FairyWREN: A Sustainable Cache for Emerging Write-Read-Erase Flash Interfaces

Sara McAllister Yucong "Sherry" Wang Benjamin Berg* Daniel S. Berger†
George Amvrosiadis Nathan Beckmann Gregory R. Ganger

Carnegie Mellon University **UNC Chapel Hill* *†Microsoft Azure and University of Washington*

Abstract

Datacenters need to reduce embodied carbon emissions, particularly for flash, which accounts for 40% of embodied carbon in servers. However, decreasing flash's embodied emissions is challenging due to flash's limited write endurance, which more than halves with each generation of denser flash. Reducing embodied emissions requires extending flash lifetime, stressing its limited write endurance even further. The legacy Logical Block-Addressable Device (LBAD) interface exacerbates the problem by forcing devices to perform garbage collection, leading to even more writes.

Flash-based caches in particular write frequently, limiting the lifetimes and densities of the devices they use. These flash caches illustrate the need to break away from LBAD and switch to the new Write-Read-Erase iNterfaces (WREN) now coming to market. WREN affords applications control over data placement and garbage collection. We present FairyWREN[1], a flash cache designed for WREN. FairyWREN reduces writes by co-designing caching policies and flash garbage collection. FairyWREN provides a 12.5× write reduction over state-of-the-art LBAD caches. This decrease in writes allows flash devices to last longer, decreasing flash cost by 35% and flash carbon emissions by 33%.

1 Introduction

DATACENTER CARBON EMISSIONS are a topic of growing concern. At current emission rates, datacenters' share of global emissions are projected to rise to 20% by 2038 [48] and 33% by 2050 [53]. In the next few decades, many companies — including Amazon [1], Google [2], Meta [11], Microsoft [71] — are looking to achieve Net Zero, i.e., greenhouse gas emissions close to zero. To achieve this goal, many datacenters are adopting renewable energy sources such as solar and wind [11, 39, 64, 71]. Google, AWS, and Microsoft are expected to complete their transition to renewable energy by 2030 [30, 49, 59]. 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 [39].

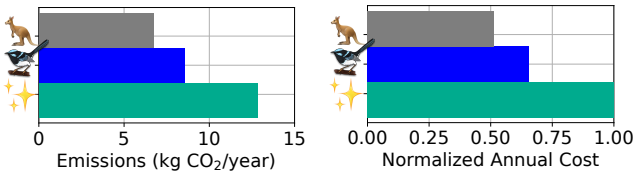


Fig. 1: Carbon emissions and cost for flash in Kangaroo (🦘), FairyWREN (🐦‍⬛), and "minimum writes" (✨)—an idealized cache with no extra writes—over a 6-year lifetime for a production Twitter trace and a target 30% miss ratio. Compared to Kangaroo, FairyWREN reduces carbon emissions by 33% and cost by 35%.

Embodied emissions are produced by one-time lifecycle events. Datacenters can reduce these emissions by: *(i)* replacing hardware with less carbon-intensive alternatives, and *(ii)* extending the lifetime of components to amortize embodied emissions over a longer period. Recent work has studied embodied emissions in processor design [24, 38, 39, 85], but considerably less attention has been paid to memory and storage, even though they constitute 46% and 40% of server emissions, respectively [64]. It is therefore crucial to both move from carbon-intensive technologies like DRAM to flash, which has 12× less embodied carbon per bit [38], *and* to extend flash lifetimes to amortize flash's embodied carbon.

However, flash introduces a new challenge: *limited write endurance*. A flash device can only be written a limited number of times before it wears out. Each new generation of flash has lower write endurance as a result of manufacturers packing more bits into each cell. This packing, however, does improve sustainability by storing more capacity in the same silicon (i.e., less carbon per bit). To realize the benefits of denser flash, applications must write to flash much less frequently. The write-rate budgets that applications must operate under to achieve longer lifetimes are tiny: to achieve a six-year lifetime on a 2 TB QLC drive, the application can write only 14 MB/s,

[1] Fairywrens (🐦‍⬛) are vibrant birds native to Australia. Common varieties include Superb Fairywrens, Splendid Fairywrens, and Lovely Fairywrens.

or 0.09% of available write bandwidth (Sec. 2).

Reducing carbon from caching. Hence, write-intensive flash applications present a major challenge in reducing overall datacenter emissions. This paper focuses on reducing carbon from flash caching, an increasingly popular use of flash in the datacenter [3,16,21,22,35,36,83]. We aim to demonstrate, through caching, how to *leverage emerging flash interfaces* to reduce writes, in particular by *re-purposing garbage collection to do useful work*.

Caching is fundamentally write-intensive, as new objects must be frequently admitted to maintain hit rates [15,18]. Datacenter caches also store many small objects [16,67], which is particularly problematic because flash can only be written at a coarse granularity. Because of this mismatch, admitting small objects to the cache can lead to significant *write amplification*: i.e., more bytes are written to the underlying flash device than requested by the application.

Most current flash devices are *Logical Block-Addressable Devices (LBAD)* that present the same block device abstraction used by hard disks. This abstraction hides significant details about how SSDs work. In particular, while the interface allows reading and writing 4KB blocks, the underlying flash device can only erase large (MB to GB) regions. To implement the LBAD interface, the flash firmware performs garbage collection, copying blocks of valid data and erasing entire regions to make room for new writes. Current flash caches have a limited ability to optimize these internal writes, which can amplify the total bytes written by 2× to 10× [67].

Opportunity: WREN. New flash SSD interfaces, such as ZNS [19] and FDP [66], allow closer integration of host-level software and flash management. The key difference between these interfaces and LBAD is that these interfaces include `Erase` as a first-order operation, allowing the cache to control garbage collection. We use the name *Write-Read-Erase iNterfaces (WREN)* to collectively refer to such interfaces, and we describe the necessary and sufficient operations for flash caches to minimize write rate. However, we also show that merely porting existing flash caches to WREN does not reduce flash writes. *Flash caches must be re-designed to leverage the additional control provided by WREN.*

Our solution: FairyWREN. We design and implement FairyWREN, a flash cache that harnesses WREN to reduce writes. The main insight in FairyWREN is that every flash write, whether from the application *or from garbage collection*, is an opportunity to admit objects to the cache. When flash is written during garbage collection, FairyWREN can admit objects "for free". This idea cannot be realized on LBAD, since these devices offer no control over garbage collection. FairyWREN uses the features of WREN to perform a "nest packing" algorithm on *every* write, *unifying cache admission and garbage collection in a single algorithm*. FairyWREN also leverages WREN to enable large-small object separation and hot-cold set-partitioning, further reducing writes.

Summary of results. We find that, without major changes to flash interfaces and cache designs, deploying denser flash will not reduce the carbon emissions of flash caches. *For current caching systems, the reduced write endurance of denser flash outweighs the gains in density.* Only by changing the flash interface and optimizing the cache to this new interface can we realize the significant emissions savings of denser flash.

To illustrate this point, we implement FairyWREN as a flash cache module within CacheLib [16]. We evaluate FairyWREN on production traces from Meta and Twitter using both simulation and a real ZNS SSD. *FairyWREN reduces flash writes by 12.5×* vs. the research state-of-the-art. By enabling caching on denser flash, *FairyWREN reduces flash's carbon emissions by 33%* vs. the research state-of-the-art (Fig. 1). FairyWREN performs close to an idealized, minimum-write cache on both carbon emissions and cost.

Contributions. This paper contributes the following:

- *Flash trends (Sec. 2):* By studying flash trends, we identify opportunities for more sustainable flash caching as well as challenges that prevent current flash caches from realizing these benefits (Sec. 3).
- *Critical elements of flash interfaces (Sec. 4):* We identify the `Erase` operation and control over garbage collection as the essential features of emerging flash interfaces. We describe tradeoffs and fundamental constraints of flash interfaces, showing that some features are, contrary to prior work, unhelpful for caching.
- *FairyWREN (Sec. 5):* FairyWREN's key insight is to leverage emerging flash interfaces to unify garbage collection and cache admission as one operation, greatly reducing overall flash writes. FairyWREN further reduces writes by partitioning objects by size and popularity (hot vs. cold).
- *Analysis of flash emissions (Sec. 6):* We develop a model to analyze carbon emissions from flash. We show that FairyWREN's write reduction allows flash caches to improve sustainability using denser flash for longer lifetimes.

2 Opportunities in flash caching

Flash is an increasingly attractive option for caching [16, 21,22,35,57,67,68,83]. In this section, we discuss how trends in the design of flash devices present growing opportunities to reduce the cost and carbon emissions of caching.

Opportunity 1: *Flash is less carbon-intensive than DRAM, so caches are more sustainable with less DRAM.*

DRAM often makes up 40% to 50% of server cost [58, 79, 82] and is no longer scaling (Fig. 2). DRAM also has a large embodied carbon footprint and has large operational emissions due to requiring up to half of system power [38].

Flash is cheaper per-bit, embodies 12× less carbon, and requires less power per-bit than DRAM [38]. Thus, datacenters should use flash over DRAM whenever possible [37], even for traditionally DRAM workloads, such as caching [16, 35, 67, 68] or machine learning [95].

