

Toward Sustainable Datacenters through Efficient Data Retrieval

Sara McAllister

CMU-CS-25-126

August 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Nathan Beckmann, Co-Chair

Gregory R. Ganger, Co-Chair

George Amvrosiadis

Daniel S. Berger, Microsoft Azure & University of Washington

Margo Seltzer, University of British Columbia

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

Copyright © 2025 Sara McAllister

This work is supported by the National Science Foundation under award numbers, CNS-2402838 and CMMI-1938909, and CSR-1763701, as well as a National Defense Science and Engineering Graduate (NDSEG) Fellowship and a Siebel Scholarship.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Datacenters, sustainability, storage, caching, flash, hard disk drives

*To my grandmother, Jane, who showed me that anyone can be a computer scientist and
to my parents, who encouraged me to follow my dreams.*

Abstract

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

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

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

Acknowledgments

Getting my doctorate was a long and hard journey. I would not have been able to do it without the many people who supported me throughout the journey, including my advisors, family, collaborators, mentors, and friends. There's too many of you all to mention everyone, but I am deeply thankful that you are all in my life.

Thanks first and foremost to my advisors: Nathan Beckmann and Greg Ganger. I could not have asked for better advisors — you helped me figure out every stage of PhD and set me up to succeed. You were both always there to the questions that came up as I went through the PhD. I hope that I can live up to both of your examples when I advise students. Nathan, thanks for helping me figure out how to craft a compelling research story and encouraging me to discover the more fundamental math principles that underline my work (and for making sure I appropriately used m-dashes instead of n-dashes). I also appreciate your strong support of my work to build a more inclusive program. Greg, thanks for your wealth of wisdom. I am continually impressed at your ability to find very on-point pieces of advice to give at each stage of my PhD, whether that be refocusing me on research, instilling confidence, or just that the text on my slides is too small.

I also want to thank the other members of my committee: George Amvrosiadis, Daniel Berger, and Margo Seltzer. George, thanks for joining the team starting with FairyWREN and always having sound advice on research and beyond. Daniel, I am grateful that we met when you were still a post-doc at CMU. You showed me the ropes of how to do systems research, and it has been a pleasure continuing to work with you and brainstorming about fun research ideas. And finally, but not least, thanks to Margo for agreeing to join my committee even though we had only met briefly at SOSP. You have been an invaluable source of feedback on my research and encouragement as I have gone through the last stage of my PhD and applying to academic jobs.

I also need to thank all my collaborators. I believe systems research often requires a group and I enjoyed working with all of you. Ben Berg, for helping me on research throughout the PhD and always helping out when I got stuck with mathematical modeling. Thanks to Sanjith Athlur, Tim Kim, and Theo Gregersen for going on the Declarative IO adventure with me. Thanks also all my co-authors for the work that ended up in this thesis who I have not yet mentioned: Ricardo Bianchini, Yiwei Chen, Rodrigo Fonseca, Kali Frost, Sathya Gunasekar, Saurabh Kadekodi, Fiodar Kazhamiaka, Jimmy Lu, Sarvesh Tandon, Julian Tutuncu-Macias, Arif Merchant, Aaron Ogus, Maneesh Sah, Rashmi Vinayak, Lucy Wang, Sherry Wang, and Juncheng Yang. To Sophia Cao, Akshath Karanam, and all the other students I have had the pleasure to work with throughout my PhD, meeting with you all was always a fun part of my week.

Thanks to everyone in industry who have given me feedback about this

research, including but by no means limited to Mike Allison, Matias Bjorling, Javier Gonzalez, Brian Gold, Hans Holmberg, Ajay Joshi, and Ross Stenfort. I 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.

Thanks also to all the staff who have ensured that everything in CSD kept running, including Deb Cavlovich, Jenn Landefeld, Matthew Stewart, and Charlotte Yano. Thanks to Karen Lindenfelser. I do not know how PDL would keep running without you, and thanks for making sure I always had the room reservations, talk slots, and everything else I needed to keep my research running. Thanks also to Jason Boles, Bill Courtright, Chad Dougherty, Mitch Franzos, and Joan Digney for helping me on everything from getting posters together to ensuring the servers I needed were running with the correct hardware. Thanks to all the CMU faculty who supported me, particularly as I went onto the job market. Thanks to everyone in PDL, CORGi group, and architecture lunch for the great feedback on research.

I also want to thank those who helped me get to CMU. Thanks to Geoff Kuenning who inspired me to pursue systems, supported trips to my first couple systems conferences, and introduced me to many people who are now my peers. Thanks to Don Porter for taking a chance on me when I was an undergraduate student who was tackling systems research for the first time. Thanks also to Colleen Lewis who helped me learn more about CS research and encouraged me to not limit where I applied for during the PhD application cycle, including recommending that I apply to CMU.

I also cannot leave out the invaluable support my friends have been throughout the PhD. I would not have made it through without you all. Bailey Flanagan, Ananya Joshi, and Catalina Vajiac, thanks for being my accomplices in this PhD journey. I will forever be impressed by what we accomplished together. Ray Ware, thanks for always being in to play more board games, there is no better board game buddy. Jennifer Brana, Rose Silver, Naama Ben-David, Katherine Kosaian, I could wish for no better office mates, and I would not have made it through without all your support. Thank you to Jessie Groson who was always on hand with either a meme or listening ear to get us both through another tough week. Dorian Chan, Justin Raizes, and Hugo Sadok, thank you for the good times from our first year on. Thanks also to everyone I have met in Pittsburgh through improv, you helped keep me sane throughout my PhD.

I also want to thank all my friends outside of Pittsburgh. Thanks to my Mudd crew of Charles Dawson, Maggie Gelber, Camille Goldman, Katie Gruenhagen, Alex Quinn, and Lydia Sylla for all the weekend chats and fall foliage trips. Maya Josyula, thank you for being a steadfast friend from when we met in kindergarten.

Last, thanks to my family — my parents, Lauri and Curtis, and my brother David. Thanks for always being there for me and encouraging me to ask ques-

tions and discover answers for them. Dad, thanks for encouraging me to try computer science even when I thought I would never choose it as a career. Mom, thanks for always being a source of support and inspiration, and starting my journey in research early by giving me some “light” reading on the latest flu research in high school. David, thanks for all the late night debates and keeping me up to date on what is happening in machine learning. Thanks also to my extended family for always supporting my journey through yet more years of school and specifically to my grandmother, Jane, for listening to all my PhD adventures and always believing in me. And finally, thanks to Bear, the best pandemic puppy.

Contents

1	Introduction	1
1.1	Overview of Contributions	3
1.1.1	Storage emissions (Ch. 3): What they are and how to reduce them .	3
1.1.2	Kangaroo (Ch. 4): Caching Billions of Tiny Objects in Flash	3
1.1.3	FairyWREN (Ch. 5): A Sustainable Cache with Write-Read-Erase Flash Interfaces	4
1.1.4	Declarative IO (Ch. 6): Scaling the IO-per-TB wall in Bulk Storage	4
2	Background	7
2.1	Data retrieval in the datacenter	7
2.2	Caching in the datacenter	8
2.2.1	Tiny objects are important and numerous	9
2.2.2	Caching tiny objects	9
2.2.3	Flash Solid State Drives (SSDs)	10
2.2.4	Challenges in flash caching	11
2.2.5	Shortcomings of existing solutions	14
2.3	Bulk storage	15
2.3.1	Hard-disk drives	16
2.3.2	How do we quantify IO on disk?	16
2.3.3	Higher-capacity HDDs are increasingly IO-bound	17
2.3.4	IO demand in distributed storage systems	18
2.4	Datacenter sustainability	19
2.4.1	Prior work on reducing storage emissions	20
2.4.2	Prior solutions to reduce compute emissions	21
3	Reducing storage emissions results in IO limitations	23
3.1	Where do storage emissions come from?	23
3.1.1	Operational emissions	23
3.1.2	Embodied emissions	24
3.2	How can datacenters reduce storage emissions?	25
3.2.1	Denser and longer lifetimes to reduce flash emissions	25
3.2.2	IO-per-capacity wall limits HDDs	26

4	Kangaroo: Caching Billions of Tiny Objects on Flash	29
4.1	Kangaroo Overview and Motivation	31
4.2	Theoretical Foundations of Kangaroo	33
4.2.1	Baseline set-associative cache	35
4.2.2	Add KLog, no admission policies	36
4.2.3	Add threshold admission before KSet	38
4.2.4	Add probabilistic admission before KLog	39
4.2.5	Modeling results	40
4.3	Kangaroo Design	41
4.3.1	Pre-flash admission to KLog	41
4.3.2	KLog	41
4.3.3	KLog \rightarrow KSet: Minimizing flash writes	44
4.3.4	KSet	44
4.4	Experimental Methodology	46
4.4.1	Kangaroo implementation and parameterization	47
4.4.2	Comparisons	47
4.4.3	Simulation	47
4.4.4	Workloads	48
4.4.5	Metrics	48
4.4.6	Scaling traces	48
4.5	Evaluation	52
4.5.1	Main result: Kangaroo significantly reduces misses vs. prior cache designs under realistic constraints	52
4.5.2	Kangaroo performs well as constraints change	53
4.5.3	Parameter sensitivity and benefit attribution	56
4.5.4	Production deployment test	60
5	FairyWREN: A Sustainable Cache for Emerging Write-Read-Erase Flash Interfaces	61
5.1	Sustainable design constraints in flash caching	64
5.2	Write-Read-Erase iNterfaces (WREN)	65
5.2.1	Today's interface is LBAD	65
5.2.2	Challenges of new interface design	65
5.2.3	What makes an interface WREN?	66
5.2.4	WREN alone is not a cure for WA	67
5.3	FairyWREN Overview and Design	69
5.3.1	Overview	69
5.3.2	The LOC	70
5.3.3	The SOC	71
5.3.4	Optimizing the SOC	72
5.4	Evaluation	78
5.4.1	Experimental setup	78
5.4.2	Carbon emissions and cost model	79
5.4.3	Carbon emissions of flash caches	80

5.4.4	On-flash experiments	81
5.4.5	FairyWREN reduces carbon emissions	82
5.4.6	Where are benefits coming from?	85
5.4.7	Operating on a fixed flash device	87
5.5	Related Work	89
6	Scaling the IO-per-TB wall with Declarative IO	91
6.1	Maintenance tasks	93
6.1.1	Maintenance tasks are essential	94
6.1.2	Challenges in reducing maintenance IO	96
6.1.3	Opportunity: Maintenance tasks are flexible	97
6.2	Declarative IO	98
6.2.1	Interface Overview	98
6.2.2	Interface Details	99
6.2.3	Converting maintenance tasks to Declarative IO	100
6.2.4	Consistency with Declarative IO	102
6.3	DINGOS Design	104
6.3.1	DINGOS Overview	104
6.3.2	Scheduling in DINGOS’s IO Planner	105
6.3.3	DINGOS’s dispatcher minimizes cache space	106
6.4	Evaluation	107
6.4.1	DINGOS on top of HDFS	107
6.4.2	DINGOS in Simulation	109
6.5	Related Work	112
7	Conclusion	115
7.1	Future work	116
7.1.1	Flash caching	116
7.1.2	Declarative IO	117
7.1.3	Sustainable storage	118
	Bibliography	121

List of Figures

2.1	Datacenter storage overview.	8
2.2	Overview of caching at Meta	9
2.3	Cost of flash and DRAM over time	10
2.4	Flash architecture	12
2.5	Effect of over-provisioning on write amplification	13
2.6	HDD IO supply vs demand	17
2.7	IO demand across the datacenter	18
2.8	Operational and embodied emissions at Azure	19
3.1	Flash carbon emissions vs write rate	26
3.2	HDDs bandwidth-per-TB is dropping	27
4.1	Overview of Kangaroo design and results	30
4.2	Looking up objects in Kangaroo	32
4.3	Inserting and evicting objects in Kangaroo	32
4.4	Markov Model for Kangaroo	35
4.5	Modeled ALWA in Kangaroo with different KLog sizes	40
4.6	Modeled ALWA in Kangaroo with different thresholds	41
4.7	Overview of KLog operations	42
4.8	Kangaroo’s RRIParoo eviction policy	46
4.9	Miss ratio over time	52
4.10	Miss ratio vs device-level write rates	54
4.11	Miss ratio vs flash capacity	55
4.12	Miss ratio vs device size	56
4.13	Miss ratio vs average object size	57
4.14	Sensitivity study on Kangaroo parameters	58
4.15	Production deployment of Kangaroo	59
5.1	Overview of FairyWREN results	63
5.2	DLWA for different EU sizes	69
5.3	Overview of FairyWREN architecture	70
5.4	Nest packing in FairyWREN’s small-object cache	72
5.5	FwSets architecture	73
5.6	FwLog architecture	74
5.7	FwLog space overhead comparison	76

5.8	FWLog with slicing	77
5.9	Caches' carbon emissions breakdown	81
5.12	Kangaroo vs FairyWREN	81
5.13	Cost and emissions for different miss ratios	83
5.14	Emissions for different flash densities	83
5.15	Emissions for different lifetimes	84
5.16	Emissions for different lifetimes and densities	85
5.17	Lifetimes vs miss ratios	85
5.18	FairyWREN benefit attribution	86
5.19	Miss ratio vs write rate vs write amplification	87
5.20	DRAM capacity vs miss ratio	88
6.1	Declarative IO exploits task flexibility to reduce IO	92
6.2	DINGOS architecture overview	93
6.3	Imperative IO architecture	94
6.4	declare call	99
6.5	Supporting files in Declarative IO	100
6.6	Declarative scrubbing	101
6.7	Declarative capacity balancing	101
6.8	Declarative LSM compaction	102
6.9	Block-file mappings	103
6.10	Block choice in scrubbing.	103
6.11	DINGOS overview.	105
6.12	DINGOS scheduler	107
6.13	Declarative vs imperative IO reads over time	108
6.14	CDF of blocks accessed by maintenance tasks	109
6.15	IO savings with different supply and demand	111
6.16	Cache size vs IO savings	112

List of Tables

2.1	SSD vs HDD Servers	16
3.1	Operational emissions at Azure	24
3.2	Embodied emissions at Azure	24
4.1	Variables in Kangaroo model	34
4.2	DRAM overhead in Kangaroo	43
4.3	Kangaroo’s default parameters	47
4.4	Key parameters in trace scaling methodology	49
5.1	Comparison of FairyWREN vs. prior cache designs	64
5.2	Variables in analytical model of FIFO+	67
5.3	FairyWREN memory overhead	77
5.4	Experimental parameters	78
5.5	Flash density scaling factors	79
6.1	Description of common maintenance tasks	95
6.2	Workload parameters	110

Chapter 1

Introduction

“Given the advancements in system energy efficiency and the increasing use of renewable energy, most carbon emissions now come from infrastructure and the hardware.”

Udit Gupta et al. [122]

DATACENTER CARBON EMISSIONS are on par with the aviation industry [155], and growing. In the next decade, datacenters will account for over 20% of the world’s carbon emissions [138]. By 2050, datacenters may account for up to 33% of the world’s carbon emissions [155]. To avoid this outcome, datacenters need to be more sustainable.

Both companies and governments are pushing to reduce datacenter emissions. In the next few decades, many companies — including Amazon [6], Google [13], Meta [34], and Microsoft [186] — want to achieve Net Zero, i.e., greenhouse gas emissions close to zero. Global government regulations are also requiring emissions reductions across industries, such as the EU’s Fit for 55 which aims for a 55% reduction in EU emissions by 2030 and climate-neutrality by 2050 [189].

Most emissions reduction efforts are focused on reducing datacenter energy usage and its carbon footprint. Many datacenters are adopting renewable energy sources such as solar and wind [34, 122, 173, 186]. Google, AWS, and Microsoft are expected to complete their renewable-energy transition by 2030 [91, 139, 165]. 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 [122]. To continue reducing datacenters’ carbon footprint, datacenters’ embodied emissions need to be reduced.

Why focus on storage? Existing work focuses primarily on reducing emissions of *general-purpose compute* [39, 73, 122, 123, 230, 241, 248, 251]. This focus will reduce a datacenter’s *energy* usage or operational emissions, but it fails to account for storage emissions. Researchers and practitioners have frequently considered storage a less important source of emissions. This could not be further from the truth.

Storage comprises a sizable portion of both operational and embodied carbon emissions in hyperscale datacenters. For instance, Azure’s storage-related emissions — including

storage racks and local storage devices — make up 33% of operational and 61% of embodied emissions [180, 251]. Storage racks alone account for 24% of operational and 45% of embodied emissions [251]. Thus, as datacenters continue to target compute emissions and deploy renewable energy, storage will dominate overall datacenter emissions due to storage’s embodied emissions.

How can we reduce storage systems emissions? Unfortunately, we cannot just use optimizations on compute emissions for storage, because storage has fundamentally different constraints, such as ensuring data durability and availability. While the high-level techniques — including reducing power consumption, shifting power consumption to regions and times where renewable energy is available [14, 39, 55, 207, 230, 255], using fewer devices [81, 98, 99, 117, 223], and extending device lifetime [174, 241, 248, 249, 250] — still apply to storage, they face different challenges and tradeoffs.

One primary difference is that storage has much higher embodied emissions than operational emissions. Unlike compute, to reduce storage emissions, we need to focus on reducing embodied emissions. There are two main strategies to reduce embodied emissions: (1) extend the lifetime of devices and (2) reduce the quantity of hardware, which in storage means using more dense storage technology to store the same amount of data with fewer devices.

Storage emissions face an IO bottleneck. These emission-reduction strategies are not without trade-offs. Most importantly and the focus of this thesis, these emission-reduction strategies, for both flash and hard drive systems, reduce IO. For flash devices, the IO bottleneck is writes. Flash wears out with each write, and eventually it becomes unusable. Thus, writes limit the lifetime of flash devices. This limitation is more restrictive for denser flash devices. Hard disk drives (HDDs) have a slightly different IO bottleneck — while they are becoming denser, they do not have more IO-per-second available. Thus, these HDDs are increasingly IO-bottlenecked, requiring more devices to achieve the same number of reads and writes per stored bit — negating any emissions benefit of denser devices.

To reduce storage emissions, we need to reduce IO. How can we reduce IO while maintaining the performance of our storage system? This dissertation shows that redesigning storage software to reduce IO is possible. We accomplish this IO reduction through increasing the utility of each IO to the device, e.g. having each read or write fulfill multiple objectives. Unfortunately, many of these optimizations are limited today by the IO interfaces — both at the device level and the distributed storage system level. Therefore, this dissertation considers what new interfaces are needed to support sustainable storage systems. More specifically, this dissertation demonstrates that:

Thesis Statement: *Reducing IO, through increasing the utility of each read and write and developing more expressive and symbiotic interfaces, enables more sustainable storage systems in datacenters.*

To support this statement, this dissertation first demonstrates that storage is essential to datacenter sustainability and that IO limits storage sustainability for both flash and hard disk drives (Ch. 3). Then, we introduce the flash cache, Kangaroo, which shows that we

can increase write utility in flash caches through leveraging hash collisions, reduces writes without sacrificing miss ratio (Ch. 4). Next, we discuss the flash cache, FairyWREN, which illustrates that to further reduce writes, we need to change the interface to flash devices and build a cache to leverage those changes (Ch. 5). Together, Kangaroo and FairyWREN enable flash caches with half of the emissions compared to prior work. Finally, we show that the same principles of increasing IO utility through better interfaces can be applied to distributed storage systems with Declarative IO, which enables expressing flexibility to reduce IO from maintenance tasks (Ch. 6).

1.1 Overview of Contributions

The insights and contributions of this dissertation are summarized below.

1.1.1 Storage emissions (Ch. 3): What they are and how to reduce them

We identify storage as a necessary target for emissions reductions and break down both operational and embodied emissions in Azure’s storage, showing the impact of both SSD and HDD storage servers. We then discuss how IO is a major limitation for both SSDs and HDDs.

1.1.2 Kangaroo (Ch. 4): Caching Billions of Tiny Objects in Flash

Social networks, microblogs, and emerging sensing applications in the Internet of Things (IoT) need to access tiny objects such as graph edges, text, and metadata. However, tiny objects are uniquely challenging for flash caches, because the objects are orders-of-magnitude smaller than the write size of flash, causing either write endurance problems or requiring unreasonable memory overheads.

Prior flash caches. Existing cache designs either require too many writes or too much memory. Log-structured caches [105] minimize writes by storing all objects sequentially, achieving close to the minimum writes. Unfortunately, these caches need a full index to quickly find objects, which, even when highly optimized, requires too much DRAM to be sustainable. Set-associative caches [60] minimize DRAM indexing by mapping objects to fixed size locations, “sets”, on flash using a hash of the object’s key. However, set-associative caches suffer from too many writes: they have to write 4 KB for every 100 byte object, causing a *write amplification* of 40x. These additional writes lead to early wear-out due to flash’s limited write endurance.

Kangaroo [178, 179]. To optimize both writes and DRAM usage, we developed a hybrid flash cache called Kangaroo that combines prior designs to get the best of both worlds. Kangaroo’s main insight is that a small log-structured cache (KLog) minimizes write amplification by amortizing set writes in a large set-associative cache (KSets) over multiple objects. Kangaroo first writes to KLog, taking advantage of its low write amplification.

Once KLog fills, Kangaroo flushes each object in a log segment to the set in KSets corresponding to the key’s hashed value along with *all other objects in KLog that map to the same set*. Essentially, Kangaroo reduces writes by creating hash collisions. Since Kangaroo is a cache, if there are not enough collisions for a set, objects are evicted. Kangaroo’s design has a very low memory overhead of 7.0 bits/obj, a 4.3x improvement over prior work.

Thus, Kangaroo has both low memory overhead and low write amplification, unlike prior systems. Avoiding both of these pitfalls causes Kangaroo to have 29% fewer misses under realistic workloads. The deployment of Kangaroo on production servers at Meta showed a 38% reduction in writes compared to their production tiny-object cache.

1.1.3 FairyWREN (Ch. 5): A Sustainable Cache with Write-Read-Erase Flash Interfaces

While Kangaroo greatly reduces writes without a large memory overhead, long-lived and dense flash requires further write reductions. Unfortunately, Kangaroo cannot address the main remaining source of writes: rewrites inside the flash device. These writes exist because flash’s current interface, *Logical-Block-Addressable Devices (LBAD)*, allows 4 KB writes even though flash’s erase granularity is closer to a gigabyte (flash must erase before overwriting data). This mismatch requires the device to reclaim space through garbage collection. Garbage collection leads to uncontrollable writes, particularly bad in caches where every write is a decision on whether to keep objects or evict them.

Flash interfaces. New emerging flash SSD interfaces, such as ZNS [66] and FDP [44], allow closer integration of host-level software and flash management. Importantly, these interfaces expose erase operations. We introduce Write-Read-Erase iNterfaces (WREN), an interface categorization that captures the necessary operations for sustainable flash caches. However, WREN does not immediately reduce writes — it only gives the cache control of all writes.

FairyWREN [177, 181]. FairyWREN combines previously device-controlled garbage collection and the caching logic in Kangaroo into one operation called *nesting*, reducing writes. With several other optimizations, FairyWREN decreases writes 12.5x over Kangaroo. This write reduction translates to a 33% decrease in flash emissions over Kangaroo and a >50% reduction in total emissions over log-structured caches, the prior state-of-the-art.

1.1.4 Declarative IO (Ch. 6): Scaling the IO-per-TB wall in Bulk Storage

Most storage devices are in bulk storage, where datacenters use hundreds of thousands of hard disk drives. These hard disk drives’ capacities are greatly increasing due to new technology — from <20 TB in 2020 to an expected >50 TB in 2025. Deploying denser drives promises to reduce embodied emissions. Unfortunately, these drives’ bandwidth and their IOPS per drive has not increased to match their capacity. This “*IO wall*” prevents the

adoption of denser drives. To deploy these more sustainable, denser drives, bulk storage needs to reduce IO.

Most bulk storage IO is maintenance. Surprisingly, most IO does not come from storage users, since most of this user IO is highly cacheable. Rather, *my work identifies maintenance tasks as the primary source of IO in bulk storage*, based on discussions with hyperscalars including Google, Meta, and Microsoft. These tasks are essential, because they ensure data is stored accessibly and reliably with little space overhead. For instance, scrubbing requires reading every byte of data every few months to ensure the data still exists. Decreasing the frequency of scrubbing increases the risk of data loss, an unacceptable outcome for a storage system. Making the problem worse, maintenance tasks come from throughout the datacenter: from databases (e.g., compaction, table statistics), to object stores (e.g., transcoding, object checksums), to distributed file systems (e.g., scrubbing, rebalancing, reconstruction), to disks (e.g., garbage collection).

Insight: Imperative IO destroys flexibility. Today, maintenance tasks use an imperative interface that allows tasks to request only a *specific* piece of data *now*, i.e., read this block now. This unnecessarily limits these tasks' inherent time and order flexibility. For instance, scrubbing must currently be implemented via sequential requests to read individual blocks rather than a directive to read a set of blocks at some time in the next month. If the imperative interface did not force these tasks to eliminate their flexibility, there is massive potential for overlap *between* these tasks. By coordinating IO, storage systems could issue a single command to

Declarative IO: Exposing and exploiting IO overlap to enable denser HDDs. To reduce bulk storage IO, we introduce a new interface, Declarative IO, that allows maintenance tasks to declare that they need to read a set of data by a target deadline. Returning to the scrubbing example, rather than writing a large loop through all data, declarative IO allows scrubbing to declare that it needs all data every couple months. Other tasks, such as re-encoding and garbage collection, can do the same. Then, when DINGOS, our implementation of a system to support Declarative IO, decides it is a good time to read a piece of data for garbage collection, scrubbing and any other declarative tasks get that data *for free*. DINGOS demonstrates that Declarative IO is a promising new direction to reduce disk IO in bulk storage, reducing read IO by 40% relative to imperative IO for maintenance tasks.

Chapter 2

Background

“Data really powers everything that we do.”

Jeff Weiner

STORAGE is a critical component of a datacenter. Cloud users rely on storage to guarantee access to their data across applications from machine learning to social media and beyond. This chapter provides an overview of the storage stack in a datacenter, from the application to bulk storage (Sec. 2.1). Then, it provides a more detailed description of caching, particularly flash caching, in the datacenter (Sec. 2.2) and bulk storage systems (Sec. 2.3) along with their design challenges. Finally, it discusses work on increasing datacenter sustainability and its relation to storage (Sec. 2.4).

2.1 Data retrieval in the datacenter

Cloud storage is predominantly backed by distributed storage systems. Most data is permanently stored in *storage servers* grouped into storage racks, separated from compute.

Datacenters deploy a complex, interconnected collection of services to manage its data, including caching, bulk storage, and data management services (Fig. 2.1). Generally, requests from applications first encounter a data management service (i.e., data lakehouse or database), which then leads to storage — both more caches and eventually bulk storage.

Data Management Services. Data management services exist to organize data and make it accessible to applications. These services include table stores, data warehouses, and data lakes. In addition to sending IO to storage based on application requests, these services also send IO to storage to manage their data. For example, many data management services deploy log-structured merge trees [20, 24, 208, 210, 252], which require compacting their data repeatedly to ensure their performance and to achieve reasonable space utilization.

Caching layers (Sec. 2.2). Caches exist throughout the datacenter to minimize the latency of services and reduce the load on backend services, including data management services and bulk storage. Caches farther from user requests need to be large in order to effectively cache requests since much of the locality in requests is removed by earlier

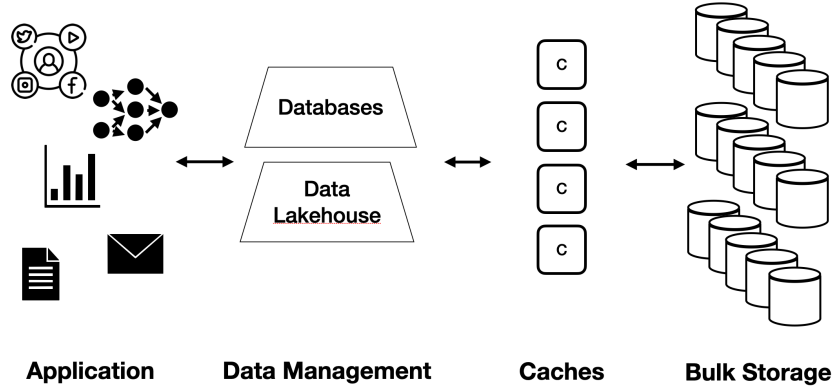


Figure 2.1: Datacenter storage overview. When applications need data, they typically first go to a data management service, then to the caches, and finally to bulk storage. For this thesis, we focus primarily on storage — the caching and bulk storage layers.

caching stages. To accommodate these large caches, many datacenters deploy flash caching as their last layer of caching before going to storage [60, 68, 69, 105, 266].

Bulk Storage (Sec. 2.3). Data is stored for long-term retention in bulk storage. Bulk storage consists primarily of 100Ks of HDDs [119, 194, 224, 254], which are together combined into a single storage system. The bulk storage system generally stores large blocks of data spread across many devices and servers, using a metadata service to track where data resides and any other metadata about the blocks. Since bulk storage needs to retain data until it is deleted (usually years later if it is ever deleted), it also runs many maintenance services, from scrubbing [132, 190, 215] to reconstruction [82, 121] to load balancing to transcoding [142, 144]. These maintenance tasks result in IO to the hard drives.

2.2 Caching in the datacenter

Caching is essential to the performance of large-scale storage systems and thus caches are prevalent (as seen in Fig. 2.2, which shows caching at Meta.). Caches are used to both reduce the latency of data, since caches are often physically closer to applications and on lower latency media. They also reduce the load sent to lower layers of the storage stack, such as bulk storage. Specifically, in this dissertation, we focus on the last layer of caches — right before bulk storage. These caches are typically large (thousands of servers each with at least several TB of flash), flash-based caches that see a mix of workloads, including key-value stores, data management services, and more. One notable feature of these workloads is that they often contain many tiny objects (objects around 100 bytes in size).

In this section, we first discuss the presence of tiny objects in caching workloads (Sec. 2.2.1). Then, we examine why caching on flash makes sense at a datacenter scale (Sec. 2.2.2) and describe, at a high-level, what flash SSDs are and their limitations (Sec. 2.2.3). Then, we discuss the challenges of caching tiny objects in flash (Sec. 2.2.4), particularly

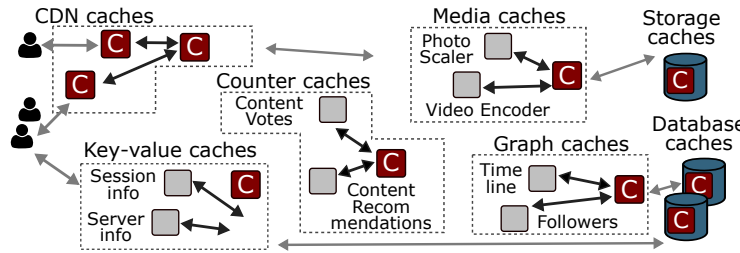


Figure 2.2: Overview of caching at Meta. Caching is deployed in many different services across Meta from the content delivery network, CDN, all the way back to storage. [60]

how write amplification limits flash cache design. Finally, we discuss existing approaches to caching tiny objects in flash and their shortcomings (Sec. 2.2.5).

2.2.1 Tiny objects are important and numerous

Tiny objects are prevalent in many large-scale systems:

- At Meta, small objects are prevalent in the Facebook social graph. For example, the average social-graph edge size is under 100 B. Across edges, nodes, and other objects, the average object size is less than 700 B [60, 71]. This has led to the development of a dedicated flash caching system for small objects [60].
- At Twitter, tweets are limited to 280 B, and the average tweet is less than 33 characters [198]. Due to the massive and growing number of tweets, Twitter seeks a cost-effective caching solution [263].
- At Microsoft Azure, sensor updates from IoT devices in Azure Streaming Analytics are a growing use case. Before an update can be processed (e.g., to trigger a real-time action), the server must fetch metadata (the sensor’s unit of measurement, geolocation, owner, etc.) with an average size of 300 B. For efficiency and availability, it caches the most popular metadata [111]. Another use case arises in search advertising, where Azure caches predictions and other results [160, 161].

Each of these systems accesses billions of objects that are each significantly less than the 4KB minimum write granularity of block-storage devices. For example, Facebook records 1.5 billion users daily [27] and friendship connections alone account for hundreds to thousands of edges per user [71, 243]. Twitter logs over 500 million new tweets per day and serves over 190 million daily users [30]. While IoT update frequencies and ad impressions are not publicly available, the number of connected devices is estimated to have surpassed 50 billion in 2020 [94], and the average person was estimated to see 5,000 ads every day as early as 2007 [232].

2.2.2 Caching tiny objects

While individual objects in the above applications are tiny, application working sets on individual servers still add up to TBs of data. To reduce throughput demands on back-

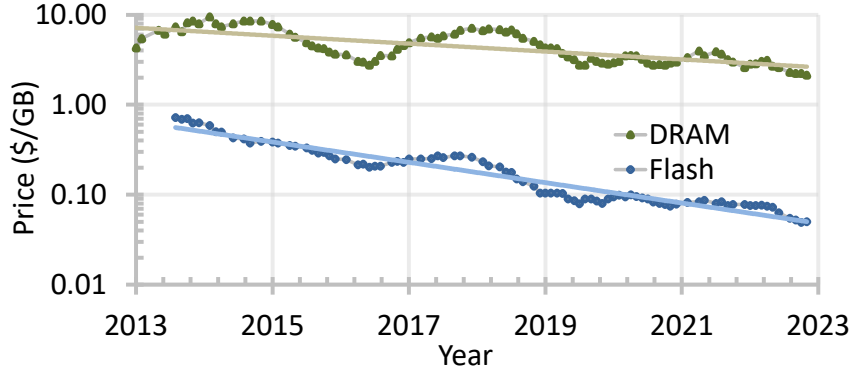


Figure 2.3: Cost of flash and DRAM over time. Cost for flash and DRAM over the last 10 years [10, 12]. Flash prices have decreased over $14\times$, while DRAM prices have only decreased by $\approx 2\times$.

end data-management systems, applications rely on large-scale, cost-efficient caches, as a single caching server can replace tens of backend servers [60]. Unfortunately, as described below, current caching systems are inefficient for tiny objects. There is therefore a need for caching systems optimized specifically for large numbers of tiny objects.

Why not just use memory? DRAM is expensive, both in terms of acquisition and power cost per bit. DRAM often makes up 40% to 50% of server cost [163, 222, 236] and is no longer scaling (Fig. 2.3). DRAM capacity is also often limited due to operational power concerns. Moreover, DRAM is often in high demand, so all applications are encouraged to minimize DRAM usage. For example, the trend in recent years at Meta is towards less DRAM and more flash per server [60, 236]. DRAM also has a large embodied carbon footprint and has large operational emissions due to requiring up to half of system power [123].

Why flash? Flash is cheaper per-bit, embodies $12\times$ less carbon, and requires less power per-bit than DRAM [123]. Thus, datacenters should use flash over DRAM whenever possible [120], even for traditional DRAM workloads, such as caching [60, 105, 178, 179] or machine learning [265].

Flash currently provides the best combination of performance and cost among memory and storage technologies and is thus the technology of choice for most large-scale caches [60, 68, 69, 105, 229]. It is much faster than mechanical disks. While flash-based caches do use DRAM for metadata and “hot” objects, the bulk of the cache capacity is flash to reduce end-to-end system cost.

2.2.3 Flash Solid State Drives (SSDs)

Flash SSDs are built from NAND flash memory. Flash memory has two main limitations: it wears out with writes and does not allow small-granularity overwrites. SSDs have *limited write endurance* — after too many writes, their flash cells can no longer store data [125]. If applications write too much, flash’s lifetime can be extremely short.

SSDs have been getting denser. SSD density has increased through two main mechanisms: increasing the number of layers and increasing cell density. SSDs have been 3D-stacking layers of cells, growing flash storage “vertically.” Today, flash devices can have over 200 layers, and the number of layers is quickly increasing [1, 25]. 3D stacking increases device density but also increases embodied emissions. Flash is also becoming denser by packing bits into cells. Most datacenter SSDs today use tri-level cells (TLC), which store 3 bits per cell. Flash SSDs will soon use quad-level cells (QLC) (4 bits/cell) and penta-level cells (PLC) (5 bits/cell) [203]. Unfortunately, increasing cell density causes disproportionately lower write endurance.

2.2.4 Challenges in flash caching

Flash presents many problems not present in DRAM caches due to its limited write endurance. Without care, caches can quickly wear out flash devices as they rapidly admit and evict objects [60, 105]. Hence, many existing flash caches over-provision capacity, suffering more misses in order to slow wear out [60, 69].

Exacerbating the endurance issue, flash drives suffer from *write amplification*. Write amplification occurs when the number of bytes written to the underlying flash exceeds the number of bytes of data originally written. Write amplification is expressed as a multiplier of the number of bytes written, so a value of $1\times$ is minimal, indicating no extra writes. Flash devices suffer from both device-level write amplification and application-level write amplification [125].

Wherefore device write amplification? *Device-level write amplification* (DLWA) [105, 159, 237] occurs because flash does not allow overwrites at a small granularity. Instead, flash devices can write only new values after 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) [70, 105, 125, 156, 159, 168, 237] 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 $3\times$.) DLWA can be large: a factor of $2\times$ to $10\times$ is common [178]. DLWA causes write-intensive applications to quickly wear out flash devices, increasing their replacement frequency and embodied emissions over time.

DLWA is primarily caused by the physical limitations of flash storage. Flash devices are organized in a physical hierarchy (Fig. 2.4). 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* [41, 65, 66, 125, 172]. 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 [66].

The mismatch between the granularity of writes and erases is the root cause of DLWA. To maintain the 4 KB read/write page interface (confusingly an interface called *Logical Block Addressing* (*LBA*)), flash devices *garbage collect* (*GC*), moving live pages from partially

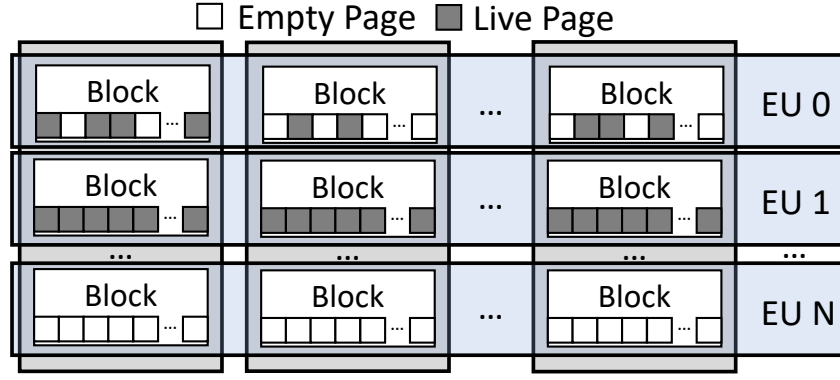


Figure 2.4: Flash architecture. The internal arrangement of flash devices into planes, blocks, pages, and EUs. Each EU has multiple blocks and each block has multiple pages. EU 0 is a partially full, EU 1 is entirely full, and EU N has just been erased.

empty EUs (such as EU 0 in Fig. 2.4) 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.

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 [23]. Additionally, because denser flash has to differentiate between more voltage levels, even small voltage changes can make data unreadable. TLC uses two-phase writes¹ and more frequent refresh to prevent data loss [184]. 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.

One might hope that, as a counterbalance, technological advances would decrease EU sizes, closing the gap between write and erase granularities. However, *flash EU sizes have become larger as flash has gotten denser*. Effective block sizes on an SLC flash device were 128 KB[242], MLC and TLC flash devices are around 20 MB [234], and QLC devices are 48 MB [235]. Striping these blocks with hundreds of 3D-stacked layers [235] produces EUs in the gigabyte range [66, 182]. Thus, DLWA will increase with flash density.

Overprovisioning to reduce DLWA. If DLWA and its write endurance problems are going to get worse, how can we reduce it? Generally speaking, DLWA worsens as more of the raw flash capacity is utilized and as workloads contain more of small, random writes. A common approach to reduce DLWA is *overprovisioning*, i.e., only exposing a fraction of the raw flash capacity in the LBA namespace, so that cleaning tends to find fewer live pages in victim erase blocks [60, 69]. Fig. 2.5 shows DLWA vs. utilized capacity for random

¹Two-phase writes first write pass aims to get each cell close to the desired voltage. The device then probes the cells to see their actual voltage and does a second, finer programming pass to achieve the desired voltage. This two-phase process reduces variability in voltage ranges such as from interference from programming neighboring cells.

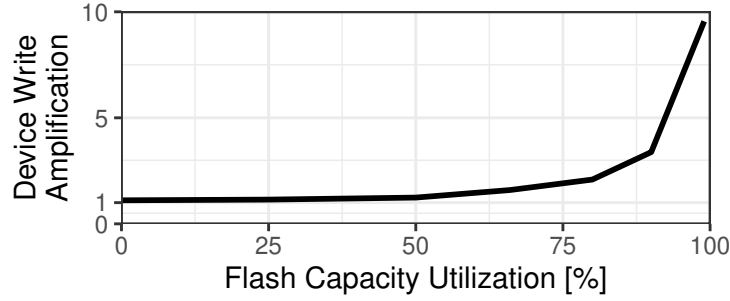


Figure 2.5: Effects of overprovisioning on write amplification. The effect of flash over-provisioning on device-level write amplification (DLWA) of random writes of various sizes. DLWA increases as over-provisioning decreases.

