Nabu: Unlocking Better Cache Performance at Lower Cost with Expiration Time-based Flash Caching

Theano Stavrinos

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Advisers: Wyatt Lloyd, Ethan Katz-Bassett (Columbia University)

May 2023

Abstract

Caches are crucial building blocks of web services. They keep data close to users and other services, reducing request latencies, expensive network traversals, and requests to resource-constrained backend servers. Today's web services need high-capacity, high-performance caches for their massive working set sizes and to meet stringent performance requirements. Flash-based SSDs meet this need by providing excellent performance and high capacity at low cost. However, caching on flash involves a fundamental tradeoff. On the one hand, caches aim for low miss ratios by keeping useful objects in the cache. On the other hand, caches must protect SSDs from write-induced wear-out, which increases when useful objects are copied forward during garbage collection. Flash caches are often forced to choose between good cache performance (i.e., low cache miss ratios) and acceptable device lifespans.

This dissertation describes Nabu, a flash caching framework for static content that unlocks new positions along the Pareto frontier of the miss ratio/device lifespan tradeoff, enabling more effective caching at lower cost than existing frameworks. At the foundation of Nabu's design are expiration times, which specify an object's earliest possible eviction time. In particular, Nabu's garbage collection procedure uses expiration times in a cost/benefit analysis to choose the best flash block to erase, balancing write amplification from copying forward useful objects and increased miss ratios from evictions. Nabu also uses a novel clustering algorithm to group objects together by their expiration times, increasing the likelihood of low write amplification during garbage collection. Our evaluation shows that, for a range of CDN traces, Nabu significantly improves the object miss ratios achievable at a given total write volume compared to the state-of-the-art flash caching framework. Nabu also achieves better byte miss ratios at lower write volumes at most points along the byte miss ratio/write volume Pareto frontier.

Acknowledgements

Thank you to my committee, Wyatt Lloyd, Ethan Katz-Bassett, Kai Li, Amit Levy, and Ravi Netravali; your input and feedback has greatly improved this work. Daniel Berger, thank you for collaborating with us on this project and helping ground it in the real world. Nick Kaashoek, thank you for wrangling with CacheLib on behalf of the project. Nicki Mahler, Tom, Tonka, and the rest of the folks who keep the CS department running, thank you for all your hard work doing so. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1564242, CNS-1835253, and CNS-1910390. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

It feels appropriate to start with thanking Ms. Harbin, my kindergarten teacher, who led me through writing my first book way back in the day and who cultivated my love of reading and writing early on. I've since had more great teachers and mentors than I can count, so I'll jump forward to CIND and give a special shout-out to Mike, Norbert, Diana, and Cheryl. I am here now because you gave me the freedom to try new things well beyond what I thought I was capable of. Dr. Miodrag Potkonjak, you guided me through two challenging years at UCLA, and gave me the opportunity to try out this research thing. I am so grateful to you, Jia Guo, and Teng Xu for your mentorship.

I was fortunate to do a bunch of wonderful internships during my graduate studies. Matt, Todd, Megan, Katie, Ted, Sean, Cody, Ben, Marrisa, Caroline, and the rest of the 3Scan team: thank you for inviting me into your world for two summers. I learned a lot about what it takes to build ridiculously complicated, amazing things in that old hair salon on Mission. Kate, I had such an awesome time during my Google internship, in no small part thanks to your support and kindness. Dr. Miroslav Zivkovic and the team at UvA, thank you for showing me what a vibrant and tight-knit research community looks like. Satadru, it was great interning with you at Facebook and writing the Tectonic paper—thank you for all the time you spent patiently explaining the nitty-gritty details of the system to me. Dushyanth, Nat, and Xingbo, thank you for immediately making me feel at home at MSR Cambridge—with all that quiet on the fifth floor we managed to get quite a lot done.

I've had the love and support of so many friends through all of this. Carmella, Dennis, Sarah L., Peter, Ian, J.Ho, Vince, Arpit, Josh, Sean, Ed, the way-back folks—seeing you here and there over the years has been a constant joyful reminder that there is a life outside this thing. Breakfast Club—Adam, Peter, Alix, and Seghel—CIND was a great place but you really elevated it to something special. Every time I make Spam musubi or eat soup dumplings I think of you all. Peter with the

boat, thank you for showing me how liberating it can be to never be told to be careful. Natia, I am pretty sure I would not have survived my Master's if it were not for you; thank you for your encouragement, conversation, and friendship all these years.

GSG Slack cycling crew—Joe, Jessica W., Lindsay, Arseniy, Nick Q., Connor, Jason, Max, David, Glen, many others—thank you for joining me on adventures and making the pandemic a little easier to bear. Special shout out to Joe for getting me up at the crack of dawn for Princeton cycles and Cambridge runs. 18 Cash Money Boulevard—Alex, Rob, Nick H., Mito, and of course, Kevin—thank you for all the chats and BBQs and baked goods and occasional movies, and for lavishing Nathan with pets and scritches even when it took her a little time to warm up to you (sorry, Alex!). Brad, Daniel V., Matt M., Robert M., and many folks already mentioned, thanks variously for the dinner parties, birthday parties, Sunday brunches, bonfires, and karaoke nights. Chris, we struggled through a monster of a project that, by some weird poetic turn, got accepted not a month before we defend. Thank you for being a great cohort-mate through all of this. Thanks also to the rest of SNS, especially Khiem and Haonan, Zhenyu, Jeff, Natalie, Yue, Sam, David, Andrew, Jianan, Jennifer, Anja, Nick, Nan, Shai, Chris B., Leon, Ryan, Ashwini, Dongsheng, Neil, Marcela, Amy, hopefully that's everyone?!—for the conversations, the distractions, the happy hours and coffee hours, the Systems lunches, the conference shenanigans, for being a home these last six years (+1 for the USC folks).

Nathan Scott Phillips, my little cartoon character, I'm so lucky you foster-failed into my life. Thank you for all the laughs and comfort you've selflessly given me.

Doris, you were one of the first people I met when I moved to the West Coast. I still have the blender you gave me. Thank you for watching over me. Adam, you were my crashpad in SF when things got weird and my crashpad in LA when I started on this path. I'll be thinking of you when I walk my walk; I so wish I could have seen you do this, too. Abuelito, all the hours we spent in the garden catching lizards and enjoying Miami's natural abundance are some of my happiest memories. Thank you for instilling in me a love of nature, which got me through many difficult days. TQM.

Kevin, you inspire me in so many ways to be a better version of myself, in work and in life. Thank you for encouraging me to be more curious, more methodical, more patient, and less judgemental, which among other things has made me a better researcher. I'm looking forward to many more years of discovery with you.

Wyatt and Ethan—me completing this PhD would have been impossible without your guidance. Wyatt, I have imprinted in my brain the image of you pointing to a piece of paper with the word "Why?" in enormous font on your office wall. I feel so fortunate to have had the opportunity to

talk through a whole bunch of fascinating problems with you. Ethan, because of you, I will never (knowingly) use the word “this” in academic writing without an accompanying noun. I am a much better writer today than when I started out, in no small part from the many hours you spent giving me feedback over the years. Thank you, also, for continually setting an example of how to create a stronger, more welcoming CS community. To you both—this wasn’t easy, but it was awesome. Thank you.

Finally, of course, my family. Mom, I am here now because of your love and unwillingness to accept anything less than the best for us. TQM. Dad, your constant encouragement and my memories of our trips to Greece kept me going when the work felt too difficult and never-ending. Mica and Panos, you’ve done such awesome things while I’ve been mashing a keyboard over here—I’ll never be as cool as you, but I at least try to live up to your example of being hard-working, kind, and community-oriented Stavrinoses. And Aleida, thanks for all the ways, big and small, you’ve supported me since I was a kid. Fam, without you this all would have been pointless. Thank you for everything.

To my family.

Contents

Abstract . . . iii

Acknowledgements . . . iv

1 Introduction **1**

2 Background **7**

2.1 Datacenter storage hierarchies . . . 7

2.2 Caching . . . 8

2.3 NAND flash-based SSDs . . . 9

2.4 Flash Caching Frameworks . . . 11

3 Design **13**

3.1 High-level Overview . . . 14

3.2 Expiration Times . . . 15

3.2.1 Scores are inadequate for flash caching. . . . 15

3.2.2 Earliest eviction times support good flash caching decisions. . . . 17

3.2.3 Computing expiration times . . . 17

3.3 Grouping Policy . . . 19

3.3.1 Handling expiration time drift . . . 20

3.3.2 Closing open containers . . . 21

3.4 Container Erasure . . . 21

3.5 Copy-Forward Policy . . . 23

4 Evaluation **24**

4.1 Implementations . . . 24

4.1.1 Nabu Implementation . . . 24

4.1.2 Baseline Implementation: RIPQ . . . 25

4.1.3 Common Parameters . . . 26
4.2 Configuration . . . 26
4.3 Workloads . . . 27
4.4 Nabu pushes out the miss ratio/write volume Pareto frontier for object misses. . . . 27
4.4.1 Object miss ratio . . . 28
4.4.2 Byte miss ratio . . . 30
4.4.3 Discussion . . . 31
4.5 Write volume is influenced by copy-forward write volume and misses . . . 32
4.6 Sensitivity Analysis . . . 33
4.6.1 Sensitivity to lifetime cap, `lcap` . . . 34
4.6.2 Sensitivity to score threshold, `thresh` . . . 35
4.6.3 Sensitivity to size penalty, `szmod` . . . 35
4.6.4 Sensitivity to ϕ . . . 36
4.6.5 Sensitivity to open container count, k . . . 37
4.6.6 Sensitivity to copy-forward filter, `cffilter` . . . 38
4.7 Takeaways . . . 39

