[68] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. Modular data storage with anvil. In *ACM SOSP*, 2009.

[69] Bill Martin, Yoni Shternhell, Mike James, Yeong-Jae Woo, Hyunmo Kang, Anu Murthy, Erich Haratsch, Kwok Kong, Andres Baez, Santosh Kumar, and et al. Nvm express technical proposal 4146 flexible data placement, Nov 2022.

[70] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *ACM SOSP*, 2021.

[71] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Theory and practice of caching billions of tiny objects on flash. *ACM Transactions on Storage*, 2022.

[72] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elas-

tic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 461–477, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/min.

[73] Christian Monzio Compagnoni, Akira Goda, Alessandro S. Spinelli, Peter Feeley, Andrea L. Lacaita, and Angelo Visconti. Reviewing the evolution of the nand flash technology. *Proceedings of the IEEE*, 105(9):1609–1633, 2017. doi: 10.1109/JPROC.2017.2665781.

[74] Melanie Nakagawa. On the road to 2030: Our 2022 environmental sustainability report. https://blogs.microsoft.com/on-the-issues/2023/05/10/2022-environmental-sustainability-report/, 2022.

[75] Council of the European Union. Fit for 55. https://www.consilium.europa.eu/en/policies/green-deal/fit-for-55-the-eu-plan-for-a-green-transition/, 2024.

[76] Alina Oprea and Ari Juels. A Clean-Slate Look at Disk Scrubbing. 2010.

[77] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *ASPLOS*, 2014.

[78] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

[79] Francisco Pires. Solidigm introduces industry-first plc nand for higher storage densities. https://www.tomshardware.com/news/solidigm-plc-nand-ssd, 2022.

[80] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *ACM SOSP*, 2017.

[81] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *ACM SOSP*, 1991.

[82] Thomas JE Schwarz, Qin Xin, Ethan L Miller, Darrell DE Long, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings.*, pages 409–418. IEEE, 2004.

[83] Seagate. The Digitization of the World From Edge to Core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2018.

[84] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. Optimizing flash-based key-value cache systems. In *USENIX HotStorage*, 2016.

[85] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. Didacache: an integration of device and application for flash-based key-value caching. *ACM Transactions on Storage (TOS)*, 14 (3):1–32, 2018.

[86] Anton Shilov. Seagate's HAMR Update: 32 TB in Early 2024, 40+ TB Two Years Later. https://www.anandtech.com/show/21125/seagates-hamr-update-32-tb-in-early-2024-40-tb-two-years-later, 2023.

[87] Shigeru Shiratake. Scaling and performance challenges of future dram. In *2020 IEEE International Memory Workshop (IMW)*, pages 1–3, 2020. doi: 10.1109/IMW48823.2020.9108122.

[88] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010. doi: 10.1109/MSST.2010.5496972.

[89] Billy Tallis. Micron 3d nand status update. https://www.anandtech.com/show/10028/micron-3d-nand-status-update, .

[90] Billy Tallis. 2021 nand flash updates from isscc: The leaning towers of tlc and qlc. https://www.anandtech.com/show/16491/flash-memory-at-isscc-2021, .

[91] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *USENIX OSDI.*, 2020.

[92] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: advanced photo caching on flash for facebook. In *USENIX FAST*, 2015.

[93] Swamit Tannu and Prashant J Nair. The Dirty Secret of SSDs: Embodied Carbon. In *HotCarbon*, 2022.

[94] Amanda Tomlinson and George Porter. Something Old, Something New: Extending the Life of CPUs in Datacenters. In *HotCarbon*, 2022.

[95] Ted Tso. Aligning filesystems to an ssd's erase block size. https://tytso.livejournal.com/2009/02/20/.

[96] Haitao Wang, Zhanhuai Li, Xiao Zhang, Xiaonan Zhao, Xingsheng Zhao, Weijun Li, and Song Jiang. OC-Cache: An Open-channel SSD Based Cache for Multi-Tenant Systems. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–6, Orlando, FL, USA, November 2018. IEEE. ISBN 978-1-5386-6808-5. doi: 10.1109/PCCC.2018.8711079.

[97] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in Log-Structured storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, Santa Clara, CA, February 2022. USENIX Association. ISBN 978-1-939133-26-7. URL https://www.usenix.org/conference/fast22/presentation/wang.

[98] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, High-Performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, 2006.

[99] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Trans. Storage*, 13(3), oct 2017. ISSN 1553-3077. doi: 10.1145/3121133. URL https://doi.org/10.1145/3121133.

[100] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory

key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[101] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the memory wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023.

[102] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission Optimization for Google Datacenter Flash Caches. page 17.

[103] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 1, USA, 2012. USENIX Association.