4 KB writes to a 1.9 TB flash drive. As expected, DLWA significantly increases as over-provisioning decreases, from $\approx 1\times$ at 50% utilization to $\approx 10\times$ at 100% utilization. Thus, overprovisioning is not a panacea, as it reduces the effective capacity of the device.

Application-level write amplification. Unfortunately, DLWA is not the only source of write amplification in flash caches. *Application-level write amplification* (ALWA) occurs when the storage application re-writes some of its own data as part of its storage management. One form of this is akin to flash translation layer (FTL) cleaning, such as cleaning in log-structured file systems [156, 210] or compaction in log-structured merge trees [20, 24]. Another form is caused by having to write an entire logical block. To write a smaller amount of data, the application must read the block, install the new data, and then write the entire block [191]. For example, writing 1 KB of new data into a 4 KB logical block involves rewriting the other 3 KB, giving ALWA of $4\times$. Ideally, the unmodified data in the block would not have been rewritten.

Why caching tiny objects is hard. The size of tiny objects makes caching them on flash challenging. Tracking billions of tiny objects individually in large storage devices can require huge metadata structures [105], which either require a huge amount of DRAM, additional flash writes (if the index lives on flash), or both. To amortize tracking metadata, one could group many tiny objects into a larger, long-lived “meta-object”. This can be inefficient, however, if individual objects in the meta-object are accessed in disparate patterns.

Tiny objects are also a major challenge for write amplification. Traditional cache designs (i.e., for DRAM caches) freely re-write objects in place, leading to small, random writes; i.e., the worst case for DLWA. Since tiny objects are much smaller than a logical block, re-writing them in place would additionally involve substantial ALWA — $40\times$ for a 100 B object in a 4 KB logical block — which is *multiplicative* with DLWA. Grouping tiny objects into larger meta-objects, as mentioned above, shifts ALWA from logical blocks to meta-objects but does not address the problem.

2.2.5 Shortcomings of existing solutions

This section discusses existing solutions for flash caching and their shortcomings for caching tiny objects.

Key-value stores: Flash-efficient key-value stores have been developed and demonstrated [24, 97, 166, 208, 257], and it is tempting to consider them when a cache is needed. But key-value stores generally assume that deletion is rare and that stored values must be kept until told otherwise. In contrast, caches delete items frequently and at their own discretion (i.e., every time an item is evicted). With frequent deletions, key-value stores experience severe write amplification, much lower effective capacity, or both [59, 69, 105, 237, 257].

As a concrete example, consider SILT [166], the key-value store that comes closest to Kangaroo in its high-level design. Like Kangaroo, SILT uses a multi-tier flash design to balance memory index size vs. write amplification. Unfortunately, SILT’s design is poorly suited to caching. For example, SILT’s two main layers, which hold >99% of entries, are immutable. Because those layers are immutable, DELETE operations are logged and do not immediately reclaim space. Thus, cache evictions result in holes (i.e., reduced cache capacity) until the next compaction (merge and re-sort) occurs. One can reduce the lost cache capacity with more frequent compactions, but at a large penalty to performance and ALWA.

Similar issues with DELETES affect most key-value stores, often with this same trade-off between compaction frequency and holes in immutable data structures. One may be able to reduce these overheads somewhat by coordinating eviction with compaction operations, but this is not trivial and not how these systems were designed. For instance, Netflix used RocksDB [24] as a flash cache and had to over-provision by 67% due to this issue [69]. Some key-value stores reduce ALWA by making reads less efficient [168, 208, 257] but do not sidestep the fundamental challenge of DELETES wasting capacity. In contrast, flash caches have the freedom to evict objects when convenient. This lets flash caches co-design data structures and policies so that DELETES are efficient and minimal space is wasted.

Log-structured caches: To reduce write amplification, many flash caches employ a log structure on flash with an index in DRAM to track objects’ locations [61, 105, 159, 220, 229, 237]. While this solution often works well for larger objects, it requires prohibitively large amounts of DRAM for tiny objects, as the index must keep one entry per object. The index can spill onto flash [257], but spilling adds flash reads for lookups and flash writes to update the index as objects are admitted and evicted.

Even Flashield [105], a recent log-structured cache design for small objects, faces DRAM problems for larger flash devices. After optimizing its DRAM usage, Flashield needs 20 bits per object for indexing plus approximately 10 bits per object for Bloom filters. Thus, Flashield would need 75 GB of DRAM to track 2 TB of 100 B objects. In fact, Flashield’s DRAM usage is much higher than this, because it relies on an in-memory cache to decide which objects to write to flash. The DRAM cache must grow with flash capacity or else prediction accuracy will suffer, leading to more misses.

Thus, the total DRAM required for a log-structured cache can quickly exceed the amount available and significantly increase system cost and power. Technology trends will

make these problems worse over time, since cost per bit continues to decrease faster for flash than for DRAM [80, 258].

Set-associative flash caches: Metadata to locate objects on flash can be reduced by restricting their possible locations [196]. Meta’s CacheLib [60] implements such a design for small objects (<2 KB), e.g., items in the social graph [71]. CacheLib’s “small-object cache” (SOC) is a set-associative cache with variable-size objects, using a hash function to map each object to a specific 4 KB set (i.e., a flash page). With this scheme, SOC requires no index and only ≈ 3 bits of DRAM per object for per-set Bloom filters.

Although more DRAM-efficient, set-associative designs suffer from excessive write rates. Inserting a new object into a set means rewriting an entire flash page, most of which is unchanged, incurring $40\times$ ALWA for a 100 B object and 4 KB page as discussed above. In addition, flash writes are a worst case for DLWA: small and random (Fig. 2.5). The multiplicative nature of ALWA and DLWA compounds the harmful effect on device lifetimes.

Set-associative flash caches limit their flash write-rate through two main techniques. To reduce DLWA, set-associative flash caches are often massively over-provisioned. For example, CacheLib’s SOC is run in production with over half of the flash device empty [60]. That is, the cache requires more than *twice* the physical flash to provide a given cache capacity. Additionally, to limit ALWA, CacheLib’s SOC employs a pre-flash admission policy [60, 105] that rejects a fraction of objects before they are written to flash. Unfortunately, both techniques reduce the cache’s achievable hit ratio.

Summary: Prior work does not adequately address how to cache tiny objects in flash at low cost. Log-structured caches require too much DRAM, and set-associative caches add too much write amplification. As discussed in Ch. 3, both of these problems lead to excessive emissions.

2.3 Bulk storage

Bulk storage, supported by distributed storage systems, are the backbone of modern datacenters, containing exabytes of data across 100Ks of *hard disk drives (HDDs)* that support everything from ML training to internet applications. As these storage systems continue to grow, datacenters continue to deploy cheaper storage media to support unrelenting data growth while continuing to maintain a low-cost service. New HDD technological advancements support this goal, reducing cost with denser drives. Unfortunately, because these drives do not have proportionately more IO supply, distributed storage systems are running into an IO-per-TB wall.

This section describes hard-disk drives, quantify IO supply and demand in disk-based systems (Sec. 2.3.2), how advancements in HDD density lead to the IO-per-TB wall (Sec. 2.3.3), and some background on distributed storage systems (Sec. 2.3.4).

	Size	Blades/Rack	Count	Capacity ('17)	Capacity ('24)
SSD [218]	1U	8	16	128 TB	246 TB [31]
HDD [126]	4U	36	88	1.2 PB	2.6 PB [51]

Table 2.1: SSD vs HDD Servers. Comparison of Project Olympus’ storage servers including the blade size, blades per rack, storage device count, and the capacity of the entire blade from both the original Project Olympus 2017 specifications and updating the storage devices to 2024 capacities.

2.3.1 Hard-disk drives

An HDD server’s purpose is to store lots of data cheaply. To accomplish this, each server holds many disks (e.g., 88 in Project Olympus [126] Table 2.1), referred to as "Just a bunch of disks" or JBODs. These servers store an order-of-magnitude more data per server than SSD servers and about 2.6x more data per rack space.

HDDs contain multiple circular platters that store data magnetically. To write or read data to a platter, the platter’s head has to seek to the correct track and wait for the disk to spin to the correct sector. Thus, a key factor in request latency is the speed that the HDD is spinning, i.e., its rotations-per-minute (RPM). RPM affects both wait time and device bandwidth, since the HDD can only transfer data that passes under its active head. Unfortunately, even after significant optimization effort, RPM has not improved much for the past decade [9, 194].

HDDs are growing denser, maintaining their capacity-cost advantage over SSDs. HDDs have grown from one to 20 terabytes over the last decade without changing their form factor [147] through density improvements such as shingled-magnetic recording (SMR) drives that overlap the write tracks to increase bits stored per disk area [45, 256]². The next frontier of HDD density is heat-assisted magnetic recording (HAMR), which allows denser packing of bits by heating disks during writes [35]. HAMR promises to increase device capacities to 50 TB and beyond [221].

2.3.2 How do we quantify IO on disk?

IO is commonly measured either by throughput (bytes/s) or request rate (IOPS). However, each only shows half the equation for HDDs. A workload that issues many small reads will show low throughput but high IOPS, while a workload that issues few large reads will show high throughput but low IOPS. Given just one of these two metrics, it is impossible to infer the overall utilization of a disk.

Instead, we quantify IO by its disk-head time [59, 93]: the combined time required to serve an IO request by seeking to the correct head position and then having the head transfer the data. Given the disk’s average positioning latency and transfer rate, disk-head

²SMR drives also introduce a variant of the erase problem seen on flash since the overlapped tracks have to be rewritten together.

time can be estimated as:

$$\text{disk-head time} = \text{seek latency} \times \# \text{ of requests} + \frac{\text{bytes}}{\text{transfer rate}}$$

and the portion of total time the disk spends on the IO as:

$$\text{disk-head utilization} = \frac{\text{disk-head time}}{\text{total time}}$$

2.3.3 Higher-capacity HDDs are increasingly IO-bound

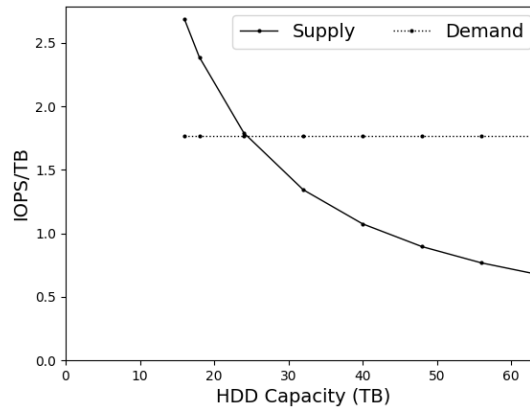


Figure 2.6: HDD IO supply vs demand. Projected HDD IO-per-TB supply trend as disks become denser against the IO demands at Google based on their public Thesios traces [199]. Drives bigger than 24 TB will not be able to support the IO requirements at hyperscalars. We assume each IO operation to be 2 MB in size, average repositioning latency of 10 ms, and a transfer speed of 150 MBps.

As hard drives have seen significant increases in capacity, their seek latencies and transfer rates have not improved at the same pace due to physical limitations of the devices. While HDD transfer rates have increased with capacity due to higher areal density, these improvements are small compared to the increases in capacity. Thus, the disk-head time of a given request has remained roughly constant as HDD capacities have grown, meaning that a smaller percentage of the disk can be accessed in any given amount of time as disk capacity increases.

To measure this effect, Fig. 2.6 shows the IO supplied by the disks, both current and projected, versus the IO demanded of the disk, based on Google’s Thesios traces [199], in *IO-per-TB* (measured in units of disk-head time per TB). As distributed storage systems either add additional capacity or replace older less dense disks using newer, denser disks, the IO-per-TB supplied by the storage systems is decreasing, i.e. we move to the right on the graph. Meanwhile, as the new and existing data stored by these systems continues to be accessed at the same rate, the IO-per-TB demanded of storage will remain

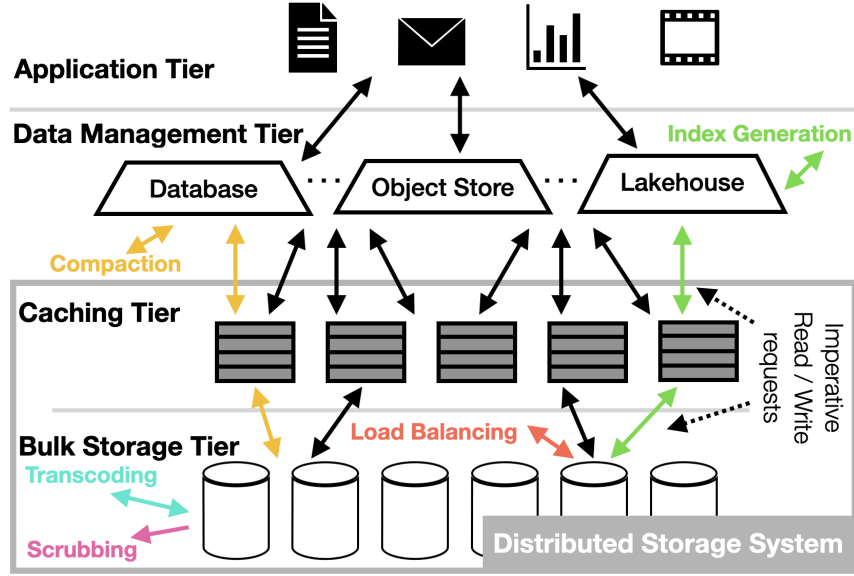


Figure 2.7: IO demand across datacenter. All requests into the caching and bulk storage tiers are imperative, independent of where they are from.

roughly constant. This trend in IO supply is set to accelerate in the next ten years as HAMR devices, which have recently come to market, lead to continued increases in HDD capacity [15, 16, 17, 154]. Hence, distributed storage systems are quickly approaching a IO-per-TB wall, where IO demand will significantly outstrip IO supply. When HDD capacities exceed 22TB, we project that the storage system will hit the IO-per-TB wall, forcing system operators to forgo the cost, power, and footprint advantages of denser drives.

2.3.4 IO demand in distributed storage systems

Today’s IO demand comes from across the datacenter, from applications down to the storage system (Fig. 6.3). Unsurprisingly, applications want data, but also data management, caches, and the distributed storage system all need data to fulfill their roles on top of the data requested directly by applications.

All these data accesses send *imperative* IO requests (read-this, write-that now) to the distributed storage system. For requests for data that is indeed needed immediately, these imperative requests are the right option. We refer to these immediate requests as *application IO* (the black arrows in Figure 6.3). Other requests may have some flexibility, but this flexibility is currently obscured by the imperative interface. We refer to these flexible requests as *maintenance IO* (the colorful arrows). The caching tier is generally effective in absorbing IO demand from application IO. However, maintenance IO is generally less cacheable, and therefore makes up a disproportionate fraction of the IO demand that reaches the bulk storage tier.

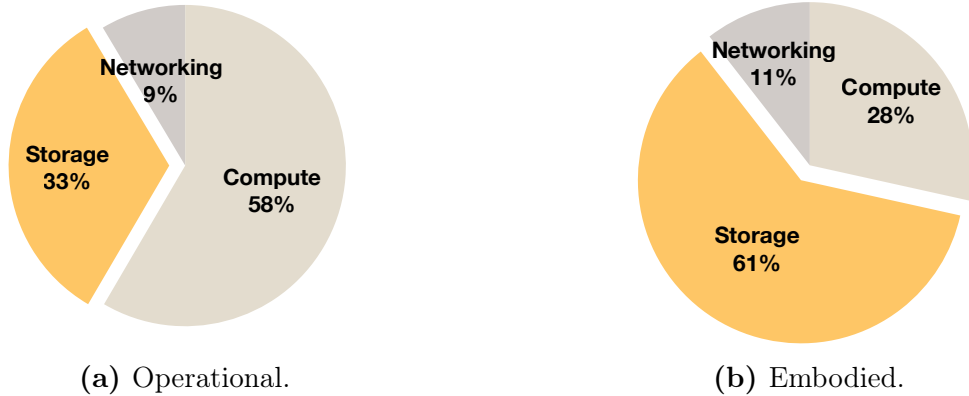


Figure 2.8: Operational and embodied emissions at Azure. Breakdown of operational and embodied emissions at Azure [180, 251].

2.4 Datacenter sustainability

The information and communication technology (ICT) sector has increasingly been recognized as a large contributor to global energy demand and emissions [122], the cloud more specifically [155, 251]. We are just starting to explore the way in which choices of datacenter design influence their emissions [122, 123, 173, 251].

Generally, we divide emissions into three components — direct emissions (Scope 1), operational emissions (Scope 2), and embodied emissions (Scope 3) — based on the Greenhouse Gas Protocol’s definitions [122, 123, 251]. Scope 1 emissions (for instance, from company vehicles or on-site generators) are negligible for hyperscalars [251], so we do not discuss them in this dissertation. Operational emissions are emissions from the electricity used to power and cool datacenters. They are about 25-58% of emissions today [36, 251], but are expected to decrease as hyperscalars actively deploy more renewable energy [122, 123]. Embodied emissions are indirect emissions or emissions from the supply chain, such as from manufacturing, transporting, and end-of-lifing hardware. These emissions are increasing [36] as hyperscalars continue to build and expand datacenters to keep up with demand.

Fig. 2.8 shows a breakdown of operational and embodied emissions between compute, storage, and networking at Azure from 2023. Storage is responsible for 33% of operational emissions and 61% of embodied emissions. While we expect storage’s operational emissions to be less significant due to AI workloads’ large and increasing energy demands, storage will continue to dominate embodied emissions [180]. Thus, to reduce embodied emissions, the large and growing fraction of datacenter emissions, we need to focus on storage’s embodied emissions.

The remainder of this section discusses prior work on reducing storage emissions (Sec. 2.4.1). Since there are limited approaches to reduce storage emissions in prior work, we will also discuss prior work on reducing computer emissions, where there is more research so that we can assess how well that work can be applied to storage (Sec. 2.4.2).

2.4.1 Prior work on reducing storage emissions

While there has not been much prior work on reducing storage emissions, we identify that prior work has identified the influence of media choice on storage emissions, discussed how to increase lifetime of storage through overprovisioning and data degradation, and identified how to reduce energy consumption in storage.

Media choice influences storage emissions. DRAM often makes up 40% to 50% of server cost [163, 222, 236]. DRAM also has a large embodied carbon footprint (46% of a server in Azure [173]) and has large operational emissions due to requiring up to half of system power [123]. Flash embodies $12\times$ less carbon and requires less power per-bit [123] (i.e. less operational emissions per-bit assuming the same energy source). Thus, datacenters should use flash over DRAM to reduce emissions whenever possible [120] even for traditionally DRAM workloads, such as caching [60, 105, 178, 179] or machine learning [265]. HDDs are even less carbon-intensive, $\approx 3\text{-}10\times$ less embodied carbon per-bit than flash [123, 238]. This means that hard drives are going to be an essential part of a sustainable storage stack, particularly more dense drives that have similar hardware manufacturing requirements.

Reducing storage energy consumption Prior work has tried to reduce energy, particularly for HDDs that require constant power, since the early 2000s. This work fits into two categories: (i) caching to create idle IO periods and (ii) distributing data to enable power proportionality. Unfortunately, these solutions do not consider embodied emissions.

HDDs use far less power when in idle or sleep modes [26], so prior work often aims to create idle periods in HDD traffic. Prior work used different strategies, such as using a relatively low-power disk to store most hot data [76], separating hot and cold data [131, 151, 162, 200], or employing better prefetching and caching policies [205, 228]. These policies often do not work for common maintenance tasks that stream through large amounts of data sequentially, such as scrubbing [53, 216], and are hard to change in cloud environments where much of the work is *not under the cloud provider’s control*. As shown in Pelican [53], datacenter power provisioning would also have to change to accommodate variable power, since typically power is provisioned for storage server’s peak — when all disks are running.

Alternatively, some work has suggested leveraging data replication to turn off storage servers when either IO is low enough [46, 53, 89, 240] or there is not enough available green energy [150]. Turning off the entire server allows for energy savings from all components, not just disks [116]. Unfortunately, most prior work assumes data *replication*, where the storage system stores multiple copies of data, whereas today’s datacenters use more space-efficient erasure codes [145, 146]. One prior paper did consider leveraging redundancy in erasure-code-based distributed storage systems [201], but much more work can be done especially factoring in embodied emissions.

More recently, these ideas have been revisited for flash and operational emissions. For instance, if one assumes flash is power-proportional, we could move background IO accesses to when there is more renewable energy and reduce operational emissions [209]. While this approach is more promising, because it does not require additional embodied emissions, we find that flash in datacenters today is not power-proportional (Sec. 3.1). None of this research also addresses the largest fraction of storage emissions — embodied emissions.

Decreasing storage embodied emissions. There are two main pieces of prior work on reducing embodied emissions in storage: one that increases device lifetime through overprovisioning and one that argues for deploy lower-emission, lower-endurance flash by accepting degraded data quality.

Increasing device lifetime is a key way to reduce embodied emissions, since embodied emissions are independent of device lifetime. Unfortunately, flash has a limited lifetime due to its limited write endurance (Sec. 2.2.3), so prior work has suggested increasing overprovisioning to increase lifetime [123]. Using overprovisioning to decrease embodied emissions is a trade-off. Increasing overprovisioning decreases embodied emissions for a while, because it lowers the device-level write amplification. Eventually though, the capacity loss from overprovisioning means that the the embodied emissions per stored bit increases. For 2 year lifetimes, the optimal overprovisioning that Gupta found was 16% [123], higher than the traditional 7% device overprovisioning for commodity SSDs and at much less the expected lifetimes in datacenters (which is 5-6 year [173]).

A different way to reduce flash emissions is to use high-density, lower-emission flash by accepting degraded data quality and the potential deletion or loss data as the dense flash degrades [270]. This approach does not work well in the datacenter due to explicit durability guarantees.

This prior research leaves open whether, through system design, we can reduce flash’s embodied emissions without sacrificing durability better than just relying on overprovisioning. It also does not address reducing embodied emissions of hard disk drives despite their prevalence in bulk storage.

2.4.2 Prior solutions to reduce compute emissions

Since prior work on reducing storage embodied emissions in system design is limited, we now shift our attention to prior work on reducing compute emissions and see whether we can apply it to storage. Prior work on compute often focuses on reducing energy usage [83, 112, 114, 115, 135, 170, 171, 183, 204, 231, 268] and moving computation to times/locations with more renewable energy [14, 39, 55, 207, 230, 255]. Since both of these approaches focus on operational emissions, they are less relevant for storage. Instead, compute shows two other, more promising approaches that are more relevant to storage: using fewer compute devices [81, 98, 99, 117, 223] and increasing lifetimes by identifying places where older device performance is adequate [174, 241, 248, 249, 250]. Unfortunately, as we will show in Ch. 3, these approaches both have new challenges in storage that need to be overcome to be viable strategies to reduce storage emissions.

Chapter 3

Reducing storage emissions results in IO limitations

“Remarkably, the manufacturing of storage devices alone contributed to an estimated 20 million metric tonnes of CO₂ emissions in the year 2021”

Tammu and Nair. [238]

DISTRIBUTED STORAGE is a large emitter [251]. Unfortunately, there is no standard break down of storage emission sources as they are deployed in datacenters, which is necessary to understand and reduce storage emissions. Therefore, we begin by showing how each component of storage servers in a datacenter rack contributes to emissions at Azure (Sec. 3.1). We then introduce a model to frame the discussion on how to reduce emissions. This model shows how using fewer, denser drives and extending lifetime are the two ways to reduce storage’s embodied emissions. We also present how both of these solutions lead the key challenge of sustainable storage — a lack of IO (Sec. 3.2).

3.1 Where do storage emissions come from?

We now present the emissions from both a SSD storage rack and an HDD storage rack in Azure, focusing on the key components (CPU, DRAM, SSD, and HDD). We use the “Other” category to group rack overheads, such as fans, network switches, power supplies, and power delivery units. For embodied carbon, the “Other” category also includes passive material like sheet metal and plastics. Sec. 3.1.1 discusses operational emissions, e.g., from power generation, and Sec. 3.1.2 discusses embodied emissions, e.g., from semiconductor fabs.

3.1.1 Operational emissions

Table 3.1 shows the relative operational emissions of each Azure rack type. To determine energy consumption and therefore operational emissions of different components, we take component energy draws measured under a representative load. Notably, an SSD storage

Operational Emissions	CPU	DRAM	SSD	HDD	Other
Compute Rack	42%	18%	19%	0%	21%
SSD Rack	32%	8%	38%	1%	21%
HDD Rack	26%	5%	7%	41%	21%

Table 3.1: Operational emissions at Azure. Operational emission breakdown for Azure different rack types and for different components of each rack type.

Embodied Emissions	CPU	DRAM	SSD	HDD	Other
Compute Rack	4%	40%	30%	0%	26%
SSD Rack	1%	9%	80%	1%	9%
HDD Rack	2%	11%	14%	41%	33%

Table 3.2: Embodied emissions at Azure. Embodied emission breakdown for Azure different rack types and for different components of each rack type.

rack has approximately $4\times$ the operational emissions per TB of an HDD storage rack.

Storage devices (SSDs and HDDs) are the largest single contributor of operational emissions. For SSD racks, storage devices account for 39% of emissions, whereas for HDD racks they account for 48% of emissions. These numbers contradict the conventional wisdom that processing units dominate energy consumption [123, 251]: storage servers carry so many storage devices that they become the dominant energy consumers. Thus, the best way to reduce operational emissions in a storage server is to reduce the storage *devices*’ energy.

Since CPUs still cause the next largest portion of the emissions, improving the energy efficiency of CPUs in storage servers may still provide benefits. However, one has to be careful with energy efficiency improvements that increase embodied carbon emissions [238, 251]. For example, advanced semiconductor fabrication nodes reduce operational emissions but increase manufacturing emissions and electricity use [54, 104, 233]. This consideration is particularly important in storage, which is already embodied emission heavy.

3.1.2 Embodied emissions

We show the relative embodied emissions of each Azure rack type in Table 3.2. To estimate embodied emissions, we use raw material numbers from vendors, the device’s silicon area, and then leverage IMEC [18] and Makersite [21] to determine average emissions for manufacturing processes. We ensure that manufacturing and shipping emissions are only counted once and are amortized across components, so that our embodied emissions results are comparable to our operational emissions results.

SSD racks contribute approximately *10x the embodied emissions per TB* as that of HDD storage racks. The storage devices themselves dominate embodied emissions, accounting for 81% and 55% of emissions in SSD and HDD racks, respectively. While DRAM is the largest embodied emissions contributor in compute servers, this is not true for storage servers due to the many storage devices in these servers. Across both operational and

embodied emissions in distributed storage clusters, there is a clear need to reduce emissions from the storage devices themselves.

3.2 How can datacenters reduce storage emissions?

We now consider the important opportunities and challenges in reducing storage emissions. To structure the discussion, we model emissions by breaking apart operational and amortized embodied emissions:

$$\begin{aligned} \text{Annual Carbon Emissions} &= \text{Operational Emissions} + \text{Embodied Emissions} \\ &= \sum_{\text{Devices}} \left(\frac{\text{Watt-Hours}}{\text{Device}} \times \frac{\text{Carbon}}{\text{Watt-Hour}} + \frac{\text{Carbon}}{\text{Device}} \times \frac{1}{\text{Lifetime}} \right) \end{aligned}$$

This simple model tells us that emissions can be reduced in five ways: using fewer devices, lowering power, reducing carbon intensity of power, reducing per-device embodied emissions, or increasing server and device lifetime.

Thus, to specifically reduce embodied emissions, we can use fewer, less embodied-emissions-per-bit devices or increase lifetimes. Unfortunately, these improvements in flash and hard drive emissions-per-bit come with drawbacks in the form of decreased IO. Deploying denser flash and increasing its lifetime both lower the number of writes that flash can withstand without wearing out. Denser hard drives cannot increase their bandwidth to keep up with their capacity. Without addressing these IO limitations, at best, these technologies cannot be deployed, at worse, they can increase emissions. For the rest of this section, we describe these IO limitations in more detail for both flash (Sec. 3.2.1) and HDDs (Sec. 3.2.2).

3.2.1 Denser and longer lifetimes to reduce flash emissions

To reduce flash emissions, we can use fewer, denser devices and increase lifetime. The denser the flash, the more it reduces embodied emissions, since more bits are packed onto roughly the same silicon. Moving to longer lifetimes also amortizes embodied carbon. Traditionally, datacenter hardware replacement cycles have been around 3 years [173] due to the rate of improvement in hardware performance and power-efficiency. Today, datacenters deploy devices for longer. Longer replacement cycles have become common due to their cost advantages and the slowing of Moore’s Law. For example, Microsoft Azure increased the depreciable lifetime of servers from 4 to 6 years [127, 174], and Meta recently started planning for servers to last 5.5 years [39]. Additionally, hyperscalers are finding that servers do not fail quickly: failure rates at Azure have little evidence of increasing before 8 years [64, 173].

Lowering carbon-intensity lowers write rate. Unfortunately, as flash becomes denser and its lifetime increases, its write endurance drops significantly. Fig. 3.1 models how write rate affects both emissions when varying lifetimes and flash density. Each line shows

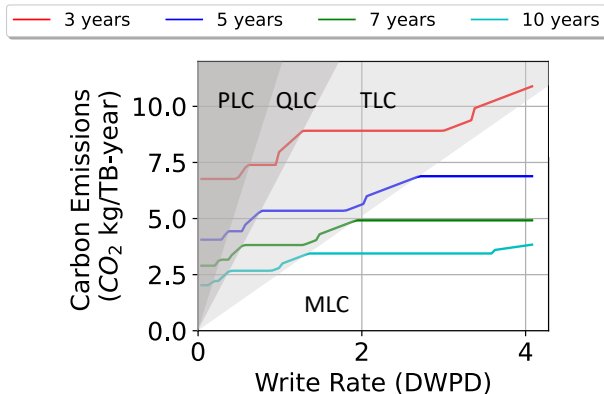


Figure 3.1: Flash carbon emissions vs write rate. The annual carbon emissions of flash depending on the required average write rate and desired lifetime. Lifetime has a much larger impact on emissions than density, but both are important to lower emissions.

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. Device lifetime is the most important factor in reducing carbon emissions. Moreover, denser flash has the potential to improve sustainability, but only if flash write rate is very small — much less than one device-write per day. However, if flash applications use dense flash, but cannot reduce their write rate enough for a long lifetime, then their carbon emissions increase over using a less dense flash device for a longer lifetime. To have low-emissions flash deployments, we need to overcome flash’s write limitation.

3.2.2 IO-per-capacity wall limits HDDs

Although hard drives are not currently write-limited like flash, they are quickly running into their own IO limitation if we want to deploy denser drives. IO bottlenecks are already becoming a challenge in datacenters for HDDs, primarily because higher-capacity HDDs do not increase their bandwidth. For instance, Seagate has LCA analysis for its Exos HDDs show that its 18 TB HDD has 59.6% fewer kg CO_2e per TB-year compared to its 10 TB drive [32, 33]. However, the 18 TB HDD’s bandwidth increases only 8.4% and has no increase in random 4KB IOPS [28, 29]. In order to use the 18 TB drives instead of 10 TB drives, we would need to reduce IO per GB stored. But there is little headroom available — many storage applications already saturate today’s HDD bandwidth (Sec. 2.3.3).

This IO limitation is universal for hard drives. Fig. 3.2 shows the decrease in bandwidth-per-TB, a $\sim 8.5\%$ reduction per year. HDDs are already running into bandwidth limitations in bulk storage. Therefore, deploying HDDs that have lower bandwidth is not possible in many datacenters today, preventing datacenters from realizing the carbon-savings of these denser drives unless bulk storage can decrease its IO requirements.

To deploy these denser drives, we need to reduce IO, but this is difficult. Storage systems already deploy large caches to take advantage of most locality in the storage

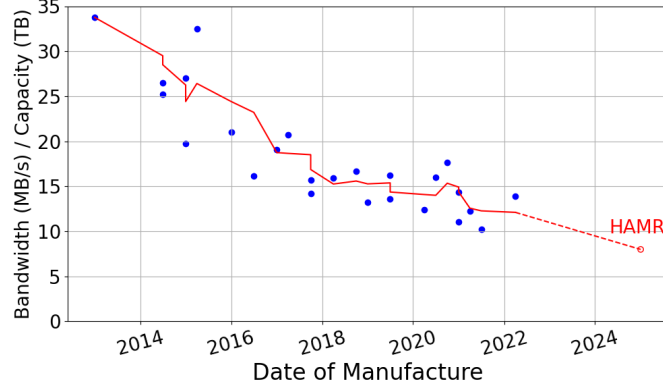


Figure 3.2: HDDs bandwidth-per-TB is dropping. HDD Sustained Bandwidth-per-TB trend over years [38]. The red empty circle denotes the speculated bandwidth/TB cost after the introduction of future disk technologies like HAMR [15, 16, 17].

accesses [60, 93, 178, 261]. Additional caching capacity also needs to be weighed against the cache’s emissions. Caching also does not help with low-locality workloads, like LSM compaction [24, 79, 95, 96, 103, 157, 208]. Thus, we need to develop new solutions to reduce IO so we can deploy fewer, denser drives and reduce emissions.

Extending lifetime in HDDs. Deploying denser drives is especially important for HDDs, since extending lifetime in HDDs is difficult due to device failure. Unlike flash, where device failures correlate to wear out, HDD failure rates increase with age. Reported annual failure rates can double going from three to six year lifetimes [143] as HDDs enter end of life [106, 107, 260]. Therefore, we focus on increasing HDD density through lowering IO to decrease HDD emissions.

Chapter 4

Kangaroo: Caching Billions of Tiny Objects on Flash

“The species has an unusual eating practice. The kangaroo regurgitates grass and shrubs that it has already eaten and chews it once more before swallowing it for final digestion.”

PBS Nature, Kangaroo Fact Sheet. [187]

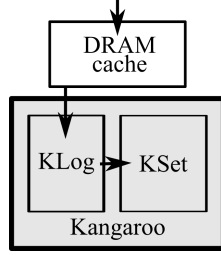
“Methane is a greenhouse gas that is roughly 25 times more powerful than carbon dioxide at trapping heat in the atmosphere, making it a key target of global efforts to fight climate change. Dairy cows and beef cattle are a significant source of methane, which is released as a byproduct of food digestion in the stomach of cows and other hoofed animals. Kangaroos, like cows, also have microbes in their gut which aid in digesting plant material. But these microbes release acetic acid instead of methane.”

Sheraz Sadiq on “Reducing methane production from rumen cultures by bioaugmentation with homoacetogenic bacteria”. [149, 213]

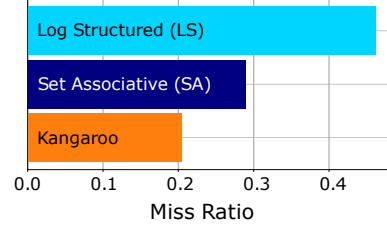
MANY WEB SERVICES require fast, cheap access to billions of tiny objects, each a few hundred bytes or less. Examples include social networks like Facebook or LinkedIn [60, 71, 253], microblogging services like Twitter [261, 262], ecommerce [63], and emerging sensing applications in the Internet of Things [111, 160, 161]. Given the societal importance of such applications, there is a strong need to cache tiny objects at high performance and low cost (i.e., capital and operational expense).

Among existing memory and storage technologies with acceptable performance, flash is by far the most cost-effective. DRAM and non-volatile memories (NVMs) have excellent performance, but both are an order-of-magnitude more expensive than flash. Thus, cost argues for *using of large amounts of flash with minimal DRAM*.

Flash’s main challenge is its limited write endurance; i.e., flash can only be written so many times before wearing out. Wearout is especially problematic for tiny objects because flash can be read and written only at multi-KB granularity. For example, writing a 100 B object may require writing a 4 KB flash page, amplifying bytes written by 40× and rapidly



(a) Overview.



(b) Kangaroo reduces misses by 29%.

Figure 4.1: Overview of Kangaroo design and results.(a) High-level illustration of Kangaroo’s design. (b) Miss ratio achieved on a production trace from Facebook by different flash-cache designs on a 1.9 TB drive with a budget of 16 GB DRAM and three device-writes per day. Prior designs are constrained by either DRAM or flash writes, whereas Kangaroo’s design balances these constraints to reduce misses by 29%.

wearing out the flash device. Thus, cost also argues for *minimizing excess bytes written to flash*.

The problem. Prior flash-cache designs either use too much DRAM or write flash too much. *Log-structured caches* write objects to flash sequentially and keep an index (typically in DRAM) that tracks where objects are located on flash [61, 105, 159, 220, 229, 237]. By writing objects sequentially and batching many insertions into each flash write, log-structured caches greatly reduce the excess bytes written to flash. However, tracking billions of tiny objects requires a large index, and even a heavily optimized index needs large amounts of DRAM [105]. *Set-associative caches* operate by hashing objects’ keys into distinct “sets,” much like CPU caches [60, 71, 196]. These designs do not require a DRAM index because an object’s possible locations are implied by its key. However, set-associative caches write many excess bytes to flash. Admitting a single small object to the cache requires re-writing an entire set, significantly amplifying the number of bytes written to the flash device.

Our solution: Kangaroo. We introduce Kangaroo, a new flash-cache design optimized for billions of tiny objects. The key insight is that existing cache designs each address half of the problem, and they can be combined to overcome each other’s weaknesses while amplifying their strengths.

Kangaroo adopts a hierarchical design to achieve the best of both log-structured and set-associative caches (Fig. 6.1).

To avoid a large DRAM index, Kangaroo organizes the bulk of cache capacity as a set-associative cache, called *KSet*. To reduce flash writes, Kangaroo places a small (e.g., 5% of flash) log-structured cache, called *KLog*, in front of *KSet*. *KLog* buffers many objects, looking for objects that map to the same set in *KSet* (i.e., hash collisions), so that each flash write to *KSet* can insert multiple objects. Our insight is that even a small log will yield many hash collisions, so only a small amount of extra DRAM (for *KLog*’s index) is needed to significantly reduce flash writes (in *KSet*).

The layers in Kangaroo’s design complement one another to maximize hit ratio while

minimizing system cost across flash and DRAM. Kangaroo introduces three techniques to efficiently realize its hierarchical design and increase its effectiveness. First, Kangaroo’s *partitioned index* lets it efficiently find all objects in KLog that map to the same set in KSet while using a minimal amount of DRAM. Second, since Kangaroo is a cache for immutable objects, not a key-value store, it is free to drop objects instead of admitting them to KSet. Kangaroo’s *threshold admission policy* exploits this freedom to admit objects from KLog to KSet only when there are enough hash collisions — i.e., only when the flash write is sufficiently amortized. Third, Kangaroo’s “RRIParoo” *eviction policy* improves hit ratio by supporting intelligent eviction in KSet, even though KSet lacks a conventional DRAM index to track eviction metadata.

Summary of results. We implement Kangaroo as a module in CacheLib [60] (cachelib.org). We evaluate Kangaroo by replaying production traces on real systems and in simulation for sensitivity studies. Prior designs are limited by DRAM usage or flash write rate, whereas Kangaroo optimizes for both constraints. For example, under typical DRAM and flash-write budgets, Kangaroo reduces misses by 29% on a production trace from Facebook (Fig. 4.1b), lowering miss ratio from 0.29 to 0.20. Moreover, in simulation, we show that Kangaroo scales well with flash capacity, performs well with different DRAM and flash-write budgets, and handles different access patterns well. We break down Kangaroo’s techniques to see how much each contributes. Finally, we show that Kangaroo’s benefits hold up in the real world through a test deployment at Facebook.

Contributions. This chapter contributes the following:

- *Problem:* We show that, for tiny objects, prior cache designs require either too much DRAM (log-structured caches) or too many flash writes (set-associative caches).
- *Key idea:* We show how to combine log-structured and set-associative designs to cache tiny objects on flash at low cost.
- *Theoretical foundations:* We develop a rigorous Markov model that shows Kangaroo reduces flash writes over a set-associative flash design without any increase in miss ratio.
- *Kangaroo design & implementation:* Kangaroo introduces three techniques to realize and improve the basic design: its partitioned index, threshold admission, and RRI-Paroo eviction. These techniques improve hit ratio while keeping DRAM usage, flash writes, and runtime overhead low.
- *Results:* We show that, unlike prior caches, Kangaroo’s design can handle different DRAM and flash-write budgets. As a result, Kangaroo is Pareto-optimal across a wide range of constraints and for different workloads.