5 Related Work **40**
5.1 Flash translation and abstraction layers . . . 40
5.1.1 Grouping data on flash . . . 41
5.1.2 Using cost/benefit analysis for container erasure . . . 42
5.2 Flash caching frameworks . . . 42
5.2.1 RIPQ . . . 42
5.2.2 Pannier . . . 44
5.2.3 Other flash caching frameworks . . . 44
5.3 Time-to-live and grouping-based caching . . . 45
5.3.1 TTL-based caching . . . 45
5.3.2 Grouping-based caching . . . 46

6 Future work **47**
6.1 Supporting object deletions and updates . . . 47
6.2 Incorporating more learning into flash caching . . . 48

7 Conclusion **50**

8 Appendix **52**

8.1 Grouping Policy Clustering Algorithm 52

8.2 Nabu and RIPQ performance relative to FIFO 55

8.3 Nabu and RIPQ performance for 2TiB cache 55

List of Figures

1.1 An example web service architecture. 2

3.1 Nabu system diagram. 14
3.2 A schematic of a cluster. 19

4.1 Nabu parameters and where they are used. 25
4.2 Properties of traces used in evaluation. 27
4.3 Absolute values used to compute percent differences from RIPQ minimums in Figures 4.4 and 4.5. 28
4.4 Object miss ratio vs write volume for Nabu and RIPQ for a 1TiB cache. 29
4.5 Byte miss ratio vs write volume Pareto frontier for Nabu and RIPQ for a 1TiB cache. 30
4.6 Breakdown of bytes written by type for a selection of configurations in Nabu and RIPQ. 32
4.7 Nabu's sensitivity to the lifetime cap. 34
4.8 Nabu's sensitivity to the size penalty. 36
4.9 Nabu's sensitivity to the ϕ parameter. 37
4.10 Nabu's sensitivity to the open container count. 38
4.11 Nabu's sensitivity to the copy-forward filter. 39

8.1 Function to compute an object's distance to a cluster/container. 53
8.2 Pseudocode for the grouping policy's clustering algorithm. 54
8.3 Absolute values used to compute percent differences from FIFO minimums in Figures 8.4 and 8.5. 55
8.4 Object miss ratio vs write volume for Nabu, RIPQ, and FIFO for a 1TiB cache. . . 56
8.5 Byte miss ratio vs write volume Pareto frontier for Nabu, RIPQ, and FIFO for a 1TiB cache 57

8.6 Absolute values used to compute percent differences from RIPQ minimums in Figures 8.7 and 8.8. 57
8.7 Object miss ratio vs write volume for Nabu and RIPQ for a 2TiB cache. 58
8.8 Byte miss ratio vs write volume Pareto frontier for Nabu and RIPQ for a 2TiB cache. 59

Chapter 1

Introduction

Modern web services manage petabytes to exabytes of data, storing, processing, and serving that data to millions of users. Services face high peak request rates as well as high performance expectations from users. Services must balance the resource demands with the resource costs of supporting these workloads. To do so, they rely on complex, layered service architectures that exploit the differing strengths and price points of a variety of storage technologies.

At a high level, this architecture commonly consists of three *tiers* within a datacenter: a storage tier, an application logic tier, and a web server tier (Figure 1.1). The storage tier typically consists of fleets of hard disk drives (HDDs), whose low cost, durability, and high capacity make it possible to store the huge volumes of data managed by the service. In the next tier are servers that implement the service's application logic. These servers use fast solid-state disks (SSDs) and faster memory (DRAM) to support low-latency, high-throughput data processing. Finally, the upper tier consists of web servers that handle client requests to the service. Web servers similarly rely on SSDs and DRAM, since responding to client requests should be done with low latency and high throughput.

These tiers are supported by *caches*. Caches are critical for the performance and efficiency of web services [5]. For a given *backing store*, e.g., a storage tier database, a cache stores its most popular data objects and serves them more cheaply or efficiently than the backing store itself could serve them. Caching can significantly improve user-visible performance and reduce the resources needed to run a service. For example, Facebook reports order-of-magnitude decreases in user request latencies through caching, and claims that a single cache server "can replace tens of backend database servers" [5].

A cache's ability to make these performance and resource improvements depends largely on

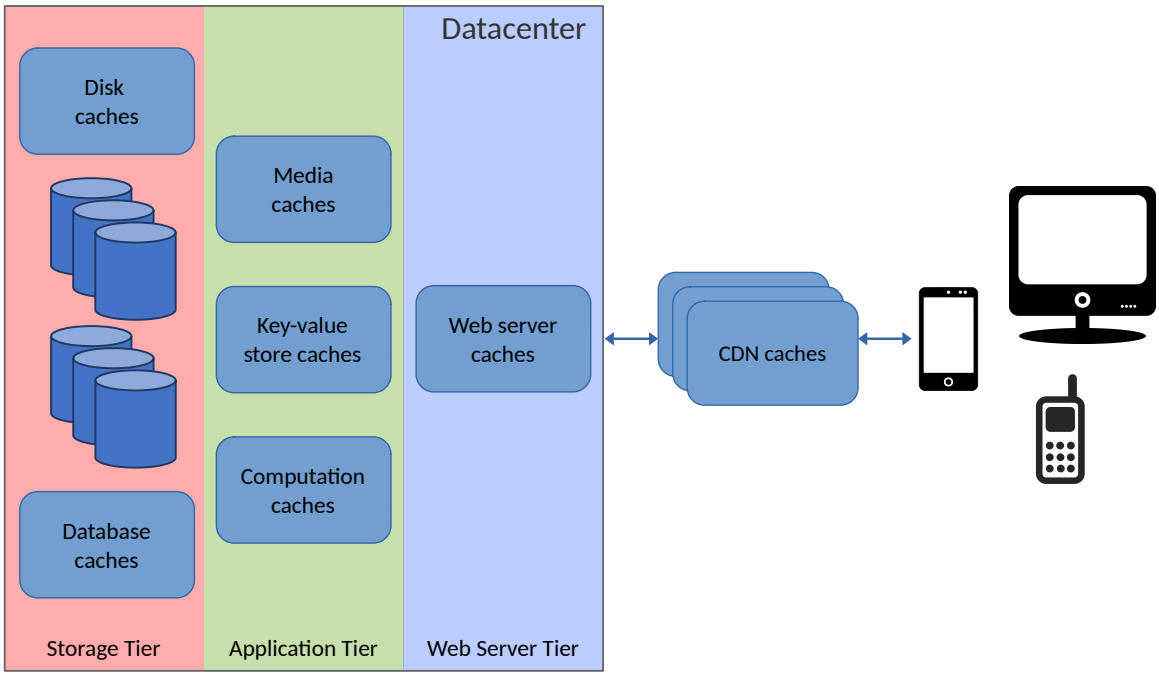


Figure 1.1: An example web service architecture with categories of caches supporting the service's efficiency and performance within the service's tiers [5, 76]. Disk and database caches support low-latency access to data in the storage tier. They also shield hard disks from high load. Media and key-value stores in the application tier store user data that may be processed by internal services, e.g., machine learning-based services, or served to users. Computation caches store intermediate results from processing. Web server tier caches may store very popular media items and user connection state. Content delivery network (CDN) caches store particularly popular or low-latency data outside the datacenter, close to densely-populated areas.

how many requests it serves on behalf of its backing store. However, a cache is much smaller than its backing store, since serving requests cheaply or efficiently is typically a tradeoff for higher cost per unit of capacity than its backing store. For instance, the cache's hardware may be more expensive, as with storage server disk block caches on SSD or DRAM, both of which are significantly more expensive per gigabyte than HDDs. The cache may also have higher operating expenses; for instance, content delivery networks (CDNs) are server deployments near densely populated areas, where electricity and physical space are more expensive than the remote locations where datacenters are typically located (Figure 1.1).

Given the cache's size constraints, a primary challenge in caching is to choose the right objects to store in the cache to serve the most requests. A cache's effectiveness is typically reported as its *miss ratio* (i.e., objects or bytes not served by the cache versus objects or bytes requested). For a given cache size and workload, the miss ratio depends on the *caching algorithm* used to determine the cache's contents. Caching algorithms leverage object access patterns in workloads to decide which objects to keep in the cache and which to evict.

Caching algorithms are continually improving to support lower byte or object miss ratios in a wide variety of settings. Unfortunately, even the best caching algorithms will achieve poor miss ratios when the cache is too small for a workload's *working set size*, or the size of the set of objects which are in use at a given time. The working set sizes of modern service workloads can be enormous; the working set size at individual servers can exceed the cost-effective size of DRAM by an order of magnitude [5, 45]. At the same time, caching on HDDs often defeats the purpose of a cache in the first place, since most workloads are backed by HDDs anyway.

The benefit of increasing cache capacity and the prohibitive per-unit-capacity cost of DRAM have led to increased interest in caching on SSDs. While SSD read/write latencies are around 100 μs, three orders of magnitude higher than DRAM [26], SSDs are around an order of magnitude less expensive per gigabyte than DRAM. This additional latency over DRAM is often a fraction of the other latencies involved in serving a request. For instance, one-way latency within a datacenter is a few hundred microseconds, while wide-area network latencies are around a hundred milliseconds. SSD read/write latencies are also significantly lower than HDD latencies, which are around ten milliseconds. SSDs thus have acceptable performance as a caching medium within web services.

Unfortunately, caching workloads exacerbate SSDs' main limitation, which is their limited endurance. SSDs are made up of NAND flash chips. Flash accumulates wear with each write, eventually failing when it is so worn it cannot reliably hold data. For instance, the flash underpinning datacenter-grade SSDs is typically rated for a maximum of a few thousand writes per flash chip [30, 31, 57]. However, caches have high write rates as they continually insert new objects and evict old objects to attempt to reduce miss ratios. There is thus a tension between reducing miss ratios and preserving SSD lifespans. The tradeoff between cache effectiveness and acceptable SSD lifespan is the fundamental challenge of caching on flash.