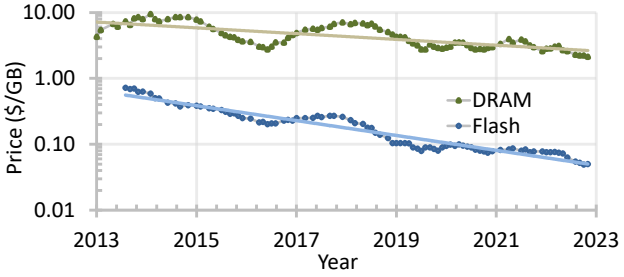


Fig. 2: Cost for flash and DRAM over the last 10 years [4, 6]. Flash prices have decreased over 14×, while DRAM prices have only decreased by ≈2×.

Opportunity 2: *Flash caches should use denser flash where possible to reduce emissions.*

Flash is becoming denser, 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)* and *penta-level cells (PLC)* [73]. Denser flash is cheaper; e.g., PLC is forecast to be 40% cheaper per-bit than TLC [9]. Denser flash also reduces carbon emissions, since more bits are packed onto roughly the same silicon.

Opportunity 3: *Lengthening device lifetime is an effective way to improve datacenter sustainability.*

Traditionally, datacenter hardware replacement cycles have been around three years [64] due to the rate of improvement in hardware performance and energy 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 four to six years [42, 65], 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 eight years [17, 64].

Moving to longer lifetimes amortizes both cost and embodied carbon. As datacenters shift to renewable energy, they are rapidly reducing operational carbon. As a result, embodied carbon now dominates datacenter carbon emissions [12, 38, 39, 84]. The major challenge, though, is how to extend *flash* lifetime, given its limited write endurance.

3 Challenges in flash caching

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 fail. Hence, while flash caching presents carbon-saving opportunities (Sec. 2), caches must severely limit the amount they write. Here, we discuss the challenges of flash caching in detail and describe how current systems fail to address these challenges.

3.1 Wherefore device write amplification?

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. The *device-level write amplification* (DLWA) [23, 35, 41, 54, 57, 62, 83] captures this 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 [67]. DLWA causes write-intensive applications to quickly wear out flash devices, increasing their replacement frequency and embodied emissions over time.

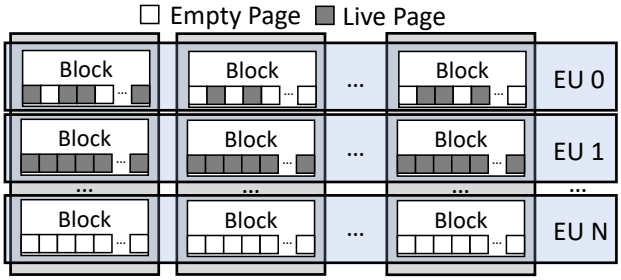


Fig. 3: The internal arrangement of flash devices into planes, blocks, pages, and EUs. Each EU has blocks in multiple pages. EU 0 is a partially full, EU 1 is entirely full, and EU N has just been erased.

DLWA is primarily caused by the physical limitations of flash storage. Flash devices are organized in a physical hierarchy (Fig. 3). 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, 63]. 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 to gigabytes [19].

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. 3) 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.

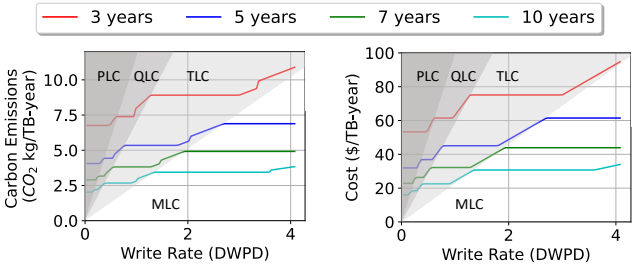
One might hope that technological advances would decrease EU sizes, closing the gap between write and erase granularities. However, *flash EU sizes have gotten larger as flash has gotten denser*. Effective block sizes on an SLC flash device were 128 KB [86], MLC and TLC flash devices are around 20 MB [81], and QLC devices will be 48 MB [80]. Striping these blocks with hundreds of 3D-stacked layers [80] results EUs in the gigabyte range [19, 69].

Lesson for flash caches: Write amplification is caused by the size mismatch between writes and erases in flash. This mismatch will keep increasing.

3.2 Denser flash has lower write endurance

As flash becomes denser, its write endurance drops significantly. For example, while PLC flash is up to 40% denser than TLC, PLC is forecast to have only *16%* of TLC's writes [9]. 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 [70]. 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 requires closing an active EU, incurring more internal device writes.

Fig. 4 models how write rate affects both emissions and cost 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 for each technology to achieve the desired lifetime at a given write rate. For example, a device lasting 7 years (green) has lower annualized carbon emissions than one lasting 3 or 5 years, and it should use dense flash (e.g., TLC) only at write rates below two device-writes-per-day.



(a) Carbon emissions. *(b) Cost.*

Fig. 4: The annual carbon emissions and cost of flash depending on the required average write rate and desired lifetime.

Lesson for flash caches: Device lifetime is the most important factor in reducing carbon emissions. Moreover, denser flash can improve sustainability, but only if flash write rate is very small — much less than one device-write per day.

3.3 Shortcomings of existing solutions

To limit embodied emissions, sustainable flash caches must minimize *(i)* idle flash space — which incurs emissions for no benefit; *(ii)* DRAM usage for object metadata — which can add up to tens of GBs [35, 67]; and *(iii)* flash write rates — which wear out the device, reducing lifetime. No prior flash-cache design meets these criteria (Table 1). In particular, although caches must admit new objects to maintain hit rates, flash caches must be designed to minimize application- and device-level write amplification to extend device lifetime.

Flash caches $\neq$ DRAM caches. Both flash caches and DRAM caches try to reduce misses, but flash caches must also contend with flash's limited write endurance, leading to much different designs. Flash caches are designed to achieve low end-to-end write amplification, i.e., the product of *application-level write amplification* (ALWA) (e.g., from having to write 4KB to flash to admit a 100B object) and DLWA.

	Flash caches should minimize ...			
	Unused flash	**DRAM**	**ALWA**	**DLWA**
Key-value stores	✗	✓	✓	✓
Log-structured caches	✓	✗	✓	✓
Set-associative caches	✗	✓	✗	✗
Kangaroo [67]	✓	✓	✓	✗
FairyWren	✓	✓	✓	✓

Table 1: Comparison of FairyWren vs. prior cache designs. FairyWren is the only design to minimize all important overheads.

Flash caches $\neq$ key-value stores. KV stores [5, 7, 33, 55, 60, 75, 90] support a similar read-write interface as caches and likewise minimize flash writes and DRAM overhead. However, flash caches have significantly different design goals.

The main difference is that `delete` operations are uncommon in KV stores, but very frequent in caches. Caches frequently evict objects and must reclaim space immediately to admit new objects [67]. Most KV stores do not support deleting objects quickly enough to implement cache eviction policies. Specifically, standard KV store data structures like LSM trees [5, 7, 31, 32, 60, 75, 90] will not work well for caching unless the KV store is massively overprovisioned, often by more than 2× the cache capacity [21, 22, 83].

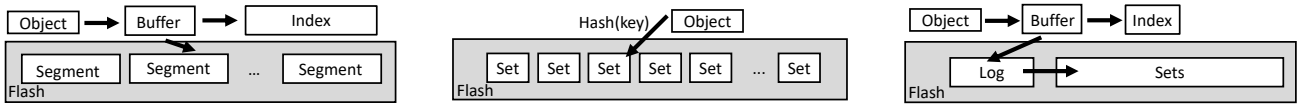
Moreover, KV stores do not exploit a cache's biggest advantage: caches are free to evict objects whenever it is convenient. Evicting objects opportunistically can greatly reduce writes and maximize space utilization, but KV stores are not built to exploit this cache-specific optimization.

Existing flash caches do not address DLWA. Because of the unique challenges of flash caching, there is a growing body of work devoted to improving flash cache designs. Prior flash caches generally fall under three categories (Fig. 5): log-structured, set-associative, and hierarchical.

Log-structured caches. To minimize writes, many flash caches are log-structured [16,27,35,83]. These caches append objects to an on-flash log (Fig. 5a), locating objects through a DRAM index and evicting objects in large groups. The log allows large sequential writes to flash and thus achieves nearly ideal write amplification.

While log-structured caches work well for larger objects, the DRAM index becomes prohibitively large for small objects, even if it is highly optimized [67], significantly increasing overall emissions and cost (see Fig. 11). Flash caches are thus often partitioned, using a log-structured cache for large objects and a different design for small objects [16].

Set-associative caches. Set-associative caches, such as the Small Object Cache in Meta's CacheLib [16], replace the DRAM index with a hash function that maps each object to a unique set (usually a 4 KB page) on flash (Fig. 5b).



(a) Log-structured *(b) Set-associative* *(c) Kangaroo*

Fig. 5: Designs of prior flash caches: (a) Log-structured caches write objects segments to flash sequentially, (b) Set-associative cache write objects to a set based on the key's hash, and (c) Kangaroo is a hierarchical design that combines a log-structured and a set-associative cache.