4.1 Kangaroo Overview and Motivation

Kangaroo is a new flash-cache design optimized for billions of tiny objects. Kangaroo maximizes hit ratio while minimizing DRAM usage and flash writes. Like some key-value stores [79, 166, 175], Kangaroo adopts a hierarchical design, split across memory and flash. Fig. 4.2 depicts the two layers in Kangaroo’s design: (i) KLog, a log-structured flash cache

and (ii) KSet, a set-associative flash cache; as well as a DRAM cache that sits in front of Kangaroo.

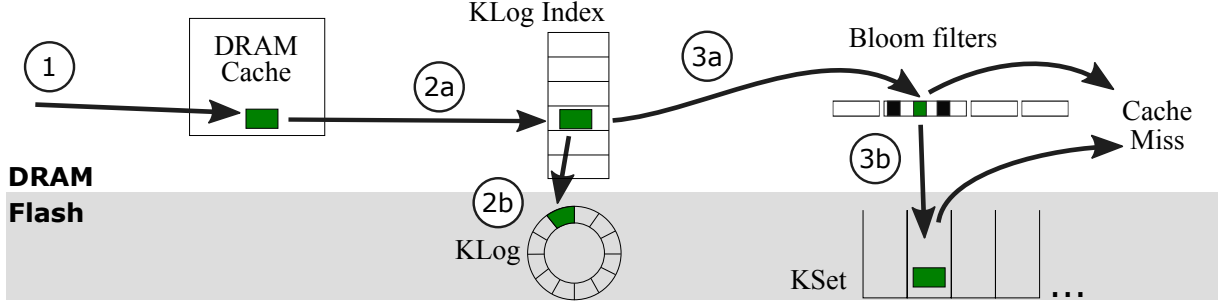


Figure 4.2: Looking up objects in Kangaroo. Lookups in Kangaroo first check a tiny DRAM cache; then KLog, a small on-flash log-structured cache with an in-DRAM index; and finally KSet, a large on-flash set-associative cache. Kangaroo uses little DRAM because KLog is small and KSet has no DRAM index.

Basic operation. Kangaroo is split across DRAM and flash. As shown in Fig. 4.2, ① lookups first check the DRAM cache, which is very small (<1% of capacity). ② If the requested key is not found, requests next check KLog ($\approx 5\%$ of capacity). KLog maintains a DRAM index to track objects stored in a circular log on flash. ③ If the key is not found in KLog’s index, requests check KSet ($\approx 95\%$ of capacity). KSet has no DRAM index; instead, Kangaroo hashes the requested key to find the set (i.e., the LBA(s) on flash) that might hold the object. ③a If the requested key is not in the small, per-set Bloom filter, the request is a miss. Otherwise, the object is probably on flash, so ③b the request reads the LBA(s) for the given set and scans for the requested key.

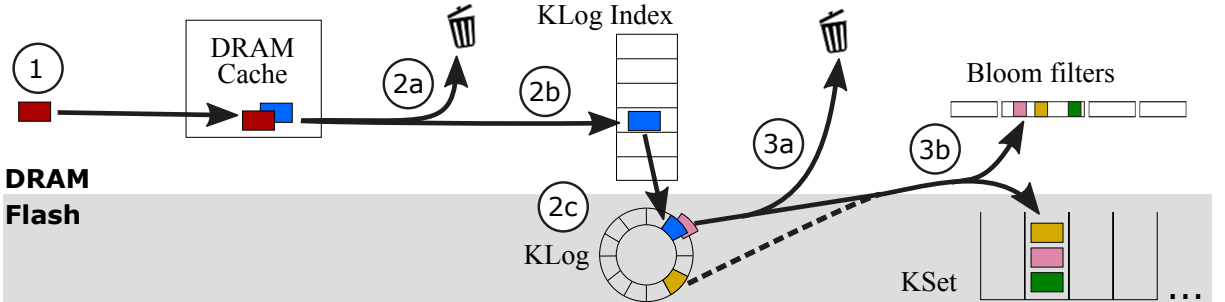


Figure 4.3: Inserting and evicting objects in Kangaroo. Objects are first inserted into the tiny DRAM cache, then appended to KLog, and finally moved — along with all other objects mapping to the same set — to KSet. Kangaroo significantly reduces write amplification because KLog is written sequentially and each write to KSet inserts multiple objects.

Insertions follow a similar procedure to reads, as shown in Fig. 4.3. ① Newly inserted items are first written to the DRAM cache. This likely pushes some objects out of the

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

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

On top of this basic design, Kangaroo introduces three techniques to minimize DRAM usage, minimize flash writes, and reduce cache misses. (i) Kangaroo’s *partitioned index* for KLog can efficiently find all objects in KLog mapping to the same set in KSet, and is split into many independent partitions to minimize DRAM usage. (ii) Kangaroo’s *threshold admission* policy between KLog and KSet only admits objects to KSet when at least n objects in KLog map to the same set, reducing ALWA by $\geq n\times$. (iii) Kangaroo’s “RRIParoo” eviction improves hit ratio in KSet by approximating RRIP [133], a state-of-the-art eviction policy, while only using a single bit of DRAM per object.

4.2 Theoretical Foundations of Kangaroo

We develop a Markov model of Kangaroo’s basic design, including threshold admission, to analyze Kangaroo’s miss ratio and flash write rate. This model rigorously demonstrates that Kangaroo can greatly reduce ALWA compared to a set-only design, without any increase in miss ratio.

Assumptions. For tractability, this analysis makes several simplifying assumptions that do not hold in our design (Sec. 4.3), implementation (Sec. 4.4), or evaluation (Sec. 4.5). We assume the independent reference model (IRM), in which each object has a fixed reference popularity, drawn independently from a known object probability distribution. However, we make no assumptions about the object popularity distribution; our model holds across any popularity distribution (uniform, Zipfian, etc.). We also assume that all objects are fixed-size and that KSet uses FIFO eviction. Similar assumptions are common in prior cache models [43, 61, 62, 88, 92, 113, 141, 197, 211].

We model a cache consisting of two layers: a log-structured cache and a set-associative cache, called KLog and KSet, as in Kangaroo — but note that the model simplifies Kangaroo’s operation significantly. We assume that an object is first admitted to KLog. Once KLog fills up, it flushes all objects to KSet. KLog and KSet may employ an admission

Variable	Description
O	Out-of-cache state.
Q	KLog state.
W	KSet state.
w	Capacity of each set.
s	Number of sets in KSet.
q	Capacity of KLog.
r_i	Probability of requesting object i .
m	Miss probability.
f	Flash write rate.
$\pi_{i,X}$	Stationary probability of object i in state X .
$X \rightarrow Y$	Transition from state X to state Y .

Table 4.1: Variables in Kangaroo model. Key variables in the Markov model with descriptions.

policy that drops objects instead of admitting them, as described below. Our goals are *(i)* to compute the miss probability and flash writes per cache access and *(ii)* to show that Kangaroo improves miss ratio for a given write rate vs. the baseline set-associative cache.

Modeling approach. We model how a single object moves through KLog and KSet. Fig. 4.4 shows our simple continuous-time Markov model, which has three states: an object can be out-of-cache (O), in KLog (Q), or in KSet (W). To compute the miss probability m , we need to know each object’s probability of being requested, which is fixed according to the IRM, and the probability that it is out-of-cache (in state O). To find the latter, we need to know each state’s stationary probabilities, π , i.e., the likelihood of an object being in a given state once the cache reaches its steady-state behavior. To compute these probabilities and to find flash write rate, we require the transition rates between states, e.g., how often an object transitions $O \rightarrow Q$ when an object is admitted to KLog. Table 4.1 summarizes the key variables in the model.

Summary of model results. Through building our model from a baseline set-associative cache, we will show that Kangaroo’s ALWA follows Theorem 1 and that Kangaroo’s miss ratio is the same as a set-associative flash cache.

Theorem 1. *Suppose KLog contains q objects; KSet contains s sets with w objects each; objects are admitted to flash with a p probability; and objects are only admitted to KSet if at least n new objects are being inserted. Kangaroo’s app-level write amplification is*

$$ALWA_{Kangaroo} = p \left(1 + \frac{w \bar{F}_X(n)}{\bar{F}_X(1) \mathbb{E}[X|X \geq n]} \right), \quad (4.1)$$

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

For a reasonable parameterization of Kangaroo on a 2 TB drive with 5% of flash dedicated to KLog ($q = 5 \cdot 10^8$, $s = 4.6 \cdot 10^8$, $w = 40$, $p = 1$, and $n = 2$), Kangaroo’s ALWA will

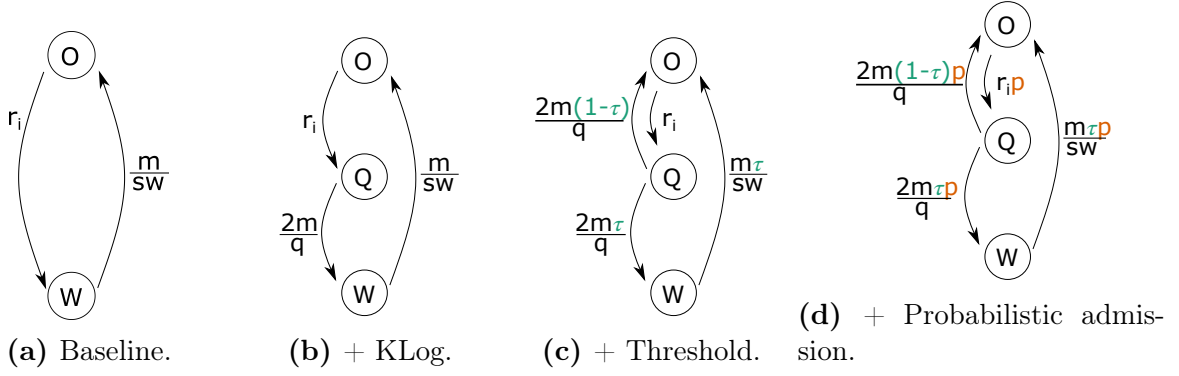


Figure 4.4: Markov Model for Kangaroo. The continuous Markov model for Kangaroo’s basic cache design (a) with no log, (b) with no admission policies, (c) with Kangaroo’s threshold admission before KSet, and (d) with probabilistic admission before KLog.

be ≈ 5.8 . In contrast, a set-associative cache of the same size and admission probability, $\mathbb{P}[X \geq n | X \geq 1] \approx 0.45$, gets $\text{ALWA}_{\text{Sets}} = w \cdot 0.45 = 17.9\times$. That is, Kangaroo improves ALWA by $\approx 3.08\times$, a large decrease in ALWA with only a small percentage of flash dedicated to KLog.

4.2.1 Baseline set-associative cache

We first analyze a baseline set-associative cache (i.e., without KLog) and build up to Kangaroo. This design has all objects admitted directly to KSet.

Transition rates: Each object i is requested with probability r_i . When an object is requested and not in the cache, there is a miss and the object moves to KSet. So, the transition rate from $O \rightarrow W$ is r_i .

Each set in KSet holds w objects. Since we are modeling FIFO eviction of fixed-sized objects, KSet evicts each object after w newer objects are inserted into the same set. With s sets, each set only receives $1/s$ of misses, so the probability of writing a new object to a set is $\frac{m}{s}$. Since there needs to be w newer objects in the set to evict an object, the transition rate from $W \rightarrow O$ is $\frac{m/s}{w}$.

Stationary probabilities: With the transition rates, we calculate the stationary probabilities using two properties of stationary probabilities: (i) the sum of all the stationary probabilities is 1 and (ii) the likelihood of entering and leaving a state is equal since stationary probabilities occur at steady-state behavior. From these, we reach the equations:

$$1 = \pi_{i,O} + \pi_{i,W} \quad (4.2)$$

$$r_i \pi_{i,O} = \frac{m}{s w} \pi_{i,W} \quad (4.3)$$

which means that the stationary probabilities are:

$$\pi_{i,O} = \frac{m}{m + s w r_i} \quad (4.4)$$

$$\pi_{i,W} = \frac{s w r_i}{m + s w r_i} \quad (4.5)$$

Miss ratio: The miss ratio is computed by summing the probability that an object will miss when it is requested for all objects. Object i is requested with probability r_i and misses when it is out-of-cache with probability $\pi_{i,O}$. Hence, the overall miss ratio m is:

$$m_{\text{baseline}} = \sum_i r_i \pi_{i,O} = \sum_i \frac{m r_i}{m + s w r_i} \quad (4.6)$$

Without knowing the popularity distribution $\{r_i\}$, we cannot go further than this; yet we will see it is sufficient to show that Kangaroo's design does not compromise miss ratio under our model.

Flash writes: To compute the flash write rate, we assign a write-cost to each edge in Fig. 4.4. For the baseline set-associative cache, each transition $O \rightarrow W$ re-writes the entire set, and so the transition has a write-cost of w . Transitions $W \rightarrow O$ do not write anything to flash, and so they have no write-cost. The flash write rate f is the average bytes written to flash on each access; that is, we compute write rate in logical time. In the baseline design, this is:

$$f_{\text{baseline}} = \sum_i r_i \cdot \pi_{i,O} \cdot w = w \cdot m_{\text{baseline}} \quad (4.7)$$

The application-level write amplification (ALWA) is the flash write rate divided by the miss rate, since each miss should ideally write exactly one object. Hence, for the baseline:

$$\text{ALWA}_{\text{baseline}} = \frac{f_{\text{baseline}}}{m_{\text{baseline}}} = w, \quad (4.8)$$

which matches our expectations for set-associative designs, since w is just the size of each set.

4.2.2 Add KLog, no admission policies

Next we add KLog, a log-structured cache, in front of KSet, as shown in Fig. 4.4b. KLog's operation in our simple model is to buffer objects until full, and then flush the log's contents to KSet.

Transition rates: The transition rate $O \rightarrow Q$ with KLog is the same as $O \rightarrow W$ in the baseline, since the only difference is that objects are written to KLog instead of KSet.

In our simplified Markov model, KLog flushes all objects in KLog to KSet when KLog is completely full, i.e., it has q objects. KLog starts with 0 objects and each miss inserts one object, so KLog is half-full when an object is admitted on average. Therefore, on average, $q/2$ objects need to be inserted until the next flush, and the transition rate from $Q \rightarrow W$ is $\frac{m}{q/2} = \frac{2m}{q}$. Finally, the transition rate from $W \rightarrow O$ is the same as the baseline.

Stationary probabilities:

$$\pi_{i,O} = \frac{2m}{q r_i + 2m + 2s w r_i} \approx \frac{m}{m + s w r_i} \quad (4.9)$$

$$\pi_{i,Q} = \frac{q r_i}{q r_i + 2m + 2s w r_i} \quad (4.10)$$

$$\pi_{i,W} = \frac{2 s w r_i}{q r_i + 2m + 2s w r_i} \quad (4.11)$$

The approximation for $\pi_{i,O}$ holds when $q \ll sw$ (i.e., when KLog is much smaller than KSet). We find that Eq. 4.9 is the same as Eq. 4.4, demonstrating that adding KLog does not significantly affect the probability an object is out-of-cache, so long as KLog is small.

Miss ratio: As a result, the miss ratio does not change either:

$$m_{\text{KLog-only}} = \sum_i r_i \pi_{i,O} \quad (4.12)$$

$$= \sum_i r_i \times \frac{2m}{q r_i + 2m + 2s w r_i} \quad (4.13)$$

$$\approx \sum_i r_i \times \frac{m}{m + s w r_i} \quad (4.14)$$

$$= m_{\text{baseline}} \quad (4.15)$$

Flash write rate: Writes are much cheaper with KLog. Since log-structured caches write out objects sequentially in batches, newly admitted objects to KLog only write one object to flash per miss. Hence the write-cost of $O \rightarrow Q$ edge is 1.

Writes to KSet are also cheaper because, even though w objects are still written to flash at a time, these writes are amortized across all objects in KLog that map to the same set. The number of objects admitted to each set is a balls-and-bins problem. Specifically, it follows a binomial distribution $X \sim B(q, 1/s)$. Each transition is amortized across $\mathbb{E}[B|B \geq 1]$ objects, as KSet only writes the set if at least one object is admitted. The total flash write rate is:

$$f_{\text{KLog-only}} = \sum_i r_i \cdot \pi_{i,O} \cdot 1 + \frac{2m}{q} \cdot \pi_{i,Q} \cdot \frac{w}{\mathbb{E}[X|X \geq 1]} \quad (4.16)$$

Which means every object suffers write amplification of:

$$\text{ALWA}_{\text{KLog-only}} = 1 + \frac{w}{\mathbb{E}[X|X \geq 1]} \quad (4.17)$$

Deriving Eq. 4.17 in detail:

$$\begin{aligned}
f_{\text{KLog-only}} &= \sum_i \frac{r_i \cdot 2m \cdot 1 + \frac{2m}{q} \cdot qr_i \cdot \frac{w}{\mathbb{E}[X|X \geq 1]}}{qr_i + 2m + 2swr_i} \\
&= \left(1 + \frac{w}{\mathbb{E}[X|X \geq 1]}\right) \times \left(\sum_i \frac{2mr_i}{qr_i + 2m + 2swr_i}\right) \\
&= \left(1 + \frac{w}{\mathbb{E}[X|X \geq 1]}\right) \times m_{\text{KLog-only}}
\end{aligned}$$

This means that KLog is responsible for 1 object write, and the rest of the writes come from KSet.

4.2.3 Add threshold admission before KSet

Next, we consider the impact of adding Kangaroo's threshold admission policy (Sec. 4.3.3), which only admits objects to a set in KSet if at least n objects map to that set. For instance, a threshold of 2 means that if KLog has only one object mapping to a set, then that object is discarded (evicted) instead of being inserted into KSet. To represent threshold admission, we add an edge in the Markov model (Fig. 4.4c) back from $Q \rightarrow O$, denoting the discarded objects.

Transition rates: We denote the probability that a set has at least n objects mapped to it during a flush of KLog as $\tau(n)$. The exact value of τ can be computed from the binomial distribution, X , where $X \sim \text{Binomial}(q, 1/s)$ given that there is one object mapping the the set, i.e., $X \geq 1$. Since objects are admitted to KSet if there are greater than n :

$$\tau(n) = \frac{\bar{F}_X(n)}{\bar{F}_X(1)}, \quad (4.18)$$

the probability that X has a value greater than n given that X has a value greater than 1.

The stream of objects flushed from KLog are split between the transition $Q \rightarrow O$ and $Q \rightarrow W$. The added edge back from $Q \rightarrow O$, represents the $1 - \tau$ fraction of the discarded objects. The remaining τ fraction transition $Q \rightarrow W$ as before. For the Markov model, the transition probabilities along those edges are multiplied by their probability.

The admission policy also reduces the admission rate to state W , which in turn causes an object to spend more time in state W . This is reflected in the transition rate $W \rightarrow O$, which is scaled by τ .

Stationary probability and miss ratio: Threshold admission adds an edge, which causes the stationary equations to be more complicated:

$$1 = \pi_{i,O} + \pi_{i,Q} + \pi_{i,W} \quad (4.19)$$

$$r_i \pi_{i,O} = \frac{m \tau}{s w} \pi_{i,W} + \frac{2 m (1 - \tau)}{q} \pi_{i,Q} \quad (4.20)$$

$$\frac{m \tau}{s w} \pi_{i,W} = \frac{2 m \tau}{q} \pi_{i,Q} \quad (4.21)$$

Surprisingly, the threshold admission policy does not change the stationary probabilities in the Markov model. Hence, the miss ratio is also unchanged:

$$m_{\text{threshold}} = m_{\text{KLog-only}} \approx m_{\text{baseline}} \quad (4.22)$$

Flash write rate: Threshold admission further reduces the write rate in two ways: (i) objects are less likely to enter KSet at all; and (ii) the write-cost of KSet is reduced because at least n objects are written. This is reflected in the flash write rate and write amplification:

$$f_{\text{threshold}} = \sum_i r_i \cdot \pi_{i,O} \cdot 1 + \frac{2m\tau}{q} \cdot \pi_{i,Q} \cdot \frac{w}{\mathbb{E}[X|X \geq n]} \quad (4.23)$$

$$\text{ALWA}_{\text{threshold}} = 1 + \frac{w}{\mathbb{E}[X|X \geq n]} \cdot \tau \quad (4.24)$$

This formula is derived similar to Eq. 4.17 above. Note that the ALWA can be easily read off from Fig. 4.4 at a glance by “following the write loop” from O back to O , adding up write costs for each edge and scaling them by their transition rate relative to KLog-only; e.g., by a factor of τ for the second term.

Kangaroo’s threshold admission policy thus greatly decreases ALWA in KSet’s set-associative design by guaranteeing a minimum level of amortization on all flash writes.

4.2.4 Add probabilistic admission before KLog

The above techniques — KLog and threshold admission — are Kangaroo’s main tricks to reduce flash writes. However, the design thus far always has write amplification at least $1\times$ because all objects are admitted to KLog. It is possible to achieve write amplification below $1\times$ by adding an admission policy in front of KLog. We now consider the effect of adding a probabilistic admission policy that drops objects with a probability p before they are admitted to KLog, as shown in Fig. 4.4d.

Transition rates: If only a fraction p of objects are admitted to KLog, then the transition rate $O \rightarrow Q$ decreases by a factor p . This factor of p propagates to all of the other transition rates.

Stationary probability and miss ratio: As with the threshold admission policy, stationary probabilities and miss ratio do not change by adding a probabilistic admission policy before KLog.

This insensitivity to admission probability reflects a limitation of the model: we assume static reference probabilities, so all popular objects will eventually make it into the cache. In practice, object popularity changes over time, so miss ratio decreases at very low admission probabilities because the cache does not admit newly popular objects quickly enough.

Flash write rate: The write-cost of each edge does not change, but probability of traversing each edge changes by a factor p . Thus:

$$f_{\text{Kangaroo}} = \sum_i p r_i \cdot \pi_{i,O} \cdot 1 + \frac{2m p \tau}{q} \cdot \pi_{i,Q} \cdot \frac{w}{\mathbb{E}[X|X \geq n]} \quad (4.25)$$

$$\text{ALWA}_{\text{Kangaroo}} = p \left(1 + \frac{w}{\mathbb{E}[X|X \geq n]} \cdot \tau \right) \quad (4.26)$$

This equation is Theorem 1, as expected.

4.2.5 Modeling results

With a full model of Kangaroo, we can not only show the ALWA benefits of Kangaroo over the set-associative baseline, but we can also predict the effect of different parameters on Kangaroo’s ALWA.

Fig. 4.5 shows the effect of KLog’s size on ALWA without an admission policy. As Kangaroo devotes more space to KLog, its ALWA greatly decreases. For instance, doubling the percent of the cache dedicated to KLog from 2.5% to 5% decreases ALWA by up to 38%. Since the overall amount of flash dedicated to KLog is small, there is only a small increase in DRAM overhead for this configuration of Kangaroo. Thus, even a small KLog greatly decreases ALWA.

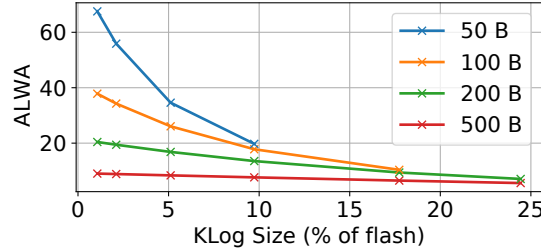


Figure 4.5: Modeled ALWA in Kangaroo with different KLog sizes. Modeled ALWA for Kangaroo with different percentages of the flash cache devoted to KLog, assuming 4 KB sets, 100% probabilistic admission, and no admission threshold.

Fig. 4.6 shows the effect of thresholding on ALWA and KSet’s admission probability for different object sizes using Theorem 1, keeping KLog at 5% of cache size. With no thresholding ($n = 1$), no objects are rejected; but as the threshold increases more objects are rejected (Fig. 4.6a). Also, since more objects fit in the KLog when objects are smaller, smaller objects are more likely to be admitted. Thresholding significantly reduces ALWA (Fig. 4.6b). Importantly, the ALWA savings are larger than the fraction of objects rejected, unlike purely probabilistic admission. For instance, with 100 B objects, threshold $n = 2$ admits 44.4% of objects, but its write rate is only 22.8% of the write rate with threshold $n = 1$.

The ALWA results are consistent with results in our experiments (Sec. 4.5), giving us confidence when using the model to explore the ALWA design space.

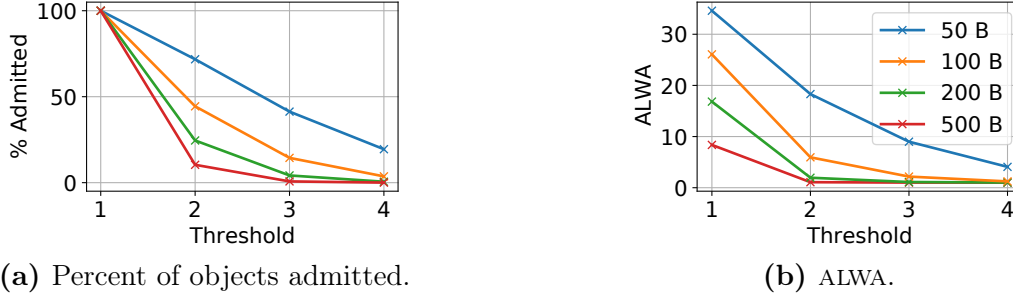


Figure 4.6: Modeled ALWA in Kangaroo with different thresholds. Modeled (a) admission percentage and (b) ALWA for Kangaroo with different threshold values and object sizes, assuming 4 KB sets and KLog w/ 5% of capacity.

4.3 Kangaroo Design

This section describes the techniques introduced in KLog and KSet to reduce DRAM, flash writes, and miss ratio.

4.3.1 Pre-flash admission to KLog

Like previous flash caches, Kangaroo may not admit all objects evicted from the DRAM cache [60, 86, 87, 105, 108, 110]. It has a pre-flash admission policy that can be configured to randomly admit objects to KLog with probability p , decreasing Kangaroo’s write rate proportionally without additional DRAM overhead. Compared to prior designs, Kangaroo can afford to admit a larger fraction of objects to flash than prior flash caches due to its low ALWA; in fact, except at very low write budgets, Kangaroo admits almost all objects to KLog.

4.3.2 KLog

KLog’s role is to minimize the flash cache’s ALWA without requiring much DRAM. To accomplish this, it must support three main operations: LOOKUP, INSERT, and ENUMERATE-SET. ENUMERATE-SET allows KLog to find all objects mapping to the same set in KSet. LOOKUP and INSERT operate similarly to a conventional log-structured cache with an approximate index. However, the underlying data structure is designed so that ENUMERATE-SET is efficient and has few false positives.

Operation. Like other log-structured caches, KLog writes objects to a circular buffer on flash in large batches and tracks objects via an index kept in DRAM. To support ENUMERATE-SET efficiently, KLog’s index is implemented as a hash table using separate chaining. Each index entry contains an `offset` to locate the object in the flash log, a `tag` (partial hash of the object’s key), a `next-pointer` to the next entry in the chain (for collision resolution), eviction-policy metadata (described in Sec. 4.3.4), and a `valid` bit.

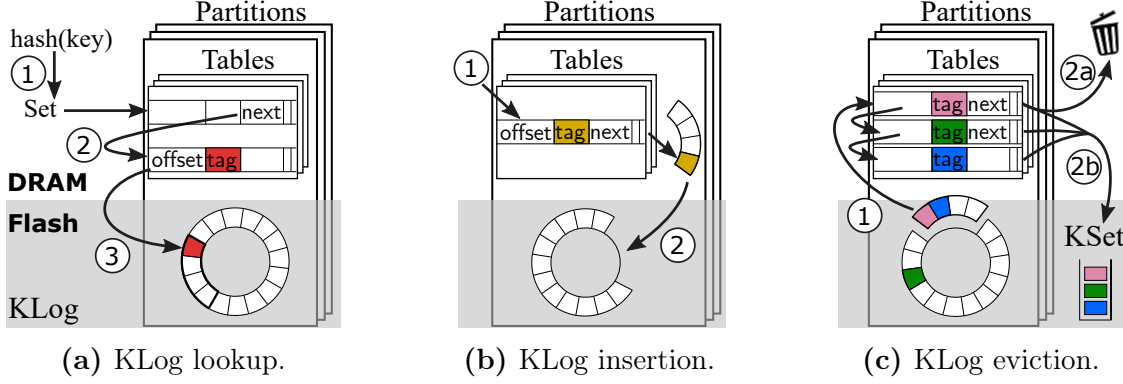


Figure 4.7: Overview of KLog operations. KLog allows for lookup, insertion, and eviction of objects.

LOOKUP: To look up a key (Fig. 4.7a), ① KLog determines which bucket it belongs to by *computing the object’s set* in KSet. ② KLog traverses index entries in this bucket, ignoring invalid entries, until a **tag** matches a hash of the key. If there is no matching **tag**, KLog returns a miss. ③ KLog reads the flash page at **offset** in the log. After confirming a full key match, KLog returns the data and updates eviction-policy metadata.

INSERT: To insert an object (Fig. 4.7b), ① KLog creates an index entry, adds it to the bucket corresponding to the key’s set in KSet, and appends the object to an in-DRAM buffer. The on-flash circular log is broken into many *segments*, one of which is buffered in DRAM at a time. ② Once the segment buffer is full, it is written to flash.

ENUMERATE-SET: The $\text{ENUMERATE-SET}(x)$ operation returns a list of all objects currently in KLog that map to the same set in KSet as object x . This operation is efficient because, by construction, all such objects will be in the same bucket in KLog’s index. That is, KLog *intentionally exploits hash collisions* in its index so that it can enumerate a set simply by iterating through all entries in one index bucket.

Internal KLog structure. As depicted in Fig. 4.7, KLog is structured internally as multiple *partitions*. Each partition is an independent log-structured cache with its own flash log and DRAM index. Moreover, each partition’s index is split into multiple *tables*, each an independent hash table.

This partitioned structure reduces DRAM usage, as described next, but otherwise changes the operation of KLog little. The table and partition are inferred from an object’s set in KSet. Hence, all objects in the same set will belong to the same partition, table, and bucket; and operations work as described above within each table.

Reducing DRAM usage in KLog. Table 5.3 breaks down Kangaroo’s DRAM usage per object vs. a naïve log-structured design as a standalone cache (“Naïve Log-Only”) and as a drop-in replacement for KLog (“Naïve Kangaroo”).

The flash **offset** must be large enough to identify which page in the flash log contains the object, which requires $\log_2(\text{LogSize}/4 \text{ KB})$ bits. By splitting the log into 64 partitions, KLog reduces *LogSize* by $64\times$ and saves 6 b in the pointer.

	Component	Naïve Log-Only	Naïve Kangaroo	Kangaroo
KLog Index	offset	29 b	25 b	19 b
	tag	29 b	29 b	9 b
	next-pointer	64 b	64 b	16 b
	Eviction metadata	67 b	58 b	3 b
	valid	1 b	1 b	1 b
	<i>Sub-total</i>	<i>190 bits/obj</i>	<i>177 bits/obj</i>	<i>48 bits/obj</i>
KSet	Bloom filter	–	3 b	3 b
	Eviction	–	5 b	1 b
	<i>Sub-total</i>	–	<i>8 bits/obj</i>	<i>4 bits/obj</i>
Overall	Index buckets	≈ 3.1 b	≈ 3.1 b	≈ 0.8 b
	Log size	100% = 181 b	5% = 8.9 b	5% = 2.4 b
	Set size	0%	95% = 7.6 b	95% = 3.8 b
	Total	193.1 bits/obj	19.6 bits/obj	7.0 bits/obj

Table 4.2: DRAM overhead in Kangaroo. Breakdown of DRAM per object for a 2 TB cache, comparing Kangaroo to a naïve log-structured cache and Kangaroo with a naïve log index. Bucket and LRU overhead assume 200 B objects.

The **tag** size determines the false-positive rate in the index; i.e., a smaller tag leads to higher read amplification. KLog splits the index into 2^{20} tables. Since the table is inferred from the key, all keys in one table effectively share 20 b of information, and KLog can use a much smaller tag to achieve the same false positive rate as the naïve design.¹

KLog’s structure also reduces the **next-pointer** size. We only need to know the offset into memory allocated to the object’s index table. Thus, rather than using a generic memory pointer, we can store a 16 b offset, which allows up to 2^{16} items per table. KLog can thus index 2^{36} items as parameterized (12.5 TB of flash with 200 B objects), which can be increased by splitting the index into more tables.

In a naïve cache using LRU eviction, each entry keeps a pointer to adjacent entries in the LRU list. This requires $2 \cdot \log_2(\text{LogSize}/\text{ObjectSize})$ bits. In contrast, Kangaroo’s RRIParoo policy (Sec. 4.3.4) is based on RRIP [133] and only needs 3 b per object in KLog (and even less in KSet).

Finally, each bucket in KLog’s index requires one pointer for the head of the chain. In naïve logs, this is a 64 b pointer. In KLog, it is a 16 b offset into the table’s memory. KLog allocates roughly one bucket per set in KSet. With 4 KB sets and 200 B objects, the per-object DRAM overhead is 3.1 b (Naïve) or 0.8 b (KLog) per object.

All told, KLog’s partitioned structure reduces the per-object metadata from 190 b to 48 b per object, a $3.96\times$ savings vs. the naïve design. Compared to prior index designs, KLog’s index uses slightly more DRAM per object than the state-of-the-art (30 b per object in Flashield [105]), but it supports ENUMERATE-SET and has fewer false positives. Most importantly, *KLog only tracks $\approx 5\%$ of objects in Kangaroo, so indexing overheads are just 3.2 b per object.* Adding KSet’s DRAM overhead gives a total of 7.0 b per object, a $4.3\times$

¹Processor caches reduce tag size vs. a fully associative cache similarly; each index table in KLog corresponds to a “set” in the processor cache.

improvement over the state-of-the-art.

4.3.3 KLog \rightarrow KSet: Minimizing flash writes

Write amplification in KLog is not a significant concern because it has a ALWA close to $1\times$ and writes data in large segments, minimizing DLWA. However, KSet’s write amplification is potentially problematic due to its set-associative design. Kangaroo solves this by using KLog to greatly reduce ALWA in KSet: namely, by amortizing each flash write in KSet across multiple admitted objects.

Moving objects from KLog to KSet. A background thread keeps one segment free in each log partition. This thread flushes segments from the on-flash log in FIFO order, moving objects from KLog to KSet as shown in Fig. 4.7c. For each victim object in the flushed segment, this thread ① calls `ENUMERATE-SET` to find all other objects in KLog that should be moved with it; ②a if there are not enough objects to move (see below), the victim object is dropped or, if popular, is re-admitted to KLog; ②b otherwise, the victim object and all other objects returned by `ENUMERATE-SET` are moved from KLog to KSet in a single flash write.

Instead of flushing one segment at a time, one could fill the entire log and then flush it completely. But this leaves the log half-empty, on average. Flushing one segment at a time keeps KLog’s capacity utilization high, empirically 80–95%. Incremental flushing also increases the likelihood of amortizing writes in KSet, since each object spends roughly twice as long in KLog and is hence more likely to find another object in the same set when flushed.

Threshold admission to KSet. Kangaroo amortizes writes in KSet by flushing all objects in the same set together, but inevitably some objects will be the only ones in their set when they are flushed. Moving these objects to KSet would result in the same excessive ALWA as a naïve set-associative cache. Thus, Kangaroo adds an admission policy between KLog and KSet that sets a *threshold*, n , of objects required to write a set in KSet. If `ENUMERATE-SET(x)` returns fewer than n objects, then x is not admitted to KSet.

To avoid unnecessary misses to popular objects that do not meet the threshold when moving from KLog to KSet, Kangaroo readmits any object that received a hit during its stay in KLog back to the head of the log. This lets Kangaroo retain popular objects while only slightly increasing overall write amplification (due to KLog’s minimal ALWA).

4.3.4 KSet

KSet’s role is to minimize the overall DRAM overhead of the cache. KSet employs a set-associative cache design similar to CacheLib’s Small Object Cache [60]. This design splits the cache into many *sets*, each holding multiple objects; by default, each set is 4 KB, matching flash’s read and write granularity. KSet maps an object to a set by hashing its key. Since each object is restricted to a small number of locations (i.e., one set), an index is not required. Instead, to look up a key, KSet simply reads the entire set off flash and scans it for the requested key.

To reduce unnecessary flash reads, KSet keeps a small Bloom filter in DRAM built from all the keys in the set. These Bloom filters are sized to achieve a false positive rate of about 10%. Whenever a set is written, the Bloom filter is reconstructed to reflect the set’s contents.

RRIParoo: Usage-based eviction without a DRAM index. Usage-based eviction policies can significantly improve miss ratio, effectively doubling the cache size (or more) without actually adding any resources [58, 61, 67, 133, 229]. Unfortunately, implementing these policies on set-associative flash caches is hard, as such policies involve per-object metadata that must be updated whenever an object is accessed. Since KSet has no DRAM index to store metadata and cannot update on-flash metadata without worsening ALWA, it is not obvious how to implement a usage-based eviction policy. For these reasons, most flash caches use FIFO eviction [4, 5, 7, 24, 60, 78, 111, 118, 219, 245], which keeps no per-object state. Unfortunately, FIFO significantly increases miss ratio because popular objects continually cycle out of the cache.

Kangaroo introduces RRIParoo, a new technique to efficiently support usage-based eviction policies in flash caches. Specifically, RRIParoo implements RRIP [133], a state-of-the-art eviction policy originally proposed for processor caches, while using only ≈ 1 bit of DRAM per object and without any additional flash writes.

Background: How RRIP works. RRIP is essentially a multi-bit clock algorithm: RRIP associates a small number of bits with each object (3 bits in Kangaroo), which represent reuse predictions from NEAR reuse (000) to FAR reuse (111). Objects are evicted only once they reach FAR. If there are no FAR objects when something must be evicted, all objects’ predictions are incremented until at least one is at FAR. Objects are promoted to NEAR (000) when they are accessed. Finally, RRIP inserts new objects at LONG (110) so they will be evicted quickly, but not immediately, if they are not accessed again. This insertion policy handles scans that can degrade LRU’s performance.

RRIParoo’s key ideas. There are two ideas to support RRIP in KSet. First, rather than tracking all of RRIP’s predictions in a DRAM index, RRIParoo stores the eviction metadata in flash and keeps only a small portion of it in DRAM. Second, to reduce DRAM metadata to a single bit, we observe that RRIP only updates predictions upon eviction (incrementing predictions towards FAR) and when an object is accessed (promoting to NEAR). Our insight is that, so long as KSet tracks which objects are accessed, *promotions can be deferred to eviction time*, so that *all* updates to on-flash RRIParoo metadata are only made at eviction, when the set is being re-written anyway. Hence, since KSet can track whether an object has been accessed using only single bit in DRAM, KSet achieves the hit-ratio of a state-of-the-art eviction policy with one-third of the DRAM usage (1 b vs. 3 b).

RRIParoo operation. RRIParoo allocates enough metadata to keep one DRAM bit per object on average; e.g., 40 b for 4 KB sets and 100 B objects. Objects use the bit corresponding to their position in the set (e.g., the i^{th} object uses the i^{th} bit), so there is no need for an index. If there are too many objects, RRIParoo does not track hits for the objects closest to NEAR, as they are least likely to be evicted.

Kangaroo also uses RRIP to merge objects from KLog. Tracking hits in KLog is trivial

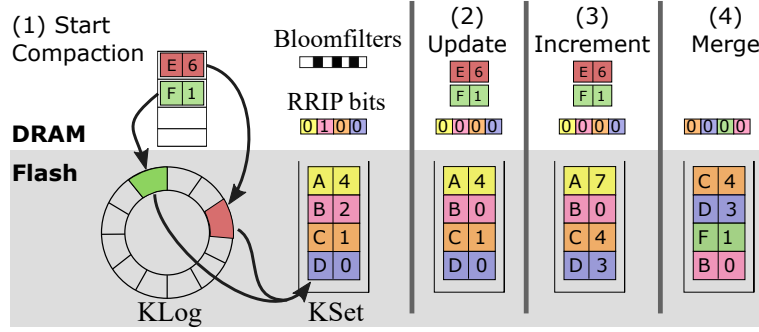


Figure 4.8: Kangaroo’s RRIParoo eviction policy. RRIParoo implements RRIP eviction with only ≈ 1 b in DRAM per object and no additional flash writes.

because it already has a DRAM index. Objects are inserted into KLog at LONG prediction (like usual), and their predictions are decremented towards NEAR on each subsequent access. Then, when moving objects from KLog to KSet, KSet sorts objects from NEAR to FAR and fills up the set in this order until out of space, breaking ties in favor of objects already in KSet.

Example: Fig. 4.8 illustrates this procedure, showing how a set is re-written in KSet. ① We start when KLog flushes a segment containing object **F**, which maps to a set in KSet. KLog finds a second object, **E**, elsewhere in the log that also maps to this set. Meanwhile, the set contains objects **A**, **B**, **C**, and **D** with the RRIP predictions shown on flash, and **B** has received a hit since the set was last re-written, as indicated by bits in DRAM. ② Since **B** received a hit, we promote its RRIP prediction to NEAR and clear the bits in DRAM. ③ Since no object is at FAR, we increment all objects’ predictions by 3, whereupon object **A** reaches FAR. ④ Finally, we fill up the set by merging objects in DRAM and flash in prediction-order until the set is full. The set now contains **B**, **F**, **D**, and **C**; **A** was evicted, and **E** stays in KLog for now since its KLog segment was not flushed. (The set on flash is only written once, after the above procedure completes.)