Flash's lifespan is affected by the total *write volume* of a workload relative to the device's endurance rating and capacity. A workload's write volume is made up of two types of writes: writes of new data, and re-writes of existing data as part of the process of clearing out other data deleted by the host, called *write amplification*.

Write amplification in an SSD is the outcome of two of flash's physical constraints. First, the unit of flash erasure, an *erase block*, is much larger than the unit of flash read/write, a *page*. An erase block must be erased as a whole to reclaim capacity from *invalid* pages in the block, or pages deleted by the host that have not yet been erased from the device.

The second constraint is that flash must be written sequentially within an erase block. Data cannot be overwritten; updates to existing data are done out of place. These two physical constraints

can result in an erase block with a mix of valid and invalid data. When the block is erased, the valid data must be *copied forward* to a new region of flash. Write amplification from copy-forward chips away at device lifespan by increasing the workload's total write volume.

Reducing write amplification has been extensively studied for generic storage systems such as file systems and key-value stores. However, techniques for reducing write amplification for storage systems leave performance on the table in the flash caching setting [16, 58, 66]. Unlike in storage systems, applications using caches do not require the cache to retain all data. The cache can evict cached objects instead of copying them forward when reclaiming capacity. However, evicting to avoid write amplification comes with a cost: valuable objects may be evicted, potentially causing high miss ratios in exchange for low write amplification. Furthermore, cache misses cause objects to be inserted into the cache, increasing write volume. Storage systems are not designed to navigate the tradeoff between write volume and miss ratios.

Special-purpose *flash caching frameworks* are used in practice to handle caching on flash [5, 23, 29, 32, 40, 41, 43, 52, 58, 66, 81]. Flash caching frameworks group cached objects into *containers* typically sized to cover a full erase block. Frameworks explicitly delete a full container at a time from the SSD. Deleting a full container means the device does not handle reclaiming capacity from individual objects, since there are no valid pages in an erase block to copy forward. The framework instead takes care of copying forward objects that should remain cached before deleting the container.

Many existing flash caching frameworks limit writes to flash by filtering insertions to flash, deduplicating objects, compressing objects, or otherwise reducing the volume of new writes to flash [5, 23, 29, 32, 40, 43, 52, 58, 81]. They then use simple caching algorithms on the SSD. This dissertation is concerned with fully leveraging SSD capacity by implementing more sophisticated algorithms on the SSD itself while minimizing the write volume generated by the flash caching workload. Other existing flash caching frameworks aim to achieve this goal, but they leave performance on the table: how they encode object value limits how effectively they can make cache management decisions.

For generic storage systems, prior work observes that grouping pages by invalidation time can keep write amplification low, if invalidation time can be predicted or known with an oracle [15, 27]. In a flash cache, invalidation time is analogous to the object's eviction time for a given caching algorithm [16]. Grouping objects on flash by their eviction time would reduce write amplification by enabling the cache to erase entire containers with little or no copy-forward. It would also simplify the task of choosing a container to erase by making it easy to find containers with many objects ready to be evicted at a given time.

The challenge is that for non-trivial caching algorithms, an object's eviction time is difficult to predict. Algorithms typically choose to keep objects in the cache longer when they receive a hit, but predicting when objects will be accessed is itself difficult to predict.

We observe that in the flash caching setting it is also useful to know an object's *earliest possible eviction time*, i.e., the time an object would get evicted if it did not get hit again. The earliest possible eviction time provides a good foundation for making caching decisions in the flash setting. Objects that get grouped together by earliest possible eviction time can be evicted together if they do not receive more hits, contributing to low write amplification. The earliest possible eviction time supports good container erasure and copy-forward decisions. It clearly expresses which objects can be evicted at a given time, and hence which containers are good candidates for erasure. Finally, the framework can confidently handle containers with objects with similar earliest possible eviction times: if a container's objects expire far in the future, the framework need not consider that container for erasure until that time. On the other hand, if most of the container's objects are past their earliest possible eviction time and have not been accessed again, the framework can erase most of the container's objects with confidence that it will likely not negatively impact miss ratios.

This dissertation demonstrates that *it is possible to predict earliest possible eviction times for a useful class of caching algorithms.* Furthermore, *earliest possible eviction times can provide a foundation for cache management decisions that significantly reduce total write volume and miss ratios over state-of-the-art flash caching frameworks.*

We demonstrate these points through the design, implementation, and evaluation of Nabu[1], a flash caching framework for static content. Nabu's key idea is to explicitly assess the write volume and potential miss ratio impact of each cache management decision it makes. Nabu achieves this by introducing *expiration times*, which specify each object's earliest possible eviction time. Expiration times express both an object's *value* with respect to the potential benefit of keeping the object in the cache, and its *evictability*, i.e., whether the object should be evicted at a given time.

Importantly, expiration times supply this information for each object in *absolute* terms, irrespective of the rest of the cache's contents. Existing flash caching systems express object values in relative terms; the best objects to evict are the worst objects in the cache [41, 66]. Relative values make it difficult to evaluate the impact of erasing a container *other* than the one containing the worst objects, even if other choices would contribute significantly less to write amplification. By instead expressing objects' values and evictabilities in absolute terms, expiration times grant Nabu freedom to erase any container in the cache. Nabu can thus erase the container that best balances the goals

[1]The system is named after Nabu, the Mesopotamian god who inscribed people's fates on clay tablets [49].

of evicting low-value objects and keeping write amplification low. To achieve this, Nabu implements a cost/benefit approach to choosing a container to erase, based on expiration times. Nabu's design also includes a novel clustering algorithm for grouping objects into containers by their expiration times. The algorithm strives to create object clusters with small expiration time ranges, to increase the likelihood that all objects in the container can be evicted together.

Our evaluation on five CDN traces shows that Nabu pushes out the Pareto frontier of the object miss ratio and write volume tradeoff. For the same write volume, Nabu can achieve up to a 20% reduction in object miss ratio compared to RIPQ, the state of the art flash caching framework [66]. This translates to significant savings for backend storage systems underlying the cache. Nabu can achieve the same object miss ratio as RIPQ with up to 30% fewer flash bytes written, which translates directly to longer flash device lifetimes (or, equivalently, the ability to use cheaper devices for the same lifetime), without sacrificing miss ratios. Nabu achieves comparable performance on byte miss ratio to RIPQ.

The contributions of this dissertation include:

- Identifying earliest possible eviction times as a useful basis for making flash caching decisions.
- A technique for predicting an object's earliest possible eviction time for an important class of caching algorithms.
- The design and implementation of Nabu, a flash caching framework that demonstrates how earliest possible eviction times can support high-performance flash caching that preserves device lifespans.
- An evaluation of Nabu in simulation using five CDN traces, showing that Nabu pushes out the Pareto frontier of object miss ratio and write volume and achieves a comparable byte miss ratio/write volume tradeoff to the state-of-the-art flash caching framework.

Chapter 2

Background

2.1 Datacenter storage hierarchies

Web services rely on extensive datacenter storage hierarchies to store and serve data. The storage hierarchy underpinning a service should meet the capacity and performance needs of the service at a reasonable cost. Services achieve this by layering persistent storage with caches, strategically deploying hard disks, SSDs, and memory to achieve capacity, performance, and cost goals.

The foundation of most services is a *backing store*. This may be a database, filesystem, or blob store [28, 50, 71]. Backing stores are typically served from hard disk drives (HDDs), which are inexpensive but are severely constrained by the read/write rates they can serve per terabyte of capacity [50]. Hard disks also have relatively high read and write latencies, typically in the tens of milliseconds. A backing store typically includes disk caches on individual nodes, as well as a memory or SSD-based caches for storing metadata and/or popular blocks [5, 50, 61]. These caches reduce read/write load to the HDDs and avoid high request latencies for popular blocks.

The data stored by the backing store is processed by datacenter-internal services to enable rich user-facing services. For instance, photos get resized for viewing in different settings, and social relationships among users are processed to support content recommendations [28, 68]. Internal services also generate logs, counters (e.g., for rate limiters and load balancing), tracking data, and metadata (e.g., indexes) that may need additional processing. Caches are commonly used to store results of such processing to avoid expensive recomputations, or to enable low-latency access to data underpinning critical internal services [5, 28, 50, 71, 76].

Web services typically also deploy caches to store popular data as close as possible to users. In

particular, content delivery networks (CDNs) are located near high concentrations of users. CDNs store content on behalf an *origin* datacenter, i.e., a datacenter where the data is durably stored by the service. In addition to serving data to users with low latency, CDNs help avoid expensive network traversals to retrieve requested data from the origin datacenter [5, 28, 62, 71].

2.2 Caching

The degree to which a cache meets its goals—e.g., reducing load to backend storage hard disks, serving content to users with low latency, keeping network costs low—depends on the cache's ability to store and serve requested content. This ability is typically expressed as the cache's *object* or *byte miss ratio*. The object miss ratio is the count of data objects requested that the cache could not serve, versus the total count of objects requested from the cache. Byte miss ratio is defined similarly. Each of these two types of miss ratio is important for different parts of a service's infrastructure. Object miss ratios tend to matter for backing stores, since each object request typically translates to a hard disk IO, and hard disk IO per second is severely constrained [5, 50]. Object miss ratios also affect user-visible performance, since the latency of retrieving a missed object from a backing store is the result of per-IO delays and does not typically depend on object size. Byte miss ratios tend to matter when bandwidth or network is constrained or costly, e.g., for CDN misses [5, 62].

Miss ratio is a useful way to measure a cache's effectiveness because it gives a clear picture of how the cache impacts downstream users, systems, and services. For instance, if CDN A achieves an object miss ratio of 10% and CDN B achieves an object miss ratio of 20%, then CDN A reduces requests to the origin datacenter by half compared to CDN B.