The downside of these caches is that they cause significantly more writes. When a set-associative cache admits a small object (say, 100 B), it must write at least one flash page (4 KB), resulting in large ALWA (40×). Even worse, these caches perform random writes, leading to DLWA of 2× to 10× [67]. Since write amplification (WA) is the product of ALWA and DLWA, *a set-associative cache's WA easily exceeds 100×*. To mitigate this, Meta's flash caches use only 50% of the drive [16], increasing miss ratio and carbon emissions.

Hierarchical. FairyWREN builds on Kangaroo [67, 68], a hierarchical flash cache for small objects that combines a small log-structured cache (KLog) and a large set-associative cache (KSet) (Fig. 5c). Kangaroo uses KLog to reduce ALWA and KSet to reduce DRAM. Objects are first buffered in KLog and then admitted in batches to KSet, amortizing its ALWA across several admitted objects. KSet comprises more than 90% of the cache capacity, limiting the DRAM needed to index KLog. Kangaroo also includes a selective admission policy to reduce flash writes and a partitioned index data structure to reduce DRAM. Due to its low DRAM overhead, Kangaroo achieves large emission reductions over a memory-optimized log-structured cache, Flashield [35], for workloads with many small objects (Fig. 11 in Sec. 6.2).

While Kangaroo optimizes both DRAM and ALWA, it still has too many writes because *Kangaroo does not address device-level write amplification*. KSet performs random 4 KB writes, the worst case for DLWA. As a result, Kangaroo's emissions do not reduce with denser flash. Fig. 4 shows that, for a 10-year lifetime, QLC requires fewer than 0.37 device-writes per day (DWPD) and PLC requires fewer than 0.16 DWPD, whereas Kangaroo performs 1.46 DWPD in our evaluation.

4 Write-Read-Erase iNterfaces (WREN)

Prior flash caches incur excessive DLWA (Sec. 3). The root causes are the mismatch between write and erase granularities and a legacy LBAD interface that hides this mismatch from software. This section discusses recent *Write-Read-Erase iNterfaces (WREN)*, such as ZNS [19] and FDP [66], that include `Erase` as a first-order operation. We show that WREN is necessary but insufficient: a new flash interface does not reduce writes by itself, changes to the cache design are required.

4.1 Today's interface is LBAD

Most flash SSDs today are *logical block addressable devices (LBAD)*, sharing the same interface as disks. LBAD presents the flash device as a linear address space of fixed-size blocks[2] that can be independently read or written.

LBAD eased the transition from HDDs to SSDs, but does not expose the erase granularity of flash (Sec. 3). As a result, the LBAD device firmware must perform garbage collection (GC) that can cause high DLWA and tail latency. Although there has been work to decrease DLWA [40, 41, 44, 56, 89, 91], LBAD devices still hide erase units and GC from applications, preventing co-optimization to minimize overall flash writes.

4.2 Challenges of new interface design

While a variety of flash interfaces have been proposed [20, 44, 51, 52, 72, 78, 88, 96], 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 [51, 52] 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.

Open-Channel SSDs [20] remove all flash-device logic and force applications to handle *all* of flash's complexities, even beyond those described in Sec. 3. While the hope was to develop layers of abstraction in software to hide some of this complexity, this software was never widely deployed.

Lesson for flash caches: 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.

4.3 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) [19] and *Flexible Data Placement* (FDP) [66] are both WREN. Both interfaces are NVMe standards with strong support from industry and provide an abstraction for writing to an EU[3]. 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 through Zone Append [96]. FDP

[2]These fixed-size blocks correspond to pages, not flash blocks (Sec. 3)
[3]This abstraction is called a *zone* in ZNS and a *reclaim unit* in 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 beyond WREN.

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

4.4 WREN alone is not a cure for WA

WREN devices make it easy to perform large, sequential writes with no WA. When writing sequentially, the user can maintain a single active EU and fill the EU completely before activating the next EU. Furthermore, 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.

Set-associative flash caches also want low WA for 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 updates sequentially to a single active EU. We find that this naive approach does not reduce WA because it just moves the GC from the device to the cache.

The impact of smaller EUs. One idea for mitigating WA under small, random writes is to reduce the EU size, e.g., from a GB to tens of MB, by removing error correction between flash blocks, which caches can tolerate. Prior systems literature uses smaller EUs to minimize GC [14, 69] 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, other prior work that analyzes the WA of FIFO GC policies [34, 46] 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 in prior work, we model the WA of a FIFO GC policy for a set-associative cache, capturing the effect of EU size.

Following the approach of [46], we approximate the distribution of the number of live pages in the EU at the tail of a log-structured store (see Appendix A for details). Our approximation shows that when EU sizes are small, 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. 6) matches simulation results, with a R^2 value of 0.9996.

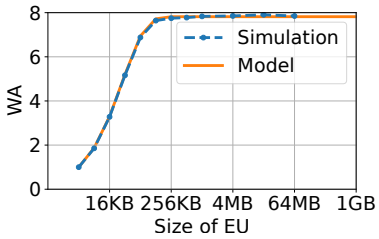


Fig. 6: The DLWA for a set-associative cache running on WREN with 7% overprovisioning. EUs have to be less than 128 KB to significantly reduce DLWA.

Lesson for flash caches: 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 (Sec. 3). Hence, we conclude that WREN alone does not reduce WA for caches. To reduce WA, we must also re-design the cache.

5 FairyWREN Overview and Design

FairyWREN uses WREN to substantially reduce WA by unifying cache admission with garbage collection. The resulting reduction in overall writes lets FairyWREN use denser flash while extending device lifetime to improve sustainability.

5.1 Overview

How FairyWREN reduces writes. FairyWREN uses WREN's control over data placement and garbage collection to reduce writes in two main ways. First, FairyWREN introduces *nest packing* to combine garbage collection with 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, by contrast, these objects would have to be rewritten separately for GC and admission/eviction.

Second, FairyWREN groups data with similar lifetimes into the same EU, separating 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. FairyWREN leverages two main techniques to enable this grouping: large-small object separation and hot-cold set partitioning.

Architecture of FairyWREN. FairyWREN partitions its capacity into a large-object cache (LOC) and a small-object cache (SOC), as seen in Fig. 7. Incoming requests first check the LOC and then check the SOC.

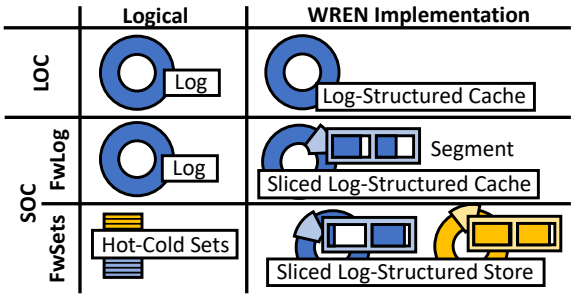


Fig. 7: The components of FairyWREN.

The *large-object cache* (Sec. 5.2) 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* (Sec. 5.3) uses a hierarchical design based on Kangaroo [67]. 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. Therefore, FwLog can have a fairly low capacity ($\approx 5\%$), keeping its DRAM overhead low. FwSets is a set-associative cache, but, since WREN does not support random writes, the sets are kept in a log-structured store. FwSets stores sets, not individual objects, in the log to minimize DRAM. When this log-structured store is garbage collected, objects are opportunistically moved from FwLog into FwSets. Finally, each set in FwSets is further partitioned into hot (frequently accessed, long-lived) objects and cold (recently admitted, short-lived) objects (Sec. 5.4).

5.2 The LOC

The LOC is a log-structured cache. Adapting log-structured caches to WREN is straightforward, since they only perform large, sequential writes. The LOC is broken into large segments, each the size of an EU. Segments can then be evicted in LRU or FIFO order with minimal WA. The LOC uses DRAM in two ways: *(i)* an in-memory, EU-sized buffer for log insertions, and *(ii)* an in-memory index tracking object locations on flash. Because the LOC stores large objects, it contains relatively few objects and needs little DRAM. Besides the segment buffer, all LOC objects are stored on flash.

Insertions. Objects are first inserted into an in-memory segment buffer and added to the in-memory log index. Once the segment buffer is full, it is written to an empty EU in the log.

Lookup. Reads look up the object's key in the log index. If found, the cache reads the object from the indicated EU.

Eviction. Eventually, the log will fill up and LOC will evict a log segment based on the eviction policy. Since log segments are aligned to EUs, eviction simply `Erases` an EU, evicting those objects from the cache. This design does not rewrite any objects, incurring minimum WA of $1\times$.

5.3 The SOC

The focus of FairyWREN is the SOC. Log-structured caches are impractical for caching small objects because a large flash cache can fit billions of small objects, requiring a large DRAM index to track them all (Sec. 3.3). FairyWREN's SOC is based on Kangaroo [67], a recent flash cache designed for small objects with low DRAM overhead and low ALWA. The SOC is a hierarchy of two levels: FwLog, a small log-structured cache, and FwSets, a large set-associative cache. FwLog contains about 5% of the SOC's capacity, with the remaining 95% for FwSets. We describe FwLog and FwSets individually, and then how they work together.