DRAM usage. As shown in Table 5.3, KSet needs up to 4 bits in DRAM per object: one for RRIParoo and three for the Bloom filters. Combined with the DRAM usage of KLog that contains about 5% of objects, Kangaroo needs ≈ 7.0 b per object, $4.3\times$ less than Flashield [105]. Moreover, the 1 b per object DRAM overhead for RRIParoo can be lowered by tracking fewer objects in each set. Taken to the extreme, this would cause the eviction policy to decay to FIFO, but it allows Kangaroo to adapt to use less DRAM if desired.

4.4 Experimental Methodology

This section describes the experimental methodology that we use for to evaluate Kangaroo in Sec. 4.5.

4.4.1 Kangaroo implementation and parameterization

We implement Kangaroo in C++ as a module within the CacheLib caching library [60]. Table 5.4 describes Kangaroo’s default parameters; we evaluate sensitivity to these parameters in Sec. 4.5.3.

Parameter	Value
Total cache capacity	93% of flash
Log size	5% of flash
Admission probability to log from DRAM	90%
Admission threshold to sets from log	2
Set size	4 KB

Table 4.3: Kangaroo’s default parameters.

4.4.2 Comparisons

We compare Kangaroo to (i) CacheLib’s small object cache (SOC), a set-associative design that currently serves the Facebook Social Graph [71] in production; and (ii) an optimistic version of a log-structured cache (LS) with a full DRAM index. For LS, we configure KLog to index the entire flash device and use FIFO eviction.

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

Except where noted, all experiments are configured to stay within 16 GB of DRAM (all-inclusive — DRAM cache, index, etc.); 62.5 MB/s flash writes, as measured directly from the device (i.e., including DLWA); and 100 K requests/s, similar to what is achieved by flash caches in production [60, 69]. To mimic a memory-constrained system, we limit LS’s flash capacity to the maximum allowed by a 16 GB index assuming 30 b/object, the best reported in the literature [105], but also grant LS an *additional* 16 GB for its DRAM cache. Note that this is optimistic for LS, as DRAM is LS’s main constraint. We use this variant as we were unable to compare to state-of-the-art systems: the source code of Flashield [105] is not available, and we were unable to run FASTER [78] as a cache. All systems use CacheLib’s probabilistic pre-flash admission policy.

4.4.3 Simulation

To explore a wide range of parameters and constraints, we implemented a trace-driven cache simulator for Kangaroo. The simulator measures miss ratio and application-level write rate. We estimate device-level write amplification based on our results in Sec. 2.2.4, using a best-fit exponential curve to the DLWA of random, 4 KB writes for SOC and Kangaroo,

and assuming a DLWA of $1\times$ (no write amplification) for LS. Note that this is pessimistic for Kangaroo, since writes to KLog incur less DLWA than SOC. Comparing the results with our experimental data shows the simulator to be accurate within 10%. The simulator does not implement some features of CacheLib including promotion to the memory cache, which can affect miss ratios, but we have found it able to give a good indication of how the full system would perform as parameters change.

4.4.4 Workloads

Our experiments use sampled 7-day traces from Facebook [60] and Twitter [261]. These traces have average object sizes of 291 B and 271 B, respectively. For systems experiments, we scale the Facebook trace to achieve 100 K reqs/s by running it $3\times$ concurrently in different key spaces.

The simulator results use sampled-down traces, and we scale-up the measurements to a full-server equivalent based on the server’s flash capacity and desired load as described below, Sec. 4.4.6. We use 1% of the keys for the Facebook trace and 10% of the keys for the Twitter trace. Unless otherwise noted, we report numbers for the last day of requests, allowing the cache to warm up and display steady-state behavior.

4.4.5 Metrics

Kangaroo is designed to balance several competing constraints that limit cache effectiveness. As such, our evaluation focuses on *cache miss ratio*, i.e., the fraction of requests that must be served from backend systems, under different constraints. We further report on Kangaroo’s raw performance, showing it is competitive with prior designs.

4.4.6 Scaling traces

Our scaling methodology allows us explore a wide range of system parameters in simulation. This methodology builds on prior analysis of scaling caches [56, 57, 152, 185, 217, 246], and Table 4.4 summarizes its key parameters.

The model involves three free parameters that let us: (i) choose the load on each server; (ii) choose the flash cache size on each server; and (iii) down-sample requests to accelerate simulations. Moreover, the methodology incorporates three constraints to exclude infeasible configurations: request throughput, flash write rate, and DRAM usage.

Goals for scaling traces. The starting point for our methodology is a trace collected from a real, production system. For simplicity and without loss of generality, we assume that the trace is gathered from a single caching server. This trace’s requests arrive at a rate R_o (measured in, e.g., requests/s).

The goal of our methodology is to use this trace to explore other system configurations. In particular, we want to explore caching systems with fewer or more caching servers and with different amounts of resources at each individual server. We do this by modeling the performance of a single server in the desired system configuration. Last but not least, we

Param	Description
R	Request rate.
ℓ	Relative load factor.
S	Flash cache size.
W	App-level write rate (w/out DLWA).
D	Device-level write rate.
k	Trace sampling rate.
Q	Per-server DRAM capacity.
x_o	Param x in original system.
x_m	Param x in modeled system.
x_s	Param x in simulated system.

Table 4.4: Key parameters in trace scaling methodology.

want to be able to do this efficiently, i.e., without needing to actually duplicate the original production system, by running scaled-down simulations.

Load factor and request rate per server. The first choice in the methodology is the load factor on each server, which changes the number of servers in the cluster. In the original system, each server serves requests at a rate R_o — by increasing or decreasing this rate, we effectively scale the number of caching servers that are needed to serve all user requests.

The parameter ℓ sets the *relative load factor* at each server. That is, the request rate at each server in the modeled system is

$$R_m = \ell \cdot R_o, \quad (4.27)$$

and the total number of caching servers in the modeled system scales $\propto 1/\ell$.

The load factor is clearly an important parameter. In general, a higher load factor is desirable, as higher load reduces the number of servers needed to serve all user requests. However, load factor is constrained by the maximum request throughput at a single server R_{\max} . Specifically, the maximum load factor is

$$\ell_{\max} = R_{\max}/R_o. \quad (4.28)$$

Higher load factors may also not be desirable because higher load increases flash write rate and, at a fixed cache size per server, increases miss ratio. Hence, the best load factor will depend on a number of factors, including properties of the trace like object size and locality (i.e., miss ratio curve).

Flash cache size. The next choice is the per-server flash cache size, S_m . This is a free parameter constrained only by flash write rate and the size of the flash device. (A log-structured cache size is also constrained by DRAM, as discussed below.)

This parameter is significant because it determines the miss ratio at each server. A bigger cache is usually better, until write amplification or DRAM usage exceed the server’s constraints. For a given cache design, at cache size S_m it will achieve miss ratio of $m_m(S_m)$

and a flash write rate (excluding DLWA) of

$$W_m \propto m_m(S_m) \cdot R_m. \quad (4.29)$$

The miss ratio $m(S)$ depends on the system, because load factor varies between systems. Application-level write rate W is scaled by a design-specific factor corresponding to the cache design’s ALWA — this factor is large for set-associative designs like SOC, smaller for Kangaroo, and essentially $1\times$ for log-structured caches like LS.

The maximum cache size S_{\max} is determined from the flash-write constraint, D_{\max} . Specifically, we multiply the application-level flash write rate W_m by the estimated DLWA to get the device-level write rate D_m . We then sweep the flash cache size S_m to find the sizes that stay within the constraint. Increasing cache size has two competing effects on write rate: larger caches generally have fewer misses, leading to fewer insertions, but they also suffer higher DLWA, increasing the cost of each insertion. As a result, the maximum size usually lies on the “knee” of the DLWA curve (see Fig. 2.5), though which size hits the knee depends on the cache design (via ALWA), the load factor (via R_m), and the trace itself (via m_m).

We are now ready, in principle, to run experiments to model the desired system. By replaying the original trace, which has a request rate of R_o , we are simulating a system at $1/\ell$ -scale of the desired system (since $R_o = R_m/\ell$). We therefore need to scale the cache size in our experiments by the same factor, simulating a cache of size $S_s = S_m/\ell$. (This is why increasing load factor can hurt miss ratio: all else equal, a larger load factor reduces effective cache capacity.) We can then interpret results by scaling them up by a factor ℓ , e.g., rescaling the measured write rate W_s to report a modeled write rate of $W_m = \ell \cdot W_s$. We can accelerate experiments further by employing the same trick more aggressively.

Accelerating simulations by sampling down. To speedup simulation, we downsample the original trace by pseudorandomly selecting keys to produce a new, sampled trace that we will use in the actual simulation experiments. This trace has a request rate of R_s , yielding an empirically measured sampling rate of

$$k = R_s/R_o \quad (4.30)$$

Downsampling by $k \ll 1$ makes simulations take many fewer requests and also lets simulated flash capacity fit in DRAM, significantly accelerating each experiment.

We must scale down the other resources in the system to match the downsampled trace. The simulated cache size is

$$S_s = k \cdot S_m. \quad (4.31)$$

With this scaling, simulated write rate needs to be scaled up by $1/k$ to compute the modeled system’s write rate

$$W_m = W_s/k. \quad (4.32)$$

However, simulated miss ratio does not change

$$m_m(S_m) = m_m(S_s/k) = m_s(S_s). \quad (4.33)$$

(Miss ratio is invariant under sampling because it is the ratio of rates, so the scaling factors cancel.)

DRAM constraints per server. In addition to other constraints, systems are constrained in their DRAM usage. This is particularly important for log-structured caches like LS, but every system includes a DRAM cache that has a (modest) impact on results. We enforce DRAM constraints by observing that the DRAM : flash ratio should be held constant between the simulated and modeled system. So, given a fixed DRAM capacity in the modeled system Q_m (e.g., 16 GB), the flash cache size in the modeled system S_m , and the simulated flash cache size S_s , it is trivial to compute the simulated DRAM budget:

$$Q_s = \frac{Q_m S_s}{S_m} \quad (4.34)$$

For each simulation, we compute the DRAM overhead for that cache design (e.g., for its DRAM index and Bloom filters), and use the remaining DRAM capacity as a DRAM cache. For LS, flash cache size is often limited by DRAM usage, not the flash write rate or device size.

The methodology in practice. The above describes the methodology from a top-down perspective, starting from the decisions that have the largest impact on performance and cost. In practice, we use this methodology to understand the parameter limitations for the simulator. Then, the scaling methodology is applied in the opposite direction, starting from the parameters of a specific simulation experiment and backing out the modeled system configuration for any given simulation.

Specifically, we run each simulation with a DRAM capacity Q_s , flash size S_s , and trace sampled at rate k . These experiments produce a miss ratio $m_s(S_s)$ and application-level flash write rate W_s .

Then, given a fixed DRAM budget in the modeled system Q_m , we compute the full properties of the modeled caching system. We compute the size of the modeled flash cache as

$$S_m = \frac{Q_m S_s}{Q_s}. \quad (4.35)$$

This is the flash cache size that respects the modeled DRAM constraint and keeps DRAM : flash ratio constant. Moreover, to maintain miss ratio, the ratio of cache sizes S_m/S_s must equal the ratio of request rates R_m/R_s . We want to model a system receiving $R_m = \ell \cdot R_o$ requests, but actually run a simulation with $R_s = k \cdot R_o$ requests. Hence, the load factor is

$$\ell = \frac{R_m}{R_s} k = \frac{S_m}{S_s} k, \quad (4.36)$$

yielding a modeled request rate of

$$R_m = \frac{S_m}{S_s} R_s \quad (4.37)$$

Finally, we scale the write rate and estimate DLWA for size S_m

$$D_m = \text{DLWA}(S_m) \cdot \frac{W_s}{k}. \quad (4.38)$$

This methodology lets us run short simulations to estimate the behavior of a wide range of modeled caching systems, while obeying constraints faced by production servers.

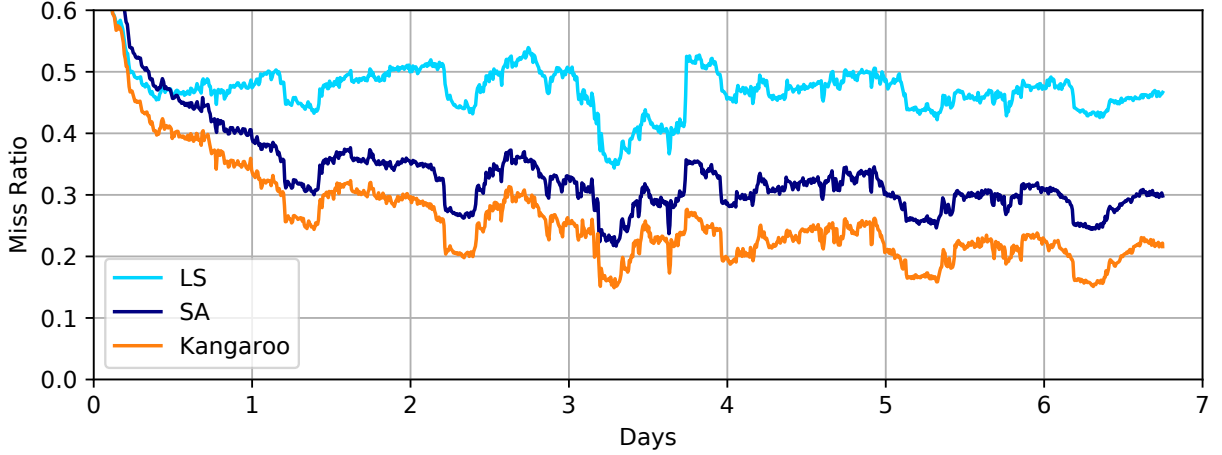


Figure 4.9: Miss ratio over time. Miss ratio for all three systems over a 7-day Facebook trace. All systems are run with 16 GB DRAM, a 1.9 TB drive, and with write rates less than 62.5 MB/s.

4.5 Evaluation

This section presents experimental results from Kangaroo and prior systems. We find that: *(i)* Kangaroo reduces misses by 29% under realistic system constraints. *(ii)* Kangaroo improves the Pareto frontier when varying constraints. *(iii)* In a production deployment, Kangaroo reduces flash-cache misses by 18% at equal write rate and reduces write rate by 38% at equal miss ratios. We also break down Kangaroo by technique to see where its benefits arise.

4.5.1 Main result: Kangaroo significantly reduces misses vs. prior cache designs under realistic constraints

Kangaroo aims to achieve low miss ratios for tiny objects within constraints on flash-device write rate, DRAM capacity, and request throughput. This section compares Kangaroo against SOC and LS on the Facebook trace, running our CacheLib implementation of each system under these constraints. We configure each cache design to minimize cache miss ratio while maintaining a device write rate lower than 62.5 MB/s and using up to 16 GB of memory and 1.9 TB of flash. Later sections will consider how performance changes as these constraints vary.

Miss ratio: Fig. 4.9 shows that Kangaroo reduces cache misses by 29% vs. SOC and by 56% vs. LS. This is because *Kangaroo makes effective use of both limited DRAM and flash writes*, whereas prior designs are hampered by one or the other. Specifically, SOC is limited primarily by its high write rate, which forces it to admit a lower percentage of objects to flash and to over-provision flash to reduce device-level write amplification. SOC uses only 81% of flash capacity. Similarly, LS is limited by the reach of its DRAM index. LS warms up as quickly as Kangaroo until it runs out of indexable flash capacity at 61% of device

capacity, even though we provision LS extra DRAM for both an index and DRAM cache (Sec. 4.4.2).

By contrast, Kangaroo uses 93% of flash capacity, increasing cache size by 15% vs. SOC and by 52% vs. LS. On top of its larger cache size, Kangaroo’s has lower ALWA than SOC and its RRIParoo policy makes better use of cache space.

Request latency and throughput: Kangaroo achieves reasonable throughput and tail latencies. When measuring flash cache performance without a backing store, Kangaroo’s peak throughput is 158 K gets/s, LS’s is 172 K gets/s, and SOC’s is 168 K gets/s. Kangaroo achieves 94% of SOC’s throughput and 91% of LS’s throughput, and it is well above typical production request rates [60, 69, 253].

In any reasonable caching deployment, request tail latency will be set by cache misses as they fetch data from backend systems. However, for completeness and to show that Kangaroo has no performance pathologies, we present tail latency at peak throughput. Kangaroo’s p99 latency is 736 μ s, LS’s is 229 μ s, and SOC’s is 699 μ s. All of these latencies are orders-of-magnitude less than typical SLAs [2, 3, 69, 267], which are set by backend databases or file systems. For instance, in production, the p99 latency in Facebook’s social-graph cache is 51 ms and Twitter’s is 8 ms, both orders-of-magnitude larger than Kangaroo’s p99 latency. In addition, Kangaroo might reduce p99 latency in practice, because its improved hit ratio reduces load on backend systems.

4.5.2 Kangaroo performs well as constraints change

Between different environments and over time, system constraints will vary. Using our simulator, we now evaluate how the cache designs behave when changing four parameters: device write budget, DRAM budget, flash capacity, and average object size.

Device write budget. Device write budgets change with both the type of flash SSD and the desired lifetime of the device. To explore how this constraint affects miss ratio, we simulate the spread of miss ratios we can achieve at different device-level write rates. To change the device-level write rate, we vary both the utilized flash capacity percentage and the admission policies for all three systems while holding the total DRAM and flash capacity constant. Note that LS can never use the entire device in these experiments, because its index is limited by DRAM capacity.

Figure 4.10 shows the tradeoff between device-level write rate budget and miss ratio. At 62.5 MB/s (the default) on both the Facebook and Twitter workloads, Kangaroo consistently performs better than both SOC and LS. At higher write budgets, Kangaroo continues to have lower miss ratio. In this range, SOC suffers both due to its FIFO eviction policy and its higher ALWA, which shifts points farther right vs. similar Kangaroo configurations. LS is mostly constrained by DRAM capacity, which is why its achievable miss ratio does not change above 15 MB/s for both traces. However, at very low device-level write budgets, LS performs better than Kangaroo, because Kangaroo is designed to balance DLWA and DRAM capacity, whereas LS focuses only on DLWA. At extremely low write budgets, Kangaroo’s higher DLWA (in KSet) forces it to admit fewer objects. (Kangaroo configurations

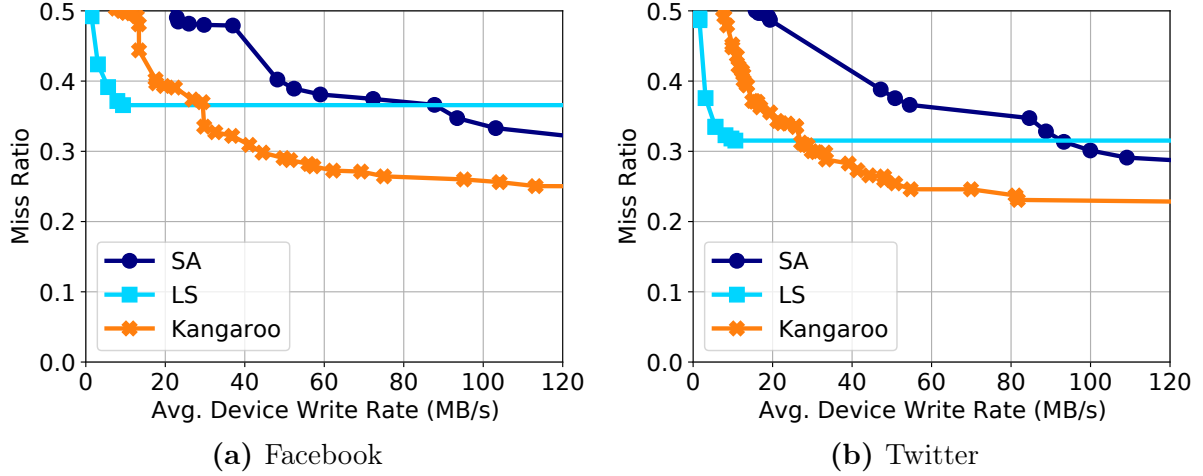


Figure 4.10: Miss ratio vs device-level write rates. Pareto curve of cache miss ratio at different device-level write rates under 16 GB DRAM and 2 TB flash capacity. At very low write rates, LS is best, but it is limited by DRAM from scaling to larger caches. Thus, for most write rates, Kangaroo outperforms both systems because it can take advantage of the entire cache capacity, has a lower write rate than SOC, and has a better eviction policy than the other two systems.

where KLog holds a large fraction of objects, which we did not evaluate, would solve this problem.)

DRAM capacity. Over time, the typical ratio of DRAM to flash in data-center servers is decreasing to reduce cost [236]. Figure 4.11 compares miss ratios for DRAM capacities up to 64 GB, holding flash-device capacity fixed at 2 TB and device-write rate at 62.5 MB/s. DRAM capacity does not greatly affect SOC. Larger DRAM capacity allows Kangaroo to use a larger log. Even so, both of these systems are mainly constrained by device-level write rate. By contrast, LS is dependent on DRAM capacity. LS approaches, though does not reach, Kangaroo’s miss ratio at 64 GB of DRAM on the Facebook trace and at 40 GB on the Twitter trace. At this point, Kangaroo is constrained from reducing misses further by device write rate (see Fig. 4.10).

Larger flash capacities will shift the lines right as the DRAM : flash ratio decreases. Assuming write budget and request throughput scale with flash capacity, a 4 TB flash device requires 60 GB DRAM to achieve the same miss ratio as a 2 TB flash device with 30 GB DRAM. This makes the left side of the graph particularly important when comparing flash-cache designs.

Flash-device capacity. As stated in the previous section, the size of the flash device greatly impacts miss ratio and the significance of write-rate and DRAM constraints. As we look forward, flash devices are likely to increase while DRAM capacity is unlikely to grow much and may even shrink [236]. Fig. 4.12 shows the miss ratio for each system as the device capacity changes. Each system can use as much of the device capacity as it desires while staying within 16 GB DRAM and 3 device writes per day.

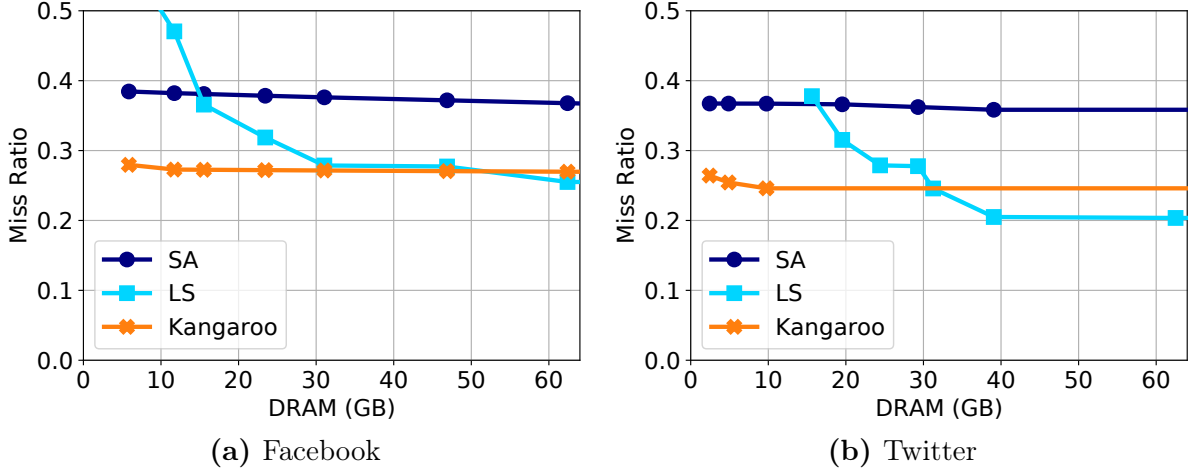


Figure 4.11: Miss ratio vs flash capacity. Pareto curve of cache miss ratio as DRAM capacity varies from 5 to 64 GB. Flash capacity is fixed at 2 TB and device write rate is capped at 62.5 MB/s. The amount of DRAM does not greatly affect SOC or Kangaroo, which are both write-rate-constrained, but has a huge effect on LS by increasing its cache size.

Except at smaller flash capacities, Kangaroo is Pareto-optimal across device capacities. At smaller device capacities (<1.2 TB for the Facebook trace and <1 TB for the Twitter trace), Kangaroo and SOC are increasingly write-rate-limited while LS is *decreasingly* DRAM-limited. However, as flash capacity increases, LS is quickly constrained by DRAM capacity. In contrast, Kangaroo and SOC both take advantage of the additional write budget and flash capacity. Kangaroo is consistently better than SOC due to lower ALWA (allowing larger cache sizes) and RRIParoo.

Object size. The final feature that we study is object sizes. Fig. 4.13 shows how miss ratio changes for each system as we artificially change the object sizes. For each object in the trace, we multiply its size by a scaling factor, but constrain the size to [1 B, 2 KB]. To study the impact of cache design as object sizes change, we keep the working-set size constant by scaling up the sampling rate (Sec. 4.4.6).

The cache designs are affected differently as object size scales. SOC writes 4 KB for every object admitted, independent of size, so its ALWA grows in inverse proportion to object size, and SOC is increasingly constrained by flash writes as objects get smaller. Similarly, LS can track a fixed number of *objects* due to DRAM limits, so its flash-cache size in *bytes* shrinks as objects get smaller. Both SOC and LS suffer significantly more misses with smaller objects.

Kangaroo is also affected as objects get smaller, but not as much as prior designs. KSet’s ALWA increases with smaller objects, but less than SOC. For example, as avg object size goes from 500 B to 50 B, Kangaroo’s ALWA increases by $4\times$, while SOC’s ALWA increases by $10\times$ (Fig. 4.6). Similarly, KLog uses more DRAM with smaller objects, but, unlike LS, Kangaroo can reduce DRAM usage by decreasing KLog’s size without decreasing overall cache size. The tradeoff is that ALWA increases slightly (see below). Kangaroo thus scales

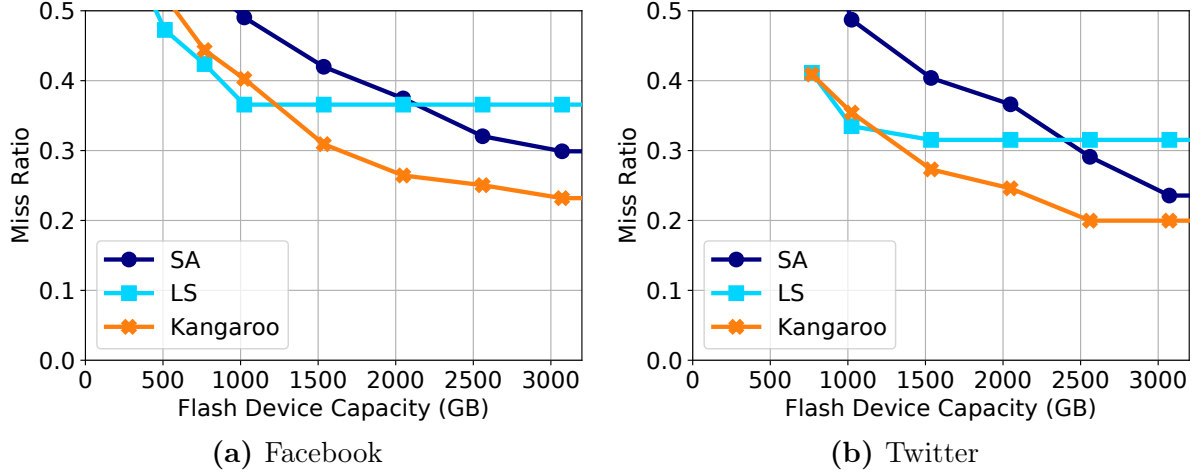


Figure 4.12: Miss ratio vs device size. Pareto curve of cache miss ratio at different device sizes. The DRAM capacity is limited to 16 GB and the device write rate to 3 device writes/day (e.g., 62.5 MB/s for a 2 TB drive).

better than prior flash-cache designs as objects get smaller: on the Twitter trace, Kangaroo reduces misses by 7.1% vs. LS with 500 B avg object size and by 41% vs. LS with 50 B avg object size.

4.5.3 Parameter sensitivity and benefit attribution

We now analyze how much each of Kangaroo’s techniques contributes to Kangaroo’s performance and how each should be parameterized. Fig. 4.14 evaluates Kangaroo’s sensitivity to four main parameters on the Facebook trace: KLog admission probability, KSet eviction policy, KLog size, and KSet admission threshold. All setups use the full 2 TB device capacity and 16 GB of memory. We build up their contribution to miss ratio and application write rate from a basic set-associative cache with FIFO eviction.

Pre-flash admission probability. Fig. 4.14a varies admission probability from 10% to 100%. As admission probability increases, write rate increases because more objects are written to flash. However, the miss ratio does not decrease linearly with admission probability. Rather, it has a smaller effect when the admission percentage is high than when the admission percentage is low. Since Kangaroo’s other techniques significantly reduce ALWA, we use a pre-flash admission probability of 90%.

Number of RRIParoo bits. Fig. 4.14b shows miss ratios for FIFO and RRIParoo with one to four bits. Although changing the eviction policy does slightly change the write rate (because there are fewer misses), we show only miss ratio for readability. RRIParoo with one bit suffers 3.4% fewer misses vs. FIFO, whereas RRIParoo with three bits suffers 8.4% fewer misses. Once RRIParoo uses four bits, the miss ratio increases slightly, a phenomenon also noticed in the original RRIP paper [133]. Since three-bit RRIParoo uses the same amount of DRAM as one-bit RRIParoo (Sec. 4.3.4), we use three bits because it

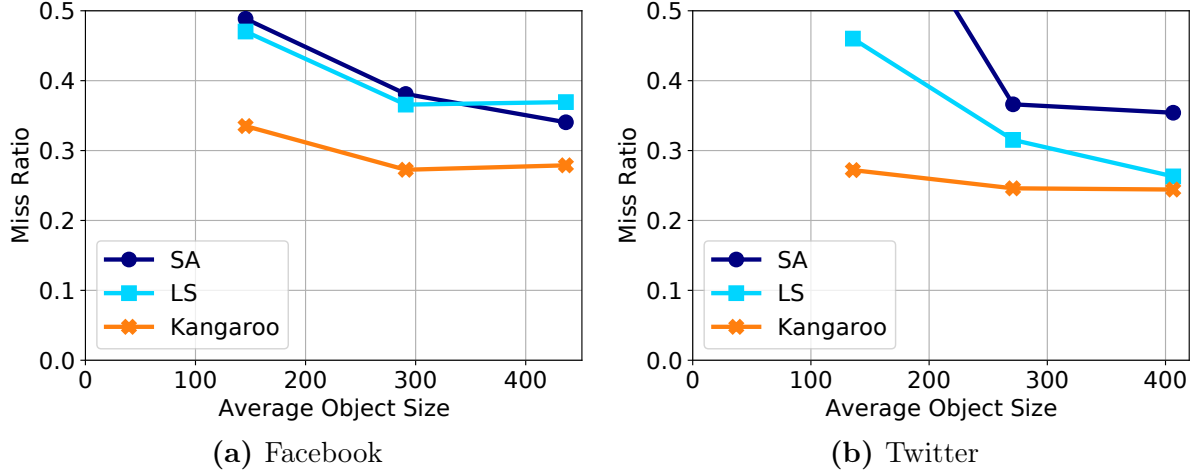


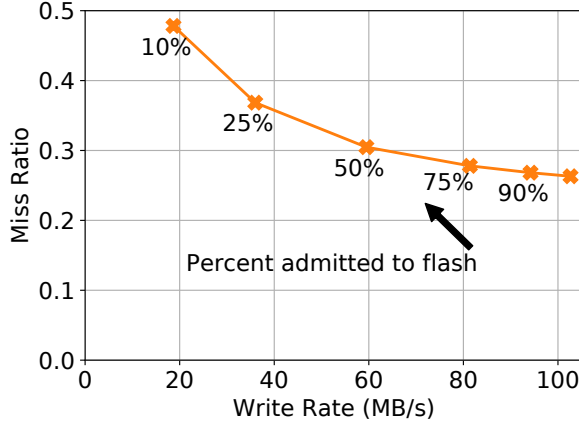
Figure 4.13: Miss ratio vs average object size. Pareto curve of cache miss ratio vs. average object size. Object sizes are limited to $[1B, 2048B]$. The write rate is constrained to 62.5 MB/s for a 2 TB flash drive with 16 GB of DRAM.

achieves the best miss ratio.

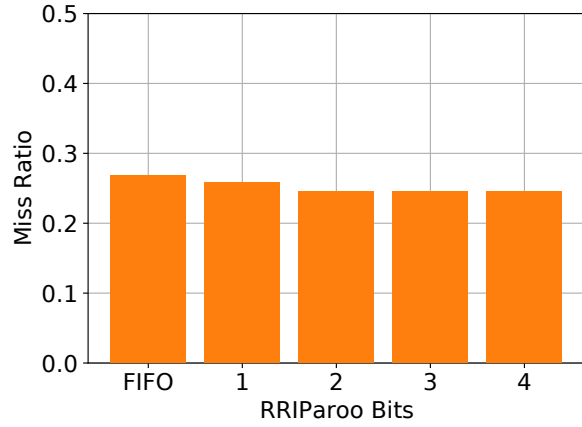
KLog size. Fig. 4.14c shows that, as KLog size increases, the flash write rate decreases significantly, but the miss ratio is unaffected ($<.05\%$ maximum difference). However, a bigger KLog needs more DRAM for its index. Thus, KLog should be as large enough to substantially reduce write amplification, but cannot exceed available DRAM nor prevent using a DRAM cache. Flash writes can be further reduced via admission policies or by over-provisioning flash capacity as needed. We use 5% of flash capacity for KLog.

KSet admission threshold. Fig. 4.14d shows the impact of threshold admission to KSet. Thresholding reduces flash write rate up to 70.4% but increases misses by up to 72.9% at the most extreme. Note that these results assume rejected objects are re-admitted to KLog if they have been hit, since re-admission reduces misses without significantly impacting flash writes. We use a threshold of 2, which reduces flash writes by 32.0% while only increasing misses by 6.9%.

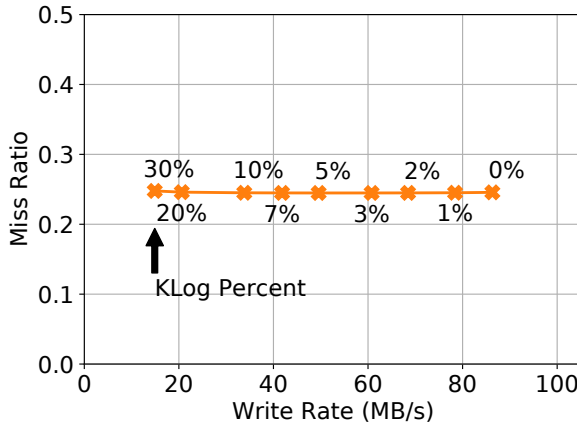
Benefit breakdown. In this configuration, Kangaroo reduces misses by 2% and decreases application write rate by 67% vs. a set-associative cache that admits everything. Most of the miss ratio benefits over SOC come from RRIParoo. Kangaroo also improves miss ratio vs. SOC at a similar write rate by reducing ALWA, which allows it to admit more objects than SOC. Each of Kangaroo’s techniques reduces write rate: pre-flash admission by 8.2%, RRIParoo by 8.3%, KLog by 42.6%, and KSet’s threshold admission by 32.0%. Kangaroo’s techniques have more varied effects on misses: pre-flash admission increases them by 1.9%, RRIParoo decreases them by 8.4%, KLog changes them little ($<0.05\%$ difference), and KSet’s threshold admission increases them by 6.9%. We found similar results on the trace from Twitter.



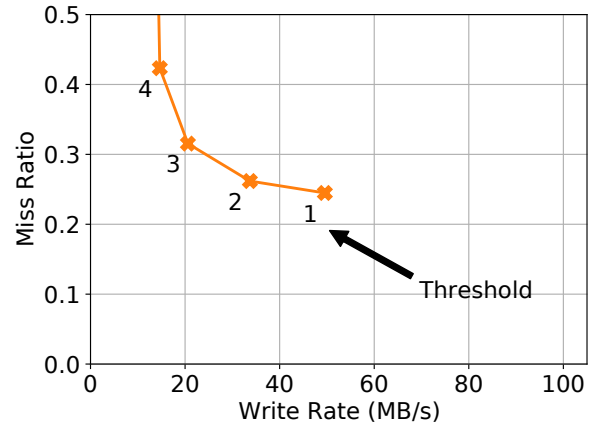
(a) Probabilistic admission.



(b) + RRIParoo

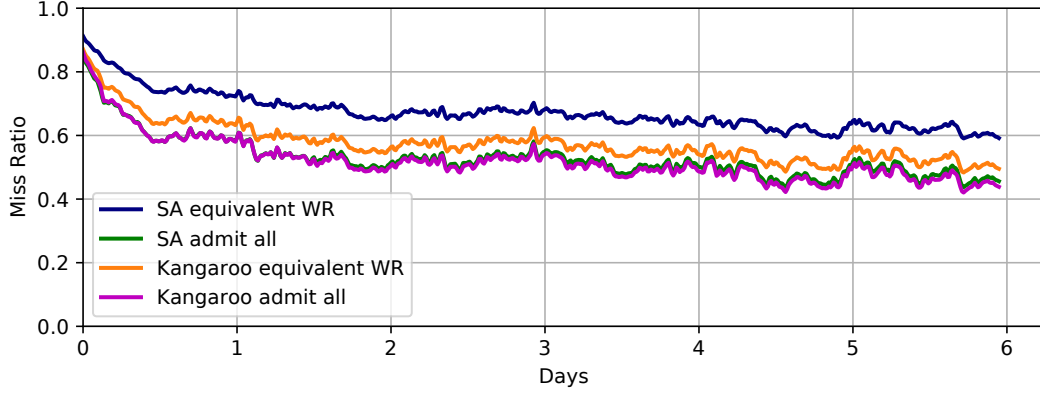


(c) + KLog.

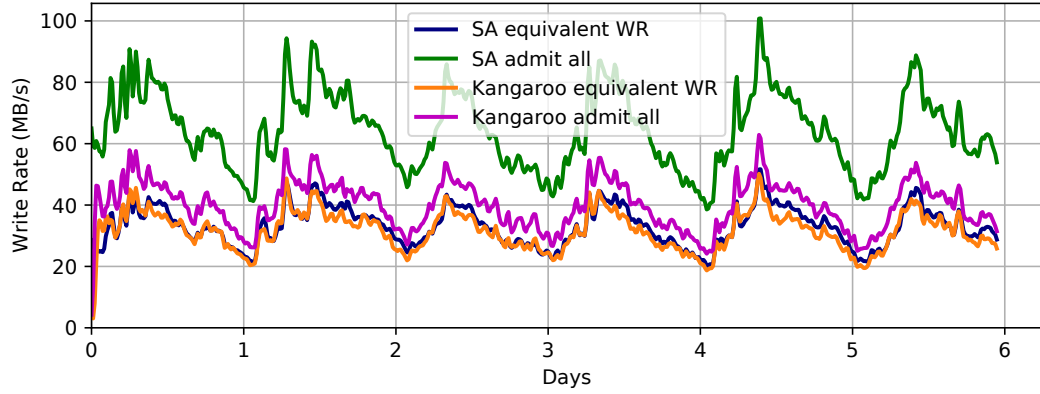


(d) + Threshold.

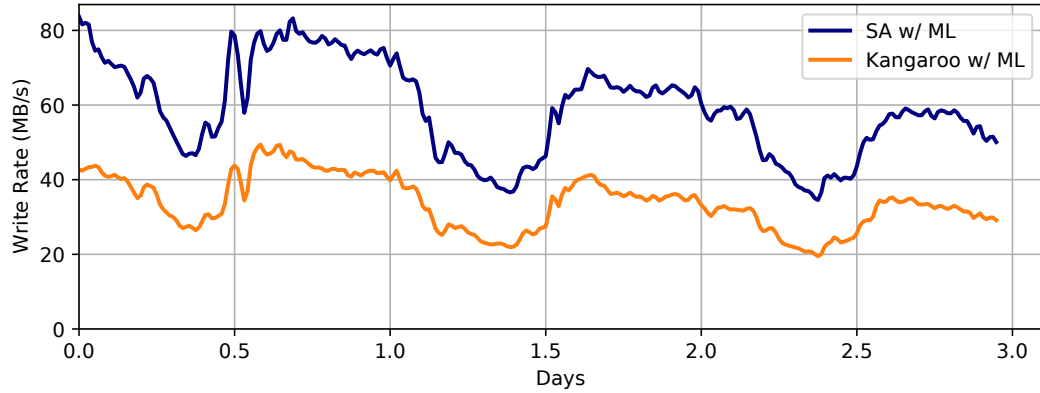
Figure 4.14: Sensitivity study on Kangaroo parameters. Miss ratio vs. application-level write rate based on various design parameters in Kangaroo: (a) KLog admission probability, (b) RRIParoo metadata size, (c) KLog size (% of flash-device capacity), and (d) KSet admission threshold.