Two factors have an especially big impact on cache effectiveness. One is the caching algorithm used to manage the cache's contents. *Admission* algorithms determine which objects are inserted into the cache [8, 22, 63]. By restricting what gets inserted into the cache, admission algorithms reduce objects' competition for cache space, allowing useful objects to stay in the cache longer. *Eviction* algorithms determine which objects should be evicted to make room for new objects [3, 4, 11, 17, 34, 46, 54, 62, 69]. No one algorithm is best for every workload or system. Algorithms differ in their metadata requirements, CPU overhead, object indexing overhead, and effectiveness in different conditions, hence the proliferation of algorithms over time.

The other factor affecting cache performance is the size of the cache [5]. Naturally, a larger cache can hold more objects at the same time than a smaller cache, increasing the likelihood that a requested object can be found in the cache. A larger cache can thus reduce cache miss ratios without

changing anything about the caching algorithm. However, storage hardware that achieves the low latency and high throughput required for caching in most settings is expensive per unit of capacity. DRAM, for instance, is on the order of $4 per GB [1]. HDDs, on the other hand, are on the order of $0.06 per GB [79]. Flash costs on the order of $0.10-1 per GB [1, 79]. Though flash sits at an attractive price point compared to DRAM, it has some critical limitations when it comes to caches and write-heavy workloads generally. We next describe flash and those limitations in more depth.

2.3 NAND flash-based SSDs

Flash's excellent performance and low cost per unit of capacity have made it extremely popular in data center storage hierarchies. However, flash has three limitations: its erase size is far larger than the write size, it can only be written sequentially, and it has limited endurance. Together, these three limitations can dramatically shorten flash device lifetimes under certain workloads, including ones common for traditional caching.

Flash's first limitation is that the unit of erasure is much larger than the IO size. *Erase blocks* are the minimum unit of erasure. An erase block is typically tens to hundreds of megabytes in size. Each erase block is composed of a group of flash pages, typically 4-16KiB in size. Pages are the minimum unit of IO. Garbage-collecting *invalid* pages, i.e., pages with content the application wishes to erase, requires erasing an entire erase block. Garbage collection causes *write amplification* when valid pages coexist with invalid pages in an erase block, since any valid pages in the block must be *copied forward* to a new erase block.

Flash's second limitation is that it must be written sequentially within an erase block. Once a flash page is written, it cannot be overwritten until its containing block is fully erased. That is, flash can only be updated *out of place*: updating a page involves writing a new page on flash, then invalidating the old page. These out-of-place updates gradually tie up more of the SSD's free capacity in invalid pages, which must be garbage-collected.

Conventional SSDs hide out-of-place updates from the host behind a *flash translation layer* (FTL). The host reads, writes, and erases data using logical page addresses. The FTL gives the appearance of a block interface by mapping logical pages to valid physical pages on flash. The FTL garbage-collects invalid pages by selecting a block to erase, copying-forward and re-mapping valid pages in the block, and finally erasing the block to make it writeable once more. A more recent SSD interface, Zoned Namespaces (ZNS), instead exposes append-only zones to the host [10]. Zones map to one or more erase blocks and are erased in full. The host becomes responsible for garbage-

collecting unneeded data, but gains control over grouping data on the device and over when and how garbage collection is carried out.

Regardless of the interface, reclaiming capacity from invalid data can cause significant write amplification when valid data gets copied forward. Random-write workloads in particular—such as those generated by caches—can generate excessive write amplification. As random logical pages get updated, their old physical flash pages get invalidated, resulting in erase blocks with a mix of valid and invalid pages. When those blocks get erased, the valid pages must be copied forward, causing write amplification.

Initial data writes and write amplification together aggravate flash's third limitation, its low *endurance*: as flash is written and erased, it accumulates wear until it eventually fails. An SSD can sustain a certain write rate depending on how long it is expected to be in operation, the device's techniques for managing wear (e.g., through overprovisioning flash to replace bad erase blocks, or the FTL's wear-leveling of erase blocks), and the underlying flash's endurance. Flash which stores single bits per flash cell has the highest endurance, while the endurance of flash storing more bits per cell decreases with increasing density. For instance, single-level cell (SLC) flash can endure hundreds of thousands of write/erase cycles, compared to quad-level cell (QLC) flash's one thousand write/erase cycles [12, 60, 65].

The endurance of an SSD is often expressed as supported drive writes per day (DWPD) over a given lifespan (e.g., 10 DWPD for 5 years). The bound on write rate can be computed with the following equation:

$$\text{capacity} \times \text{DWPD} \div \text{seconds/day} \tag{2.1}$$

So, a 1TiB SLC SSD rated for 10 DWPD over 5 years can sustain an average write rate of 121 MiB/s. On the other hand, a 1TiB QLC SSD rated for 0.1 DWPD can only sustain an average write rate of 1.2 MiB/s for a five-year lifespan. Denser flash storing multiple bits per cell is increasingly used in datacenters because of its low cost per unit of capacity. Unfortunately, workloads moved onto such high-density flash are often write-rate bound because they are forced to stay within the lower endurance limit of the flash.

2.4 Flash Caching Frameworks

Caching workloads can severely stress the limited endurance of flash. New objects are inserted frequently, and write amplification can be high in naïve cache implementations [23, 66]. However, within this challenge is an opportunity: the write amplification generated in caches is optional, unlike in traditional storage applications. A flash cache can completely avoid write amplification by evicting every object in a block being erased. The catch is that evicting the wrong objects can increase miss ratios. Higher miss ratios have two consequences. First, higher miss ratios may result in higher costs throughout a service as requests must be served from slower or more expensive storage. Second, they may result in *more* flash writes as missed objects get reinserted into the cache and cancel out the benefit of lower write amplification. Evicting and reinserting objects results is another source of write volume called *miss-driven write volume* (reinsertion write volume) [23]. The total endurance impact of a caching workload is the sum of new object insertions, write amplification, and reinsertion write volume.

Flash caching frameworks mediate between applications and SSDs to manage the complex tradeoff between miss ratios and endurance: they attempt to balance low write amplification from avoiding copy-forwards with low miss ratios and low reinsertion write volume from keeping valuable objects in the cache [5, 5, 23, 23, 29, 32, 40, 41, 43, 52, 58, 66, 81]. Flash caching frameworks typically group objects into *containers* to have full control of write amplification and of how objects are colocated in erase blocks. A container maps to one or more SSD erase blocks. Containers may be *open*, i.e., accepting new objects, or *closed*, i.e., no longer writeable. Resource limits typically determine how many containers can be open at a time. For instance, on conventional SSDs, available DRAM limits how many containers can be open at a time, since each open container is buffered in DRAM before it is flushed to flash. Buffering containers gives frameworks control over how objects are physically grouped together on a conventional device. Containers also allow frameworks to bypass the FTL's garbage collection to control write amplification.

On ZNS SSDs, the zoned interface already physically groups data together on the device if it is written to the same zone, so open containers do not need to be buffered in DRAM first. ZNS SSDs have a maximum open zone count driven by the device's on-board battery-backed RAM [10]. If each container maps to a zone, then the maximum number of open containers is limited by the device's open zone count.

All flash caching frameworks implement three cache management policies to control the miss ratio/endurance tradeoff:

- The **object grouping policy** governs how objects are grouped into open containers.
- The **container erasure policy** governs which container is erased when capacity needs to be reclaimed.
- The **copy-forward policy** governs which objects are kept, i.e., copied forward, from a container being erased.

The interplay of these policies determines the high-level caching algorithm, i.e., how and when objects are evicted, as well as the write volume of the framework when caching a particular workload. For instance, a flash caching framework implementing the FIFO caching algorithm groups objects into containers based on insertion time. Its container erasure policy erases the container with the oldest objects by insertion time. Its copy-forward policy is to copy nothing forward.

A flash caching framework implementing FIFO strongly prioritizes low write amplification. However, miss ratios may be high as a result. reinsertion write volume, and hence total write volume, may also be high. For this reason, flash caching frameworks including Nabu are designed to implement more sophisticated caching algorithms [41, 66] (we show a comparison of Nabu, RIPQ [66], and FIFO in Section 8.2).

Chapter 3

Design

Nabu is a flash caching framework for static content designed to push out the Pareto frontier of miss ratios and write volume. In particular, Nabu provides low miss ratios while keeping copy-forward write volume and reinsertion write volume low (Section 2.4), where copy-forward write volume comes from objects copied forward from erased blocks and reinsertion write volume comes from objects evicted, then reinserted to the cache after misses.

The key insight behind Nabu's design is that assigning expiration times to cached objects provides a useful foundation for making good flash caching decisions. Expiration times are predictions of the earliest time an object might be evicted from the cache under a caching algorithm. Expiration times express an object's *anticipated value*, or the degree to which it is useful to keep the object in the cache, since an expiration time predicts how much longer the algorithm would keep the object based on its value. Expiration times also clearly express an object's *evictability*, or whether the object can be evicted at a given time.

Nabu leverages expiration times to design policies that quantify the write amplification and/or potential miss ratio impact of each choice and choose the best action accordingly. Since expiration times express an object's anticipated value and evictability in *absolute* terms, without requiring expensive comparisons among objects, Nabu can quantify and weigh these impacts. Nabu also uses expiration times to support performance tuning. By adjusting a small number of parameters controlling how expiration times are computed and used in copy-forward decisions, system designers can prioritize low miss ratios or low write amplification (Section 4.1.1).