FwLog design. FwLog's goal is to buffer new small objects for insertion into FwSets. Like the LOC, FwLog is a log-structured cache that uses an in-memory segment buffer and an in-memory index to track objects in the FwLog. All other objects in the FwLog are stored on flash.

FwSets design. FwSets is a set-associative cache that maps each object to a unique set by hashing its key. When admitting an object, FwSets evicts old objects from the object's set then overwrites it. 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 to track the location of each set on flash, but, unlike prior work [56,61,78], it does not track individual objects, since this would incur too much DRAM overhead. The index's DRAM overhead is low because a set is at least 4 KB, whereas objects can be just 10s of bytes. (Larger sets reduce the size of FwSets's DRAM index, but increase average read latency.)

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[4]. However, since each set contains a mixture of popular and unpopular objects, throwing away entire sets would significantly increase miss ratio. Instead, FwSets rewrites live sets during GC before erasing the EU.

SOC operation. FwLog and FwSets operate as a hierarchy:

Lookup. Lookups first check FwLog for the object. If not found, FwSets hashes the object's id and looks up the *set's* location. The set is read and scanned for the object.

Insertion. FairyWREN first inserts objects into FwLog. When FwLog is full, objects are evicted from FwLog and inserted into FwSets, as described next. Similarly, inserting into FwSets can cause cascading eviction from FwSets.

Eviction (nest packing). If either FwLog or FwSets is running out of space, FairyWREN needs to perform *nest packing* (Fig. 8). 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 because FwSets must have space

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

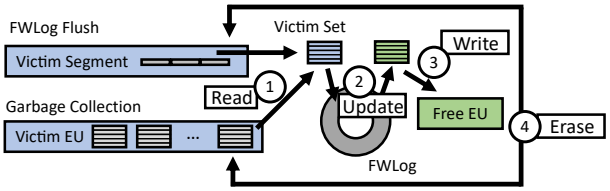


Fig. 8: Nest packing in FairyWREN's small-object cache.

to receive objects evicted from FwLog.

The victim EU is first read into memory. If evicting from FwLog, each object in the EU hashes to a *victim set*. Otherwise, when evicting from FwSets, each set in the EU *is* a victim set. Then, ① FairyWren rewrites each victim set by: ② finding all objects in FwLog that map to a given set, forming a new set containing these objects (evicting objects as necessary), and ③ rewriting the set by appending it to FwSets's log. Finally, ④ 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 FairyWren 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 is immediately rewritten to admit objects from FwLog. 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.

5.4 Optimizing the SOC

The SOC is the main source of DRAM overhead and WA in FairyWren. We employ a variety of optimizations to improve the memory and write efficiency of the SOC.

Reducing flash writes by separating hot and cold objects. Even after using nesting to decrease writes, FwSets is still the primary source of flash writes in FairyWren. FairyWren further reduces writes by separating objects by popularity, as determined by a modified RRIP algorithm [45, 67]. Instead of a set being *one unit* that is written every insertion, each set in FwSets is split in twain, into a subset for popular objects and a subset for unpopular objects, each backed by its own log-structured store. Each subset is at least a page. Paradoxically, since the unpopular objects are most likely to be evicted, the subsets with unpopular objects correspond to hot (i.e., frequently written) pages on flash. Hence, we refer to the subsets with unpopular objects as *hot subsets* and we refer to the subsets with popular objects as *cold subsets*.

With hot and cold subsets enabled, objects evicted from FwLog are inserted into the hot subset. The cold subset is not typically written during insertion. Every n nest packing operations on a subset, both the hot and cold subsets are read. In memory, these subsets are merged and redivided by object popularity, as seen in Fig. 9. Any popular objects found in the

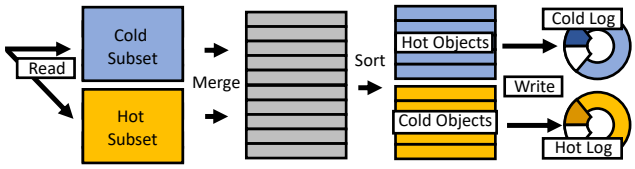


Fig. 9: FwSets *is split in two: hot subsets with cold objects and cold subsets with hot objects. Most of the time objects are inserted into the hot subset. However, every n subset updates, both subsets are read, merged, split by object popularity, and then both rewritten.*

hot subset are moved into the cold subset, since these objects are likely to remain in the cache for longer and do not need to be rewritten as frequently. The least popular objects found in the cold subset are moved into the hot subset so that FwSets can evict them if they remain sufficiently unpopular.

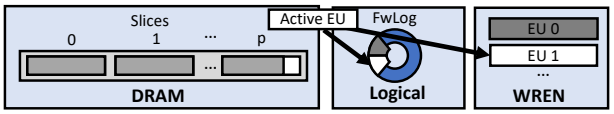
Hot-cold object separation can nearly halve FwSets's write amplification. If n is 5 and sets are 8 KB (two 4 KB subsets), FairyWren without hot-cold object separation would have to write all 8 KB on each insertion to a set. With hot-cold object separation, FairyWren writes 4 KB for the hot subset on every insertion, but only has to write 4 KB for cold subset on every fifth insertion. Thus, FwSets writes only 24 KB instead of 40 KB every five inserts to a set, a 40% reduction.

Theoretically, FairyWren could further reduce writes by further dividing sets. However, there are some practical limitations to this, namely that WREN devices only support a limited number of active EUs, often less than 10. FairyWren currently needs 4 active EUs: 1 for LOC, 1 for FwLog, and 2 for FwSets. Using only 4 active EUs allows FairyWren to run concurrently with other programs on the flash without interference and ensures compatibility with a wide range of WREN devices while still achieving low write rates.

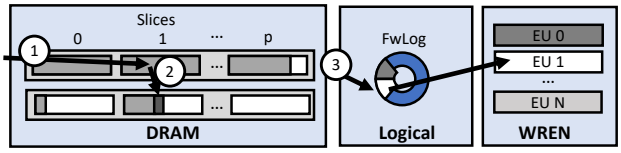
Moreover, separating objects by popularity yields diminishing returns since it increases miss ratio, which will then require more cache capacity to reduce the miss ratio. Wrong object-popularity predictions, which are frequent since very few bits of metadata are used to track each object's popularity, can lead to increases in both writes and miss ratio. The miss ratio will increase if popular objects are placed in hot subsets and evicted prematurely. This type of error becomes more frequent as one tries to separate objects by popularity at finer ganularity. In fact, even our single layer of hot-cold separations causes a modest increase in miss ratio (Sec. 6.5).

Minimizing DRAM in FwLog by slicing. Like Kangaroo [67, 68], FwLog is implemented as 64 *slices*, i.e., 64 independent log-structured caches that operate in parallel over subsets of the keyspace. This is done to save $\log_2 64 = 6$ bits per flash pointer in the DRAM index.

A naïve implementation of slicing on WREN would require one active EU for each slice. Many WREN devices do not permit 64 simultaneously active EUs due to the prohibitively large DRAM overhead this would impose on the flash device. Instead, FwLog uses a single active EU and shares segments among all 64 slices, giving each slice an equal static share of



(a) Naïve FwLog partitioning.



(b) Inserting to FwLog with overflow memory buffering.

Fig. 10: FwLog *uses partitioning to minimize memory and overflow buffers to ensure the log segments are full.*

each segment (Fig. 10a). The downside of sharing FwLog segments is that one slice could fill up its share of the segment before the others. In the worst case, one slice fills before the others contain any objects, causing internal fragmentation in FwLog. This fragmentation reduces FwLog's ability to minimize WA in FwSets. Via simulation and stochastic models, we found that fragmentation could exceed 20%.

FwLog reduces fragmentation via double buffering (Fig. 10b). On insertion, FwLog ① attempts to insert an object into its slice in the "primary" segment buffer. If the primary is full, ② the object is inserted into its slice in the secondary, "overflow" segment buffer. ③ When any slice in the overflow buffer becomes more than half full, FwLog writes the primary buffer to flash. The overflow buffer then becomes the new primary buffer and vice versa. Double buffering increases the number of objects seen before a buffer is written, reducing the variance in the number of objects in each slice. Using balls-and-bins [74] to approximate the maximum objects in a slice, we find that this optimization limits the capacity loss from fragmentation to <1%, even for small (16 MB) buffers.

Minimizing DRAM in FwSets by slicing. Like FwLog, FwSets also slices the log-structured store to reduce DRAM overhead, sharing segments to minimize active EUs and segment buffers. However, since sets are much larger than individual objects, FwSets is more susceptible to internal fragmentation than FwLog. FwSets therefore uses only 8 slices.

Reducing DRAM in FwSets by using larger sets. Finally, FwSets further reduces DRAM by using sets larger than 4 KB, reducing the number of sets that need to be tracked proportionally. Naïvely, one might expect that increasing set size would increase flash writes. In a pure set-associative cache, this would be true. However, FwLog buffers objects, and the number of objects that hash to a set also increases proportionally with set size, so FwSets's writes are roughly independent of set size. We see only a 5% increase in WA when going from 8 KB to 16 KB sets with a 4 KB hot subset and a 12 KB cold subset.