(a) Flash miss ratio.



(b) Application flash write rate.



(c) ML admission.

Figure 4.15: Production deployment of Kangaroo. Results from two production test deployments of Kangaroo and SOC, showing (a) flash miss ratio and (b) application flash write rate over time using pre-flash random admission and (c) application flash write rate over time using ML admission. With random admission at equivalent write-rate, Kangaroo reduces misses by 18% over SOC. When Kangaroo and SOC admit all objects, Kangaroo reduces write rate by 38%. With ML admission, Kangaroo reduces the write rate by 42.5%.

4.5.4 Production deployment test

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

To find appropriate production configurations, we chose seven configurations for each system that performed well in simulation: four with probabilistic pre-flash admission and three with a machine-learning (ML) pre-flash admission policy. The first production deployment ran all probabilistic admission configurations and the second ran all ML admission configurations. Since these configurations ran under different request streams, their results are not directly comparable, i.e. the probabilistic configurations (Fig. 4.15a and Fig. 4.15b) cannot be compared to the ML admission configurations (Fig. 4.15c).

Fig. 4.15a and Fig. 4.15b present results over a six-day request stream for configurations with similar write rates (“equivalent WA”) as well as configurations that admit all objects to the flash cache (“admit-all”). Kangaroo reduces misses by 18% vs. SOC in the equivalent-WA configurations, which both have similar write rates at ≈ 33 MB/s. The admit-all configurations achieve the best miss ratio for each system at the cost of additional flash writes. Here, Kangaroo reduces flash misses by 3% vs. SOC while writing 38% less.

We also tested both systems with the ML pre-flash admission policy that Facebook uses in production [60]. Fig. 4.15c presents results over a three-day request stream. The trends are the same: Kangaroo reduces application flash writes by 42.5% while achieving a similar miss ratio to SOC. Kangaroo thus significantly outperforms SOC, independent of pre-flash admission policy.

Chapter 5

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

“To distract predators from nests with young birds, the superb fairywrens may utilize a ‘rodent run’ display where they lower their head, neck and tail, hold out their wings and fluff their feathers as they run, voicing a continuous alarm call.”

Australian Wildlife Wonders. [11]

“How do parents recognize their offspring when the cost of making a recognition error is high? ... We discovered that superb fairy-wren (*Malurus cyaneus*) females call to their eggs, and upon hatching, nestlings produce begging calls with key elements from their mother’s “incubation call.”... We conclude that wrens use a parent-specific password learned embryonically to shape call similarity with their own young and thereby detect foreign cuckoo nestlings.”

Columbelli-Negrel et al. [90].

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 [138] and 33% by 2050 [155]. In the next few decades, many companies — including Amazon [6], Google [13], Meta [34], Microsoft [186] — 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 [34, 122, 173, 186]. Google, AWS, and Microsoft are expected to complete their transition to renewable energy by 2030 [91, 139, 165]. 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 [122].

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 [73, 122, 123, 241], but considerably less attention has been paid to memory and storage, even though they constitute 46% and 40% of server emissions, respectively [173]. It is therefore crucial to both move from carbon-intensive technologies like DRAM to flash, which has 12 \times less embodied carbon per bit [123], *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, or 0.09% of available write bandwidth (Sec. 5.1).

Reducing carbon from caching. Hence, write-intensive flash applications present a major challenge in reducing overall datacenter emissions. This chapter focuses on reducing carbon from flash caching, an increasingly popular use of flash in the datacenter [4, 60, 68, 69, 105, 118, 237]. 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 [58, 61]. Datacenter caches also store many small objects [60, 178], 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, such as the research state-of-the-art Kangaroo [178, 179], have a limited ability to optimize these internal writes, which can amplify the total bytes written by 2 \times to 10 \times [178].

Opportunity: WREN. New flash SSD interfaces, such as ZNS [66] and FDP [176], 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

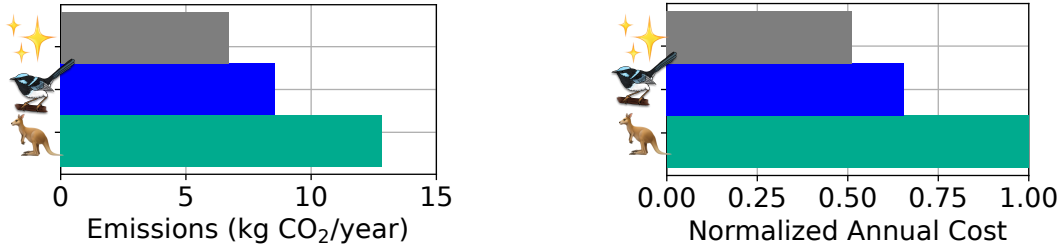


Figure 5.1: Overview of FairyWREN results. Carbon emissions and cost for flash in Kangaroo (🦘), FairyWREN (🐦), and “minimum writes” (✨)—an idealized cache with no extra writes—over a 6-year lifetime for a production Twitter trace and a target 30% miss ratio. Compared to Kangaroo, FairyWREN reduces carbon emissions by 33% and cost by 35%.

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 [60]. 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. 5.1). FairyWREN performs close to an idealized, minimum-write cache on both carbon emissions and cost.

Contributions. This chapter contributes the following:

- *Critical elements of flash interfaces (Sec. 5.2):* 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.
- *Analysis of erase granularity in WREN (Sec. 5.2.4):* We analyze the effect of **Erase**

	Flash caches should minimize ...			
	Unused flash	DRAM	ALWA	DLWA
Key-value stores	✗	✓	✓	✓
Log-structured caches	✓	✗	✓	✓
Set-associative caches	✗	✓	✗	✗
Kangaroo [178]	✓	✓	✓	✗
FairyWREN	✓	✓	✓	✓

Table 5.1: Comparison of FairyWREN vs. prior cache designs. FairyWREN is the only design to minimize all important overheads.

operation’s granularity in WREN, bridging the theoretical and systems understanding of its impact on write amplification.

- *FairyWREN (Sec. 5.3)*: 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).
- *Model of caching’s carbon emissions (Sec. 5.4.2)*: We develop a model to analyze carbon emissions from flash caching — incorporating both write rate and cache capacity to determine overall flash emissions.
- *Analysis of cache write amplification and its impact on emissions (Sec. 5.4.3-Sec. 5.4.7)*: We show that FairyWREN’s write reduction allows flash caches to improve sustainability using denser flash for longer lifetimes, without increasing the cache’s miss ratio.

5.1 Sustainable design constraints in flash caching

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 [105, 178]; and *(iii)* flash write rates — which wear out the device, reducing lifetime. No prior flash-cache design meets these criteria (Table 5.1). 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.

Why not Kangaroo? FairyWREN builds on Kangaroo [178, 179]. Kangaroo’s design allows for a tradeoff between writes and DRAM overhead. The larger KLog is, the more collisions will be found during flush operations, lowering KSet’s ALWA in exchange for higher DRAM overhead. Still, Kangaroo needs only 5-10% of flash for KLog to substantially reduce KSet’s writes. Since KSet comprises more than 90% of the cache capacity, the DRAM needed to index KLog is limited. Due to its low DRAM overhead, Kangaroo achieves large emission reductions over a memory-optimized log-structured cache, Flashield [105], for

workloads with many small objects (Fig. 5.9 in Sec. 5.4.3). This comparison shows that a carbon-efficient cache needs to have a low DRAM overhead.

While Kangaroo greatly reduces writes by limiting ALWA, it still writes too much because *Kangaroo cannot control device-level write amplification*. Kangaroo experiences high DLWA because KSet performs random 4 KB writes, the worst case for DLWA on LBAD devices. Because of its high write budget requirements, Kangaroo cannot reduce emissions by moving to denser flash. For example, Fig. 3.1 shows that, for a 10-year lifetime, QLC tolerates only 0.37 device-writes per day (DWPD) and PLC tolerates only 0.16 DWPD. Kangaroo performs 1.46 DWPD in our evaluation. Our goal is to build a sustainable cache that achieves Kangaroo’s low DRAM usage while also writing far less to flash. We find that flash caches need a different flash interface in order to reduce DLWA without adding DRAM overhead.

5.2 Write-Read-Erase iNterfaces (WREN)

Prior flash caches incur excessive DLWA. 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 [66] and FDP[176], 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.

5.2.1 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 blocks¹ 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. 2.2.4). 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 [124, 125, 129, 158, 252, 259], LBAD devices still hide erase units and GC from applications, preventing co-optimization to minimize overall flash writes.

5.2.2 Challenges of new interface design

While a variety of flash interfaces have been proposed [65, 129, 148, 153, 192, 220, 247, 269], 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 [148, 153] 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.

¹These fixed-size blocks correspond to pages, not flash blocks

Open-Channel SSDs [65] remove all flash-device logic and force applications to handle *all* of flash’s complexities. While the hope was to develop layers of abstraction in software to hide some of this complexity, this software was never widely deployed.

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.

5.2.3 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)*[66] and *Flexible Data Placement (FDP)*[176] are both WREN. Both interfaces are NVMe standards with strong support from industry and provide an abstraction for writing to an EU². 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 [269]. 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.

²This abstraction is called a *zone* in ZNS and a *reclaim unit* in FDP.

Variable	Definition
X	Random variable representing number of invalid page in an EU chosen for GC
b	Number of pages in an EU
p	Probability that a page is invalid
k	Number of writes between each GC operation
t	Total number of EUs
u	Number of EUs for user data (does not include overprovisioning)

Table 5.2: Variables in analytical model of FIFO+.

5.2.4 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.

However, not all caches can perform large, sequential writes. Set-associative flash caches also want low WA, but perform small, random writes that incur high DLWA on LBA 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. This naive approach does not reduce WA— it just moves the GC from the device to the cache (see Sec. 5.4.6).

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. Caches can tolerate removing error correction because they are not tasked with permanently storing the data, rather lost bits just translate to misses. Prior systems use smaller EUs to minimize GC [52, 182] 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 mathematically analyzes the WA of FIFO GC policies [101, 134] 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 need to model the WA of a FIFO GC policy for a set-associative cache and capture the effect of EU size.

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 a FIFO+ GC policy, which selects EUs for garbage collection in FIFO order and skips EUs which contain only valid data. The FIFO+ policy sees a random write workload from the set-associative cache since the inserted key’s hash determines which set to write, a random process assuming a perfect hash function.

We use an approach similar to that of Jeong and Dubois [134] to model the relationship between EU size and DLWA under FIFO+. While several prior papers [101, 134, 244] noted that DLWA can be approximated using W Lambert functions, this prior work focuses on

device overprovisioning rather than on the EU size.

We define X to be the random variable representing the number of invalid pages in an EU that is targeted for garbage collection (as seen in Table 5.2). 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

$$\text{DLWA} = \frac{b}{\mathbb{E}[X|X > 0]}. \quad (5.1)$$

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

$$\mathbb{E}[X] \approx b \cdot p \approx b \left(1 - \left(1 - \frac{1}{ub}\right)^k\right) \quad (5.2)$$

$$\approx b \left(1 - \left(1 - \frac{1}{ub}\right)^{t\mathbb{E}[X]}\right). \quad (5.3)$$

We can rewrite (5.3) using the W Lambert function to get the following approximation for $\mathbb{E}[X]$:

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

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

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

and thus

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

Hence, we now have an approximation that allows us to write DLWA as defined in (5.1) in terms of the device parameters t , u , and b .

Results of model. We validate our model against simulation in Figure 5.2, where we run both our simulation and the model with an overprovisioning of 7%. Our approximation (Fig. 5.2) matches simulation results, with a R^2 value of 0.9996.

Our approximation shows that when EU sizes are small, FIFO is more likely to find EUs that are mostly invalid or completely valid. This leads to a lower WA, as expected in prior systems, since these EUs require fewer rewrites of valid data. However, as EUs grow, the WA quickly stabilizes. Thus, the WA does not change for EUs larger than around 256 KB.

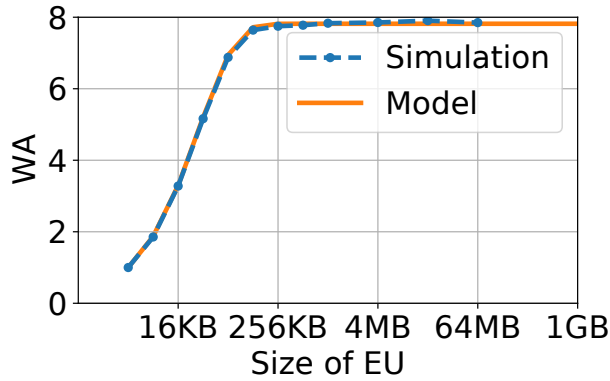


Figure 5.2: DLWA for different EU sizes. The DLWA for a set-associative cache running on WREN with 7% overprovisioning. EUs have to be less than 128 KB to significantly reduce DLWA.

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. 2.2.4). Hence, we conclude that WREN alone does not reduce WA for caches. To reduce WA, we must also re-design the cache.

5.3 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.

5.3.1 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

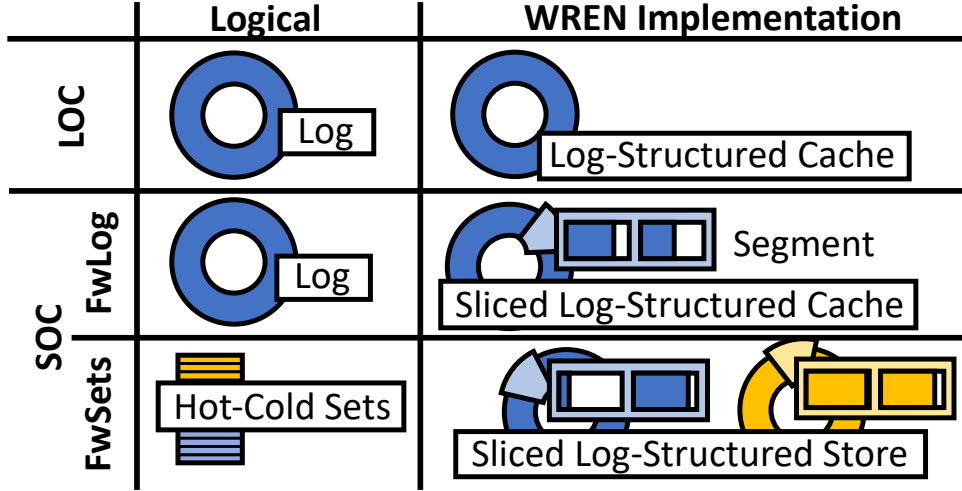


Figure 5.3: Overview of FairyWREN architecture.

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. 5.3. Incoming requests first check the LOC and then check the SOC, since requests do not know the size of the data that they are requesting.

The *large-object cache* (Sec. 5.3.2) stores objects larger than 2KB and uses a simple log-structured design, since it can tolerate higher per-object DRAM overhead.

The *small-object cache* (Sec. 5.3.3) uses a hierarchical design based on Kangaroo [178]. The SOC contains two levels: FwLog and FwSets. At a high-level this is similar to Kangaroo, but FwSets needs to operate differently due to WREN. Since WREN does not support random writes, the sets are kept in a log-structured store. FwSets store 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. 5.3.4).

5.3.2 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 $1\times$.

5.3.3 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. FairyWREN’s SOC is based on Kangaroo [178]. We describe FWSets individually, and then how they work together.

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 [158, 167, 220], 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 4KB, 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 segment³. 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:

³In this scenario, FWSets would be on a log-structured cache.

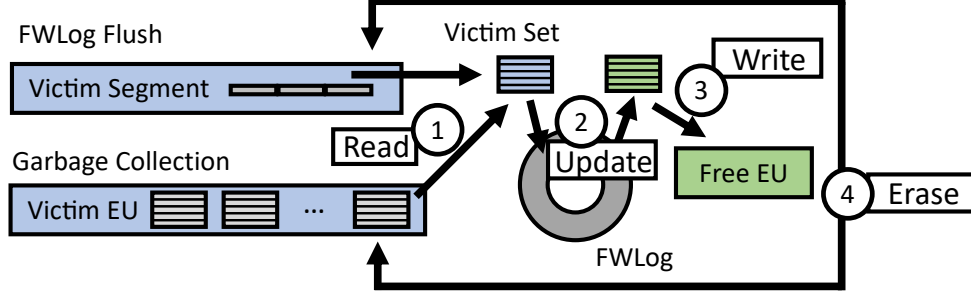


Figure 5.4: Nest packing in FairyWREN’s small-object cache.

Lookup. Lookups, like in Kangaroo, 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. 5.4). 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 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.

5.3.4 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.

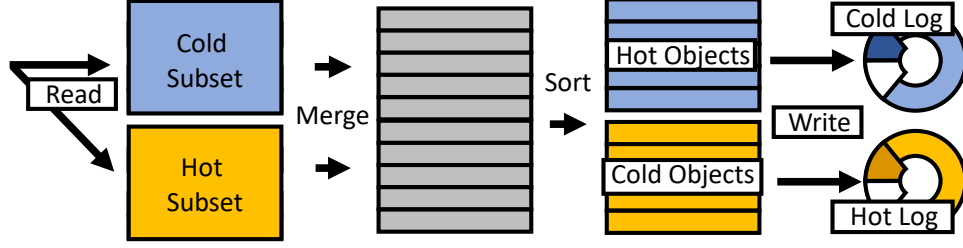


Figure 5.5: FWSets architecture. FWSets is split in two: hot subsets with cold objects and cold subsets with hot objects. Most of the time objects are inserted into the hot subset. However, every n subset updates, both subsets are read, merged, split by object popularity, and then both rewritten.

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. To further reduce these writes, FWSets separates objects by popularity, as determined by a modified RRIP algorithm [133, 178]. 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. 5.5. Any popular objects found in the hot subset are moved into the cold subset. Since popular objects are likely to remain in the cache for a while, they do not need to be rewritten as frequently. Therefore, they should be in the cold subset and not incur extra rewrites. 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. Specifically, FairyWREN writes 4 KB for the 1st, 2nd, 3rd and 4th new object written to a set, since it only has to update the hot subset with the new object. New objects have a high likelihood of being unpopular since many objects are never accessed [60] so starting them in the hot subset aligns well with our variant on the RRIP eviction policy. On the fifth insertion, FairyWREN remerges the hot and cold subsets — rewriting all 8 KB. Thus, FWSets writes only 24 KB instead of 40 KB every five inserts to a set, a 40% write reduction. Since this write reduction applies to all

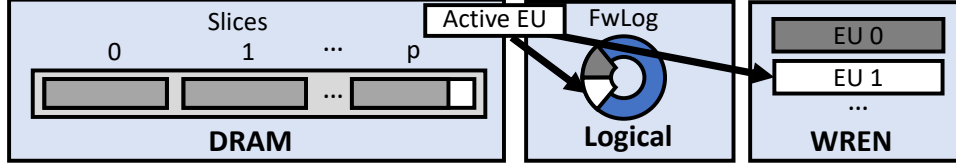


Figure 5.6: FWLog architecture. FWLog uses slicing to minimize memory overhead in FWLog.

sets, we see a 40% write reduction for FWSets overall. This translates to a large reduction in FairyWREN (Sec. 5.4.6).

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 fewer than 10. FairyWREN currently needs 4 active EUs: 1 for LOC, 1 for FWLog, and 2 for FWSets (one for the hot subsets and one for the cold subsets). 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 due to object-popularity mispredictions. To maintain miss ratio, the cache then requires more capacity — meaning FairyWREN would trade a WA problem, which may require additional capacity to maintain the required write rate, for a just a capacity problem. We expect many wrong object-popularity predictions. FairyWREN maintains very few bits of metadata to track each object’s popularity to minimize DRAM, leading to low fidelity predictions. 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 granularity. In fact, even our single layer of hot-cold separations causes a modest increase in miss ratio (Sec. 5.4.6).

Minimizing DRAM in FWLog by slicing.

Like Kangaroo [178, 179], 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 each segment (Fig. 5.6). 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, we found that fragmentation could exceed 20%.

Balls and bins approximation of slicing. To better understand how much fragmentation slicing creates, we model the process of filling a sliced buffer using a balls and bins approximation. Since FWLog hashes each object to a slice, we can model each object as a ball randomly being assigned to a bin representing one slice. To simplify the analysis, we assume each object is the same size.

We want to know how many balls, in expectation, we can throw before the maximum number of balls in any bin is greater than the number of objects that can fit in a slice (x). To answer this question, we consider the stochastic process of sequentially throwing balls into n bins. It is easy to see that the average number of balls in a given bin is $\left(\frac{m}{n}\right)$, suggesting that fragmentation should be limited. However, based on our simulation, we know that fragmentation occurs. Thus, we need to bound the deviation of the maximum number of balls in any bin from this mean.

To derive bounds on fragmentation, we define the stochastic process $\{X_m\}$ to be the maximum number of balls in any bin after m balls have been thrown. We define the random variable M to be

$$M = \min_m \{m \mid X_m > x\}.$$

Our goal is to bound $\mathbb{E}[M]$. Fortunately, the results of Raab and Steger [206] give a high-probability bound on X_m which we can use to bound $\mathbb{E}[M]$.

Specifically, Raab and Steger show that $P\{X > k_\alpha\} = o(1)$ if $\alpha > 1$ and $P\{X > k_\alpha\} = 1 - o(1)$ if $0 > \alpha > 1$, when

$$k_\alpha = \begin{cases} \frac{\log n}{\log \frac{n \log n}{\log m}} \left(1 + \alpha \frac{\log \log \frac{n \log n}{\log m}}{\log \frac{n \log n}{\log m}} \right), & \text{if } \frac{n}{\text{polylog}(n)} \leq m \ll n \log n \\ (d_c - 1 + \alpha) \log n, & \text{if } m = c \cdot n \log n \text{ for some constant } c \\ \frac{m}{n} + \alpha \sqrt{2 \frac{m}{n} \log n}, & \text{if } n \log n \ll m \leq n \text{ polylog}(n) \\ \frac{m}{n} + \sqrt{\frac{2m \log n}{n} \left(1 - \frac{1}{\alpha} \frac{\log \log n}{2 \log n} \right)}, & \text{if } m \gg n(\log n)^3 \end{cases}$$

where $\text{polylog}(x)$ is the class of functions $\bigcup_{i \geq 1} O(\log^i x)$ and d_c denotes a suitable constant depending only on c .

In our setup, we only care about the case where $m \gg n(\log n)^3$ since, for 64 slices, $n(\log n)^3 = 377$ and $m \approx 10,000$ at least.

To bound $\mathbb{E}[M]$, we note that $P\{X_m > x\} = 1 - o(1)$ if $m \geq k_1$. This gives

$$\mathbb{E}[M] = \mathbb{E}[M \mid X_{k_1} > x] \cdot P\{X_{k_1} > x\} + \mathbb{E}[M \mid X_{k_1} \leq x] \cdot P\{X_{k_1} \leq x\} \quad (5.4)$$

$$\geq k_1 \cdot P\{X > k_1\} + 0 \quad (5.5)$$

$$\geq k_1 \cdot (1 - o(1)). \quad (5.6)$$

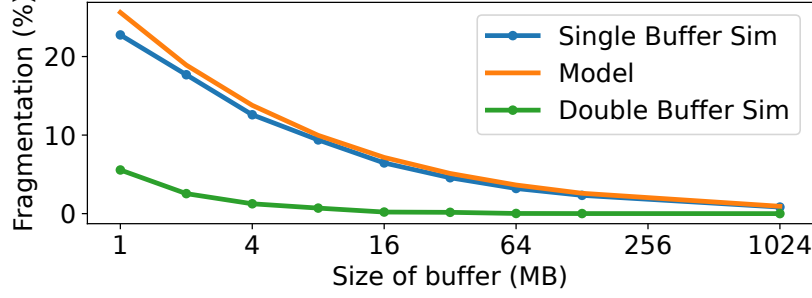


Figure 5.7: FwLog space overhead comparison. Comparison of splice model to single and double buffer simulations over a range of buffer sizes with 64 slices ($R^2 = .97$ between single buffer simulation and model).

Hence, taking limits as n becomes large gives

$$\lim_{n \rightarrow \infty} \mathbb{E}[M] \geq k_1 \quad (5.7)$$

$$= -\frac{\sqrt{n^2(2 \log n - \log \log n)(2 \log n - \log \log n + 4x)}}{2} - \frac{n \log \log n}{2} + nx + n \log n. \quad (5.8)$$

While Eq. 5.8 is an asymptotic lower bound, we find that it closely matches our simulation results, as seen in Fig. 5.7. Our simulation consists of 100 trials of the balls and bins problem at each buffer size. We plot the average of these 100 trials. We find that, unless the buffer is at least 1 GB, more than 1% of buffered capacity is wasted by our simple buffering policy. Therefore, we need to find another way to decrease our fragmentation without increasing our memory usage.

Leveraging double buffering to decrease fragmentation. FWLog reduces fragmentation via double buffering (Fig. 5.8). 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 both simulation and modeling, we find that this optimization limits the capacity loss from fragmentation to $<1\%$, even for small (16 MB) buffers (Fig. 5.7). At 16 MB, the double buffer solution has less fragmentation than 1 GB with a single buffer.

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, the capacity of each bin in our fragmentation model is smaller.

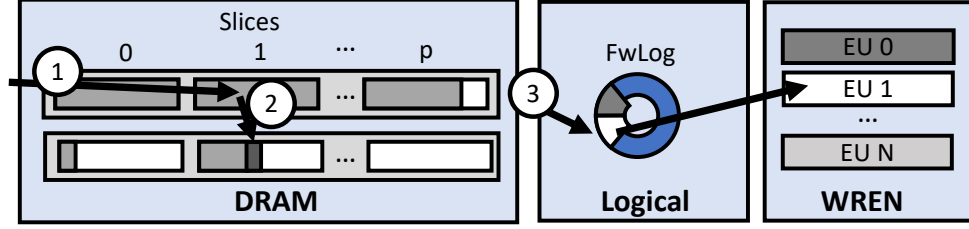


Figure 5.8: FWLog with slicing. FWLog uses overflow buffers to ensure the log segments are full when slicing.

This means that FWSets would incur more internal fragmentation than FWLog if using the same buffer size and number of slices. FWSets therefore uses only 8 slices, which keeps fragmentation to less than 1% just like slicing in FWLog.

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	FairyWREN SOC
Log total	<i>48 bits/obj</i>	<i>48 bits/obj</i>	<i>48 bits/obj</i>
Set index	–	$\approx 3.1\text{ b}$	$\approx 1.4\text{ b}$
Sets (other)	4 b	4 b	4 b
Sets total	<i>4 bits/obj</i>	<i>7.1 bits/obj</i>	<i>5.4 bits/obj</i>
Log metadata	$\approx 0.8\text{ b}$	$\approx 0.8\text{ b}$	$\approx 0.8\text{ b}$
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 5.3: FairyWREN memory overhead. 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 5.3 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

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 5.4: Experimental parameters. FairyWREN and Kangaroo experiment parameters. Both systems use the same amount of flash capacity, but FairyWREN is not required to have 7% over-provisioning for LBAD.

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 \times), netting large savings in carbon emissions and cost.

5.4 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 benefit from QLC.

5.4.1 Experimental setup

Implementation. We implement FairyWREN in C++ as a module in CacheLib [60]. All experiments were run on two 16-core Intel Xeon CPU E5-2698 servers running Ubuntu 18.04 with 64 GB of DRAM, using Linux kernel 5.15. For WREN experiments, we use a Western Digital Ultrastar DC ZNS540 1 TB ZNS SSD, using the LOC and ZNS library written by Western Digital [140]. 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 [178] 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 5.4). Both caches use 400 GB of flash capacity and achieve similar miss ratios as Kangaroo’s production experiments [178]. 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

	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 5.5: Flash density scaling factors. Scaling factors for different flash densities. We optimistically assume that increasing the bits per cell does not affect emissions or cost.

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 Ch. 4.

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 [60] and a 7-day trace from Twitter [262]. 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 [60, 262]. We present results for the last 2 days of the trace.

5.4.2 Carbon emissions and cost model

We want to evaluate carbon emissions and cost across different caching system. Our model allows different cache configuration, flash densities, and device lifetime. Since we want to compare caching systems, our model assumes that a flash device will have the same caching workload for its entire lifetime and that all flash is purchased at the start of the estimated lifetime.

Our model needs to estimate how much flash each cache needs to account for both the cache’s capacity and its writes over the desired lifetime. If the cache capacity cannot accommodate the write rate, we need to overprovision the flash for the write rate. Thus,

$$\text{Flash Capacity} = \max \left(\text{Cache Capacity}, \frac{\text{Write Rate} * \text{Desired Lifetime}}{\text{Write Endurance}} \right)$$

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. LBA devices use 7% overprovisioning, the standard on datacenter drives [22].

We combine this flash capacity requirement with the cache’s DRAM configuration and CPU to estimate both the cost and the carbon emissions, assuming that flash’s write endurance is the server’s main lifetime constraint. While we believe this constraint is reasonable for shorter lifetimes, other failures will become more common at longer lifetimes (particularly above 10 years). We base our write endurance on Micron 7300 NVMe U.2 TLC

SSDs. For other densities, we multiply the TLC write endurance by the write-endurance factors in Table 5.5, based on [23]. 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.

For cost, we account for both the power and acquisition cost of the flash, DRAM, and CPU. For the flash acquisition cost, we interpolate linearly between the Micron SSD’s flash capacities to find a cost for any flash capacity. Cost is normalized to Kangaroo with a 30% miss ratio for the Twitter trace and 20% for Meta.

To determine carbon emissions, we use the ACT model [123] to estimate operational and embodied emissions from CPUs, DDR4 DRAM, and flash.

$$\begin{aligned} \text{Carbon Emissions} &= \text{Operational Emissions} + \text{Embodied Emissions} \\ &= \sum_{\text{CPU, DRAM, Flash}}^{\text{device}} \left(\text{Energy}_{\text{device}} \times \text{Carbon Intensity} + \frac{(\text{Embodied Emissions})_{\text{device}}}{\text{Desired Lifetime}} \right) \end{aligned}$$

For the energy’s carbon intensity, we assume the grid is a 50/50 mix of wind and solar, a common renewable-energy mix [39]. The embodied emissions of both DRAM and flash depend on their capacity and we assume that the CPU uses 70% of its maximum power on average.

5.4.3 Carbon emissions of flash caches

We first examine the carbon emissions of different flash caches for a 6-year deployment. Fig. 5.9 compares FairyWREN to three systems: Minimum Writes, Kangaroo, and a Flashfield-like log-structured cache [105]. Minimum Writes is an unachievable, idealized cache with WA of $1\times$ and no DRAM overhead. Flashfield also assumes a WA of $1\times$, but requires a DRAM:SSD capacity ratio of 1:10, as originally proposed. Since we cannot faithfully replicate Flashfield’s ML eviction policy (and no working implementation is available), we assume that Flashfield achieves FairyWREN’s miss ratios.

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

Although we optimistically assumed that Flashfield incurs no write amplification, Flashfield’s overall carbon emissions are $1.7\times$ higher than Kangaroo’s. These emissions are due to its high DRAM overhead. Despite optimizations in Flashfield designed to save DRAM, high DRAM overhead is unfortunately inherent in the design of a log-structured cache. and thus we need to look beyond log-structured designs.

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 increased writes by overprovisioning flash capacity, increasing the write rate it can maintain in exchange for additional embodied emissions. While Kangaroo is far more sustainable than Flashfield, it leaves room for improvement compared to minimum writes due to its overprovisioning needs.

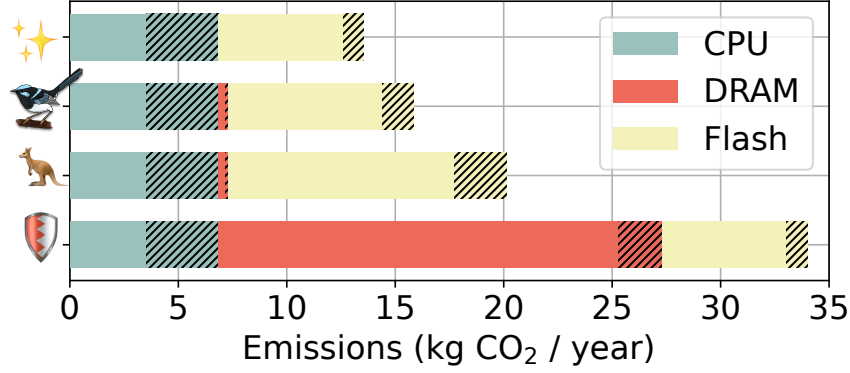


Figure 5.9: Caches’ carbon emissions breakdown. Yearly carbon emissions for 4 caching systems: minimum writes (✨) with a write amplification of 1 with no additional DRAM, FairyWREN (🦉), Kangaroo (🦘), and a Flashield-like log-structured cache (🛡️). Our results include the embodied and operational (hatched) emissions from CPU, DRAM, and flash.

FairyWREN maintains Kangaroo’s low memory overhead while greatly reducing the flash write rate, lowering its overprovisioning requirements. 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.

5.4.4 On-flash experiments

To study how FairyWREN reduces flash writes, we evaluate FairyWREN on real flash drives using the setup in Sec. 5.4.1.

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

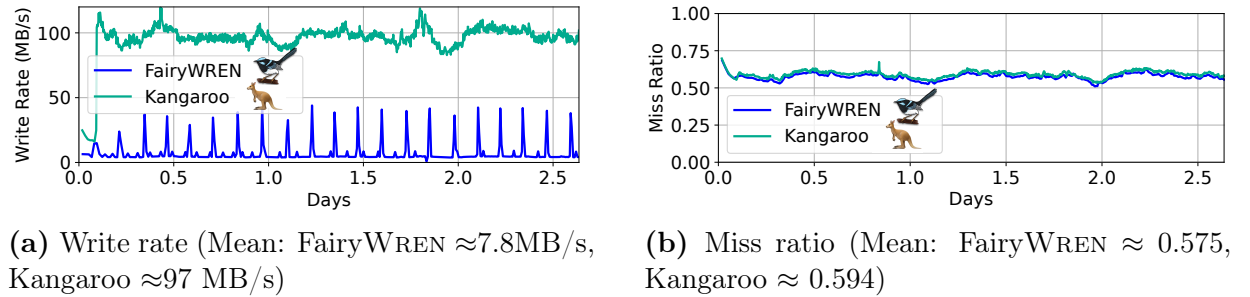


Figure 5.12: Kangaroo vs FairyWREN. The miss ratio and write rate for Kangaroo and FairyWREN.

Fig. 5.12 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 per-

forms 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. 5.12a, FairyWREN reduces writes by $12.5\times$ 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. 5.12b shows that, in fact, FairyWREN and Kangaroo have nearly identical 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 similar results for write amplification: a $12.2\times$ reduction, from $23\times$ in Kangaroo to $1.89\times$ 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 μ s and 1,370 μ s, respectively. But note that, in practice, the overall tail latency is set by the backing store, not the flash cache.

5.4.5 FairyWREN reduces carbon emissions

We now evaluate flash 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 $1\times$.

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

Fig. 5.13 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., T for TLC).

For the Twitter traces (Fig. 5.13a, Fig. 5.13b), 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. 5.4.6). 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.

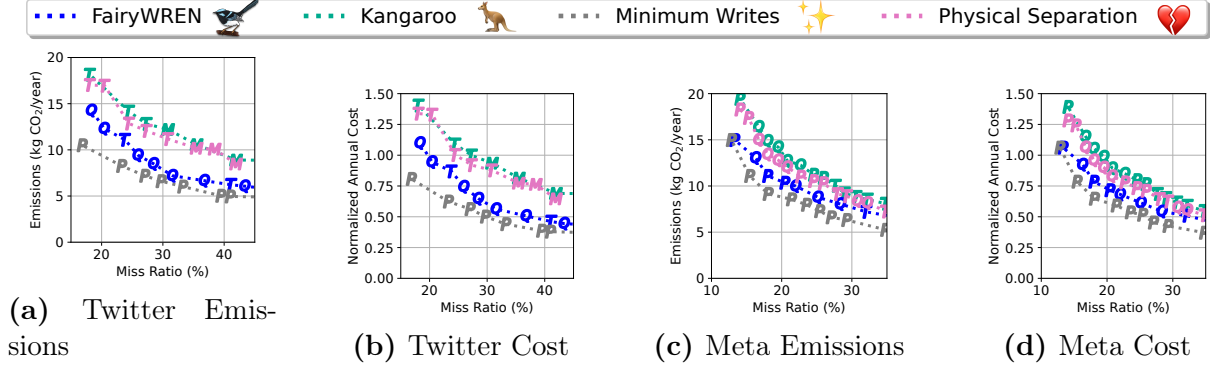


Figure 5.13: Cost and emissions for different miss ratios. The emissions and cost over six years for Kangaroo (🦘), FairyWREN (🐦), Min. Writes (🌟), and Physical Sep. (💔).

The Meta traces (Fig. 5.13c, Fig. 5.13d) 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.*

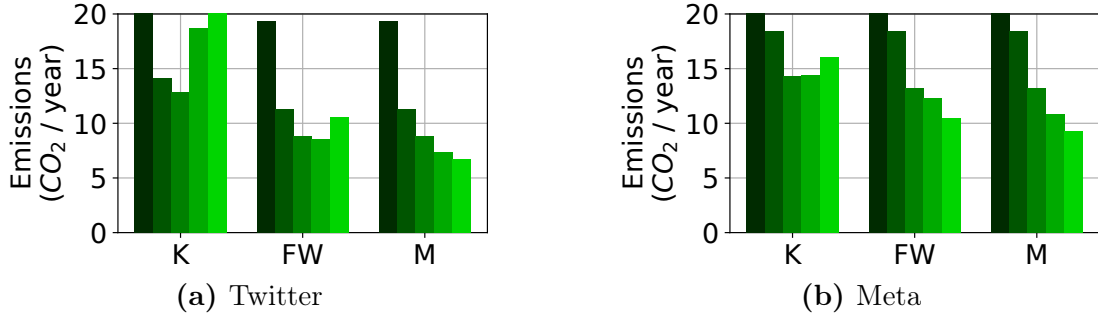


Figure 5.14: Emissions for different flash densities. The carbon emissions to achieve a 30% miss ratio on Twitter trace or 20% miss ratio on Meta trace on different flash densities for a desired lifetime of 6 years. Each bar for each cache represents a different density from SLC (left, darkest) to PLC (right, lightest).

Flash devices are becoming denser over time (Sec. 5.1). Fig. 5.14 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.*

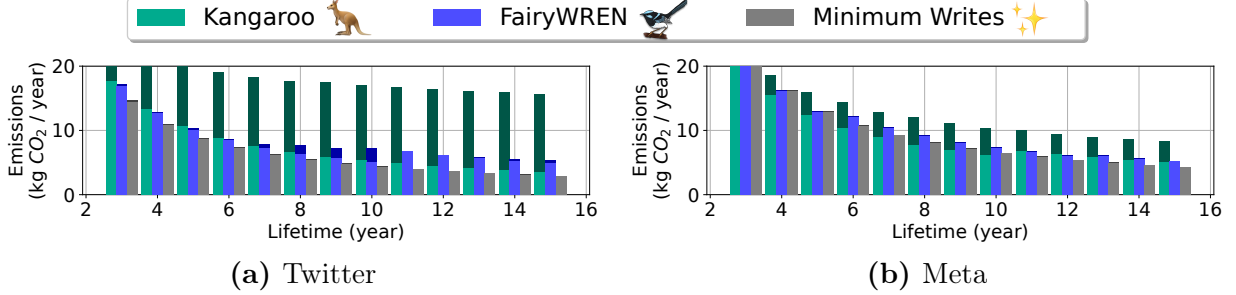


Figure 5.15: Emissions for different lifetimes. The carbon emissions to achieve a 30% miss ratio on Twitter trace or 20% miss ratio on Meta trace with different lifetimes on QLC flash. The darker part of each bar represents emissions due to overprovisioning.

To explore the trend of increasing device lifetime (Sec. 5.1), Fig. 5.15 considers the emissions for caches on QLC devices, showing emissions from overprovisioning in a darker shade.

For a 6-year lifetime, Kangaroo requires $2.2\times$ the emissions of FairyWREN on Twitter and $1.17\times$ on Meta. At 12 years, the gap increases to $2.6\times$ and $1.54\times$. 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. This lower overprovisioning leads to FairyWREN’s much lower emissions, particularly for the Twitter trace.

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. 5.16 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 and thus Kangaroo would have higher emissions if using them. 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 for a 3 year desired lifetime, but TLC for 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. Although a minimum write cache can use PLC for up to 15 years, even a slightly higher write rate quickly overcomes PLC’s limited write endurance.