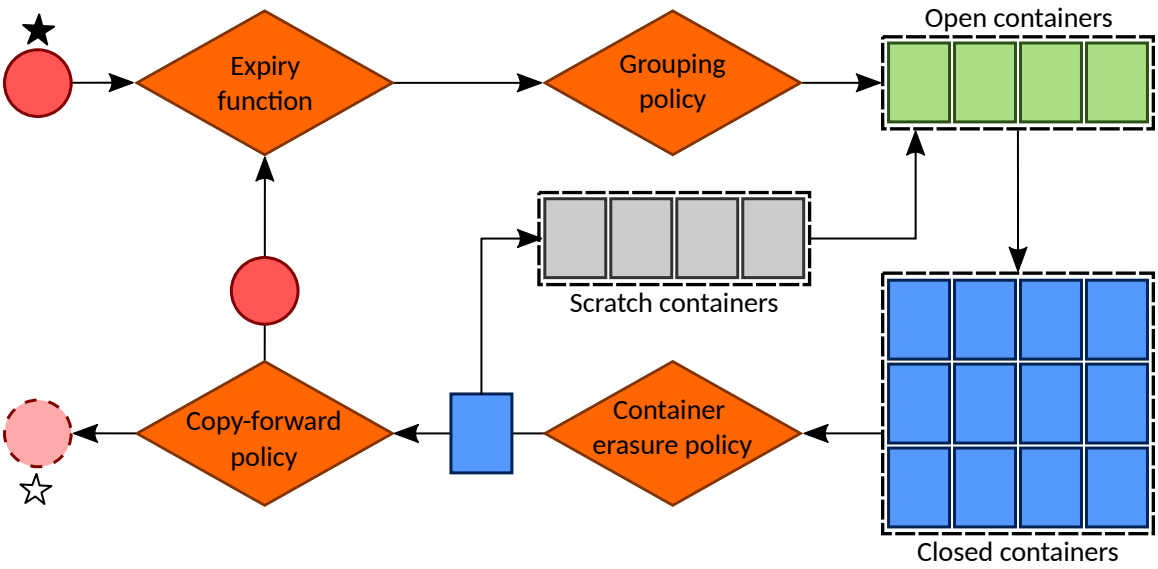


Figure 3.1: Nabu system diagram. Red circles are objects; the solid black star in the upper left indicates object insertion, while the empty star in the lower left indicates eviction. Orange diamonds indicate the expiry function and Nabu's cache management policies. Rectangles are containers; open containers are green, closed containers are blue, and scratch containers are grey. On insertion (solid black star), an object is assigned an expiration time (Section 3.2). The object is passed to the grouping policy, which places it into an open container based on its assigned expiration time (Section 3.3). When an open container becomes full, it is closed. A new container is initialized from scratch space containers (Section 3.3). If closing the container causes available scratch space to fall below a threshold, the container erasure policy selects a container to erase (Section 3.4). The copy-forward policy determines if any objects should be copied forward; those objects are passed to the expiry function to be processed (Section 3.5). Otherwise they are evicted (empty star). The container is erased and becomes scratch space.

3.1 High-level Overview

Figure 3.1 shows Nabu's high-level functionality. When an object enters Nabu, Nabu computes its expiration time, the earliest time the object will be evicted. The object's expiration time is recomputed if the object gets a hit. Expiration times are computed using a caching algorithm that evicts objects in a predictable way in the absence of hits (Section 3.2). Expiration times are the foundation for Nabu's three flash caching policies.

Nabu's grouping policy groups *incoming* objects (i.e., new and copied-forward objects) into open containers (Section 3.3). The grouping policy's goal is to reduce copy-forward write volume by grouping together objects that are likely to be evicted at the same time. Nabu's grouping algorithm groups incoming objects by their expiration times.

When the cache is full, Nabu's container erasure policy frees capacity for new objects (Section 3.4). The container erasure policy's goal is to find a closed container to erase that will best balance low copy-forward write volume and the miss ratio benefit of evicting objects from the cache

that are unlikely to get hit soon. By aggregating up-to-date object-level information based on expiration times that quantifies this tradeoff, Nabu finds the container that best balances the benefit of erasing low-quality bytes with the cost of copying unexpired objects forward.

Once a container is chosen to erase, Nabu's copy-forward policy copies forward any unexpired objects into new containers, and evicts objects whose expiration times have passed (Section 3.5).

3.2 Expiration Times

Nabu's three policies are underpinned by expiration times, which express each object's earliest possible eviction time. In this section we detail why Nabu uses expiration times instead of scores used in classical caching algorithms and how expiration times are assigned to objects.

3.2.1 Scores are inadequate for flash caching.

Useful caching algorithms such as Hyperbolic and GDSF assign *scores* to objects and use them as a basis for eviction decisions [11, 17]. Scores are values assigned to objects that are used to rank objects relative to one another. A score does not encode the absolute value of an object in a way that enables reasoning about whether to evict an object in isolation, i.e., without comparisons to other objects' scores. Scores fall short in the flash caching setting when considering the decisions that cache management policies need to make.

To make this discussion more concrete, we describe the Hyperbolic algorithm. Hyperbolic assigns a score to an object o with the following *scoring function*:

$$\frac{o.n}{t_{now} - o.t_{insertion} \times o.s} \tag{3.1}$$

where $o.n$ is the object's hit count, $t_{now} - o.t_{insertion}$ its insertion time, and $o.s$ is the object's size. Hyperbolic is a *volatile* score: the score changes as the object ages. Hyperbolic selects a sample of cached objects, ranks them, and evicts the worst of the sample. Hyperbolic's score decays over time so that older objects tend to rank lower than newer objects when sampled.

Scores fall short because they cannot convey the *evictability* of an object. That is, one object's score on its own cannot be used to decide whether to evict an object; the object must be compared to others. This limitation is problematic in a couple of ways. First, it constrains the order in which containers can be erased. When a container needs to be erased, scores offer a way to identify the lowest-value object in the cache, and hence identify a candidate container for erasure. However, the

container with the cache's worst object may contain other objects that should be copied forward. From a write amplification perspective, the container with the worst object may be a bad erasure candidate [80]. However, because there are only "better" or "worse" objects relative to one another, it is not clear how to assess the potential effect on miss ratio of evicting objects that are not the *worst* object in the cache. Scores are thus not a good basis for making container erasure decisions.

Second, because scores do not express evictability, they impact the cache's ability to make good copy-forward decisions. The copy-forward policy is a means of making up for bad object groupings. This is necessary because the grouping policy is best-effort: it cannot always group similar objects together. Flash caches have a limited number of open containers available at a given time, making it difficult to cleanly separate incoming objects when their values vary widely [15]. Furthermore, once objects have been grouped into a container, their values may evolve differently as some objects are accessed while others are not. As a result, a container may end up with dissimilar objects.

A consequence of best-effort grouping is that a container with n objects, one of which is the cache's lowest-scoring object, does not necessarily contain the next $n-1$ lowest-scoring objects. Using scores to identify which of those $n-1$ objects should be evicted is not straightforward, since scores are only meaningful relative to the scores of other objects in the cache. Many flash caching frameworks simply do not copy objects forward if they were not accessed since insertion [5, 41, 66]. Unfortunately, this makes it difficult to keep valuable objects in the cache in the face of potentially poor grouping decisions.

A strawman technique to make a score meaningful on its own is to impose a threshold that separates high- and low-value objects. For instance, for volatile scores like Hyperbolic, objects can be repeatedly evaluated. When an object's score falls below the threshold, it is no longer valuable and can be evicted. For static scores like GDSF, which do not change unless an object is accessed, the cache can maintain a ranking of objects and use the nth percentile score as the threshold. Unfortunately, the cost of this technique is likely too high to be practical given that for many workloads, a high-capacity cache may contain tens to hundreds of millions of objects [5, 66]. If using volatile scores, those scores need to be recomputed often so they are up-to-date when evaluating containers for erasure and when making copy-forward decisions. If using static scores, objects need to be ranked, likely in a priority queue, which has $O(\log n)$ time complexity for insertions, updates (on hits), and deletions (on evictions). It would be interesting to evaluate efficient implementations of cost-based algorithms, such as GD-Wheel [42], or approximations of priority queues for inexpensively ranking and thresholding objects.

3.2.2 Earliest eviction times support good flash caching decisions.

Knowing at insertion time when an object will be evicted by a caching algorithm would address the issues with scores in the flash caching context [16, 27]. Grouping objects by their eviction time would simplify deciding which container to erase and would enable better copy-forward decisions. However, no accurate prediction techniques are known for state-of-the-art caching algorithms. Eviction times are hard to predict at insertion time because algorithms dynamically decide to retain objects for longer based on access patterns, which themselves are hard to predict.

Although we cannot predict eviction times, we observe that, for a useful class of caching algorithms, we can predict the *earliest* time the algorithm would evict the object. In such algorithms, an object's score decays at a predictable rate over time until the object is evicted, unless the object gets additional hits. For instance, Hyperbolic initially assigns objects a high score that decays with the time since the object's insertion, with the decay rate determined by the object's properties such as its size and access count [11]. The function used to compute an object's score can be combined with a *score threshold* to yield a prediction of the object's earliest possible eviction time.

The earliest possible eviction time of an object is useful in the flash caching setting. First, it is a good basis for grouping decisions. Many objects will not be hit after insertion, and all objects will not be hit after their last access before eviction. The earliest possible eviction times thus are accurate eviction time predictions for many objects. Second, earliest possible eviction times are useful for the container erasure policy. The earliest possible eviction time conveys an object's usefulness in *absolute* terms, independent of the rest of the cache's contents. Earliest eviction times make it easy to examine a container's objects to determine both the write amplification impact and the potential miss ratio impact of erasing that container. This in turn makes it easy to compare containers to find the one to erase that will best balance the goals of keeping miss ratios low and keeping copy-forward write volume low. Third, at copy-forward time, earliest possible eviction times help make up for best-effort grouping decisions. An object's earliest possible eviction time clearly indicates whether it should be evicted or not, so the framework does not need to guess as to whether to evict an object in a container being erased.

3.2.3 Computing expiration times

Nabu computes an object's earliest possible eviction time, or *expiration time*, with an *expiry function*. Expiration times are expressed in terms of logical timesteps (i.e., number of requests). The intuition behind an expiration time is that it captures the time at which an object's volatile score will fall

below a threshold, at which time they are low-value enough to be evicted.