DRAM overhead breakdown. Compared to a LBAD set-associative cache, FwSets requires additional DRAM to track sets. Hot-cold object separation compounds this effect, doubling the number of (sub)sets to track.

Component	Kangaroo	Naïve SOC	SOC
Log total	*48 bits/obj*	*48 bits/obj*	*48 bits/obj*
Set index	–	≈ ***3.1 b***	≈ ***1.4 b***
Sets (other)	4 b	4 b	4 b
Sets total	*4 bits/obj*	*7.1 bits/obj*	*5.4 bits/obj*
Log metadata	≈ 0.8 b	≈ 0.8 b	≈ 0.8 b
Log size	5% = 2.4 b	5% = 2.4 b	5% = 2.4 b
Set size	95% = 3.8 b	95% = 6.7 b	95% = 5.1 b
Total	**7.0 bits/obj**	**9.9 bits/obj**	**8.3 bits/obj**

Table 2: Kangaroo and FairyWREN's SOC's DRAM overhead for a 2 TB small-object cache with a 5% log. Despite tracking sets, FairyWREN's SOC still needs fewer than 10 bits per object.

Table 2 shows the per-object DRAM overhead for Kangaroo and FairyWREN's SOC. Due to partitioning and double buffering, FairyWREN achieves the same log overhead as Kangaroo. FairyWREN's added overhead shows up in FwSets. Naïvely, when FairyWREN has 4 KB subsets and 200 B objects, each set would need 8 bytes, for 3.1 bits/obj. However, since FairyWREN uses 8 KB subsets and slices FwSets in eighths, FwSets needs just 1.4 bits/obj to track sets.

FairyWREN uses 19% more DRAM than Kangaroo, a 1.5 GB DRAM overhead increase for a 2 TB cache. However, FairyWREN's DRAM overhead is still much lower than a log-structured cache, and this modest DRAM increase allows FairyWREN to greatly decrease flash writes (by 12.5×), netting large savings in carbon emissions and cost.

6 Evaluation

We compare FairyWREN to prior flash caches and find that: (1) FairyWREN reduces flash writes by 92% over the research state-of-the-art Kangaroo, leading to a 33% carbon reduction and a 35% cost reduction, (2) FairyWREN is within 11% of the minimum write rate, and (3) FairyWREN is the first cache design to actually benefit from QLC.

6.1 Experimental setup and model

Implementation. 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

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: FairyWREN and Kangaroo experiment parameters.

	SLC	MLC	TLC	QLC	PLC
Write endurance	4.4×	4×	1×	0.32×	0.16×
Capacity discount	3×	1.5×	1×	0.75×	0.6×

Table 4: Scaling factors for different flash densities. We optimistically assume that increasing the bits per cell does not affect emissions or cost.

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 [67] 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). Both caches use 400 GB of flash capacity and achieve similar miss ratios as Kangaroo's production experiments [67]. We overprovision FWSets by 5% to ensure forward progress during nest packing, giving several free EUs to the FWSets log-structured store. Thus, FairyWREN effectively uses 383 GB. This idle capacity should decrease in larger flash devices. Kangaroo only uses 376 GB of capacity due to device-level overprovisioning. We approximate Kangaroo's DLWA based on results in the Kangaroo paper [67].

Simulation. In addition to flash experiments, we implemented a simulator to compare a much wider range of possible configurations for FairyWREN. The simulator replays a scaled-down trace to measure writes and misses from each level of the cache, including the LOC, FWLog, and FWSets.

We evaluate our cache in simulation on a 21-day trace from Meta [16] and a 7-day trace from Twitter [92]. The Meta trace accesses 6 TB of unique bytes with a 13.8% compulsory miss ratio and an average object size of 395 bytes. Small objects (<2 KB) are 95.2% of requests, and these requests account for 60.2% of bytes requested. The Twitter trace accesses 3.5 TB of unique bytes, has a 17.2% compulsory miss ratio, and an average object size of 265 bytes. Small objects are >99% of requests, and these requests account for >99% of bytes requested. Both of these traces are higher fidelity than the open-source traces [16, 92]. We present results for the last 2 days of the trace.

Carbon emissions and cost model. We evaluate carbon emissions and cost while varying cache configuration, flash density, and device lifetime. 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. We normalize all results based on device lifetime and we assume that all required flash is purchased at the beginning of deployment.

We estimate emissions and cost from the total flash needed to cover both the cache's capacity and its writes over the desired lifetime. For example, a 2 TB cache with a 6-year lifetime will require at least 2 TBs of flash, but it may require 2.5 TB of flash to accommodate the cache's write rate over 6 years. LBAD devices use 7% overprovisioning, the standard on datacenter drives [8].

We base our write endurance and cost projections on Micron 7300 NVMe U.2 TLC SSDs. For other densities, we multiply the TLC write endurance by the write-endurance factors in Table 4, based on [9]. For cost, we interpolate linearly between flash capacities and include power as the operational expense. Cost is normalized to Kangaroo with a 30% miss ratio for the Twitter trace and 20% for Meta. We optimistically assume that different flash densities will have the same cost and emissions per cell; e.g., 1 TB of PLC has the same emissions as 600 GB of TLC (5:3 ratio). Our model can incorporate more data on denser flash if it becomes available.

We use the ACT model [38] to estimate operational and embodied emissions from CPUs, DDR4 DRAM, and flash. We assume the grid is a 50/50 mix of wind and solar, a common renewable-energy mix [12].

6.2 Carbon emissions of flash caches

We first examine the carbon emissions of different flash caches for a 6-year deployment. Fig. 11 compares FairyWREN to three systems: Minimum Writes, Kangaroo, and a Flashield-like log-structured cache [35]. Minimum Writes is an unachievable, idealized cache with WA of 1× and no DRAM overhead. Flashield also assumes a WA of 1×, but requires a DRAM:SSD capacity ratio of 1:10, as originally proposed. Since we cannot faithfully replicate Flashield's ML eviction policy (and no working implementation is available), we assume that Flashield achieves FairyWREN's miss ratios.

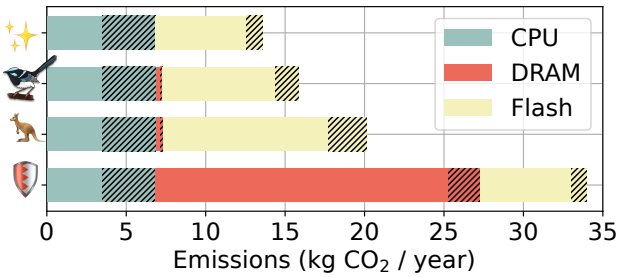
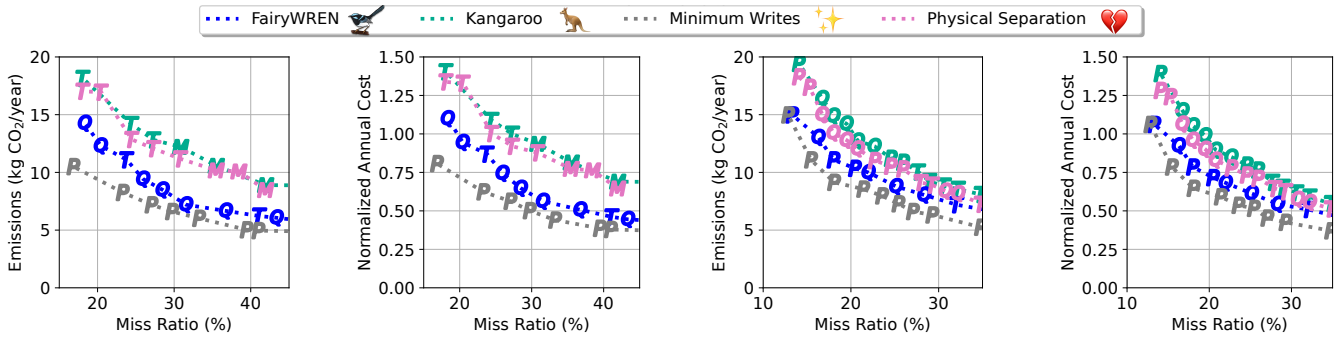


Fig. 11: Yearly carbon emissions for 4 caching systems: minimum writes (✨) with a write amplification of 1 with no additional DRAM, FairyWREN (🐦), Kangaroo (🦘), and a Flashield-like log-structured cache (🛡). Our results include the embodied and operational (hatched) emissions from CPU, DRAM, and flash.

Takeaway 0: *Sustainable flash caches must use much less DRAM than log-structured cache designs.*

Although we optimistically assumed that Flashield incurs no write amplification, Flashield's overall carbon emissions are 1.7× higher than Kangaroo's. These emissions are due to its high DRAM overhead, despite several optimizations in Flashield designed to save DRAM. High DRAM overhead is unfortunately inherent in the design of a log-structured cache.