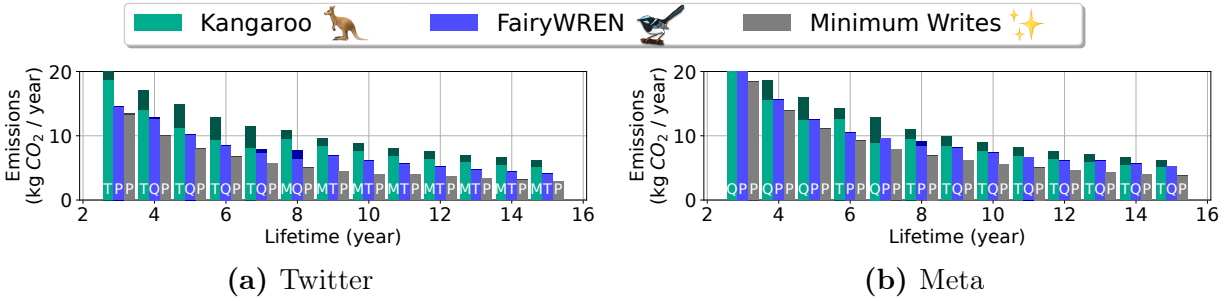


Figure 5.16: Emissions for different lifetimes and densities. The lowest carbon emissions to achieve a 30% miss ratio on Twitter trace or 20% miss ratio on Meta trace while varying both desired lifetimes and flash density. The darker part of each bar represents emissions due to overprovisioning. Letters on each bar represent the flash density

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

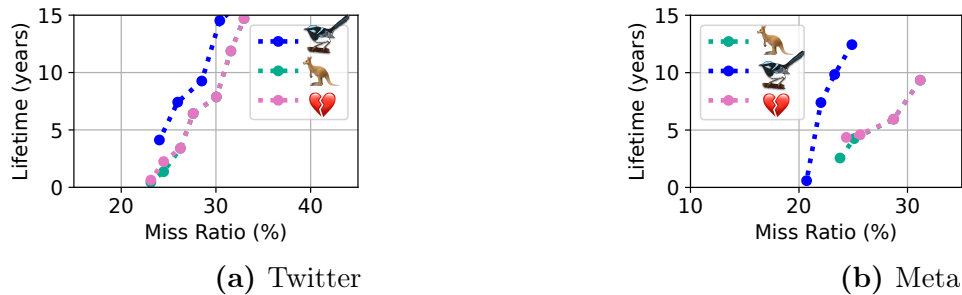





Figure 5.17: Lifetimes vs miss ratios. The lifetimes for a 3.6 TB cache for Kangaroo () , FairyWREN () , and Physical Separation () .

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, extending lifetime can still reduce emissions. Fig. 5.17 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. While Physical Separation reduces writes some over Kangaroo, both ultimately need to massively overprovision to extend lifetime — thus, increasing their miss ratio for any lifetime.

5.4.6 Where are benefits coming from?

We next explore how FairyWREN’s optimizations contribute to its write rate reduction. Fig. 5.18 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.

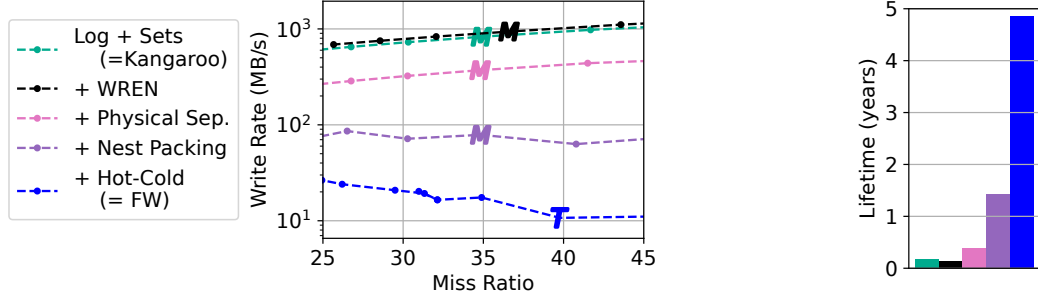


Figure 5.18: FairyWREN benefit attribution. Write rate (log-scale) and lifetime breakdown on the Twitter trace, incrementally adding optimizations to go from Kangaroo to FairyWREN.

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

Three of the lines in Fig. 5.18 are achievable with LBAD devices: Log + Sets, +WREN, and +Physical Sep (though +Physical Sep assumes an augmentation to LBAD such as streams). Log + Sets represents the current Kangaroo implementation on LBAD. +WREN is a naïve 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 ALWA as Kangaroo. +WREN has a simplistic FIFO garbage collection policy, meaning that it can be worse than just running on LBAD which often do try to separate objects belonging to different streams. This means +WREN has higher write rates than Kangaroo on LBAD. In practice, even the best LBAD implementation must perform somewhere between +WREN and +Physical Sep, which would require LBAD to perfectly predict different streams of data. But even in this best case of Physical Sep., the cache 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 $3.7\times$ and hot-cold object separation reduces writes by another $3.4\times$. Either of these optimizations alone would not achieve a close to 5 year lifetime, meaning that the cache still has too many writes to achieve a reasonable deployment lifetime today on QLC. With both optimizations, FairyWREN achieves up to a $33\times$ increase in QLC lifetime over the Kangaroo baseline and a $13\times$ increase over +Physical Sep. We also observe that, even

though hot-cold separation can increase miss ratios, the reduction in write rate and its accompanying reduction in overprovisioning outweighs this miss ratio increase.

5.4.7 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.

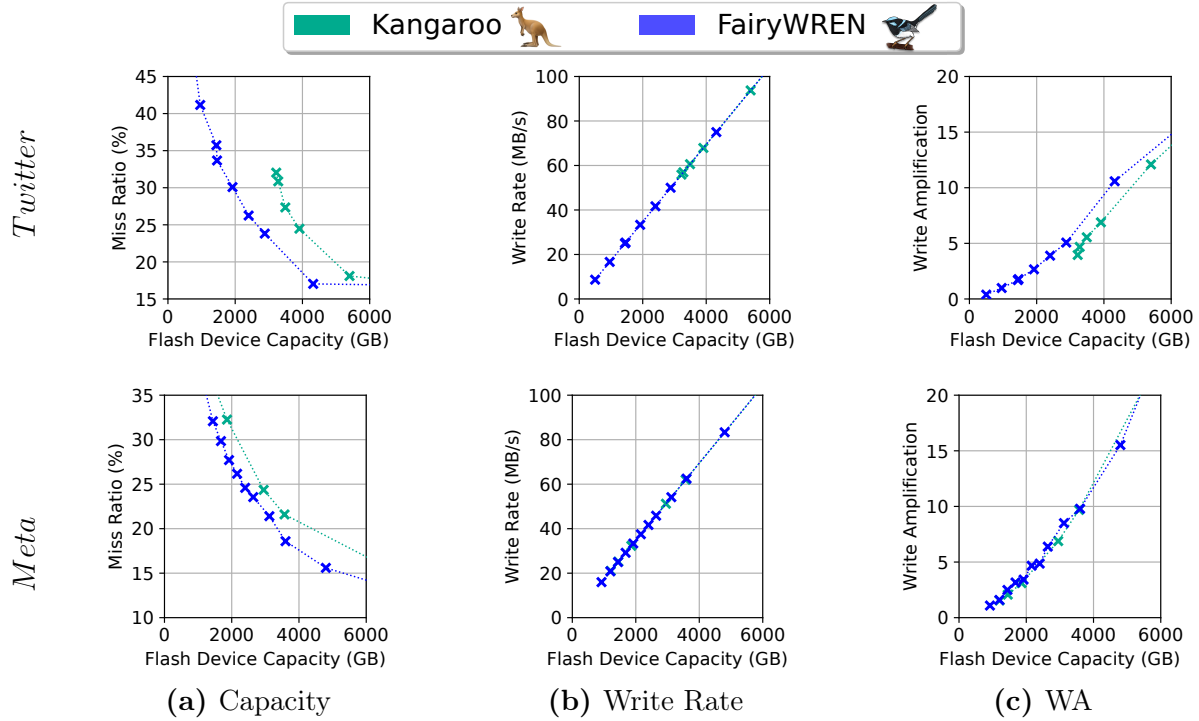


Figure 5.19: Miss ratio vs write rate vs write amplification. Pareto curve of cache miss ratio at different flash device sizes and the corresponding write rate and write amplification of these points. The DRAM capacity is limited to 32 GB, the desired lifetime is 6 years, and the caches use TLC flash.

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

Fig. 5.19 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

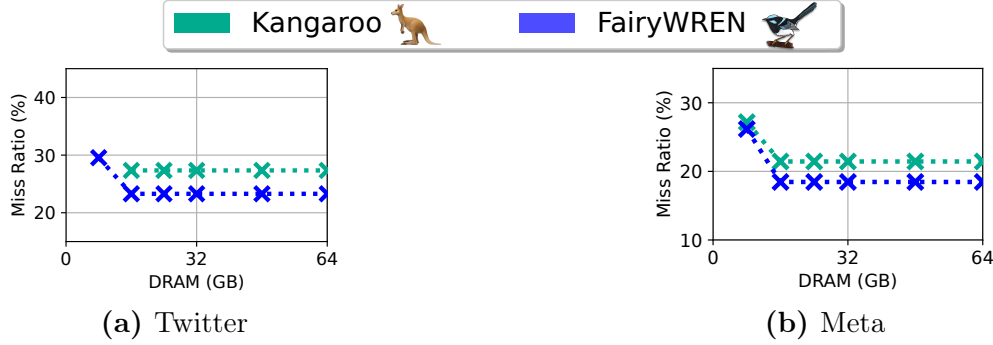


Figure 5.20: DRAM capacity vs miss ratio. Pareto curve of cache miss ratio at different DRAM sizes. The flash capacity is limited to 3.6TB, the desired lifetime is 6 years, and the caches use TLC flash.

more prominent in the Twitter trace than the Meta trace, which is less write-intensive. For the Twitter trace, the limitation of only using TLC prevents Kangaroo from achieving better miss ratios since Kangaroo’s needs much more overprovisioning, increasing the overall flash capacity needed to survive 6 years above 3.6 TB. Thus, Kangaroo’s miss ratio curve shifts to the right.

We also see that flash capacity sets the write budget for the flash device, defining the write rate that the caching system can tolerate for a desired lifetime. As the capacity increases, both FairyWREN and Kangaroo can maintain a higher write rate and both systems use that write rate to further reduce misses. 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, Fig. 5.20. Despite having a large DRAM footprint, FairyWREN maintains a constant miss ratio advantage over Kangaroo from 16 GB to 64 GB of DRAM for both traces. FairyWREN still has a low enough overhead to need less than 16 GB of DRAM for a full 3.6 TB on-flash cache. Therefore, similarly to less DRAM-constrained environments, FairyWREN’s lower write rate translates directly into using more cache capacity and a lower miss ratio.

FairyWREN’s miss ratio only begins to increase when DRAM falls to 8 GB on the both traces. On the Twitter workload, Kangaroo cannot handle the workload with only 8 GB of DRAM. As seen in the Fig. 5.19, with too small of a cache, Kangaroo actually needs more overprovisioning to handle the extra writes from the higher miss ratio. With the DRAM overhead too high to enable a larger cache, Kangaroo cannot be configured to run with 8 GB of memory and only 3.6 TB of flash capacity. Even for the Meta workload with its lower write rate, we see that FairyWREN performs slightly better than Kangaroo at 8 GB of DRAM. FairyWREN’s slightly higher DRAM overhead means its cache capacity is more constrained than Kangaroo’s, but its lower overprovisioning results in a slightly lower miss ratio. Hence, FairyWREN always outperforms Kangaroo even under severe memory

constraints.

5.5 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 [58, 133, 137, 237]. FairyWREN adapts Kangaroo’s RRIP-based eviction policy [133, 178].

Popularity is different than *deathtime*, the time when an object will be deleted [125]. To minimize GC, many storage systems will physically separate objects by their deathtime [77, 84, 125, 156, 210, 264]. 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 [77, 264]. 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 [263]. 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 [78, 105, 158, 167] 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 [220], for example, uses an Open-Channel SSD [65] 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 [85], 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 [179] and CacheLib [60]. This grouping is used to minimize memory overhead. Allocating memory using size-based slab classes is often used to reduce fragmentation [74, 128, 212, 220, 263]. 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.

Chapter 6

Scaling the IO-per-TB wall with Declarative IO

“[T]he desert dingo is intermediate between the wolf and the domestic dogs...
[D]ingoes evolved to prey on small marsupials.”

Issam Ahmed. [8]

LARGE DISTRIBUTED STORAGE SYSTEMS store exabytes over hundreds of thousands of disks [75, 119, 194, 224, 254]. Mechanical disks (HDDs) have remained more cost-effective than flash (SSDs) due to new technologies that increase HDD density, such as heat-assisted magnetic recording (HAMR) [35, 221]. Most vendors have near-term roadmaps for increasing HDD densities to over 40TB per drive, with decade-long targets for 100TB drives leading to a $6\times$ decrease in cost-per-TB [37]. These densities will allow hyperscalers to accommodate data growth rates while also minimizing the power usage and physical footprint of their storage systems.

Unfortunately, drive access speeds are not scaling proportionately to HDD capacities. In particular, the IO supply — i.e., IOPS and bandwidth per device — has remained roughly constant as capacity has increased (Sec. 2.3.3). Put differently, the IO supply per TB of HDD storage has been trending steeply downward. Systems currently match IO supply and demand by deploying flash caches that absorb application IO before it reaches HDDs [60, 93, 178, 181, 266]. However, as HDDs grow denser, we are approaching a new regime where the total IO demand of datacenter workloads on HDDs will exceed supply by the storage system — a phenomenon we refer to as the *IO-per-TB wall*. Beyond this wall, storage systems will be unable to use denser HDDs, forgoing their power, footprint, and emissions savings.

Maintenance tasks’ IO demand. To make the deployment of denser storage devices feasible, IO demand on HDDs needs to be reduced. Most of active disk time stems from various *data maintenance tasks*, such as scrubbing [132, 190, 215], reconstruction [82, 121], capacity balancing, and transcoding [142, 144, 154]. Such maintenance tasks are crucial for providing the durability and availability guarantees users have come to expect from distributed storage systems. These maintenance tasks occur from the block layer to the

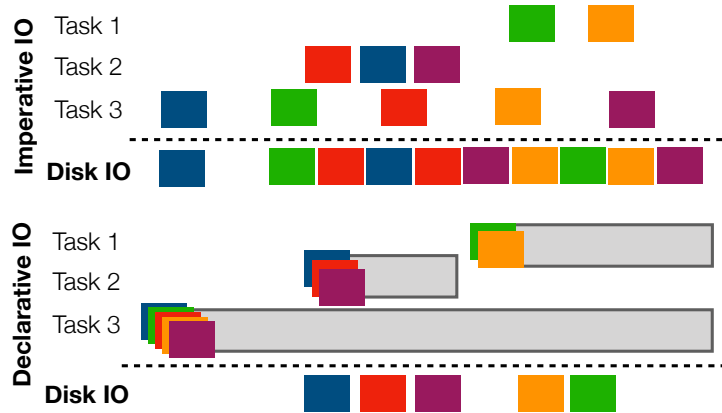


Figure 6.1: Declarative IO exploits task flexibility to reduce IO. Declarative IO allows maintenance tasks to declare their maintenance tasks’ flexibility, creating overlap in the IO requests that are too far apart in time to exploit in imperative IO as seen in a small example with 3 different maintenance tasks requesting 5 different pieces of data.

data services (e.g., table stores and database systems) running atop storage systems. Importantly, each maintenance task accesses large amounts of data with little to no reuse, rendering caches ineffective.

Opportunity: Maintenance tasks’ flexibility creates reuse. While maintenance tasks are individually hard to cache, we observe that there is significant *data overlap* between *different maintenance tasks*. Although there is little reuse within scrubbing, for instance, it overlaps with every other maintenance tasks. This data overlap across tasks occurs too far apart in time to exploit. Fortunately, maintenance tasks are generally flexible in order, time, and even data they access. A single maintenance task (e.g. "scrub a given disk") is generally composed of several lower-level *requests* (e.g. "scrub a given block"). While a maintenance task aims to complete all of its requests within a certain timeframe, the exact ordering, timing, and sometimes even the data of the requests is flexible. We can thus coordinate the overlapping requests from different maintenance tasks to avoid performing redundant IO.

Unfortunately, current *imperative* distributed storage interfaces (e.g., GET/PUT, read/write) do not allow order, time, nor data flexibility — each request is for a *specific* data unit to be accessed *now* (as seen in the top of Fig. 6.1). Co-designing all tasks to explicitly coordinate data reuse using these interfaces is not a practical option for software development at scale, where there are an ever-increasing number of maintenance tasks spanning numerous system and organizational boundaries.

Our Solution: Declarative IO. We introduce *Declarative IO*, a new interface to distributed storage systems to allows tasks to *declare* their upcoming IO needs along with a deadline for those requests . Our system, DINGOS¹, uses these declarations to coordinate

¹Dingoes are Australian canines. DINGOS is Declarative INterface for Global Optimization of Storage. Both rely on packs: dingoes to hunt kangaroos and DINGOS on packs of maintenance tasks to find IO overlap.

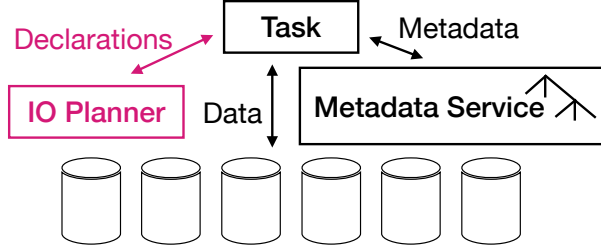


Figure 6.2: DINGOS architecture overview. Declarative IO in DINGOS adds an IO Planner to handle declarations. Tasks still access metadata and data as traditional in distributed storage systems.

IO across maintenance tasks and minimize their aggregate IO demand. DINGOS aims to read each declared block only *once* to satisfy all tasks needing that data (as seen in the bottom of Fig. 6.1). While current systems issue maintenance IO in the order and time that IO requests are issued, with declarative IO, each maintenance task declares all necessary IO, along with an associated deadline. An *IO Planner* (Fig. 6.2) then decides when data should be read from disk and notifies interested maintenance tasks of data availability. The DINGOS IO Planner uses a rate-based scheduling heuristic and hierarchical bit-vectors to find overlap across maintenance tasks for each scheduling quanta. It dispatches these overlaps by erasure blocks to ensure a tiny cache suffices to absorb IO from maintenance tasks.

Summary of results. We implement DINGOS on top of HDFS [224]. We evaluate DINGOS on a 20-node HDD cluster, deriving workloads by profiling 3 hyperscalers’ maintenance tasks. DINGOS decreases IO demand by 26%, showing that Declarative IO is a viable way to decrease disk IO in bulk storage. In simulation, we find that DINGOS reduces IO by up to 40% with more maintenance tasks and that DINGOS requires little cache overhead.

Contributions. This chapter contributes the following:

- *Sources of disk IO:* We identify that essential data maintenance tasks are prevalent, their IO requirements increase with data, and that their IO is uncacheable.
- *Declarative IO interface:* We introduce a new storage interface where tasks declare their IO, time, and data flexibility, allowing an IO Planner to exploit data overlap to reduce IO.
- *DINGOS IO Planner:* Our IO Planner uses a rate-based scheduling heuristic and hierarchical bit-vectors to achieve up to a 40% reduction in maintenance IO — showing that Declarative IO is a promising new interface for distributed storage.

6.1 Maintenance tasks

Although most IO demand to the distributed system comes from application IO, the caching tier is highly effective at absorbing these requests. Hence, the IO demand that

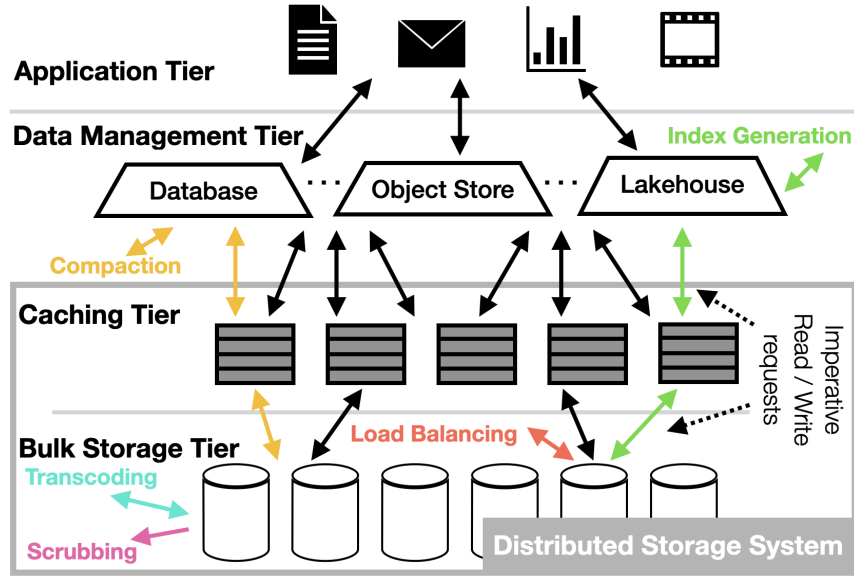


Figure 6.3: Imperative IO architecture. Distributed storage systems receive imperative IO requests from all layers of the datacenter. Requests into distributed storage from maintenance tasks are imperative today and less likely to be cacheable than other data requests.

the HDDs of the bulk storage tier fulfill is disproportionately composed of maintenance IO, which is harder to cache (Fig. 6.3). Based on conversations with multiple datacenter operators, maintenance tasks produce the *majority* of disk IO.

This section discusses why we have maintenance tasks (Sec. 6.1.1) considers the challenges in reducing maintenance IO (Sec. 6.1.2), and finds that the flexibility of maintenance tasks provides an opportunity to reduce their IO usage (Sec. 6.1.3).

6.1.1 Maintenance tasks are essential

Distributed storage systems need to ensure that applications can access their data with high-performance guarantees at low cost and increasingly low emissions and energy-consumption. Thus, storage systems are designed to optimize several objectives such as fault tolerance, reliability, request latency, and low capacity overhead while both handling user requests and failing hardware. Distributed storage systems run maintenance tasks to ensure these properties. For instance, reconstruction ensures that erasure coding’s guarantees are maintained after one part of the stripe is lost, enabling fault tolerance.

However, maintenance tasks are not limited to the distributed storage system. These tasks come from throughout the datacenter architecture (see Table 6.1 for a description of common maintenance tasks). Shingled magnetic recording (SMR) disks internally garbage collect to minimize capacity overhead [40]. Object stores also garbage collect to minimize capacity overhead. Data lakehouses transcode data as it ages to save space while storing newer data in narrower encodings to lower tail latency. Databases create indexes and

Task	Objective				Description
	Perf	Cap	FT	Rel	
Garbage collection	x	x			Removes dead objects, blocks, or entries from the object store, SMR HDD, or database
Rebalancing	x		x		Moves data for fault domains, georeplication, temperature or capacity distribution, etc
Transcoding		x			Changes data encodings to save space as data becomes colder
Reconstruction			x		Recreates data that is lost but recoverable from erasure coding
Scrubbing				x	Checks data blocks, files, or objects to ensure that they are still valid
Integrity checks				x	Checks files, objects, or entries to ensure that their metadata and data still match
Backups					Duplicates data to prevent data loss or inconsistencies
Compaction	x	x			Recombines data from different layers of log-structured merge trees
Index Generation	x				Creates indexes for databases to speed up queries
Short Stroking	x				Moves hot data to outside tracks of HDDs to help performance
Materialized views	x				Pre-computes data set to speed up database queries
Table statistics	x				Finds database distributes to facilitate better query planning

Table 6.1: Descriptions of common maintenance tasks. Each maintenance task fulfills at least one important goal for the storage stack — fault tolerance (FT), reliability (Rel), capacity (Cap), performance (Perf).

materialized views for higher performance. While these maintenance tasks have similar objectives, they are not typically distinguished from application IO by the storage system — often getting the same priority as non-maintenance application IO.

6.1.2 Challenges in reducing maintenance IO

Maintenance tasks are not only essential, they are integral to reducing disk IO since they both are prevalent throughout all sources of distributed system IO. Unfortunately, it is not straightforward to reduce maintenance IO without giving up on the important properties that they provide. We identify three crucial challenges to reducing maintenance IO: (1) no single maintenance task dominates the overall maintenance IO demand, (2) maintenance tasks’ IO requirements scale with data volume, and (3) maintenance tasks are not cache-friendly workloads. Taken together, these challenges limit the kinds of approaches that can be used to dramatically reduce maintenance IO.

No single maintenance task dominates. Based on conversations with multiple companies, no single maintenance accounts for the majority of maintenance IO — even when only considering subsets of maintenance IO from bulk storage and data management layers. Hence, there is no clear single target for optimization. Furthermore, maintenance IO is spread across different systems and different tiers of the datacenter architecture. This means that there is no single system or logical group of systems that is a clear optimization target. Instead, we must develop solutions that span many diverse maintenance tasks generated by different systems and managed by different development teams.

Maintenance tasks grow with capacity. For several maintenance tasks, it is clear that the IO demand scales with the amount of data. Scrubbing, for example, periodically reads every piece of data, and thus grows proportionally with data volume. Reconstruction, which occurs when disks fail, also demands more IO as disks grow larger.

Maintenance tasks are hard to cache. As extensively shown in this dissertation (Ch. 2, Ch. 4, Ch. 5), caching provides an effective way to reduce application IO demand generated from a diverse array of systems [60, 93]. One might hope that larger or more optimized caches be used to reduce maintenance IO. Unfortunately, maintenance IO is hard to cache. A single maintenance task generally scans through a large range of data with almost no reuse between its requests. While data reuse does occur *across* maintenance tasks that scan similar data ranges, these accesses tend to occur much farther apart in time than the requests of application IO, meaning that caching for them is ineffective. Unfortunately, this can result in maintenance IO thrashing the cache and therefore known maintenance tasks, such as in the distributed storage system, avoid sending their IO many of the caching layers.

The poor cacheability of maintenance IO is one of the central reasons that maintenance tasks have come to dominate IO demand on HDDs. Although far more application IO is generated throughout the datacenter than maintenance IO, application IO is greatly reduced by the caching tier. Maintenance IO then has an outsize impact on the bulk storage tier because it is relatively harder to cache. Therefore, we need a different solution

to reduce IO from maintenance tasks while ensuring these tasks still fulfill their objectives.

6.1.3 Opportunity: Maintenance tasks are flexible

Despite the above challenges, we can still reduce maintenance IO by exploiting data reuse. In imperative systems, reuse typically occurs too far apart in time to be captured by traditional caches. However, maintenance tasks are flexible. We can leverage this flexibility to shift overlapping requests from different maintenance tasks closer together in time, creating a highly cacheable series of requests.

Maintenance tasks often have *order-* and *time-flexibility*: a task does not need data in an exact order or at a specific time, e.g. a task’s IO requests do not have precedence constraints or per-request timing constraints. Instead, each task typically has one timing constraint, a *deadline* by which all its requests must be completed. For instance, scrubbing must read and validate every block of data, but it does not care if data x is scrubbed before or after data y (order-flexibility). A scrubbing task also does not specify when data x must be scrubbed, as long as all data is scrubbed within a specified period, typically about one month (time-flexibility).

In addition to flexibility around when requests are executed, some maintenance tasks are also flexible about *which* requests are executed – a phenomenon that we refer to as *data-flexibility*. For instance, load balancing displays data-flexibility. There are typically many hot files or blocks that are good candidates to be moved off a heavily-loaded disk. The load balancing task can make progress by moving some subset of the hot data without caring exactly which data is moved. Carefully selecting which data is requested by each maintenance task can also improve reuse and reduce IO.

Imperative IO impedes flexibility. Unfortunately, there is no way to express this flexibility in today’s imperative storage interface. The imperative interface only allows a task to specify that it needs a *specific* piece of data *now*, imposing order, time, and data constraints. As a result, the only way that reuse has been exploited across maintenance tasks has been through the explicit coupling of task implementations to manually align accesses to overlapping data ranges.

Rewriting maintenance tasks into a single task is not realistic. Maintenance tasks are often managed by different teams, organizations, and even different companies, making coordination of these tasks logistically difficult if not impossible. In addition to representing a large and complex development effort, this approach violates modular design — an essential way to design large systems with many moving parts such as distributed storage systems and the applications and data management services built on top of them. It also does not help with fault tolerance. Combining tasks creates dependencies between systems that are otherwise designed to fail independently — a losing proposition in complicated distributed systems where correctness bugs are easy to introduce. For example, in order to provide an end-to-end correctness guarantee, some object stores perform data scrubbing that is redundant to the scrubbing done by the storage system. Combining the implementation of scrubbing between these systems could violate these guarantees.

Distributed storage needs a new interface. To reduce maintenance tasks’ IO, we need to expose their flexibility while still providing modularity. Thus, we need a new, more expressive interface for distributed storage systems. In the next section, we describe our solution — Declarative IO.

6.2 Declarative IO

The imperative IO interface does not allow maintenance tasks to express their flexibility preventing IO reduction. Therefore, we introduce *Declarative IO*, a new storage interface that the system can leverage to improve data reuse. In this section, we look at Declarative IO from the perspective of someone trying to implement a maintenance task. We first give an overview of Declarative IO (Sec. 6.2.1), then define the interface more precisely (Sec. 6.2.2), discuss how maintenance tasks can adopt a declarative paradigm (Sec. 6.2.3), and explore how the interface modifies the correctness guarantees that maintenance tasks using Declarative IO can expect (Sec. 6.2.4).

6.2.1 Interface Overview

The Declarative IO interface centers on a function called **declare**. At a high level, **declare** allows maintenance applications to send a *declaration* to the storage system that describes a set of flexible read requests. The caller of **declare** passes a description of the data to be read, a deadline by which the data is needed, and a callback that will notify the caller that some of their data is ready to be read. When the callback is triggered, the maintenance task should read the corresponding data through the standard imperative IO interface.

Declarative IO has two main advantages. First, **declare** allows tasks to express time-, order-, and data-flexibility to the storage system. To express order-flexibility, **declare** allows tasks to specify multiple *sets of IO* in one call. The interface allows these IO sets to be completed in any order. The data requested in each set is then read together. To express time-flexibility, tasks declare a deadline for all sets of IO in the declarations. This states that each IO set in the declaration just needs to be completed by the deadline, rather than at a precise time (see Sec. 6.2.4 for more on timing guarantees). To express data-flexibility, tasks specify the number of sets that should be completed before the deadline. This allows a task to declare many sets, of which multiple will fulfill the tasks’ objective.

Second, by passing a callback to **declare**, tasks allow the distributed storage system to select, shift, and reorder declared requests in order to vastly improve data reuse. By exploiting time- and order-flexibility, the storage system turns data reuse that used to span weeks into a series of overlapping read requests that occur within minutes. Furthermore, by exploiting data-flexibility, the storage system can create *additional* reuse that would not exist given an imperative IO interface. Crucially, Declarative IO allows the distributed storage system to optimize across declarations from potentially disparate tasks managed by different organizations. Any tasks that use the same distributed storage system can have their IO reduced by using Declarative IO.


```

1  class BlockSet { set<Block> blocks };
2  declare(list<BlockSet> block_sets,
3         size_t sets_needed, time_t deadline,
4         void callback(list<BlockSet> block_sets, bool overloaded));

```

Figure 6.4: declare call. Maintenance tasks use the `declare` call to specify sets of data that the task needs before a deadline, communicating their flexibility explicitly.

6.2.2 Interface Details

We now describe Declarative IO’s interface in more detail. The main addition is the `declare` call, formally specified in Fig. 6.4.

Specifying data. Many tasks require data that is grouped in a specific way. For instance, transcoding requests data one stripe at a time, where a stripe consists of several blocks². Declarative IO supports data grouping via `BlockSet`, a list of distributed storage blocks. `declare` takes the argument `block_sets`, a list of `BlockSets` representing different task requests. Each `Block` in a `BlockSet` corresponds to one unit of data used by the distributed storage system (in practice, a `Block` is often tens of MBs). A `Block` specifies a logical address in the distributed storage system, not a disk block or an LBA on a specific device. The `sets_needed` argument exposes the data-flexibility of a task by specifying that any subset of `sets_needed` requests from `block_sets` will fulfill the task.

Deadline. The `deadline` argument indicates the time by which the entire task should be completed. In practice, different maintenance tasks exhibit time-flexibility on vastly different time scales. For example, a task such as reconstruction may require a short deadline (e.g. hours) whereas a task such as scrubbing might be done at the scale of weeks.

Callback. The `callback` argument is invoked when the storage system determines that it is a good time for a task to read one or more of its declared `BlockSets`. The callback function takes a list of `BlockSets` specifying which data should be read. The storage system may use the callback multiple times before a declared task is completed.

Importantly, the callback is not passed actual data, but rather a collection of identifiers describing which data to read. The task itself is then responsible for reading this data using standard imperative read requests. Said another way, the callback serves as a strong hint from the storage system to a task that reading the specified data *now* will result in IO savings. As discussed in Sec. 6.2.4, the choice to rely on imperative read calls simplifies the fallback mechanism in case a task is not completed by its deadline. At any time, a task can read any outstanding data in the declaration, potentially lowering IO savings while preserving correctness. If Declarative IO cannot complete a declaration by its deadline, it may invoke the callback with `overloaded=true`, signaling the task to fall back to the imperative interface as needed.

²Note that these are blocks in the distributed storage system, not local file system blocks. Importantly, they are often O(MB) or larger.

```

5  class Segment { string file_path,
6      off_t offset, size_t length };
7  class SegmentSet { set<Segment> segments };
8  def declareFiles(
9      list<SegmentSet> segment_sets,
10     size_t segment_sets_needed, time_t deadline,
11     void callback(
12         list<SegmentSet> segment_sets_selected, bool overloaded));

```

Figure 6.5: Supporting files in Declarative IO. Declarative IO extends the `declare` call to support files using `declareFiles`.

Extending `declare` to files. Depending on where in the datacenter a maintenance task originates, it may not express its data needs in terms of blocks. To address these cases, we extend Declarative IO to support declarations in terms of files (Fig. 6.5) in addition to blocks. `declareFiles` allows a task to declare requests as sets of file segments. Using the storage system’s metadata service, the library translates `declareFiles` into `declare` block declarations. If the file segments do not align to block boundaries, all blocks containing part of the file segment are requested as part of the corresponding segment set. This pattern can be extended to handle objects in object-based distributed storage systems or other higher-level data structures.

6.2.3 Converting maintenance tasks to Declarative IO

Declarative IO requires changing maintenance tasks to use the new interface. We highlight three key considerations for maximizing the impact of Declarative IO when converting these tasks: (1) fully express time- and order-flexibility, (2) hunt for data-flexibility, and (3) simple declarations work well. We discuss these principles below by describing our experience converting several maintenance tasks to Declarative IO.

Fully express time- and order-flexibility. Scrubbing is a particularly good target for Declarative IO, because both it reads a lot of data and its conversion is relatively simple. Scrubbing periodically checks that each block in the system is still valid over a long time horizon (e.g., monthly). Imperative scrubbing iterates over all blocks at a fixed rate, imposing both ordering and timing constraints. A naïve declarative scrubbing implementation might continue to generate scrubbing requests at a fixed rate, but allow each scrubbing request to complete at any time before the next request. Although this naïve approach introduces some time-flexibility, it does not maximize either time- or order-flexibility.

A better approach would involve a single declaration to read all blocks with a one-month deadline (Fig. 6.6). This gives the storage system much more freedom to scrub blocks at any time and in any order, creating massive potential overlap with other tasks. While this is a simple example, it illustrates how a small change to the declaration pattern can drastically change the flexibility afforded to the storage system.

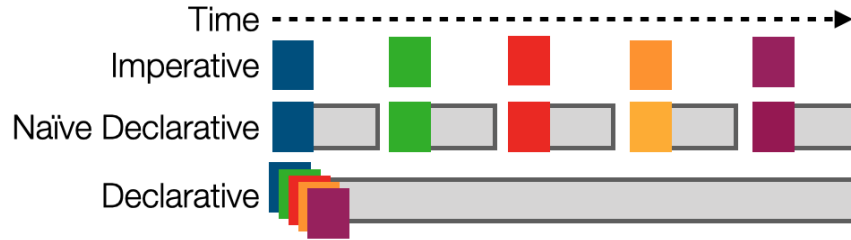


Figure 6.6: Declarative scrubbing. Comparison of how scrubbing requests blocks over time. Declarative IO allows scrubbing to express both order and time flexibility by combining the scrubbing requests into one declaration.

Hunt for data-flexibility. Now consider the capacity balancing task, which moves data between disks to ensure that all disks have about the same amount of data. We can introduce time- and order- flexibility to the capacity balancing implementation similarly to how we did in scrubbing: simply replace imperative reads with one declaration that includes reads for each chosen block (Fig. 6.7). However, this solution ignores that capacity balancing has a lot of data-flexibility.

Capacity balancing fundamentally needs to move some *amount* of data between disks, but not any specific piece of data. This data-flexibility is not immediately apparent from the imperative implementation of the task and is a new consideration in Declarative IO. A more flexible implementation of capacity balancing is to declare all blocks that *could* be moved (potentially all blocks on the disk) and how many *need* to be moved. This allows the storage system to create data reuse between tasks that may not have otherwise accessed the same data.

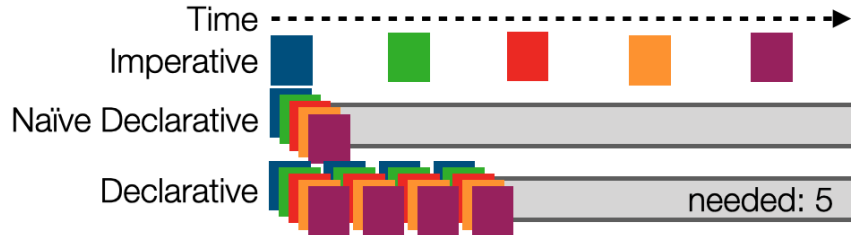


Figure 6.7: Declarative capacity balancing. Comparison of how capacity balancing requests blocks over time. Leveraging the balancing task’s data-flexibility maximizes the chance for overlap with other maintenance tasks.

Simple declarations work well. For more complicated maintenance tasks, the challenge becomes how to group work. For example, compaction in log-structured merge (LSM) trees [20, 24] has a more complex request structure (Fig. 6.8). Rather than operating on blocks, compaction operates on sets of SSTables (e.g., files), generally one SSTable in level n and all SSTables in level $n + 1$ with overlapping key ranges. Hence, the block set for a compaction task should be expressed as all the segments comprising the several files with overlapping key ranges in both the target compaction level and the one below.

It is tempting to make declarations that allow compacting all possible combinations of files (one from level n , one or more from level $n + 1$) to maximize data-flexibility. However, this design will cause problems. These declarations will overlap, meaning that one file could be targeted for compaction multiple times within a single compaction task. One solution to this problem would be to develop more complex data-flexibility semantics for Declarative IO. However, we err on the side of simplicity, and instead suggest declaring non-overlapping requests. While this approach sacrifices some potential IO savings, it is still sufficient to dramatically increase the data-flexibility of compaction. Now that we have defined the sets and how to resolve conflicts, compaction declarations are similar to rebalancing except that instead of each block set being a block, its several files worth of blocks.

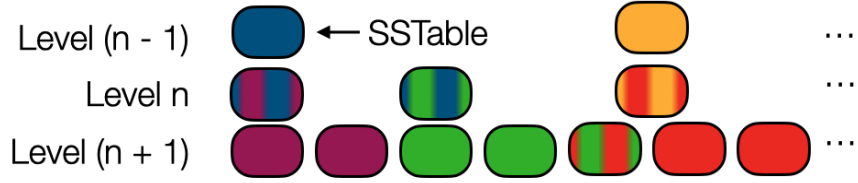


Figure 6.8: Declarative LSM compaction. Units of work in LSM compaction. When declaring compaction, we strive to declare clean cuts of work such as the purple files and the red files instead of overlapping work such as the purple files and the green files.

6.2.4 Consistency with Declarative IO

Moving a task from imperative IO to Declarative IO may change what data the task reads. This clearly applies to tasks with data-flexibility, but even a task that reads a fixed set of blocks may not see the same data. The imperative and declarative views of data may differ because they look at the system at different times (see Fig. 6.10).

Correctly rewriting maintenance tasks using Declarative IO requires understanding its consistency model and what guarantees the interface provides. The following section describes our assumptions about distributed storage system behavior, how Declarative IO provides consistency based on these assumptions, and what happens in the event that a declarative storage system cannot meet its deadlines.

Storage system assumptions. We target append-only distributed storage systems. The append-only storage model is common in hyperscalars [75, 164, 194, 224] because it greatly simplifies the consistency model. Append-only file systems, as seen in Fig. 6.9, provide a guarantee of immutability on the block-level. Here, existing files only accept append operations to add data, and any fully completed, or *sealed*, blocks are immutable. The only way to change a sealed block is through deletion. Declarative IO only accepts sealed blocks.