Our Nabu implementation comes with an expiry function based loosely on the Hyperbolic scoring function [11]. Whereas Hyperbolic penalizes objects with the time since the object's insertion, Nabu's expiry function uses the time since the object's last access. Using the time since insertion helps Hyperbolic avoid evicting new objects which are sampled for eviction by assigning them an initially high score [11]. Nabu does not evaluate objects for eviction in this manner; it is instead interested in keeping objects which have been recently hit, so penalizing objects which were accessed farther in the past is more appropriate.

Nabu's performance depends on the score threshold $\texttt{thresh} > 0$ used in the expiry function to compute each object's expiration time. Decreasing the score threshold keeps objects in the cache longer. A lower (laxer) threshold results in an object getting a later expiration time than it would with a higher (stricter) threshold, all else being equal. A lower threshold may reduce the miss ratio by keeping certain objects in the cache longer, but it may come at the cost of high copy-forward write volume. Setting a higher threshold typically results in the opposite tradeoff: though miss ratios may increase, overall write volume may decrease. However, the interplay between write volume and miss ratio is complex because of reinsertion write volume—a higher miss ratio can cancel out the benefit of lower write amplification, as missed objects get rewritten into the cache.

The expiry function is also parameterized by a *lifetime cap* $\texttt{lcap} \geq 0$, which caps the maximum expiration time an object can be assigned (i.e., lifetime is the expiration time minus the current time). The `lcap` ensures objects are evicted from the cache within a reasonable time. When the score threshold is low, expiration times can grow far beyond the maximum time observed between requests for the same object in the workload (let alone, say, the 95th percentile of inter-request times). Setting a cap on the lifetime avoids keeping objects around for so long they are unlikely to contribute to more hits.

Nabu's expiry function can tune how it accounts for object size in an object's expiration time, allowing Nabu to achieve performance on a spectrum between optimizing for bytes and optimizing for objects. The size penalty, `szmod`, is an exponent on object size between 0 and 1. Setting $\texttt{szmod} = 0$, i.e., treating all objects the same regardless of size, lets Nabu achieve the best byte miss ratios. Setting $\texttt{szmod} > 0$ increases the penalty applied to size. With $\texttt{szmod} = 1$, the expiry function fully accounts for size, penalizing larger objects and achieving the best object miss ratios.

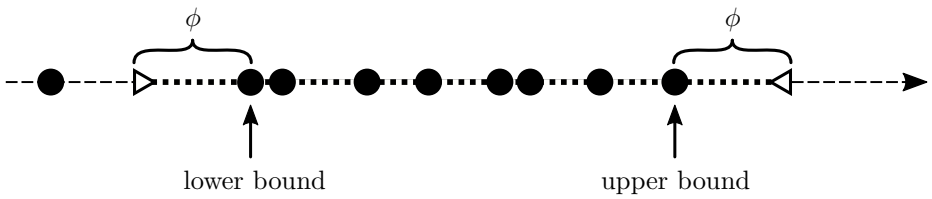


Figure 3.2: A schematic of a cluster. The dashed arrow is a timeline representing expiration times. Solid circles are expiration times of objects in the cluster. The lower and upper bounds of the cluster are marked. The empty triangles indicate ϕ away from the bounds. Any object added to the cluster will become a new upper/lower bound if its expiration time falls within the region between an upper/lower bound and a triangle marking ϕ. Objects may be added to a cluster even if their expiration times fall outside of ϕ, but they will not affect the bounds.

The full equation for calculating an object's expiration time is thus:

$$\min(o.t_n + \frac{o.n}{o.s^{\texttt{szmod}} \times \texttt{thresh}}, \texttt{lcap}) \tag{3.2}$$

where $o.n$ is the object's access count, $o.t_{n-1}$ is the object's last access time, $o.s^{\texttt{szmod}}$ is the object's size adjusted by the size penalty, `thresh` is the score threshold, and `lcap` is the lifetime cap.

Section 4.6 evaluates the impact of these parameters on Nabu's performance. Section 4.6 also describes how a system using Nabu could tune these parameters in practice.

3.3 Grouping Policy

After Nabu calculates an expiration time for an object, Nabu must select a container to group it into. The grouping policy's high-level goal is to keep copy-forward write volume low by grouping together objects with similar expected expiration times. It achieves this by treating each open container as a *cluster* of objects, then using online clustering to find the best container for each object based on its expiration time. In addition to selecting the best open container for each object, the grouping policy also manages the open containers themselves, deciding when to close open containers and when to create a new cluster with an empty open container.

Clustering is lightweight because it can create groups of similar objects without tracking the distribution of object expiration times. Nabu's clustering algorithm is inspired by an online algorithm that aims to minimize the *diameters* of clusters (EXTEND CLOSED CLUSTERS in Csirik et al. [21]). The diameter of a cluster is the range of object values in the cluster (where values are expiration times in Nabu). This objective is a good fit for Nabu's grouping policy: a small diameter means the expiration times of objects in the container are close to one another, satisfying the goal of grouping

objects together that are likely to be evicted around the same time (subject to the constraints on grouping quality described in Section 3.2.1). Nabu modifies the algorithm for flash: it supports a maximum number of open clusters (containers), and it modifies how distances among expiration times are calculated to account for their time-dependent nature.

At a high level, Nabu's clustering algorithm groups an incoming object into a container c if `Dist(`c,o`)`, the distance from the object's expiration time to the *upper/lower* bound of the container, is smaller than ϕ (Figure 3.2). The upper bound of a container is defined by the maximum expiration time added to the container under the condition of `Dist(`c,o`)` $< \phi$ at the time the object was added. The lower bound is defined similarly. If no such container exists and there is a *free* container (i.e., an open, but empty, container), the object is used to *initialize* that container as a cluster. If there are no free containers, the object is placed in the container to which it is closest. In this case, the bounds of the container are not updated since doing so would expand the container bounds by more than ϕ, leading to worse groupings. The algorithm is detailed in Figure 8.2.

An alternative to making a sub-optimal grouping for an object is to close a container and open a new one for that object. However, this strategy will waste cache capacity if the open containers are not nearly full. Furthermore, there is no guarantee that the object being grouped is not an outlier with a very high or very low expiration time; opening a new container for an outlier object would result in worse groupings later.

Nabu's clustering algorithm makes a best-effort attempt to create tight clusters of objects with small expiration time diameters. Since containers are a precious resource, it is important to only initialize a cluster for regions of the expiration time space where there is not already a container. The ϕ parameter ensures a free container is used for a cluster only when the object is a poor fit for all existing containers.

3.3.1 Handling expiration time drift

Nabu's clustering algorithm handles an additional challenge of the flash caching setting: as time moves forward, expiration times gradually drift into the past. A container whose bounds have drifted into the past as time moves forward will become increasingly difficult to fill, since incoming objects have an expiration time in the future. With no mechanism for adding new objects to such containers, open containers may remain open, with one or more stale objects, forever. Nabu handles such clusters by scaling down distances calculated to a cluster if the container's upper bound is in the past. It does so proportionally to how far the container's upper bound is in the past.

Using a distance scaling factor proportional to the container's age avoids low-quality groupings until necessary. For instance, if a container's upper bound only recently drifted into the past, it is preferable to delay closing the container to wait for incoming objects with expiration times closer to the cluster than to add objects with very distant expiration times. However, if the container's upper bound drifted into the past long ago, many of the container's objects are likely expired. Nabu should thus attempt to close the container as quickly as possible and consider it for erasure.

3.3.2 Closing open containers

Nabu closes an open container when that container is the best fit for an object but does not have space for it. Though this wastes some of the container's capacity if objects do not fit neatly into containers, in practice the wasted capacity is negligible compared to the container's size. After the container is closed, the object is reevaluated to determine if it should be added to another open container, or if the object should initialize the free cluster that replaced the closed container.

3.4 Container Erasure

Nabu periodically needs to reclaim capacity by evicting expired objects. Capacity is reclaimed by erasing a closed container; any unexpired objects in an erased container are kept in the cache by copying them forward into an open container. Nabu initiates container erasure when the cache's available *scratch space* drops below a certain threshold. Scratch space consists of containers that have been erased and are available to replace open containers that get closed.

The container erasure policy's high-level goal is to balance evicting low-quality objects in the cache and minimizing write amplification. Expiration times are key to achieving this goal: they allow Nabu to precisely compute the *cost* of erasing a container in terms of the write amplification that would be generated by erasing that container. They also express a notion of object *staleness*, i.e., how long ago the object expired and thus how long ago a caching algorithm might have evicted the object. Staleness allows Nabu to quantify the *benefit* of erasing the container, where the cache sees greater benefit from evicting more stale bytes. The container erasure policy ranks closed containers based on this cost/benefit analysis and erases the container with the greatest benefit.

The benefit of erasing a particular container is b_{rec}, the number of bytes reclaimed if the container is erased now. The reclamation benefit is scaled by st, a measure of the container's expired bytes' *staleness*. We compute st as the average time since each byte expired:

$$st = \frac{\Sigma_{i=1}^{k}(t_{now} - o_i.exp) \times o_i.s}{\Sigma_{i=1}^{k} o_i.s}$$

where o_i is the ith expired object in the container, $o_i.exp$ is its expiration time, and $o_i.s$ is its size.

The cost of erasing a container is b_{unexp}, i.e., the write amplification in bytes that would be generated from copying forward the unexpired objects in the container. The benefit to erasing each container is then:

$$\frac{b_{rec} \times st}{b_{unexp}}$$

If multiple containers need to be erased, Nabu erases containers in order of descending benefit.

Scaling the bytes reclaimed by the staleness of the bytes encourages Nabu to erase containers with very low-quality bytes, even if the write amplification impact would be worse than erasing other containers. Similarly, given two containers which, if erased, would generate the same amount of write amplification, Nabu will choose the one with more-stale bytes. Clearing low-quality bytes from the cache frees cache capacity for less-stale objects to remain cached for longer, potentially allowing them to accrue more hits.