Kangaroo reduces DRAM overhead through its hierarchical design. Unfortunately, Kangaroo also incurs a far higher write rate than a log-structured cache. Kangaroo accounts for its



(a) Twitter Carbon Emissions *(b) Twitter Cost* *(c) Meta Carbon Emissions* *(d) Meta Cost*

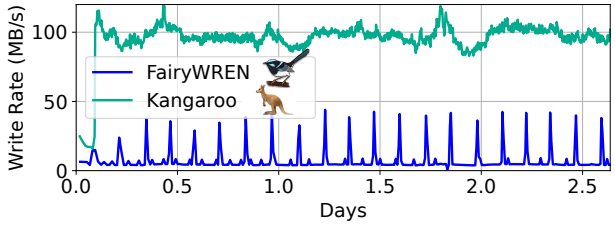
Fig. 13: The emissions and cost over six years for Kangaroo (🦘), FairyWREN (🧚), Min. Writes (✨), and Physical Sep. (💔).

increased writes by overprovisioning flash capacity, increasing embodied carbon emissions. While Kangaroo is far more sustainable than Flashield, it leaves room for improvement.

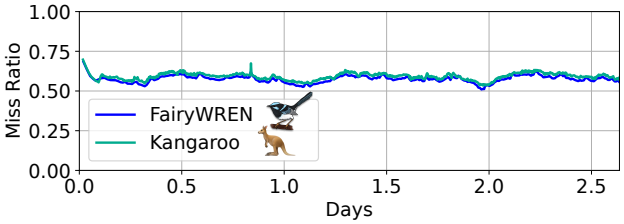
FairyWREN maintains Kangaroo's low memory overhead while greatly reducing the flash write rate. Consequently, FairyWREN reduces overall carbon emissions by 21.2% compared to Kangaroo. As this improvement comes from reducing flash emissions, we focus on flash emissions for the remainder of the evaluation.

6.3 On-flash experiments

To study how FairyWREN reduces flash writes, we evaluate FairyWREN on real flash drives using the setup in Sec. 6.1.



(a) Write rate (Mean: FairyWREN ≈7.8MB/s, Kangaroo ≈97 MB/s)



(b) Miss ratio (Mean: FairyWREN ≈ 0.575, Kangaroo ≈ 0.594)

Fig. 12: The miss ratio and write rate for Kangaroo and FairyWREN.

Takeaway 1: *FairyWREN greatly reduces flash writes while maintaining a slightly better miss ratio than Kangaroo.*

Fig. 12 plots the flash write rate and miss ratio over time for Kangaroo and FairyWREN. 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 more frequently, at 256 KB granularity.

The main goal of FairyWREN is to reduce writes, enabling the use of denser flash. In Fig. 12a, FairyWREN reduces writes by 12.5× over Kangaroo, from 97 MB/s to 7.8 MB/s. To achieve this, FairyWREN leverages WREN to combine cache logic and GC and to separate writes of different lifetimes.

However, reducing writes must not increase misses. Fig. 12b shows that, in fact, FairyWREN and Kangaroo have very similar miss ratios: on average, 0.575 for FairyWREN vs 0.594 for Kangaroo. FairyWREN's small advantage comes from reducing idle capacity due to overprovisioning.

We see very similar results for write amplification: a 12.2× reduction, from 23× in Kangaroo to 1.89× in FairyWREN. The slight difference between the write rate and WA comes from FairyWREN's slightly better miss ratio.

Takeaway 2: *FairyWREN outperforms Kangaroo for both throughput and read latency at peak load.*

While the primary performance metric for caches is miss ratio, FairyWREN must provide enough throughput that it does not require more servers — and thus more carbon emissions — to handle the same load. In our experiments, FairyWREN's throughput is 104 KOps/s whereas Kangaroo's is 40.5 KOps/s. FairyWREN's significant throughput increase is mostly due to lower write rate, but also due to better engineering that moved work off the critical path for lookups and inserts.

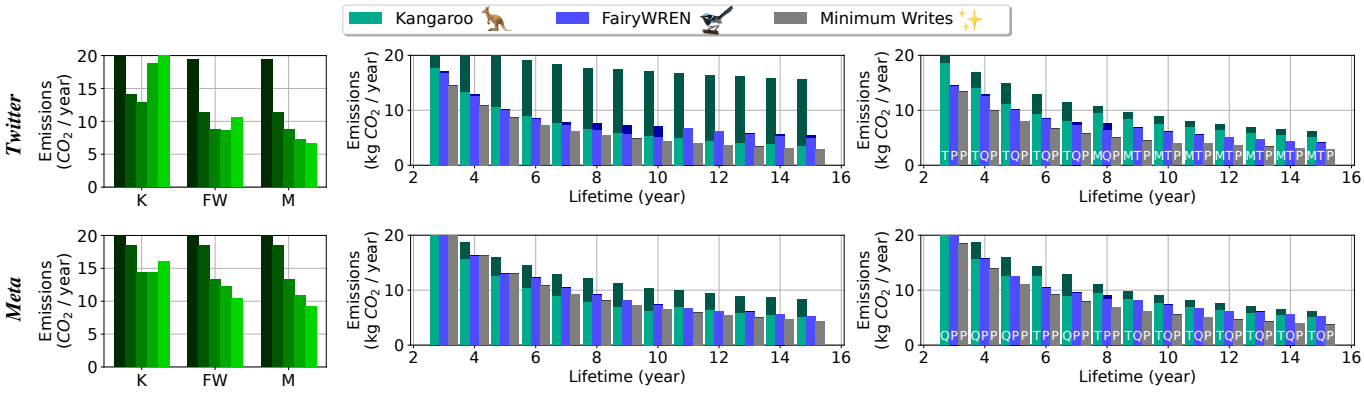
Similarly, we find that FairyWREN's and Kangaroo's 99th-percentile latencies are 170 µs and 1,370 µs, respectively. But note that, in practice, the overall tail latency is set by the backing store, not the flash cache.

6.4 FairyWREN reduces carbon emissions

We now evaluate carbon emissions and cost via simulation, comparing FairyWREN (🧚), Kangaroo (🦘), Minimum Writes (✨), and Physical Separation (💔). Physical Separation represents Kangaroo on WREN, where each cache component (e.g., LOC, KLog, KSet) is placed in its own EU to separate traffic and thereby allow LOC and KLog to have WA of 1×.

Takeaway 3: *FairyWREN's reduced writes translate into reduced carbon emissions and reduced cost across miss ratios.*

Fig. 13 plots emissions and cost for a 6-year lifetime vs. miss ratio over a wide range of cache configurations. Each point is labeled with the flash density used (e.g., TLC).



(a) Impact of density (6 years) *(b) Impact of lifetime (on QLC)* *(c) Impact of lifetime (on best flash density)*

Fig. 14: The carbon emissions to achieve a 30% miss ratio on Twitter trace or 20% miss ratio on Meta trace for (a) 6 years on SLC (darkest) to PLC (lightest), (b) different lifetimes with QLC flash, and (c) different lifetimes with any flash density. For (b) and (c), the darker part of each bar represents emissions due to overprovisioning.

For the Twitter traces (Fig. 13a, Fig. 13b), Kangaroo is limited to either MLC or TLC due to its high write rate, and likewise for Physical Separation because it does not reduce writes by much (Sec. 6.5). Meanwhile, FairyWREN leverages its low WA to use mostly QLC across miss ratios, giving it large carbon and cost reductions vs. Kangaroo. However, FairyWREN still has too many writes to use PLC. While the gap between Minimum Writes and FairyWREN grows at low miss ratios, there is only a 10.1% difference in their emissions at 20% miss ratio and a 7.7% difference in cost.

The Meta traces (Fig. 13c, Fig. 13d) are less write-intensive. However, even here we see that FairyWREN reduces cache emissions and cost compared to both Kangaroo and Physical Separation. In this case, FairyWREN is able to lower the write rate sufficiently to use QLC and PLC. As a result, FairyWREN performs close to Minimum Writes, even at low miss ratios.

Takeaway 4: *FairyWREN benefits from using denser flash when Kangaroo cannot.*

Flash devices are becoming denser over time (Sec. 2). Fig. 14a shows the carbon-optimal cache configurations over a 6-year lifetime at a target miss ratio of 30% for Twitter and 20% for Meta, varying flash density from SLC (left) to PLC (right). Kangaroo performs best when using TLC on the Twitter trace and QLC on the Meta trace. Using PLC increases Kangaroo's emissions due to the excessive overprovisioning needed to compensate for PLC's lower write endurance. FairyWREN's lower write rate enables it to use QLC for Twitter and PLC for Meta, reducing emissions and cost. Since Twitter's trace is more write-intensive, using PLC increases carbon emissions by 24% due to overprovisioning.

For Minimum Writes on Twitter, emissions decrease by 17% going from TLC to QLC and by 8% from QLC to PLC. On Meta, emissions reduce by 18% and 15%. While these numbers show that denser flash reduces emissions, they suggest diminishing returns even for an optimal cache.