Given the immutability guarantees of append-only distributed storage systems, we only need to reason about the correctness properties of deletions, creations, appends, and meta-data operations. While Declarative IO could be expanded for a general distributed storage

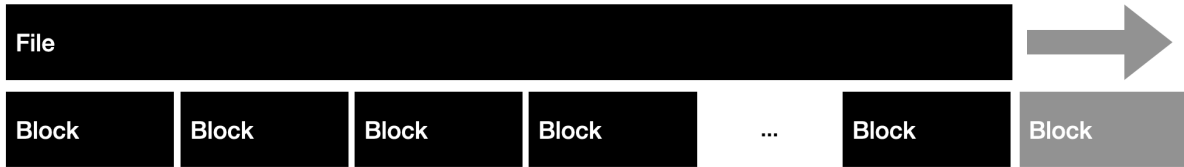


Figure 6.9: Block-file mappings. Files map to sealed blocks in append-only distributed file systems

system, several additional consistency challenges would have to be considered, such as how to handle blocks that change during the lifetime of a declaration.

Repurposing existing imperative logic for correct file deletions. A request from a maintenance task might reference blocks that will be deleted between its declaration and deadline. For instance, *Block 4* in Fig. 6.10 will be deleted before the tasks’ deadline. Depending on the order Declarative IO returns blocks, the task may or may not get *Block 4*’s data. Thus, Declarative IO only guarantees that a task will read valid data from BlockSets where its component blocks are not deleted during the entire lifetime of the declaration. Declarative IO can return callbacks to BlockSets that include a block deleted before the deadline.

Since Declarative IO only notifies maintenance tasks and these tasks still need to use a standard imperative read, declarative reads inherit the correctness guarantees of imperative reads. Specifically, existing distributed storage systems cache data about where to fetch blocks that can be stale by the time of the actual request. These systems use mechanisms such as leases from the metadata service to ensure read consistency in the face of deletion. Maintenance tasks using Declarative IO can still rely on these consistency mechanisms. Thus, even if they attempt to read deleted blocks returned in callbacks, the read will fail as expected.

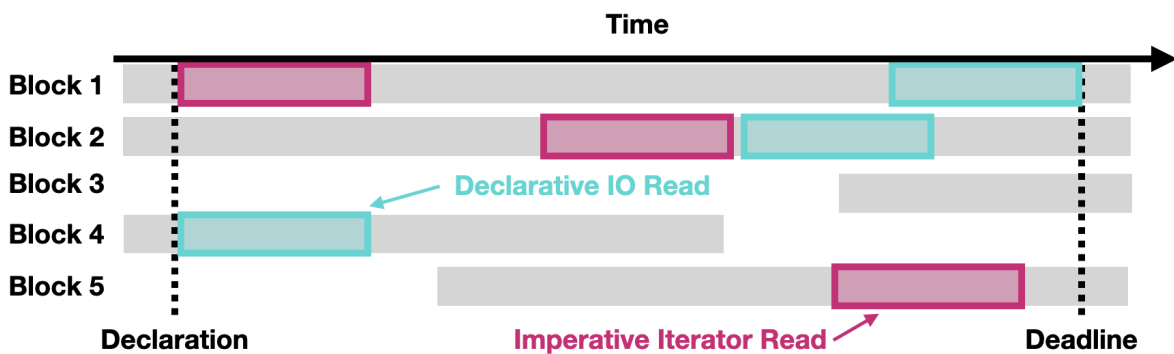


Figure 6.10: Block choice in scrubbing. An example comparison of which blocks scrubbing could find request an iterative and a declarative implementation. Each block is a BlockSet.

File creations and appends. After a maintenance task makes a request, new data might be appended to the corresponding files or new files might be created. The resulting new blocks will not be part of that request since the request’s blocks are mapped at declaration time. Tasks can create new declarations if they need these new blocks.

Most maintenance tasks already handle new files and blocks in a similar fashion. For instance, both the iterative scrubber and the declarative scrubber in Fig. 6.10 miss block 3. In fact, Declarative IO provides a more exact guarantee than a standard iterator — it will return for all blocks present at the time of declaration.

Metadata operations consistency. The final set of operations that affect Declarative IO are metadata operations, particularly renames and changing access permissions. Since we translate files into blocks, Declarative IO does not know about renaming and will send callbacks to tasks even if the access permissions on the files have changed. However, similar to deletions, this is not a problem since tasks will re-obtain permissions to read the data using the imperative interface.

Overloaded system. While Declarative IO tries to finish tasks before their deadlines, it cannot guarantee that all deadlines will be met if the storage system is overloaded. When a deadline is missed due to overload, the storage system will will inform the impacted task using the callback with the OVERLOADED flag. The task may then immediately issue any imperative requests it needs to complete, or the task may continue to wait for additional callbacks if further delay is deemed to be tolerable.

6.3 DINGOS Design

We now introduce DINGOS, a distributed storage system that supports Declarative IO. Our main addition is the DINGOS IO Planner, which receives declarations and schedules them to reduce disk IO while meeting deadlines. In this section, we provide an overview of DINGOS (Sec. 6.3.1), then discuss the IO Planner scheduler (Sec. 6.3.2) and the IO Planner dispatcher (Sec. 6.3.3).

6.3.1 DINGOS Overview

DINGOS supports Declarative IO and its new functions. It also must support existing imperative interface calls for foreground applications and as the read mechanism for declarations.

Imperative requests. As seen in Fig. 6.11, imperative requests, such as application requests, operate in DINGOS just like in other distributed storage systems (Sec. 2.3.4). These requests first go to the metadata service to translate file requests into node-specific block requests and to get a lease on the requested data. The imperative task can then query this data from the appropriate data nodes, traversing the cache to minimize disk IO.

Declarative requests. DINGOS differs from other distributed storage systems because it supports Declarative IO. Unlike imperative requests, declarative requests go to a new

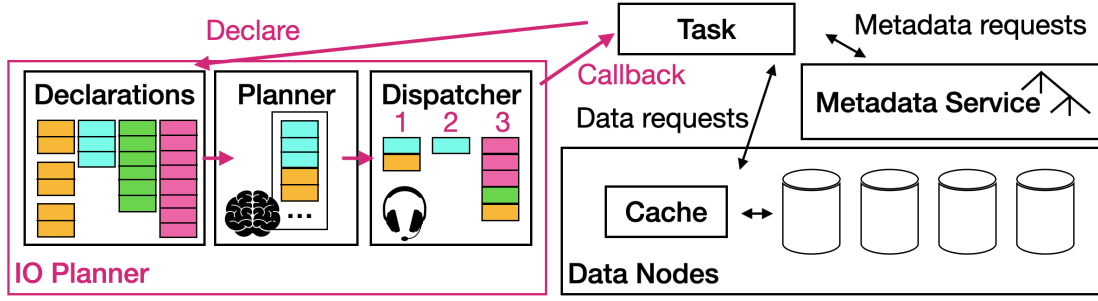


Figure 6.11: DINGOS overview. Imperative requests proceed as normal in DINGOS for both metadata and data requests. Declared requests go to the a new component: the IO Planner. The IO Planner keeps track of outstanding declarations, schedules some subset of them every time quanta, and then dispatches them to minimize cache overhead.

component in DINGOS—the *IO Planner*. The IO Planner stores declarations and periodically decides which ones to execute in the next scheduling quanta. The planner assumes that all maintenance tasks cumulatively have a total disk IO limit for a set amount of time. The planner’s goal is to reduce disk IO while adhering to the IO limit and meeting declaration deadlines. We implement a rate-based planner to optimize IO while making progress towards the deadlines (Sec. 6.3.2).

Once the planner decides which blocks to schedule for the next quanta, the dispatcher determines a dispatch ordering. The goal of this ordering is to minimize the time between callbacks that need the same data, effectively reducing the required cache space (Sec. 6.3.3). The dispatcher can leverage heuristics, such as grouping by erasure blocks, to simplify dispatching. To actually dispatch a declaration, its callback is invoked, indicating to the task that it should imperatively fetch the associated data. When multiple tasks fetch the same data at the same time, said data becomes cacheable and only one disk read must occur for each block.

6.3.2 Scheduling in DINGOS’s IO Planner

We now discuss how the IO Planner schedules declarations to reduce IO while meeting deadlines.

Problem definition. The main constraint in deploying high-capacity disks is IO availability. As such, the IO Planner’s goal is to find the least amount of disk IO needed to fulfill all declared requests within their deadlines. Specifically, the IO Planner must decide which blocks to read in the next timestep given its set of outstanding declarations and a limit on total disk IO. Declarations received during the scheduling process are not considered until the next timestep.

We assume that maintenance tasks that access the same block within the same timestep only need to read the block from disk once, i.e., that every block in a timestep can be cached for the duration of the timestep. Furthermore, maintenance tasks will request blocks within the same timestep that the block was scheduled. We also assume for simplicity that caching

cannot occur between timesteps for maintenance tasks using Declarative IO, since we want to minimize our caching overhead.

Finally, we assume that the IO Planner will schedule the maximum number of blocks allowed in the quanta unless there are no outstanding declarations. While we could potentially reduce cumulative IO by delaying reading blocks, DINGOS aims more broadly to reduce the IO limit per quanta, or the maximum IO that maintenance tasks need, since this is necessary to avoid the IO-per-TB wall.

Scheduling for reuse is NP-hard. To show that the problem is NP-hard, we take a simplification of the problem where deadlines are infinite. Assume that we have an oracle that knows all declarations and which block sets in each declaration will create the most overlap if the declaration includes any data-flexibility. We want to create a schedule for the chosen block sets that minimizes the maximum number of blocks read in any of n time periods. We can reduce this problem to a VM packing problem [225] or a bin-packing problem in the special case where there is no overlap. Since this is a simplification of our scheduling problem, scheduling for reuse in DINGOS is NP-hard.

Rate-based scheduling. For DINGOS’s scheduling heuristic, we decide to prioritize meeting deadlines over finding data reuse because (1) scheduling for reuse is NP-hard, (2) we still observe significant data overlap, and (3) not meeting deadlines leads to more imperative IO without reuse. To bias the scheduler towards meeting deadlines, we implement a rate-based scheduling heuristic. For each declaration, we calculate the rate of blocks per quantum needed to finish the declaration by its deadline; the IO Planner prioritizes declarations with higher rates. If the IO available in the scheduling quantum is less than what is needed to meet each declaration’s rate, we may have an overloaded system depending on the overlap of outstanding declarations.

In order of decreasing rate, the planner chooses block sets for the next quantum. We schedule entire block sets since they are the unit of work for each declaration. The planner selects as many block sets as it can while adhering to the total disk IO limit. Since we only need to read a block once per scheduling quantum, any blocks already scheduled in an earlier block set are considered “free” and do not count towards the limit.

Once the IO Planner reaches the total disk IO limit for the quanta, it performs a second pass over the remaining block sets to find those that are a complete subset of the blocks already scheduled. It includes these block sets to this scheduling timesteps, since they are free and do not contribute to the scheduled disk IO. This inclusion just ensures that the appropriate callbacks are invoked for these block sets, notifying the maintenance task that the data is available.

6.3.3 DINGOS’s dispatcher minimizes cache space

To match our scheduling assumptions, DINGOS needs to have cache space for all disk IO it schedules in a time period. Ideally, we would have scheduling timesteps that are as small as possible to minimize cache size (on the order of minutes instead of hours). But small quanta are not practical since our scheduling algorithm is $O(\text{block sets})$ and there can be

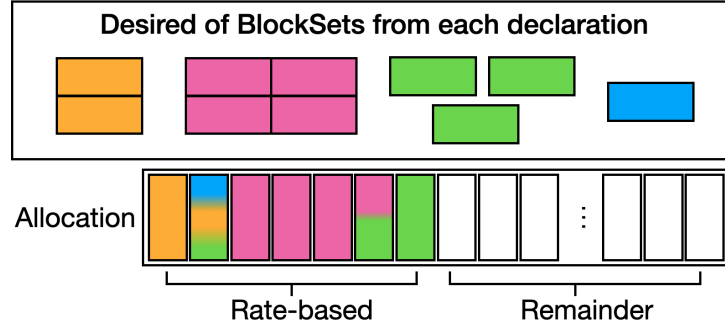


Figure 6.12: DINGOS scheduler. DINGOS uses a rate-based scheduler which biases toward completing all declarations by the deadline. It first finds the number of block sets that need to be done each scheduling quanta to meet the deadline and

many outstanding declarations. Instead, we break DINGOS’s cache size dependency on the scheduling timestep through DINGOS’s dispatcher.

The dispatcher takes the planner’s IO schedule and divides it into smaller groups of block sets. The dispatcher then selects an arbitrary ordering of these groups and, for each group, issues callbacks for all related declarations.

Erasure blocks heuristic simplifies dispatching. To simplify this partitioning problem, we use a heuristic based on *erasure blocks* — a group of blocks that are erasure coded together — to find dispatching groups. Since erasure blocks typically belong to a file, many block sets are either single blocks or a set of erasure blocks. For each dispatching group, we choose an erasure block in the IO plan and find all block sets that use any block in that erasure block. We then include any other erasure blocks in our group that those block sets use. The dispatcher keep iterating until all block sets in the dispatch group have their blocks and all blocks sets that include the chosen blocks are in the dispatch group and or the maximum cache size is reached. While dispatching to minimize cache space can decrease the reuse in highly-connected IO plans, we find that the dispatcher results in small cache requirements without losing much reuse in practice (Sec. 6.4).

6.4 Evaluation

We now present the experimental results for Declarative IO. To evaluate Declarative IO, we implement a prototype on top of HDFS, and additionally simulate a datacenter-scale distributed storage system using Declarative IO.

6.4.1 DINGOS on top of HDFS

To explore the impact of Declarative IO on a real system, we build DINGOS on top of HDFS. We modified scrubbing, reconstruction, rebalancing, and transcoding to use Declarative IO. Scrubbing and rebalancing are modified as described in Sec. 6.2.3. Both reconstruction and transcoding are modified to expose time- and order-flexibility.

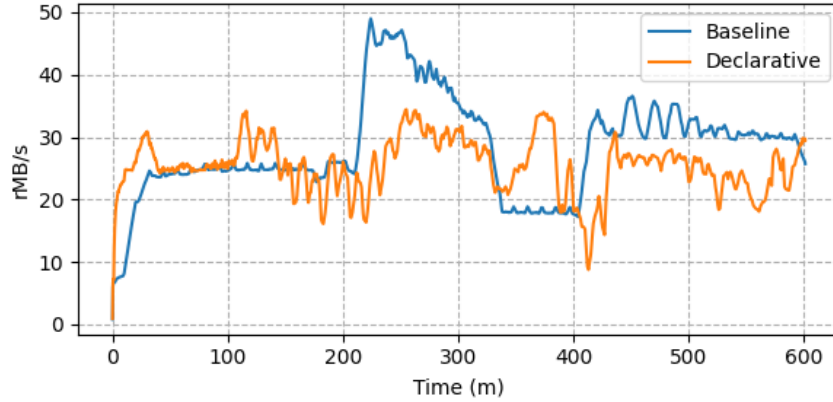


Figure 6.13: Declarative vs imperative IO reads over time. Disk read IO in the cluster for DINGOS and the baseline default HDFS. A Datanode was added at minute 8, and Datanodes were dropped at minute 204 and minute 398. 1/16th of the data is marked for transcoding every 35 minutes. DINGOS can schedule 24 MB/s of reads each quanta

Experimental setup. We evaluate our declarative version of HDFS on a local cluster with 1 Namenode, 16 Datanodes (before adding and dropping), 1 node for the Balancer, and 1 node for the transcoding workload. The cluster consists of SuperMicro 4042G-6RF nodes, each with 64 cores, 128 GiB RAM, 2x 3TB HDDs and 64GB SSD connected with 40 GbE and FDR10 InfiniBand. We compare DINGOS against a default version of HDFS. The IO Planner is configured with a period of 60 seconds.

We distinguish between logical and physical IO to determine the efficiency of Declarative IO. Logical IO represents bytes read or written by HDFS, regardless of whether those bytes were in the cache or disk. Physical IO represents disk IO. We determine logical work by collecting read and write metrics from within HDFS. For instance, if HDFS reads 24 MB for a reconstruction task, that counts as 24 MB of logical work, even if it only generated 16 MB of disk IO. To calculate physical IO, we measure disk statistics with `iostat`.

By default, HDFS leverages the OS for caching. We do not use the explicit cache pinning mechanism because it only works at a file level. To restrict cache size since we do not run a foreground workload. We limit each Datanode process to 1 GB of memory using `cgroups`. We explore the necessary cache space more thoroughly with the simulator Sec. 6.4.2.

DINGOS reduces IO by 24% over default HDFS. We now evaluate our DINGOS implementation. In this experiment, the cluster starts with 500 GB of data split into 64 MB files. All files begin in a 3-way replication scheme. Over the course of the experiment, we add one Datanode and drop three Datanodes to trigger the appropriate amount of rebalancing and reconstruction. The transcoding workload marks 11 GB of files every 12 minutes.

Fig. 6.13 illustrates disk IO in the HDFS cluster over the course of the experiment. Over the course of the experiment, Declarative IO has **24%** fewer IOs than default HDFS. The baseline configuration achieves an efficiency of $.97\times$ (logical IO per physical IO). In

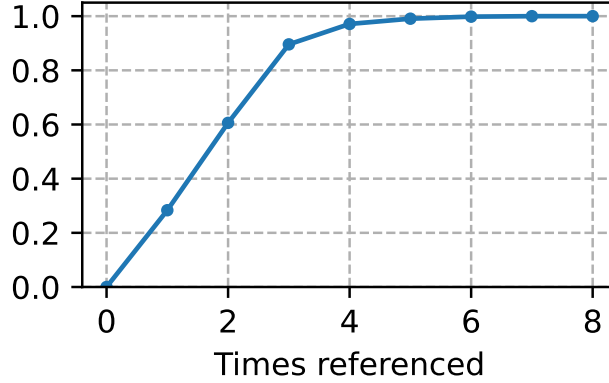


Figure 6.14: CDF of blocks accessed by maintenance tasks. Cumulative density function of block references in maintenance task declarations. The x-axis marks how many declarations a block appeared in. 71.7% of blocks are referenced more than once.

comparison, HDFS with Declarative IO has an efficiency $1.32\times$. In other words, given a maintenance budget of 1 MB/s/TB, default HDFS can achieve about 1 MB/s/TB of work, while Declarative IO achieves 1.32 MB/s/TB. DINGOS’s IO Planner expects an efficiency of $1.37\times$, but not 100% of read overlaps occur as expected. We find that this occurs mostly due to degraded reads after disk failure.

In addition, compared to the default version of HDFS, DINGOS more evenly distributes maintenance IO because DINGOS controls how much IO can occur each quanta over all of the tasks whereas imperative IO requires per task control. This control shows another benefit of Declarative IO, being able to centrally manage when maintenance IO occurs — a useful tool that could be used for example to increase maintenance IO when there is less foreground work, prioritize recovery in large-scale failures, or bias maintenance IO towards disks with more available IO (such choosing data in stripes on lower-capacity disks). Limiting maintenance IO when there is more foreground work and increasing its rate when there is less foreground work would also minimize the impact of maintenance IO on foreground work and could be done based on dynamic changes in the storage cluster, while still ensuring the maintenance IO occurs.

Most IO from maintenance tasks has overlap. Fig. 6.14 shows how often maintenance tasks access the same data. When the same blocks are frequently referenced by multiple tasks, it is more likely that Declarative IO can find a schedule that overlaps them. In this experiment, the IO Planner managed to schedule roughly 25% of the maintenance work for “free” (that is, the same block is read by multiple scheduled declarations).

6.4.2 DINGOS in Simulation

To evaluate IO savings at scale and assess the effectiveness of DINGOS, we build a time-driven cluster storage simulator. This section answers four key questions: (1) How much

Maintenance task	Event frequency	Deadline	Cluster 1	Cluster 2
Reconstruction	On disk failure	36 hours	21%	17%
Scrubbing	Once in 30 days	30 days	2%	20%
Garbage Collection	Once in 6 hours	5 days	42%	17%
File transcoding	Once in 12 hours	5 days	15%	25%
Capacity balancing	Once in a day	5 days	14%	10%
Index checking	Once in 15 days	7 days	2%	5%
File scrubbing	Once in 30 days	30 days	4%	6%

Table 6.2: Workload parameters. The simulator uses the event frequency and deadline parameters for both clusters. Each cluster has a different division of IO between the maintenance tasks.

IO does DINGOS eliminate at scale and with more workloads? (2) How does the IO supply affect DINGOS? (3) How much additional cache does DINGOS need?

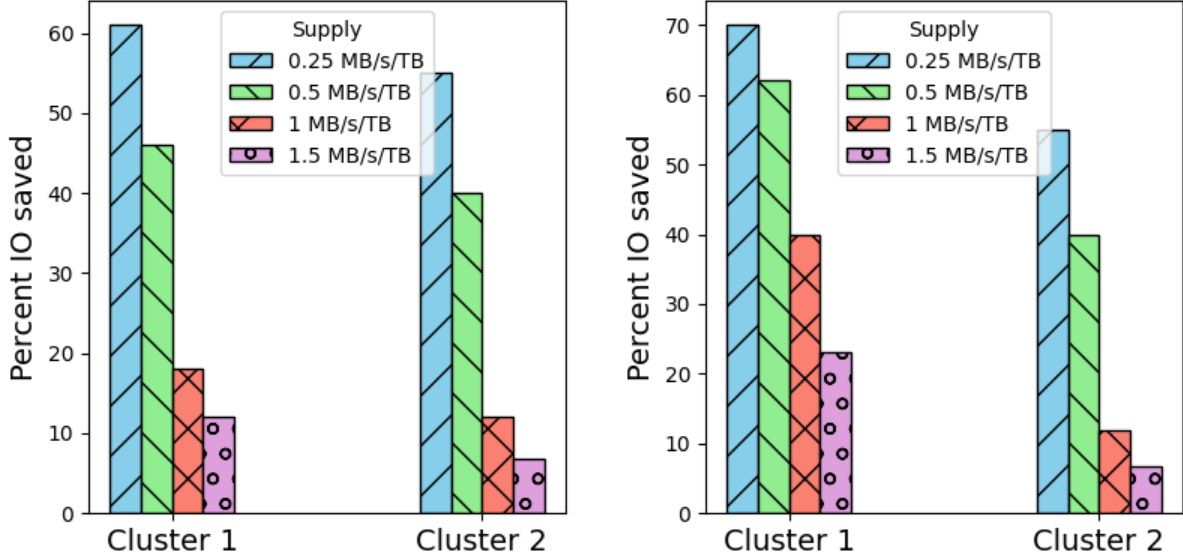
Simulator setup. The simulator models IO activity from real-world datacenter maintenance workloads in discrete 1-hour steps over a 30-day period. The simulated cluster stores data in 250 MB files, distributed in 8 MB blocks across 4 TB disks. The cluster maintains 80% storage utilization. Blocks are either 3-way replicated or part of 6-of-9 or 30-of-33 erasure-coded stripes.

We simulate seven workloads: reconstruction, disk scrubbing, garbage collection, file transcoding, index checking, file scrubbing, and rebalancing.

Reconstruction is triggered by random disk failures. Blocks from the failed disk are recovered via replication or erasure-coded stripes. Each failed disk is replaced to maintain cluster size. Disk scrubbing reads all written blocks to verify integrity. Garbage collection targets a subset of files designated as part of an object/KV store. Files are selected at random at fixed intervals, read fully, and rewritten with new blocks. Transcoding emulates age-based re-encoding for space efficiency [154]. Files are ingested in 3-way replication, then progressively transcoded to 6-of-9 and 30-of-33 EC schemes. Similar to garbage collection, files are fully read and re-written with new blocks. Index checking reads a small set of blocks to allow databases to check if their index is valid. File scrubbing reads all blocks of selected files to check data integrity. Rebalancing identifies disks with skewed usage (by capacity or IO load) and moves a fraction of their blocks to other disks. All blocks stored on the disk are candidates to be moved. Additional workload parameters—such as event frequency and deadlines—are listed in Table 6.2.

Declarative IO saves up to 40% of maintenance reads. DINGOS reduces IO demand by exploiting overlap across maintenance tasks. We measure savings as the ratio of logical work (total data read by workloads) to physical work (data read directly from disk). Declarative IO savings depend on two factors: (1) the IOPS demand of maintenance tasks, and (2) the IOPS supply, i.e. bandwidth available to them.

First, the total IO demand affects the opportunity for overlap. To model variation across clusters, we also simulate two clusters. Cluster 2 is a higher-demand scenario with 33%



(a) Savings with IO demand of 1.21 MB/s/TB (b) Savings with IO demand of 1.6 MB/s/TB

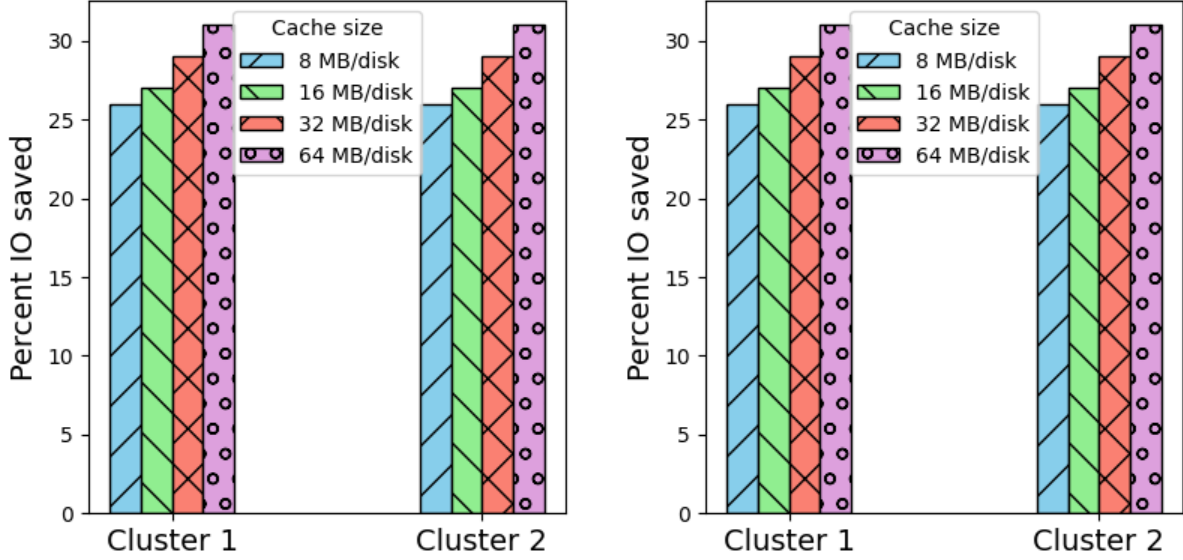
Figure 6.15: IO savings with different supply and demand.

more IO than cluster 1. They also have their IO divided differently between maintenance tasks (Table 6.2).

Second, while the *potential* for savings is independent of the supply of IOPS, the *realized* savings depend on *when* requests are scheduled. In surplus, with ample bandwidth, DINGOS is able to schedule even recently declared requests to exploit idle disk resources, without waiting for potential byte overlap with future requests. Conversely, when bandwidth is scarce (i.e., when the supply of disk IO is high), DINGOS packs multiple requests over a wider time period to exploit byte overlap. In general, bandwidth for maintenance tasks is capped by storage administrators to protect the SLOs of user-facing foreground requests. This constraint is becoming more pronounced as available IOPS-per-TB continue to decline with newer disk models.

We report IO savings as a function of available bandwidth (in MB/s per TB) in Fig. 6.15. DINGOS is able to save up to 70% of IOPS in very low bandwidth scenarios. Unfortunately, Declarative IO with an available bandwidth of 0.5 MB/s/TB causes deadline violations in both cluster. DINGOS saves up to 40% of maintenance reads under 1.0 MB/s/TB supply conditions where it can meet deadlines.

Declarative IO needs minimal cache space. Declarative IO depends on exploiting cache space to allow tasks to re-use IO. The additional cached blocks cannot have a large effect on foreground IO cache misses. We show that Declarative IO is able to effectively save IO with as little as 8 MB of cache per disk (Fig. 6.16) — essentially one block per disk.



(a) Savings with IO demand of 1.21 MB/s/TB (b) Savings with IO demand of 1.6 MB/s/TB

Figure 6.16: Cache size vs IO savings. IO savings with different cache sizes, with supply = 0.75 MB/s/TB

6.5 Related Work

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

Duet and Quartet. Duet and Quartet are the two systems closest to DINGOS. Duet [47] re-order local maintenance reads based what is in cache. Duet notifies modified maintenance tasks when data in the cached data changes. Quartet [100] extends this approach to work for map-reduce jobs. DINGOS does not rely on what is in the caches but rather pre-declared work from different maintenance tasks. This approach allows Declarative IO to make a more streamlined interface that can be used across many maintenance tasks. Declarative IO also introduces both time- and data-flexibility to increase reuse.

Disk-level optimizations. We also see the re-order of IO in some disk-level optimizations. For instance, freeblock scheduling [169] reorders low-priority IO to exploit disk-head time spent rotating to the right disk location. This can be extended to local maintenance reads [239]. These optimizations are orthogonal to Declarative IO since the maintenance reads will still go through these disk-level optimizations.

Semantically-smart disks. Semantically-smart disks [49, 226, 227] aim to understand the structure of the higher-level system such as the file system or database to make better data placement decisions or secure deletion of data. These disks increase functionality with or without changing the disk interface. Declarative IO addresses a different problem — the impending IO-wall in distributed storage system on HDDs — and takes a different

approach — embracing interface change as necessary to solve its problem. We also see the interface change as incrementally deployable since DINGOS starts seeing benefits when only a few maintenance tasks are modified.

Database-level optimizations. The database community combines data requests, such as through shared scans [42, 50]. These optimizations rely on having database queries — which do not occur for many maintenance tasks. Declarative IO instead targets the file and block level since all maintenance tasks that want data must eventually go through the distributed storage system. Thus, we need an interface specifically at that layer, and we find that simpler interfaces than those in databases are sufficient.

Reducing IO from individual maintenance tasks. There has been work trying to reduce IO from individual maintenance tasks such as transcoding [154]. We view this work as orthogonal — any improvements in reducing this IO will just add to the end goal of overcoming the IO-per-TB wall and deploying more dense hard disk drives.

RAID. RAID [82] advocated for an array of lower-capacity, *inexpensive* disks to decrease cost. Today, RAID still has a large impact on distributed storage in the form of erasure coding and replication. However, its motivation to use lower-capacity disks is no longer present because the cheapest, lowest-emissions disks are the highest-capacity disks.

Chapter 7

Conclusion

“Sustainability is not just about adopting the latest energy-efficient technologies or turning to renewable sources of power. Sustainability is the responsibility of every individual every day. It is about changing our behaviour and mindset to reduce power and water consumption, thereby helping to control emissions and pollution levels.”

Joe Kaeser [19].

THIS DISSERTATION addresses the gap on research to reduce storage emissions in datacenters, particularly focusing on embodied emissions where storage is the dominant source of emissions. We find storage emissions are primarily from the storage devices themselves and identify that IO limits the use of denser, longer-lived flash and HDDs. Since these are the two ways to reduce embodied emissions from a datacenter perspective, this dissertation focuses on reducing IO to enable more sustainable storage.

We present techniques to reduce IO through increasing the utility of each read or write in the data-retrieval system by trying to achieve multiple objectives through each IO. To accomplish higher IO utility, we had to rethink both the device-system and the storage system-user interfaces. In particular, we focused on two uses for storage devices in the datacenter: flash caching and bulk storage.

For flash caching, we present both Kangaroo and FairyWREN to reduce writes to flash, allowing for longer flash deployments with denser drives. Kangaroo reduce application-level write amplification through exploiting hash collisions between objects in its on-flash log to move multiple objects each time a set is rewritten. FairyWREN then tackles device-level write amplification through classifying and exploiting new WREN interfaces that allow caches to have control over garbage collection writes. FairyWREN then combines these writes with Kangaroo’s log flushes to reduce writes. Together, these improvements reduce writes by 28x and emissions by over 50%.

For bulk storage, we introduce Declarative IO — a new distributed storage interface that allows us to combine distributed system reads from different maintenance tasks into one disk read. Our implementation, DINGOS, allows us to reduce reads by up to 40% relative to traditional, imperative interfaces, showing that this interface is a promising approach to reduce disk IO. This IO reduction is essential for deploying new denser HDDs that have lower, emissions-per-bit.

All of these systems support the thesis: reducing IO, through increasing the utility of each read and write and developing more expressive and symbiotic interfaces, enables more sustainable storage systems in datacenters. They also are some of the first systems that show how systems-level redesigns can significantly reduce storage emissions, meaning that we do not have to wait for manufacturers to become greener to reduce embodied emissions. Rather, we can design systems to reduce emissions through identifying already existing hardware that embodies less carbon emissions and enable the deployment of that hardware through building more efficient systems that address sustainability bottlenecks.

7.1 Future work

While this dissertation makes significant progress toward sustainable data retrieval systems, there is more work to be done. In this section, we discuss future directions starting with flash caching, then for declarative IO, and finally more broadly in reducing storage emissions.

7.1.1 Flash caching

There are several opportunities to further improve flash caching.

Pushing the Pareto curve between writes, misses, and DRAM overhead. Kangaroo and FairyWREN both enable new points on the trade-off space between writes, misses, and DRAM overhead. There is potential to make a more optimal trade-off space. Particularly, we leave unexplored using multiple hash functions to increase hash collisions. This could allow a flash cache to further minimize writes, but could increase the number of reads needed to find objects or alternative the DRAM overhead for filters to minimize these reads. The design would also have to carefully consider how to minimize conflict misses. Additionally, our flash caching work is limited to using Bloom filters in memory. There are more information-optimized filters that could decrease memory overheads [109, 193, 195]. Finally, we explore separating sets by likelihood of object eviction in FairyWREN. This idea could be extended to more temperature divisions, beyond just hot and cold to potentially further reduce writes at the cost of complexity and increased likelihood of misprediction.

Can we break the lifetime vs density trade-off in flash? In FairyWREN, we found that flash writes decrease dramatically with per-cell density, and increasing lifetimes by using less dense flash leads to lower emissions. This result shows a trade-off between reducing flash embodied emissions through increasing density or increasing lifetime, since flash’s write endurance limits both. However, this trade-off is not fundamental because flash can decrease density overtime and likely regain its extended lifetime. We can investigate whether it’s possible to get the best of both worlds — a long lifetime flash device that reduces density as it wears out.

Does persistence help recovery from metastable failures? In this dissertation, we do not leverage flash’s persistence for caching. Flash is chosen instead because it is a cheaper,

lower emissions alternative to DRAM that is more performant than bulk storage’s HDDs. A problem more generally with large caches is that when they fail, the storage hierarchy enters a metastable failure [72, 130]. Essentially, the system that can handle a given load in a stable state can only handle a fraction of that load after cache failure, *even when the cache servers return*. Since the caches have to be warmed up after failure, the underlying storage system still sees higher load than under stable conditions, inhibiting returning to a stable state. We could potentially do more to leverage flash persistence to reduce this load amplification that occurs after cache failures.

7.1.2 Declarative IO

Now, we discuss future ideas to improve Declarative IO.

Can we improve DINGOS? DINGOS is an initial system that supports Declarative IO. However, we believe there is room for continued research to improve how the system works. For instance, DINGOS avoids using too much cache space to minimize its impact on foreground applications. Instead of just minimizing disruption, DINGOS could use foreground requests to further reduce IO by understanding what data is in the cache and prioritizing declarations for those requests since they would need less disk IO. This optimization would require further integrations with the caches and thus would need to make sure not to slow down the caching system.

DINGOS could also use a more optimized scheduler. DINGOS uses a rate-based scheduler that always schedules work. This policy is pretty good for basic maintenance workloads when we are aiming for lowering our maximum IO load since that maximizes disk density. However, if the declarations become more complicated, it may make sense to further optimize the scheduler, such as by delaying requests to maximize IO overlap. It would also behoove this work to have a better theoretical understanding of the optimal scheduling policy.

Extending Declarative IO beyond maintenance tasks. While we target maintenance tasks with Declarative IO because they are more flexible and are responsible for a large portion of disk IO, there are other tasks amenable to Declarative IO. Since adding more tasks would make Declarative IO more efficient, it would further to reduce disk IO to include these tasks.

User tasks such as ML training or analytics are a large source of IO and have inherent flexibility, particularly in ordering. These applications have two main challenges: they have demanding performance requirements and their developers have little knowledge of the storage stack. To target these applications, Declarative IO needs to have stronger timing guarantees and not require extensive storage knowledge to integrate.

Additionally, it would increase declared work to add a declarative interface to public cloud storage, such as an additional interface to S3. This would require better expressing SLAs and considering how to price declarations. While presumably they would result in cheaper reads to encourage adoption, there are challenges attributing the reads correctly to different declarations.

What happens if bulk storage moves to flash? If bulk storage moves to flash, writes become the fundamental limitation. Datacenters would need to reduce writes in bulk storage to deploy dense flash. To accomplish this, we need to combine writes. Distributed storage has different types of writes, such as adding new data, moving data, and regrouping data. Importantly, these writes do not always conflict, for instance, regrouping and moving data writes do not conflict. Declarative IO could be expanded to combine writes, by carefully categorizing each write’s goal. The resulting system could also incorporate lifetime so new data can also be more intelligently placed, reducing regrouping writes.

7.1.3 Sustainable storage

Finally, we step back and discuss ideas to further reduce storage emissions.

Increasing HDD lifetime. Although in this dissertation, we address increasing flash lifetimes, we do not try to increase HDD lifetimes. This discrepancy is due to HDDs dramatic increases in failure rates as lifetime increases. For instance, reported annual failure rates can double when lifetime increases from three to six year lifetimes [143] as HDDs enter end of life [106, 107, 260]. We believe that increasing HDD lifetime is possible, but would require a different approach such as adaptive redundancy or enabling partial failures can mitigate extra failures caused by extended lifetime. These approaches could also help flash lifetimes.

Adaptive redundancy was developed to enable lower capacity erasure-coding schemes during the useful life phase of HDD deployment [143, 144, 145]. For extending device lifetime, a similar idea could ensure durability at older ages — without requiring additional capacity overhead during the traditional lifetime. This reduction from embodied emissions will have to be balanced with transitioning the erasure codes with age, which causes additional IO that stresses bandwidth particularly for denser drives.

Another way to mitigate the increased failure rates is to embrace partial failures. Although storage devices present a fixed capacity, this is not the reality. SSD cells wear out at different rates. HAMR HDDs can have some lasers fail. Sectors on HDDs can develop defects. While devices today can handle a limited number of defect failures, the device must fail if it no longer has the advertised fixed capacity. Thus, these partial failures are total failures today, causing us to lose usable capacity that we have already paid the embodied emissions for. We need to reconsider total failure and enable partial failure by changing the storage stack and how clouds deploy and replace drives. For HDDs particularly, while we generally know that annual failure rates increase with age [202, 214], we do not have the telemetry to know exactly why the device failed, limiting our ability to determine the emission benefits of partial failure. For instance, a HAMR drive with one laser failing results in a partial failure whereas the drive’s only actuator no longer being reliable results in a complete drive failure since no part of the device is readable.

Reconsidering which storage media to choose. If we push using fewer, denser devices to the extreme, we need to consider media typically meant for archival storage: tape [216], glass [48], and DNA [102, 188]. All of these media have much longer access times, so we would need workloads that can tolerate these longer access times. The potential benefit is

lower emissions. Tape has the potential to lower emissions by 87% per bit [136]. Unfortunately, this estimate does not include the robots and climate control needed to deploy tape, which significantly offsets its emissions reduction. Both glass and DNA are much denser than tape, so they have the potential to reduce emissions, but we cannot determine their emissions potential until more data is available on their lifecycle embodied and operational emissions, particularly when factoring in their achievable IO.

Better metrics and transparency. To truly build sustainable datacenters, we need to better understand where carbon emissions come from and how to reduce them so that systems designers can include sustainability as a first-order metric when considering future computer systems. Finding up-to-date, accurate carbon emissions estimates is non-trivial. This difficulty has real consequences, such as not appreciating the impact of storage on embodied emissions [180]. While this dissertation aims to add some transparency to the carbon emissions of storage, increasing this knowledge further is essential so that the research community can focus on projects that have the most impact and understand the trade-offs that are inherent to designing sustainable systems.