The benefit of using expiration times in container erasures is apparent: determining the number of bytes reclaimable from a container is simply a matter of counting the number of bytes belonging to objects with expiration times in the past. The calculation of a container's staleness is similarly simple and computationally cheap. At the same time, this container erasure strategy requires calculations based on the expiration time and size of every cached object. Two factors make this practical. First, the calculations can be parallelized, since each container is evaluated independently of others until containers get ranked. Second, the analysis need not be repeated for every container erasure. Containers at the bottom of the ranking tend not to significantly change rank between container erasures. The ranking can therefore be recycled for a few container erasure cycles before it should be refreshed with the most up-to-date cost/benefit analysis [48]. We found for our traces that recycling the ranking 10 times had negligible impact on cache performance (i.e., write volume or miss ration) compared to the exact method, but conferred a significant speedup (up to 2×) in simulation runtime.

3.5 Copy-Forward Policy

Nabu's copy-forward policy is designed to rescue valuable objects from containers selected for erasure. Such objects may be in a container being erased because they were accessed after insertion, while colocated objects were not. Alternatively, objects with widely varying expiration times may have been initially grouped into the same container because there were no better containers available when the objects were written. Copying such objects forward is a lazy update of the object's position to a container with objects more similar to itself.

Expiration times makes it easy to identify objects which a caching algorithm would most likely keep, so that they can be copied forward. Expired objects have a low anticipated value and should be evicted, since Nabu predicted that the object would have been evicted by the caching algorithm, while those objects which are unexpired should be copied forward.

Some workloads benefit from copy-forward filtering, where objects which have not yet expired but have little time remaining before expiration are evicted. In those workloads, copy-forward filtering can reduce write amplification without significantly impacting miss ratios (Section 4.6). Nabu accepts a copy-forward filter parameter that specifies a minimum remaining lifetime for objects to be copied forward. Any object in a container being erased that has less than the minimum lifetime remaining before it expires is evicted.

Chapter 4

Evaluation

We evaluate Nabu to answer the following questions:

- How does Nabu compare with RIPQ, the state of the art flash caching system (Section 4.4)?
- How do types of write volume contribute to total write volume in Nabu (Section 4.5)?
- How do Nabu's parameters impact its performance (Section 4.6)?

4.1 Implementations

4.1.1 Nabu Implementation

We implement a Nabu simulator, as well as a prototype of Nabu. The simulator is implemented in approximately 5K lines of C++. The prototype is integrated into the CacheLib caching framework [5]; unfortunately, at the time of writing, the prototype was not complete, so we do not show results for it here.

We simulate Nabu as if it were on a conventional SSD exposing a block interface. That is, objects are buffered in open containers in DRAM and flushed to the SSD when they are full (Section 3.3). Flash capacity consists of closed containers and *scratch space*. Scratch space is container-sized regions of erased flash to which open containers can immediately be flushed. We allocate scratch space equivalent to twice the number of open containers. For instance, for a 1024 GiB SSD with 1 GiB containers, if there are 8 open containers, the total flash capacity available for storing cached objects is 1008 GiB. Containers are erased when the scratch space needs to be replenished (i.e., the scratch container count falls below 2× the open container count). For efficiency, our implementation

Parameter	Purpose	Where used?
`thresh`	Score threshold for tuning miss ratio/write volume	Expiry function (Section 3.2.3)
`szmod`	Modifies object size penalty to tune write volume	Expiry function (Section 3.2.3)
`lcap`	Caps maximum object lifetime	Expiry function (Section 3.2.3)
ϕ	Controls opening new containers	Grouping policy (Section 3.3)
`cffilter`	Preemptively evicts objects nearing expiration	Copy-forward policy (Section 3.5)

Figure 4.1: Nabu parameters and where they are used.

ranks containers for erasure and recycles this ranking 10 times before recomputing it, as described in Section 3.4.

Nabu parameters Nabu-specific parameters are summarized in Figure 4.1. Section 4.6 evaluates the impact of these parameters on Nabu's performance.

Metadata Overhead Nabu maintains metadata to compute expiration times and for container erasure. For objects, it maintains a total of 24 bytes, allocated as follows:

- 8 byte identifier used as a key
- 4 byte object size
- 4 byte last read time
- 2 byte hit count
- 2 byte container identifier
- 4 byte offset in container, needed to read object

Nabu also uses 4 bytes per container for the container's size (i.e., capacity consumed by objects).

For a cache with 50 million objects, this is around 1.5 GiB, comparable to other flash caching frameworks [5, 66].

4.1.2 Baseline Implementation: RIPQ

RIPQ is the state-of-the-art flash caching framework [66]. RIPQ's grouping, container erasure, and copy-forward policies are designed to approximate a priority queue. RIPQ supports caching algorithms that can be implemented with priority queues, such as GDSF [17] and SLRU [34]. RIPQ is described in more detail in Section 5.2.1.

We implement a RIPQ simulator in approximately 3K lines of C++. All RIPQ metadata is stored in DRAM for fairness, i.e., to avoid consuming cache capacity with metadata, since Nabu's

current design stores its metadata in DRAM. As with Nabu, RIPQ is implemented as if it were stored on a conventional SSD (the version of RIPQ in the paper is implemented in this way, as well). RIPQ also requires scratch space for flushing containers from DRAM to the device; the scratch space is allocated and managed in the same way as in Nabu.

RIPQ parameters RIPQ's θ parameter filters copy-forwards that have a sub-threshold priority. RIPQ supports customizable scoring functions. We use the GDSF scoring function [17] to optimize for low object miss ratio, and SLRU with 2, 3, and 4 segments [34] to optimize for low byte miss ratio, as presented in the paper [66]. For GDSF, which uses RIPQ's absolute priority interface, we set the number of priority buckets to 100, which achieved good performance. Similarly, setting the maximum access count to 3 achieved the best performance. Both settings align with observations in the RIPQ paper.

By default, RIPQ erases the tail container of the priority queue. Our RIPQ implementation includes a production optimization that avoids excessive copy-forward write volume by skipping tail containers with too many bytes to copy forward [80]. If the tail container's *copy-forward ratio* (ratio of bytes to copy forward to container size) is higher than a threshold, RIPQ will not erase the container. The container will instead be promoted to the head of the queue, and all objects' virtual block pointers will be erased. RIPQ will continue promoting tail containers until one is found whose copy-forward ratio is below the threshold.

4.1.3 Common Parameters

Both Nabu and RIPQ are parameterized by the number of open containers (Figure 4.1). Our evaluation uses 8 open containers, which showed good results across systems and traces. We explore the effect of using more/fewer open containers in Nabu in the sensitivity analysis (Section 4.6).

4.2 Configuration

All simulations are run with a DRAM cache in front of the flash cache, to simulate a common production setup. The DRAM cache is 1% of the simulated device size and uses the LRU eviction policy. Our DRAM cache borrows from Kangaroo's memory-only cache simulator [44].

Measurement details We take measurements after a warmup period of 50% of the trace. Each run generates at least 40× the device capacity in writes (including warmup and measurement peri-

Trace	Trace1	Trace2	Trace3	Trace4	Trace5
Duration (wall-clock)	9 days	9 days	9 days	4 days	0.5 days
# Requests (total)	1.8 B	2.3 B	2 B	2.6 B	1 B
# Requests (filtered)	1.2 B	1.5 B	1.5 B	2.6 B	1 B
Unique objs. requested	102 M	88.2 M	85.1 M	102.3 M	14.6 M
Total bytes requested	637.7 TiB	525.3 TiB	574.6 TiB	4.9 PiB	3.8 PiB
Unique bytes requested	57.1 TiB	29.9 TiB	40.8 TiB	195.1 TiB	55.6 TiB
Obj. size (mean)	603 KiB	339 KiB	515 KiB	2 MiB	4 MiB
Obj. size (max)	1 GiB	1 GiB	1 GiB	2 MiB	4 MiB

Figure 4.2: Properties of traces used in evaluation. Except for *# Requests (total)*, all numbers shown are for objects requested that are filtered, i.e.,larger than 2 KiB and smaller than 1 GiB (the container size).

ods). Object miss ratio is calculated as the number of objects which were requested but were not found in the cache, divided by the total objects requested. Byte miss ratio is defined similarly. Total write volume is measured as the number of containers flushed to the SSD times the container size.

4.3 Workloads

We evaluate Nabu and the baseline systems on CDN traces. CDN traces consist of read requests for immutable objects. Three traces, Trace1, Trace2, and Trace3, contain variable-sized objects. Neither Nabu nor RIPQ are designed to handle very small objects, so we do not cache requests smaller than 2KiB. A typical flash cache can hold billions of small objects; the associated overhead of indexing that many objects is prohibitive. Small objects are handled by special-purpose small object caches in practice [5, 45].

Trace4 and Trace5 contain fixed-size objects of 2MiB and 4MiB, respectively. Trace properties are summarized in Figure 4.2.

4.4 Nabu pushes out the miss ratio/write volume Pareto frontier for object misses.

Figures 4.4 and 4.5 show the best miss ratio/write volume tradeoffs achievable by Nabu and RIPQ. The frontier is constructed from the best configurations in each system. That is, the frontier points represent configurations where no other configuration has a lower write volume at a given miss ratio.

For RIPQ, we sweep the full range of the θ parameter and the full range of the copy-forward ratio (i.e., 0-1, inclusive, for each) (Section 4.1.2). We sweep these parameters for GDSF, as well as SLRU with 2-4 segments. For parameters to which Nabu is sensitive—the object lifetime cap, `lcap`; the

Trace	WV (TiB)	Object MR	Byte MR
Trace1	74.8	14.8%	21.4%
Trace2	21.3	7.7%	6.9%
Trace3	36.1	7.8%	11.6%
Trace4	480.9	18.0%	–
Trace5	206.3	9.6%	–

Figure 4.3: Absolute values used to compute percent differences from RIPQ minimums in Figures 4.4 and 4.5. WV is the write volume in terabytes; MR is miss ratio.

score threshold, `thresh`; the size penalty, `szmod`; and the copy-forward filter, `cffilter`—we sweep reasonable ranges of those parameters until performance plateaus. For other parameters, e.g., the cluster range value ϕ, we choose a value that works well and use it in all configurations. Parameter sensitivity is discussed in detail in Section 4.6.