Takeaway 5: *FairyWREN's low WA allows it to avoid massive overprovisioning on dense flash as lifetime is increased.*

To explore the trend of increasing device lifetime (Sec. 2), Fig. 14b considers the emissions for caches on QLC devices, showing emissions from overprovisioning in a darker shade.

For a 6-year lifetime, Kangaroo requires 2.2× the emissions of FairyWREN on Twitter and 1.17× on Meta. At 12 years, the gap increases to 2.6× and 1.54×. Due to the DLWA in LBAD devices, Kangaroo's emissions are lowest when it has some amount of overprovisioning. FairyWREN does not need this overprovisioning due to its lower WA.

Takeaway 6: *Increasing flash density does not necessarily improve sustainability, as lifetime matters more than density.*

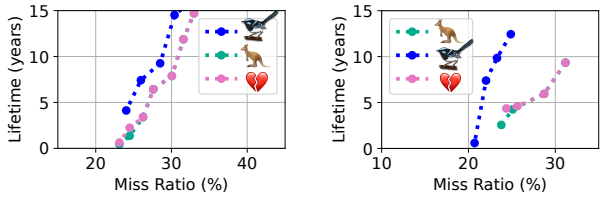
To minimize emissions, we need to optimize both lifetime and flash density. Fig. 14c shows each system's emissions for all lifetimes, with the best density displayed on each bar. Kangaroo usually prefers MLC and TLC because, to provide enough write endurance, QLC and PLC require too much overprovisioning. FairyWREN has fewer emissions than Kangaroo at all lifetimes and stays within 30% of Minimum Writes.

The best flash density decreases for longer lifetimes. FairyWREN prefers PLC on Twitter over 3 years, but TLC over 9 years. At these long lifetimes, the reduced write endurance of denser flash outweighs its sustainability benefits, and extending lifetime is more important than using denser flash.

Takeaway 7: *For a given flash device, FairyWREN extends lifetime by at least a couple of years.*

So far, we have evaluated emissions when deploying the optimal drive for a given lifetime and flash density. However, flash deployments are often constrained to specific devices with a pre-determined capacity and density. In these situations, emissions reductions come from extending lifetime. Fig. 15 evaluates device lifetime for a 3.6 TB drive at different miss ratios. Compared to Kangaroo, FairyWREN is able to extend the device's lifetime by at least 2 years and by over 5 years

on the Meta trace. By contrast, Physical Separation barely improves lifetime vs. Kangaroo.



(a) Twitter *(b) Meta*

Fig. 15: The lifetimes for a 3.6 TB cache for Kangaroo (🦘), FairyWREN (🧹), and Physical Separation (💔).

6.5 Where are benefits coming from?

We next explore how FairyWREN's optimizations contribute to its write rate reduction. Fig. 16 shows the write rate on the Twitter trace starting with Kangaroo on LBAD (Log + Sets). We then add the optimizations of FairyWREN incrementally. First, we port Kangaroo naively to WREN (+WREN), then we physically separate the large and small objects into different erase units (+Physical Sep.). Then we add nest packing (+Nest Packing), and, finally, hot-cold object separation (+Hot-Cold) to realize FairyWREN. We first present the write rates for the different systems across different capacities and miss ratios, showing the emissions-optimal flash density for one capacity. We then show how the lifetimes of each design would vary if deployed on a QLC drive.

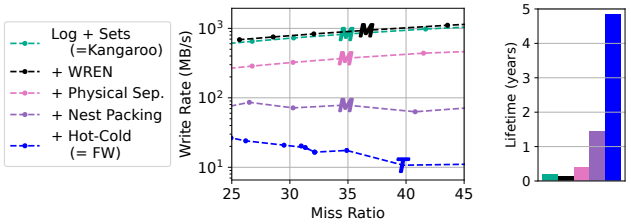


Fig. 16: Write rate (log-scale) and lifetime breakdown on the Twitter trace, incrementally adding optimizations to go from Kangaroo to FairyWREN.

Takeaway 8: *Caches on optimal LBAD devices cannot achieve the same write rate as FairyWREN.*

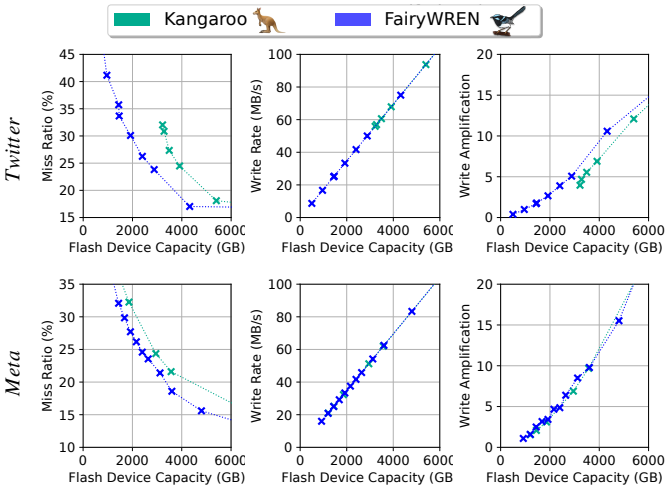
Three of the lines in Fig. 16 are achievable with LBAD devices: Log + Sets, +WREN, and +Physical Sep. Log + Sets represents the current Kangroo implementation on LBAD. +WREN is a naive port of Kangaroo to WREN devices that redirects all cache writes to a single log-structured store using FIFO garbage collection. This naïve port does not attempt any separation of objects by expected lifetime, and we assume it has the same WA as Kangaroo. However, current LBAD devices do try to separate objects belonging to different, concurrent streams, so one would expect an LBAD device to perform, in practice, somewhere between +WREN and +Physical Sep. But even in the best case, Physical Sep. still incurs far too many writes, limiting the lifetime of a QLC device to less than half a year.

Takeaway 9: *Both nest packing and hot-cold object separation are essential to FairyWREN's write reduction.*

The other two systems we compare in this breakdown are +Nest packing and +Hot-Cold (i.e., FairyWREN with all optimizations). Nest packing reduces writes by at least 3.7× and hot-cold object separation reduces writes by another 3.4×. We also observe that, while hot-cold separation can increase miss ratios, the reduction in write rate outweighs this increase, leading to a 33× increase in QLC lifetime over the Kangaroo baseline and a 13× increase over +Physical Sep.

6.6 Operating on a fixed flash device

We now compare Kangaroo and FairyWREN with respect to miss ratio given a fixed flash capacity. We enforce the same constraints of a 6-year flash lifetime, TLC flash density, and 32 GB of DRAM for both systems. Unlike prior figures where we minimize emissions, FairyWREN cannot not gain an advantage for using denser flash, and Kangaroo cannot increase write endurance by using less-dense flash. We show that FairyWREN under the same capacity constraints, and thus write rate constraints, improves miss ratio over Kangaroo through its reduction in writes allowing it to more effectively use the capacity.



(a) Capacity *(b) Write Rate* *(c) WA*

Fig. 17: Pareto curve of cache miss ratio at different flash device sizes and the corresponding write rate and write amplification of these points. The DRAM capacity is limited to 32 GB, the desired lifetime is 6 years, and the caches use TLC flash.

Takeaway 10: *FairyWREN achieves the same miss ratio at lower flash capacities than Kangaroo.*

Fig. 17 shows the effects of changing the flash capacity on miss ratio for both traces. For each flash capacity, we also plot the write rate and WA of both systems. We find that FairyWREN needs less flash capacity than Kangaroo to achieve a given miss ratio. FairyWREN also requires less overprovisioning due to its lower write rate. This trend is more

prominent in the Twitter trace than the Meta trace, which is less write-intensive. For the Twitter trace, Kangaroo's use of TLC prevents it from achieving higher miss ratios. Kangaroo's higher write rate requires much more overprovisioning, increasing the overall flash capacity needed to survive 6 years above 3.6 TB.

We also see that flash capacity sets the write budget for the flash device, defining the write rate that the system can tolerate for a desired lifetime. One might expect a similar relationship for write amplification. However, the systems have different miss ratios, causing Kangaroo to need to have a lower WA through massive overprovisioning.

Takeaway 11: *FairyWREN maintains its advantage under a DRAM constraint.*

We investigated how DRAM restrictions affect Kangaroo and FairyWREN when both caches use 3.6 TB of TLC flash for a 6-year lifetime. FairyWREN maintains a constant miss ratio advantage over Kangaroo from 16 GB to 64 GB of DRAM for both traces. FairyWREN's miss ratio only begins to increase when DRAM falls to 8 GB on the Twitter trace. However, Kangaroo cannot handle the Twitter workload for 6 years with only 8 GB of DRAM. Hence, FairyWREN always outperforms Kangaroo in these experiments.

7 Related Work

This section discusses additional related work with similar techniques and goals to FairyWREN.

Hot-cold objects and deathtime. In caching, hot objects are the most popular objects. Caches use eviction policies to retain popular objects [15, 45, 47, 83]. FairyWREN adapts Kangaroo's RRIP-based eviction policy [45, 67].