Bibliography

- [1] Is there a limit to the number of layers in 3d-nand? <https://semiengineering.com/is-there-a-limit-to-the-number-of-layers-in-3d-nand/>.
- [2] Azure cache for redis. <https://azure.microsoft.com/en-us/services/cache/#what-you-can-build> 5/5/21.
- [3] Amazon dynamodb. <https://aws.amazon.com/dynamodb/features/> 5/5/21.
- [4] Fatcache. <https://github.com/twitter/fatcache>.
- [5] Redis on flash. <https://docs.redislabs.com/latest/rs/concepts/memory-architecture/redis-flash/>.
- [6] Amazon sustainability. <https://sustainability.aboutamazon.com/climate-solutions>.
- [7] Apache traffic server. URL <https://trafficserver.apache.org>. Accessed: 2019-04-22.
- [8] Dingoes aren't just feral dogs, says study. <https://phys.org/news/2022-04-dingoes-feral-dogs.html>. (Accessed on 06/15/2025).
- [9] Disk prices. <https://jcmnit.net/diskprice.htm>.
- [10] Memory prices. <https://jcmnit.net/memoryprice.htm>.
- [11] Superb fairy wren. <https://wildlifewonders.org.au/wild-lives/superb-fairy-wren>.
- [12] Flash prices. <https://jcmnit.net/flashprice.htm>.
- [13] Climate change is humanity's next big moonshot. <https://blog.google/outreach-initiatives/sustainability/dear-earth/>, .
- [14] Helping you pick the greenest region for your Google Cloud resources. <https://cloud.google.com/blog/topics/sustainability/pick-the-google-cloud-region-with-the-lowest-co2>, . (Accessed on 04/26/2024).
- [15] How Lasers Could Unlock Hard Drives With 10 Times More Data Storage. <https://www.popularmechanics.com/technology/a20078/heating-magnets-lasers-could-be-the-key-magnetic-recording/>, .
- [16] Seagate Reveals HAMR HDD Roadmap: 32TB First, 40TB Follows. <https://www.tomshardware.com/news/seagate-reveals-hamr-roadmap-32-tb-comes-first>, .
- [17] Seagate: HAMR is nailing it – no looming 20TB to 30TB capacity problem. <https://blocksandfiles.com/2021/09/24/seagate-hamr-on-course-no-looming-20-to-30tb-capacity-problem/>, .
- [18] IMEC netzero virtual fab. <https://netzero.imec-int.com/>. (Accessed on 04/26/2024).
- [19] Sustainability is everyone's responsibility. <https://gulfnews.com/opinion/op-eds/>

- [sustainability-is-everyones-responsibility-1.1280327](#). (Accessed on 06/16/2025).
- [20] LevelDB. <https://github.com/google/leveldb>.
 - [21] Makersite Data Platform. <https://makersite.io/makersite-ai-data-apps/>. (Accessed on 04/26/2024).
 - [22] Ssd over-provisioning and its benefits. <https://www.seagate.com/blog/ssd-over-provisioning-benefits-master-ti/>.
 - [23] Wd and tosh talk up penta-level cell flash. <https://blocksandfiles.com/2019/08/07/penta-level-cell-flash/> 5/17/22.
 - [24] Rocksdb. <http://rocksdb.org>.
 - [25] Samsung plans big capacity jump for ssds, preps 290-layer v-nand this year, 430-layer for 2025. <https://www.tomshardware.com/pc-components/ssds/samsung-plans-big-capacity-jump-for-ssds-preps-290-layer-v-nand-this-year-430-layer-for-2025/>
 - [26] Skyhawk datasheet. https://www.seagate.com/www-content/datasheets/pdfs/skyhawk-3-5-hddDS1902-6-1710US-en_US.pdf, 2017.
 - [27] Facebook reports first quarter 2020 results. *investor.fb.com*, Apr 2020.
 - [28] Exos x10 data sheet. https://www.seagate.com/files/www-content/datasheets/pdfs/exos-x-10DS1948-1-1709-GB-en_GB.pdf, 2021.
 - [29] Exos x18 data sheet. https://www.seagate.com/content/dam/seagate/migrated-assets/www-content/datasheets/pdfs/exos-x18-channel-DS2045-4-2106US-en_US.pdf, 2021.
 - [30] Twitter first quarter 2021 results. *investor.twitterinc.com*, May 2021.
 - [31] Micron 7450 ssd with nvme. <https://www.micron.com/content/dam/micron/global/public/products/product-flyer/7450-nvme-ssd-product-brief.pdf>, 2022.
 - [32] Exos x10 sustainability report. <https://www.seagate.com/content/dam/seagate/assets/esg/planet/product-sustainability/images/exos-x10-10tb-sustainability-report-2022/files/exos-x10-10tb.pdf>, 2022.
 - [33] Exos x18 sustainability report. <https://www.seagate.com/content/dam/seagate/assets/esg/planet/product-sustainability/images/exos-x18-sustainability-report/files/Exos-X18-18TB-Sustainability-Report-2023.pdf>, 2022.
 - [34] Our path to net zero. <https://sustainability.fb.com/wp-content/uploads/2023/07/Meta-2023-Path-to-Net-Zero.pdf>, 2023.
 - [35] Heat assisted magnetic recording HAMR. <https://www.seagate.com/innovation/hamr/>, 2024.
 - [36] Google 2024 Environmental Report, 2024. URL <https://www.gstatic.com/gumdrop/sustainability/google-2024-environmental-report.pdf>.
 - [37] 60tb hard drives arriving in 2028 according to industry roadmap — hdd capacity forecast to double in four years. <https://www.tomshardware.com/pc-components/storage/60tb-hard-drives-arriving-in-2028-according-to-industry-roadmap-hdd-capacity-forecast-to-2028>.
 - [38] HDD User Benchmarks. <http://hdd.userbenchmark.com/>, (accessed July 5, 2023).

- [39] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.
- [40] Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. Skylight—a window on shingled disk operation. *ACM Trans. Storage*, 11(4), October 2015. ISSN 1553-3077. doi: 10.1145/2821511. URL <https://doi.org/10.1145/2821511>.
- [41] 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.
- [42] Parag Agrawal, Daniel Kifer, and Christopher Olston. Scheduling shared scans of large data files. *Proc. VLDB Endow.*, 1(1):958–969, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453960. URL <https://doi.org/10.14778/1453856.1453960>.
- [43] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of optimal page replacement. *J. ACM*, 1971.
- [44] Michael Allison, Arun George, Javier Gonzalez, Dan Helmick, Vikash Kumar, Roshan R. Nair, and Vivek Shah. Towards efficient flash caches with emerging nvme flexible data placement ssds. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys ’25, page 1142–1160, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3696091. URL <https://doi.org/10.1145/3689031.3696091>.
- [45] Ahmed Amer, JoAnne Holliday, Darrell DE Long, Ethan L Miller, Jehan-François Pâris, and Thomas Schwarz. Data management and layout for shingled magnetic recording. *IEEE Transactions on Magnetics*, 47(10):3691–3697, 2011.
- [46] Hrishikesh Amur, James Cipar, Varun Gupta, Gregory R. Ganger, Michael A. Kozuch, and Karsten Schwan. Robust and flexible power-proportional storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC ’10, page 217–228. Association for Computing Machinery, 2010. doi: 10.1145/1807128.1807164. URL <https://doi.org/10.1145/1807128.1807164>.
- [47] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP ’15, page 457–473, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815424. URL <https://doi.org/10.1145/2815400.2815424>.
- [48] Patrick Anderson, Erika Blancada Aranas, Youssef Assaf, Raphael Behrendt, Richard Black, Marco Caballero, Pashmina Cameron, Burcu Canakci, Thales De Carvalho, Andromachi Chatzieleftheriou, Rebekah Storan Clarke, James Clegg, Daniel Cletheroe, Bridgette Cooper, Tim Deegan, Austin Donnelly, Rokas Drevinskas, Alexander Gaunt, Christos Gkantsidis, Ariel Gomez Diaz, Istvan Haller, Freddie Hong, Teodora Ilieva, Shashidhar Joshi, Russell Joyce, Mint Kunkel, David Lara, Sergey Legtchenko, Fanglin Linda Liu, Bruno Magalhaes, Alana Marzoev, Marvin Mcnett, Jayashree Mohan, Michael Myrah, Trong Nguyen, Sebastian Nowozin, Aaron Ogus, Hiske Overweg, Antony Rowstron, Maneesh Sah, Masaaki Sakakura, Peter Scholtz, Nina Schreiner, Omer Sella, Adam Smith, Ioan Stefanovici, David Sweeney, Benn Thomsen, Govert Verkes, Phil Wainman, Jonathan West-

- cott, Luke Weston, Charles Whittaker, Pablo Wilke Berenguer, Hugh Williams, Thomas Winkler, and Stefan Winzeck. Project silica: Towards sustainable cloud archival storage in glass. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23*, 2023.
- [49] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, and Muthian Sivathanu. Semantically-smart disk systems: past, present, and future. *SIGMETRICS Perform. Eval. Rev.*, 33(4):29–35, March 2006. ISSN 0163-5999. doi: 10.1145/1138085.1138093. URL <https://doi.org/10.1145/1138085.1138093>.
- [50] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10*, page 519–530, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300322. doi: 10.1145/1807167.1807224. URL <https://doi.org/10.1145/1807167.1807224>.
- [51] Desire Athrow. Seagate launches biggest hard drive ever — 30tb exos mozaic 3+ hdd can store more than 1,000 blu-ray movies and, yes, everyone will be able to buy them. <https://www.techradar.com/pro/seagate-launches-biggest-hard-drive-ever-30tb-exos-mozaic-3-hdd-can-store-more-than-1000-2024>.
- [52] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. What you can't forget: Exploiting parallelism for zoned namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems, HotStorage '22*, page 79–85, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393997. doi: 10.1145/3538643.3539744. URL <https://doi.org/10.1145/3538643.3539744>.
- [53] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony Rowstron. Pelican: A building block for exascale cold data storage. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 351–365, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/balakrishnan>.
- [54] M Garcia Bardon, P Wuytens, L-Å Ragnarsson, G Mirabelli, D Jang, G Willems, A Mallik, A Spessot, J Ryckaert, and B Parvais. Dtco including sustainability: Power-performance-area-cost-environmental score (ppace) analysis for logic technologies. In *2020 IEEE International Electron Devices Meeting (IEDM)*, pages 41–4. IEEE, 2020.
- [55] Noman Bashir, Tian Guo, Mohammad Hajiesmaili, David Irwin, Prashant Shenoy, Ramesh Sitaraman, Abel Souza, and Adam Wierman. Enabling Sustainable Clouds: The Case for Virtualizing the Energy System. In *Symposium on Cloud Computing*, 2021.
- [56] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *IEEE HPCA*, 2015.
- [57] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. Scaling distributed cache hierarchies through computation and data co-scheduling. In *IEEE HPCA*, 2015.
- [58] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd: Improving hit rate by maximizing

hit density. In *USENIX NSDI*, 2018.

- [59] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. Small refinements to the dam can have big consequences for data-structure design. In *ACM SPAA*, 2019.
- [60] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosf, 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.
- [61] 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.
- [62] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. In *ACM SIGMETRICS*, 2018.
- [63] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching—dynamic reallocation from cache-rich to cache-poor. In *USENIX OSDI*, 2018.
- [64] Daniel S. Berger, Fiodar Kazhamiaka, Esha Choukse, Inigo Goiri, Celine Irvine, Pulkit A. Misra, Alok Kumbhare, Rodrigo Fonseca, and Ricardo Bianchini. Research avenues towards net-zero cloud platforms. *Workshop on NetZero Carbon Computing*, 2023.
- [65] Matias Björling, 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.
- [66] Matias Björling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the block interface tax for flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL <https://www.usenix.org/conference/atc21/presentation/bjorling>.
- [67] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC.*, 2017.
- [68] Netflix Technology Blog. Application data caching using ssds. <https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef>, 2016.
- [69] Netflix Technology Blog. Evolution of application data caching : From ram to ssd. <https://bit.ly/3rN73CI>, 2018.
- [70] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *USENIX FAST*, 2010.
- [71] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook’s distributed data store for the social graph. In *USENIX ATC*, 2013.
- [72] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS ’21, page 221–227, New York, NY, USA, 2021. Association for Computing

Machinery. ISBN 9781450384384. doi: 10.1145/3458336.3465286. URL <https://doi.org/10.1145/3458336.3465286>.

- [73] 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.
- [74] 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. ISBN 9781450372060. doi: 10.1145/3357526.3357562. URL <https://doi.org/10.1145/3357526.3357562>.
- [75] Brad Calder, Ju Wang, Aaron Ogus, Niranjana Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatrri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 143–157, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309776. doi: 10.1145/2043556.2043571. URL <https://doi.org/10.1145/2043556.2043571>.
- [76] Enrique V. Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, page 86–97, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137338. doi: 10.1145/782814.782829. URL <https://doi.org/10.1145/782814.782829>.
- [77] Chandranil Chakrabortii 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. ISBN 978-1-4503-8398-1. doi: 10.1145/3456727.3463784.
- [78] 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.
- [79] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.
- [80] Andromachi Chatzileftheriou, Ioan Stefanovici, Dushyanth Narayanan, Benn Thomsen, and Antony Rowstron. Could cloud storage be disrupted in the next decade? In *USENIX HotStorage*, 2020.
- [81] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In *Symposium on Networked Systems Design and Implementation*, 2008.
- [82] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson.

- Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.
- [83] Shuang Chen, Christina Delimitrou, and José F Martínez. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.
 - [84] 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.
 - [85] 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. URL <https://www.usenix.org/conference/hotstorage20/presentation/choi>.
 - [86] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*, 2016.
 - [87] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *USENIX ATC*, 2017.
 - [88] Edward G. Coffman and Peter J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973. ISBN 0136378684.
 - [89] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 47–47, 2002. doi: 10.1109/SC.2002.10058.
 - [90] Diane Colombelli-Négrel, Mark E. Hauber, Jeremy Robertson, Frank J. Sulloway, Herbert Hoi, Matteo Griggio, and Sonia Kleindorfer. Embryonic Learning of Vocal Passwords in Superb Fairy-Wrens Reveals Intruder Cuckoo Nestlings. *Current Biology*, 22(22):2155–2160, November 2012. ISSN 0960-9822. doi: 10.1016/j.cub.2012.09.025.
 - [91] Amanda Peterson Corio. Five years of 100carbon-free future. <https://cloud.google.com/blog/topics/sustainability/5-years-of-100-percent-renewable-energy>.
 - [92] Asit Dan and Don Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *ACM SIGMETRICS*, 1990.
 - [93] Carson Molder Sathya Gunasekar Jimmy Lu Snehal Khandkar Abhinav Sharma Daniel S. Berger Nathan Beckmann Greg Ganger Daniel Lin-Kit Wong, Hao Wu. Baleen: Ml admission and prefetching for flash caches. In *FAST*, 2024.
 - [94] Gary Davis. 2020: Life with 50 billion connected devices. In *IEEE International Conference on Consumer Electronics*, pages 1–1, 2018.
 - [95] 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. doi: 10.1145/3183713.3196927.
 - [96] 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. doi: 10.1145/3035918.3064054.

- [97] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *ACM SIGMOD*, 2011.
- [98] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.
- [99] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.
- [100] Francis Deslauriers, Peter McCormick, George Amvrosiadis, Ashvin Goel, and Angela Demke Brown. Quartet: harmonizing task scheduling and caching for cluster computing. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, page 1–5, USA, 2016. USENIX Association.
- [101] Peter Desnoyers. Analytic modeling of ssd write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–10, 2012.
- [102] George Dickinson, Golam Mortuza, William Clay, Luca Piantanida, Christopher Green, Chad Watson, Eric Hayden, Tim Andersen, Wan Kuang, Elton Graugnard, and William Hughes. An alternative approach to nucleic acid memory. *Nature Communications*, 12, 04 2021. doi: 10.1038/s41467-021-22277-y.
- [103] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. Disaggregating rocksdb: A production experience. *Proc. ACM Manag. Data*, 2023. doi: 10.1145/3589772. URL <https://doi.org/10.1145/3589772>.
- [104] Lieven Eeckhout. Focal: A first-order carbon model to assess processor sustainability. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 401–415, 2024.
- [105] 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.
- [106] J.G. Elerath. Afr: problems of definition, calculation and measurement in a commercial environment. In *Annual Reliability and Maintainability Symposium. 2000 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.00CH37055)*, pages 71–76, 2000. doi: 10.1109/RAMS.2000.816286.
- [107] J.G. Elerath. Specifying reliability in the disk drive industry: No more mtbf's. In *Annual Reliability and Maintainability Symposium. 2000 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.00CH37055)*, pages 194–199, 2000. doi: 10.1109/RAMS.2000.816306.
- [108] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI*, 2013.
- [109] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 75–88,

- New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332798. doi: 10.1145/2674005.2674994. URL <https://doi.org/10.1145/2674005.2674994>.
- [110] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.
 - [111] Peter Freiling and Badrish Chandramouli. Microsoft. personal communication.
 - [112] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, Xin Peng, Wenli Zheng, and Minyi Guo. QoS-Aware and Resource Efficient Microservice Deployment in Cloud-Edge Continuum. In *International Parallel and Distributed Processing Symposium*, 2021.
 - [113] Massimo Gallo, Bruno Kauffmann, Luca Muscariello, Alain Simonian, and Christian Tanguy. Performance evaluation of the random replacement policy for networks of caches. *SIGMETRICS Perform. Eval. Rev.*, 40(1):395–396, June 2012.
 - [114] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinisky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.
 - [115] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.
 - [116] Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. The case for sleep states in servers. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower ’11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309813. doi: 10.1145/2039252.2039254. URL <https://doi.org/10.1145/2039252.2039254>.
 - [117] Jiechao Gao, Haoyu Wang, and Haiying Shen. Smartly Handling Renewable Energy Instability in Supporting A Cloud Datacenter. In *International Parallel and Distributed Processing Symposium*, 2020.
 - [118] 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/>.
 - [119] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.
 - [120] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O’Connor, and Onur Mutlu. What your dram power models are not telling you: Lessons from a detailed experimental study. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(3), dec 2018. doi: 10.1145/3224419. URL <https://doi.org/10.1145/3224419>.
 - [121] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, EECS Department, University of California, Berkeley, 12 1990. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/1990/6373.html>.
 - [122] 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.
- [123] 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.
 - [124] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL <https://www.usenix.org/conference/osdi20/presentation/hao>.
 - [125] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The unwritten contract of solid state drives. In *ACM EuroSys*, 2017.
 - [126] Bruce Hoch and Sage Shih. Open Cloud Server - Project Olympus JBOD. <http://files.opencompute.org/oc/public.php?service=files&t=ea8af1772e9eea08a0fc0f8e1691418b>, 2017.
 - [127] Amy Hood, July 2022.
 - [128] 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. ISBN 978-1-931971-225. URL <https://www.usenix.org/conference/atc15/technical-session/presentation/hu>.
 - [129] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, Santa Clara, CA, February 2017. USENIX Association. ISBN 978-1-931971-36-2. URL <https://www.usenix.org/conference/fast17/technical-sessions/presentation/huang>.
 - [130] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL <https://www.usenix.org/conference/osdi22/presentation/huang-lexiang>.
 - [131] Jiao Hui, Xiongzi Ge, Xiaoxia Huang, Yi Liu, and Qiangjun Ran. E-hash: an energy-efficient hybrid storage system composed of one ssd and multiple hdds. In *Proceedings of the Third International Conference on Advances in Swarm Intelligence - Volume Part II, ICSI’12*, page 527–534, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 9783642310195. doi: 10.1007/978-3-642-31020-1_63. URL https://doi.org/10.1007/978-3-642-31020-1_63.
 - [132] Ilias Iliadis, Robert Haas, Xiao-Yu Hu, and Evangelos Eleftheriou. Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):241–252, 2008.

- [133] Aamer Jaleel, Kevin Theobald, Simon Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction. In *ISCA-37*, 2010.
- [134] Jaeheon Jeong and Michel Dubois. Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers*, 55(4):353–365, 2006.
- [135] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.
- [136] Brad Johns. Reducing data center energy consumption and carbon emissions with modern tape storage. <https://datastorage-na.fujifilm.com/wp-content/themes/fuji/images/sustainability/BJC-Reducing-Carbon-Emission-Whitepaper-LR-1120.pdf>, 2020.
- [137] 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. ISBN 1558601538.
- [138] Nicola Jones et al. How to stop data centres from gobbling up the world’s electricity. *Nature*, 561(7722):163–166, 2018.
- [139] 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/>.
- [140] Ajay Joshi. Cachelib on zns. <https://github.com/ajaysjoshi/CacheLib-zns>, 2022.
- [141] E. G. Coffman Jr and Predrag Jelenković. Performance of the move-to-front algorithm with markov-modulated request sequences. *Oper. Res. Lett.*, 25(3):109–118, October 1999.
- [142] Saurabh Kadekodi, K V Rashmi, and Gregory R Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. 2019.
- [143] Saurabh Kadekodi, K. V. Rashmi, and Gregory R. Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 345–358, Boston, MA, February 2019. USENIX Association. ISBN 978-1-939133-09-0. URL <https://www.usenix.org/conference/fast19/presentation/kadekodi>.
- [144] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, KV Rashmi, and Gregory Ganger. Pacemaker: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.
- [145] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, K. V. Rashmi, and Gregory R. Ganger. Tiger: Disk-Adaptive redundancy without placement restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 413–429, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL <https://www.usenix.org/conference/osdi22/presentation/kadekodi>.
- [146] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. Practical design considerations for wide locally recoverable codes (LRCs). In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 1–16, Santa Clara, CA, February 2023. USENIX Association. ISBN 978-1-939133-32-8. URL <https://www.usenix.org/>

[conference/fast23/presentation/kadekodi](https://www.usenix.org/conference/fast23/presentation/kadekodi).

- [147] Rainer W. Kaese. From 20 megabytes to 20 terabytes: 40 years of hard disk drive technology. https://www.toshiba-storage.com/wp-content/uploads/2023/02/Toshiba_40years_HDD_technology_screen.pdf, 2022.
- [148] 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.
- [149] Supriya C. Karekar and Birgitte K. Ahring. Reducing methane production from rumen cultures by bioaugmentation with homoacetogenic bacteria. *Biocatalysis and Agricultural Biotechnology*, 47:102526, 2023. ISSN 1878-8181. doi: <https://doi.org/10.1016/j.bcab.2022.102526>. URL <https://www.sciencedirect.com/science/article/pii/S1878818122002535>.
- [150] William Katsak, Íñigo Goiri, Ricardo Bianchini, and Thu D. Nguyen. Greencassandra: Using renewable energy in distributed structured storage systems. In *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2015. doi: 10.1109/IGCC.2015.7393711.
- [151] Rini T. Kaushik and Milind Bhandarkar. Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower’10, page 1–9, USA, 2010. USENIX Association.
- [152] Richard E. Kessler, Mark D Hill, and David A Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, 1994.
- [153] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully automatic stream management for Multi-Streamed SSDs using program contexts. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 295–308, Boston, MA, February 2019. USENIX Association. ISBN 978-1-939133-09-0. URL <https://www.usenix.org/conference/fast19/presentation/kim-taejin>.
- [154] Timothy Kim, Sanjith Athlur, Saurabh Kadekodi, Francisco Maturana, Dax Delvira, Arif Merchant, Gregory R. Ganger, and K. V. Rashmi. Morph: Efficient file-lifetime redundancy management for cluster file systems. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP ’24, page 330–346, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi: 10.1145/3694715.3695981. URL <https://doi.org/10.1145/3694715.3695981>.
- [155] Bran Knowles. Acm techbrief: Computing and climate change, 2021.
- [156] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *USENIX FAST*, 2015.
- [157] 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. doi: 10.1145/3341301.3359628.
- [158] Cheng Li, Philip Shilane, Fred Douglass, 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.

- [159] Cheng Li, Philip Shilane, Fred Douglass, and Grant Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage*, 13(3):24, 2017.
- [160] Conglong Li, David G Andersen, Qiang Fu, Sameh Elnikety, and Yuxiong He. Workload analysis and caching strategies for search advertising systems. In *ACM SoCC*, 2017.
- [161] Conglong Li, David G Andersen, Qiang Fu, Sameh Elnikety, and Yuxiong He. Better caching in search advertising systems with rapid refresh predictions. In *WWW.*, pages 1875–1884, 2018.
- [162] Daping Li, Xiaoyang Qu, Jiguang Wan, Jun Wang, Yang Xia, Xiaozhao Zhuang, and Changsheng Xie. Workload scheduling for massive storage systems with arbitrary renewable supply. *IEEE Transactions on Parallel and Distributed Systems*, 29(10):2373–2387, 2018. doi: 10.1109/TPDS.2018.2820070.
- [163] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3578835. URL <https://doi.org/10.1145/3575693.3578835>.
- [164] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jiaji Zhu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. More than capacity: Performance-oriented evolution of pangu in alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 331–346, Santa Clara, CA, February 2023. USENIX Association. ISBN 978-1-939133-32-8. URL <https://www.usenix.org/conference/fast23/presentation/li-qiang-deployed>.
- [165] 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/>.
- [166] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *ACM SOSP*, 2011.
- [167] Jian Liu, Kefei Wang, and Feng Chen. Tscache: An efficient flash-based caching scheme for time-series data workloads. 2021.
- [168] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wiskey: Separating keys from values in ssd-conscious storage. In *USENIX FAST*, 2016.
- [169] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, and David F. Nagle. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, OSDI '00, USA, 2000. USENIX Association.

- [170] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Symposium on Cloud Computing*, 2021.
- [171] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.
- [172] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. Improving 3d nand flash memory lifetime by tolerating early retention loss and process variation. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(3), dec 2018. doi: 10.1145/3224432. URL <https://doi.org/10.1145/3224432>.
- [173] Jialun Lyu, Jaylen Wang, Kali Frost, Chaojie Zhang, Celine Irvine, Esha Choukse, Rodrigo Fonseca, Ricardo Bianchini, Fiodar Kazhamiaka, and Daniel S. Berger. Myths and misconceptions around reducing carbon embedded in cloud platforms. In *2nd Workshop on Sustainable Computer Systems (HotCarbon23)*. ACM, July 2023. URL <https://www.microsoft.com/en-us/research/publication/myths-and-misconceptions-around-reducing-carbon-embedded-in-cloud-platforms/>.
- [174] Jialun Lyu, Marisa You, Celine Irvine, 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.
- [175] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. Modular data storage with anvil. In *ACM SOSP*, 2009.
- [176] 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.
- [177] Sara McAllister, Yucong "Sherry" Wang, Benjamin Berg, Daniel S. Berger, George Amvrosiadis, Nathan Beckmann, and Gregory R. Ganger. FairyWREN: A sustainable cache for emerging Write-Read-Erase flash interfaces. mar .
- [178] 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.
- [179] 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.
- [180] Sara McAllister, Fiodar Kazhamiaka, Daniel S Berger, Rodrigo Fonseca, Kali Frost, Aaron Ogus, Maneesh Sah, Ricardo Bianchini, George Amvrosiadis, Nathan Beckmann, et al. A call for research on storage emissions. In *HotCarbon Workshop on Sustainable Computer Systems 2024*, 2024.
- [181] Sara McAllister, Yucong "Sherry" Wang, Benjamin Berg, Daniel S. Berger, George Amvrosiadis, Nathan Beckmann, and Gregory R. Ganger. FairyWREN: A sustainable cache for emerging Write-Read-Erase flash interfaces. In *18th USENIX Symposium on Operating*

- [182] 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. ISBN 978-1-939133-34-2. URL <https://www.usenix.org/conference/osdi23/presentation/min>.
- [183] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F Wenisch. Parslo: A Gradient Descent-based Approach for Near-optimal Partial SLO Allotment in Microservices. In *Symposium on Cloud Computing*, 2021.
- [184] Christian Monzio Compagnoni, Akira Goda, Alessandro S. Spinelli, Peter Feeley, Andrea L. Lacaita, and Angelo Visconti. Reviewing the evolution of the nand flash technology. *Proceedings of the IEEE*, 105(9):1609–1633, 2017. doi: 10.1109/JPROC.2017.2665781.
- [185] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Whirlpool: Improving dynamic cache management with static data classification. In *ASPLOS*, 2016.
- [186] 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.
- [187] PBS Nature. Kangaroo fact sheet. <https://www.pbs.org/wnet/nature/kangaroo-mob-kangaroo-fact-sheet/7444/?repeat=w3tc>.
- [188] Bichlien Nguyen, Julie Sinistore, Jake Smith, Praneet S. Arshi, Lauren M. Johnson, Tim Kidman, T.J. diCaprio, Doug Carmean, and Karin Strauss. Architecting datacenters for sustainability: Greener data storage using synthetic dna. In *Electronics Goes Green 2020*. Fraunhofer IZM, IEEE, September 2020. URL <https://www.microsoft.com/en-us/research/publication/architecting-datacenters-for-sustainability-greener-data-storage-using-synthetic-dna/>.
- [189] Council of the European Union. Fit for 55. <https://www.consilium.europa.eu/en/policies/green-deal/fit-for-55-the-eu-plan-for-a-green-transition/>, 2024.
- [190] Alina Oprea and Ari Juels. A Clean-Slate Look at Disk Scrubbing. 2010.
- [191] James O’Toole and Liuba Shrira. Opportunistic log: Efficient installation reads in a reliable storage server. In *USENIX OSDI*, 1994.
- [192] 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.
- [193] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA ’05*, page 823–829, USA, 2005. Society for Industrial and Applied Mathematics. ISBN 0898715857.
- [194] Satadru Pan, Theano Stavrinou, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook’s tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.
- [195] Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, and Rob Johnson. Vector quotient filters: Overcoming the time/space trade-off in filter design.

- In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1386–1399, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3452841. URL <https://doi.org/10.1145/3448016.3452841>.
- [196] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.
 - [197] Ramtin Pedarsani, Mohammad Ali Maddah-Ali, and Urs Niesen. Online coded caching. *IEEE/ACM Transactions on Networking*, 2016.
 - [198] Sara Perez. Twitter’s doubling of character count from 140 to 280 had little impact on length of tweets. *Tech Crunch*, 2018. URL <https://techcrunch.com/2018/10/30/twitters-doubling-of-character-count-from-140-to-280-had-little-impact-on-length-of-tweets>
 - [199] Phitchaya Mangpo Phothilimthana, Saurabh Kadekodi, Soroush Ghodrati, Selene Moon, and Martin Maas. Thesios: Synthesizing accurate counterfactual i/o traces from i/o samples. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 1016–1032, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703867. doi: 10.1145/3620666.3651337. URL <https://doi.org/10.1145/3620666.3651337>.
 - [200] Eduardo Pinheiro and Ricardo Bianchini. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, page 68–78, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138393. doi: 10.1145/1006209.1006220. URL <https://doi.org/10.1145/1006209.1006220>.
 - [201] Eduardo Pinheiro, Ricardo Bianchini, and Cezary Dubnicki. Exploiting redundancy to conserve energy in storage systems. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, page 15–26, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933190. doi: 10.1145/1140277.1140281. URL <https://doi.org/10.1145/1140277.1140281>.
 - [202] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *Conference on File and Storage Technologies*, 2007.
 - [203] Francisco Pires. Solidigm introduces industry-first plc nand for higher storage densities. <https://www.tomshardware.com/news/solidigm-plc-nand-ssd>, 2022.
 - [204] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *Symposium on Operating Systems Design and Implementation*, 2020.
 - [205] Xiaoyang Qu, Jiguang Wan, Jun Wang, Liqiong Liu, Dan Luo, and Changsheng Xie. Green-match: Renewable-aware workload scheduling for massive storage systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 403–412, 2016. doi: 10.1109/IPDPS.2016.24.
 - [206] 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. ISBN 978-3-540-

- 65142-0 978-3-540-49543-7. doi: 10.1007/3-540-49543-6_13. URL http://link.springer.com/10.1007/3-540-49543-6_13. Series Title: Lecture Notes in Computer Science.
- [207] Ana Radovanović, Ross Koningstein, Ian Schneider, Bokan Chen, Alexandre Duarte, Binz Roy, Diyue Xiao, Maya Haridasan, Patrick Hung, and Nick Care. Carbon-Aware Computing for Datacenters. *Transactions on Power Systems*, 38:1270–1280, 2022.
 - [208] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *ACM SOSP*, 2017.
 - [209] Varsha Rao and Andrew A. Chien. Understanding the operational carbon footprint of storage reliability and management. *SIGENERGY Energy Inform. Rev.*, 4(5):180–187, April 2025. doi: 10.1145/3727200.3727227. URL <https://doi.org/10.1145/3727200.3727227>.
 - [210] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *ACM SOSP*, 1991.
 - [211] Elisha J. Rosensweig, Jim Kurose, and Don Towsley. Approximate models for general cache networks. In *IEEE INFOCOM*, 2010.
 - [212] 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. ISBN ISBN 978-1-931971-08-9. URL <https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble>.
 - [213] Sheraz Sadiq. How kangaroo gut bacteria could help cut a potent source of greenhouse gas emissions. <https://www.pbs.org/wnet/nature/kangaroo-mob-kangaroo-fact-sheet/7444/?repeat=w3tc>, 2023.
 - [214] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. USENIX Association. URL <https://www.usenix.org/conference/fast-07/disk-failures-real-world-what-does-mttf-1000000-hours-mean-you>.
 - [215] Thomas JE Schwarz, Qin Xin, Ethan L Miller, Darrell DE Long, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *The IEEE Computer Society’s 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings.*, pages 409–418. IEEE, 2004.
 - [216] T.J.E. Schwarz, Qin Xin, E.L. Miller, D.D.E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *The IEEE Computer Society’s 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings.*, pages 409–418, 2004.
 - [217] Rathijit Sen and David A. Wood. Reuse-based online models for caches. In *ACM SIGMETRICS.*, 2013.
 - [218] Mark A. Shaw. Project Olympus Flash Expansion FX-16. <http://files.opencompute.org/oc/public.php?service=files&t=14ab3cf25170b7a0a439e11a3d818c96>, 2017.
 - [219] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. Optimizing flash-based key-value cache systems. In *USENIX HotStorage*, 2016.

- [220] 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.
- [221] Anton Shilov. Seagate’s HAMR Update: 32 TB in Early 2024, 40+ TB Two Years Later. <https://www.anandtech.com/show/21125/seagates-hamr-update-32-tb-in-early-2024-40-tb-two-years-later>, 2023.
- [222] Shigeru Shiratake. Scaling and performance challenges of future dram. In *2020 IEEE International Memory Workshop (IMW)*, pages 1–3, 2020. doi: 10.1109/IMW48823.2020.9108122.
- [223] Junaid Shuja, Kashif Bilal, Sajjad A. Madani, Mazliza Othman, Rajiv Ranjan, Pavan Balaji, and Samee U. Khan. Survey of Techniques and Architectures for Designing Energy-Efficient Data Centers. *IEEE Systems Journal*, 10:507–519, 2016.
- [224] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010. doi: 10.1109/MSST.2010.5496972.
- [225] Michael Sindelar, Ramesh K. Sitaraman, and Prashant Shenoy. Sharing-aware algorithms for virtual machine colocation. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’11, page 367–378, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307437. doi: 10.1145/1989493.1989554. URL <https://doi.org/10.1145/1989493.1989554>.
- [226] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST ’03, page 73–88, USA, 2003. USENIX Association.
- [227] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST’05, page 18, USA, 2005. USENIX Association.
- [228] Minseok Song, Yeongju Lee, and Euseok Kim. Saving disk energy in video servers by combining caching and prefetching. *ACM Trans. Multimedia Comput. Commun. Appl.*, 10(1s), jan 2014. ISSN 1551-6857. doi: 10.1145/2537856. URL <https://doi.org/10.1145/2537856>.
- [229] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *USENIX NSDI*, 2020.
- [230] Abel Souza, Noman Bashir, Jorge Murillo, Walid Hanafy, Qianlin Liang, David Irwin, and Prashant Shenoy. Ecovisor: A Virtual Energy System for Carbon-Efficient Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.
- [231] Akshitha Sriraman and Thomas F Wenisch. μ Tune: Auto-Tuned Threading for OLDI Microservices. In *Conference on Operating Systems Design and Implementation*, 2018.
- [232] Louise Story. Anywhere the eye can see, it’s likely to see an ad. *The New York Times*, 15(1), 2007. Available at <https://www.nytimes.com/2007/01/15/business/media/15everywhere.html>, 9/6/2020.

- [233] Chetan Choppali Sudarshan, Nikhil Matkar, Sarma Vrudhula, Sachin S Sapatnekar, and Vidya A Chhabria. Eco-chip: Estimation of carbon footprint of chiplet-based architectures for sustainable vlsi. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 671–685. IEEE, 2024.
- [234] Billy Tallis. Micron 3d nand status update. <https://www.anandtech.com/show/10028/micron-3d-nand-status-update>, .
- [235] 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>, .
- [236] 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.
- [237] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: advanced photo caching on flash for facebook. In *USENIX FAST*, 2015.
- [238] Swamit Tannu and Prashant J Nair. The Dirty Secret of SSDs: Embodied Carbon. In *HotCarbon*, 2022.
- [239] Eno Thereska, Jiri Schindler, John Bucy, Brandon Salmon, Christopher R. Lumb, and Gregory R. Ganger. A framework for building unobtrusive disk maintenance applications. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST ’04, page 213–226, USA, 2004. USENIX Association.
- [240] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys ’11, page 169–182, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306348. doi: 10.1145/1966445.1966461. URL <https://doi.org/10.1145/1966445.1966461>.
- [241] Amanda Tomlinson and George Porter. Something Old, Something New: Extending the Life of CPUs in Datacenters. In *HotCarbon*, 2022.
- [242] Ted Tso. Aligning filesystems to an ssd’s erase block size. <https://tytso.livejournal.com/2009/02/20/>.
- [243] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.
- [244] 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.
- [245] Francisco Velázquez, Kristian Lyngstøl, Tollef Fog Heen, and Jérôme Renard. *The Varnish Book for Varnish 4.0*. Varnish Software AS, March 2016.
- [246] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *USENIX ATC*, 2017.
- [247] Haitao Wang, Zhanhuai Li, Xiao Zhang, Xiaonan Zhao, Xingsheng Zhao, Weijun Li, and Song Jiang. OC-Cache: An Open-channel SSD Based Cache for Multi-Tenant Systems.

- In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–6, Orlando, FL, USA, November 2018. IEEE. ISBN 978-1-5386-6808-5. doi: 10.1109/PCCC.2018.8711079.
- [248] Jaylen Wang, Udit Gupta, and Akshitha Sriraman. Peeling Back the Carbon Curtain: Carbon Optimization Challenges in Cloud Computing. In *Workshop on Sustainable Computer Systems*, 2023.
 - [249] Jaylen Wang, Udit Gupta, and Akshitha Sriraman. Giving Old Servers New Life at Hyper-scale. In *Workshop on Hot Topics in System Infrastructure*, 2023.
 - [250] Jaylen Wang, Udit Gupta, and Akshitha Sriraman. Characterizing Datacenter Server Generations for Lifetime Extension and Carbon Reduction. In *Workshop on NetZero Carbon Computing*, 2023.
 - [251] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvine, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warriar, Chetan Bansal, Jonathan Stern, Ricardo Bianchini, and Akshitha Sriraman. Designing cloud servers for lower carbon. In *ISCA*, 2024.
 - [252] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in Log-Structured storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, Santa Clara, CA, February 2022. USENIX Association. ISBN 978-1-939133-26-7. URL <https://www.usenix.org/conference/fast22/presentation/wang>.
 - [253] Rui Wang, Christopher Conrad, and Sam Shah. Using set cover to optimize a large-scale low latency distributed graph. In *USENIX HotCloud*, 2013.
 - [254] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, High-Performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, 2006.
 - [255] Philipp Wiesner, Ilja Behnke, Dominik Scheinert, Kordian Gontarska, and Lauritz Thamsen. Let’s Wait Awhile: How Temporal Workload Shifting Can Reduce Carbon Emissions in the Cloud. In *International Middleware Conference*, 2021.
 - [256] Roger Wood, Mason Williams, Aleksandar Kavcic, and Jim Miles. The feasibility of magnetic recording at 10 terabits per square inch on conventional media. *IEEE Transactions on Magnetics*, 45(2):917–923, 2009. doi: 10.1109/TMAG.2008.2010676.
 - [257] 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.
 - [258] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, et al. Bluecache: A scalable distributed flash-based key-value store. *VLDB*, 10(4):301–312, 2016.
 - [259] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Trans. Storage*, 13(3), oct 2017. ISSN 1553-3077. doi: 10.1145/3121133. URL <https://doi.org/10.1145/3121133>.
 - [260] J. Yang and Feng-Bin Sun. A comprehensive review of hard-disk drive reliability. In *Annual Reliability and Maintainability Symposium. 1999 Proceedings (Cat. No.99CH36283)*, pages 403–409, 1999. doi: 10.1109/RAMS.1999.744151.

- [261] Juncheng Yang, Yao Yue, and Rashmi Vinayak. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *USENIX OSDI*, 2020.
- [262] 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.
- [263] 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.
- [264] 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. URL <https://www.usenix.org/conference/hotstorage19/presentation/yang>.
- [265] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhung Park, Jin yong Choi, Eeye 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.
- [266] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. Cache-Sack: Admission Optimization for Google Datacenter Flash Caches. page 17.
- [267] Yao Yue. Taming tail latency and achieving predictability. <https://twitter.github.io/pelikan/2020/benchmark-adq.html>, 2020.
- [268] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.
- [269] Yiyang 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.
- [270] Aviad Zuck, Donald Porter, and Dan Tsafir. Degrading data to save the planet. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS ’23, 2023.