Popularity is different than *deathtime*, the time when an object will be deleted [41]. To minimize GC, many storage systems will physically separate objects by their deathtime [26, 28, 41, 54, 76, 94]. Grouping objects with similar deathtimes reduces WA. Hence, accurately predicting deathtimes is vital for minimizing write amplification within LBAD. Recent work uses ML to make these predictions [26, 94]. Unfortunately, ML solutions require additional hardware that can increase emissions and cost.

Caches have more control over deathtimes than storage systems. Deathtimes are set by the eviction policy, and thus determining an object's deathtime is more straightforward. For instance, in caches that evict based on TTLs, the TTLs can be used to group objects [93]. FairyWREN leverages its eviction policy's popularity rankings and the WREN interface to physically group objects by deathtime.

Eviction and garbage collection. Prior flash caches have attempted to reduce in-device garbage collection. Many log-structured caches [27, 35, 56, 61] group objects into large segments and trim these segments during eviction to minimize garbage collection. These systems attempt to evict segments before device-level GC rewrites them. Unfortunately, this does not ensure GC is prevented on LBAD devices, so some work has proposed leveraging newer interfaces to guarantee alignment. DidaCache [78], for example, uses an Open-Channel SSD [20] to guarantee its segments will align with erase units. Other proposals to use more expressive interfaces re-implement LBAD-like GC on top of a ZNS SSD [29], prohibiting optimizations like FairyWREN's nest packing. All of these log-structured approaches suffer from high DRAM overheads and cannot evict individual objects without additional writes.

Grouping by object size. FairyWREN separates objects into two object size classes, large and small, similar to Kangaroo [68] and CacheLib [16]. This grouping is used to minimize memory overhead. Allocating memory using size-based slab classes is often used to reduce fragmentation [25, 43, 77, 78, 93]. Introducing additional object size classes in FairyWREN would result in additional flash accesses, since FairyWREN does not index the size classes to save memory. Instead, FairyWREN reduces fragmentation by grouping objects into either large segments in the LOC or sets in FWSets. These segments and sets are periodically rearranged to prevent fragmentation.

8 Conclusion

FairyWREN reduces flash's carbon emissions and cost by integrating flash management with cache policies. Doing so requires redesigning the cache to transition from old LBAD flash interfaces to a WREN interface. Experiments show that FairyWREN decreases flash writes by 12.5× vs. the state-of-the-art, allowing longer flash lifetimes that reduce carbon emissions by 33% and cost by 35%.

9 Acknowledgements

Sara McAllister is supported by a NDSEG Fellowship. We thank the members and companies of the PDL Consortium (Amazon, Google, Hitachi, Honda, IBM Research, Intel, Jane Street, Meta, Microsoft Research, Oracle, Pure Storage, Salesforce, Samsung, Two Sigma, Western Digital) for their interest, insights, feedback, and support. We thank our shepherd, Gala Yadgar, and our anonymous reviewers for their helpful comments and suggestions. We also thank Western Digital for providing resources and technical expertise; we especially thank Matias Bjørling, Ajay Joshi, and Hans Holmberg. We also specifically thank Javier Gonzalez and Mike Allison, at Samsung, and Ross Stenfort, at Meta, for providing their technical expertise on FDP. We thank the PDL staff, particularly Jason Bowles, for their support.

References

[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] Fatcache. https://github.com/twitter/fatcache.

[4] Flash prices. https://jcmit.net/flashprice.htm.

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

[6] Memory prices. https://jcmit.net/memoryprice.htm.

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

[8] Ssd over-provisioning and its benefits. https://www.seagate.com/blog/ssd-over-provisioning-benefits-master-ti/.

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

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

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

[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] 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.

[15] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd: Improving hit rate by maximizing hit density. In *USENIX NSDI*, 2018.

[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, 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.

[18] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *USENIX NSDI*, 2017.

[19] 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.

[20] 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.

[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] Erik Brunvand, Donald Kline, and Alex K. Jones. Dark silicon considered harmful: A case for truly green computing. In *2018 Ninth International Green and Sustainable Computing Conference (IGSC)*, 2018.

[25] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. Faster slab reassignment in memcached. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 353–362, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, Haifa Israel, June 2021. ACM.

[27] 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.

[28] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, 29(3):267–290, 1999.

[29] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. A new LSM-style garbage collection scheme for ZNS SSDs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020.

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

[31] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, 2017.

[32] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, 2018.

[33] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *ACM SIGMOD*, 2011.

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

[35] 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.

[36] Alex Gartrell, Mohan Srinivasan, Bryan Alger, and Kumar Sundararajan. Mcdipper: A key-value cache for flash storage. `https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/`.

[37] 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.

[38] 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.

[39] 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.

[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.

[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] Amy Hood, July 2022.

[43] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, Santa Clara, CA, July 2015. USENIX Association.

[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.

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

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

[47] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[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] 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.

[52] 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.

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

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

[55] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[56] 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.

[57] 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.

[58] 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.

[59] 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-using-aws/`.

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

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

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic 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.

[70] 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.

[71] 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.

[72] 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.

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

[74] Martin Raab and Angelika Steger. Balls into Bins: A Simple and Tight Analysis. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Michael Luby, JosÃ© D. P. Rolim, and Maria Serna, editors, *Randomization and Approximation Techniques in Computer Science*, volume 1518, pages 159–170. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. Series Title: Lecture Notes in Computer Science.

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

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

[77] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for DRAM-based storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, February 2014. USENIX Association.

[78] 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.

[79] Shigeru Shiratake. Scaling and performance challenges of future dram. In *2020 IEEE International Memory Workshop (IMW)*, pages 1–3, 2020.

[80] 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`.

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

[82] 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.

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

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

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

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

[87] Benny Van Houdt. A mean field model for a class of garbage collection algorithms in flash-based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):191–202, 2013.

[88] 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.

[89] 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.

[90] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *USENIX ATC*, 2015.

[91] 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.

[92] 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.

[93] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *USENIX NSDI*, 2021.

[94] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. Reducing garbage collection overhead in SSD based on workload prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[95] 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.

[96] 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.

Appendix A Modeling of DLWA Under Random Writes

Our goal is to model the effect of EU size on DLWA. Specifically, we want to analyze the performance of the FIFO+ GC policy, which selects EUs for garbage collection in FIFO order and skips EUs which contain only valid data. Several prior papers [34,46,87] have noted that DLWA can be approximated using W Lambert functions, but this prior work tend to focus on the level of device overprovisioning rather than on the EU size. We use an approach similar to that of [46] to model the relationship between EU size and DLWA under FIFO+.

We define X to be the random variable representing the number of invalid pages an EU that is targeted for garbage collection. Because FIFO+ will erase an EU only if it contains invalid pages, our goal is to approximate $\mathbb{E}[X|X>0]$. This tells us the number of new pages that can be written every time GC is performed. Hence, if we let b be the number of pages in an EU, we can compute the DLWA as

$$\text{DLWA} = \frac{b}{\mathbb{E}[X|X>0]}. \tag{1}$$

Our approximation makes two simplifying assumptions.

First, we assume that each of the b pages in the target EU is invalid independently with probability p. This is reasonable when writes are random and the total number of pages in the device is large. This assumption implies that $X \sim \text{Binomial}(b,p)$. To approximate the expectation of X, we must approximate p.

Second, we assume that an EU is targeted for GC every k writes, where k is a constant. Specifically, we define t to be the total number of EUs in the device and assume $k = t\mathbb{E}[X]$. This is a reasonable approximation because k is the expected number of writes that occur between GC operations on a given EU and the total number of EUs, t is large. A particular page will be invalid if at least one of the k writes targets the page. Hence, the probability p that a page is invalid is

$$p = 1 - \left(1 - \frac{1}{ub}\right)^k$$

where u is the number of EUs available to store valid user data. Note that u is typically smaller than t, and $\frac{t}{u}$ represents the amount of overprovisioning in the device.

Combining these assumptions yields

$$\mathbb{E}[X] \approx b \cdot p \approx b\left(1 - \left(1 - \frac{1}{ub}\right)^k\right) \tag{2}$$

$$\approx b\left(1 - \left(1 - \frac{1}{ub}\right)^{t\mathbb{E}[X]}\right). \tag{3}$$

We can rewrite (3) using the W Lambert function to get the following approximation for $\mathbb{E}[X]$:

$$\mathbb{E}[X] = b - \frac{W\left(bt\left(1-\frac{1}{ub}\right)^{tb}\ln\left(1-\frac{1}{ub}\right)\right)}{t\ln\left(1-\frac{1}{ub}\right)}.$$

To compute $\mathbb{E}[X \mid X>0]$, we note that

$$\mathbb{E}[X \mid X>0] = \sum_{i=1}^{b} i \cdot \frac{P(X=i)}{P(X>0)} = \frac{1}{P(X>0)} \sum_{i=0}^{b} i \cdot P(X=i)$$

and thus

$$\mathbb{E}[X \mid X>0] = \frac{\mathbb{E}[X]}{P(X>0)} = \frac{\mathbb{E}[X]}{1-(1-p)^k}.$$

Hence, we now have an approximation that allows us to write DLWA as defined in (1) in terms of the device parameters t, u, and b. This approximation is validated against simulation in Figure 6.