Certain trace and system combinations have a frontier constructed of very few points, e.g., Trace4 in RIPQ (Figure 4.4d). In these cases, there were a few specific parameter combinations that outperformed all other configurations. Other plots show apparent discontiguities in the frontier; e.g., for Trace1 in Nabu, there is a sharp increase in write volume around -5% miss ratio difference (Figure 4.4a). Such discontiguities are typically a result of two different configurations with different behaviors contributed to the frontier.

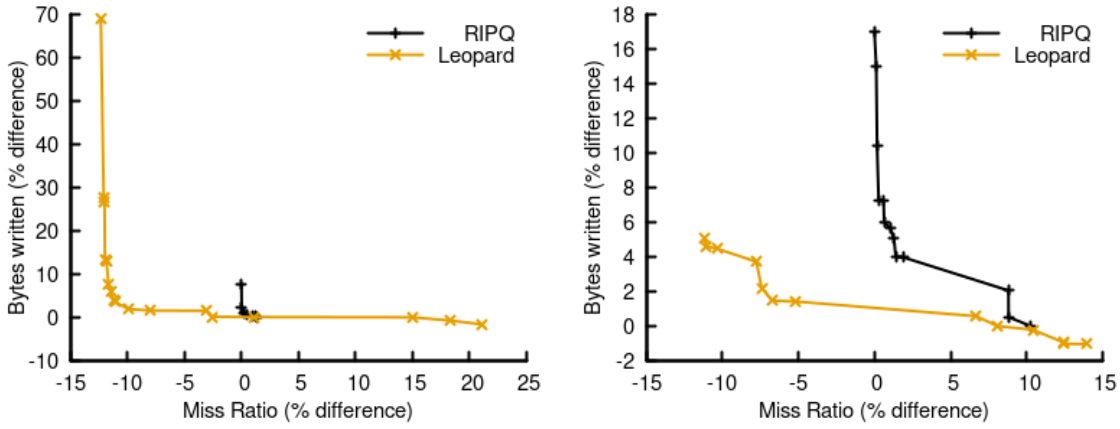
To simplify comparing Nabu's performance to RIPQ's, miss ratios are expressed as the percent difference from RIPQ's minimum miss ratio, i.e., $(MR - \min(MR_{RIPQ})) \div \min(MR_{RIPQ})$. Write volumes are similarly shown relative to RIPQ's lowest write volume. The absolute values used to calculate the percent differences are listed in Figure 4.3.

We show the results for a 1TiB cache; results for a 2TiB cache are comparable and shown in Section 8.3.

4.4.1 Object miss ratio

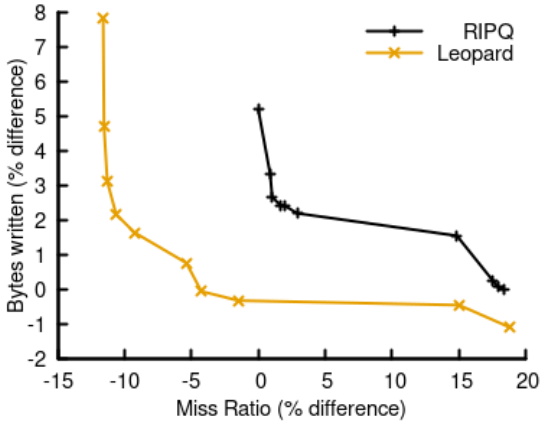
The results show that both systems are subject to the miss ratio/write volume tradeoff inherent in flash caching. As miss ratios decrease, write volume tends to increase.

However, for object miss ratios, Nabu makes the better tradeoff: Nabu pushes out the Pareto frontier of object miss ratio and write volume from what was previously achievable with RIPQ (Figure 4.4). At the lowest common write volume between the two systems, Nabu achieves object miss ratios up to 20% lower than RIPQ's (e.g., Figure 4.4c at 0% difference in bytes written). Nabu's minimum object miss ratios are up to 12% lower than RIPQ's (e.g., the leftmost points for each framework in Figures 4.4a, 4.4b and 4.4c). In both frameworks, write volume tends to spike quickly

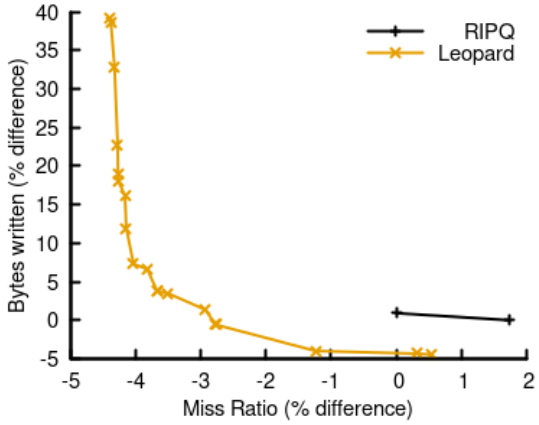


(a) Trace1 object miss ratio

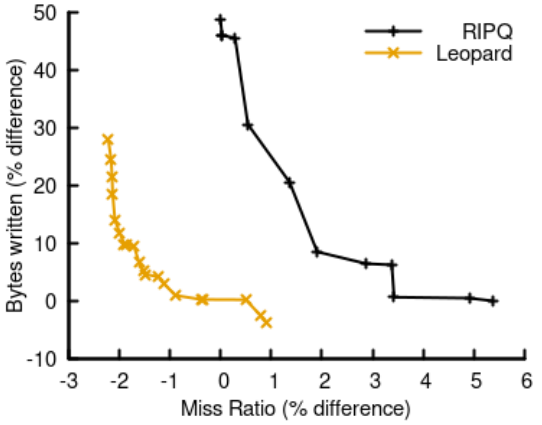
(b) Trace2 object miss ratio



(c) Trace3 object miss ratio



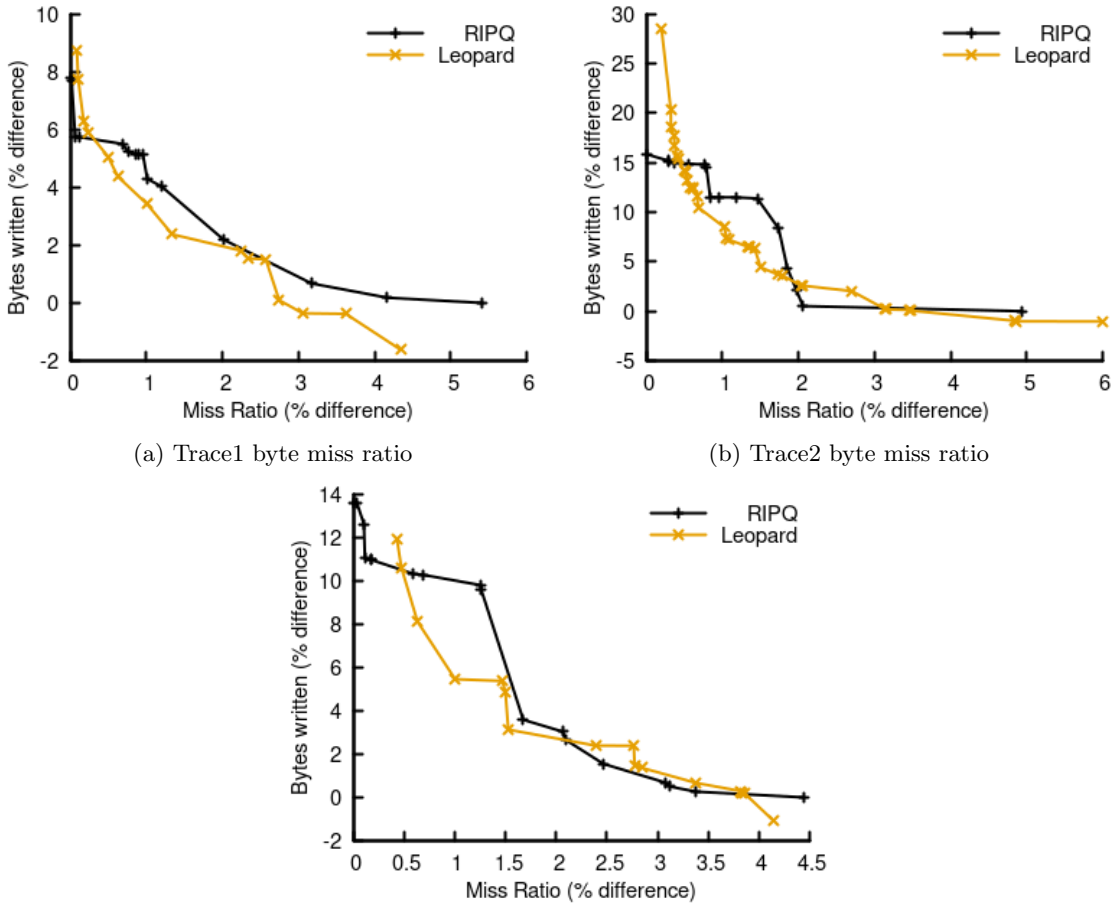
(d) Trace4 object miss ratio



(e) Trace5 object miss ratio

Figure 4.4: Object miss ratio vs write volume for Nabu and RIPQ for a 1TiB cache. Nabu pushes out the Pareto frontier of object miss ratio/write volume.

at its lowest miss ratios; a small miss ratio increase can be traded off for a large decrease in write volume. Nabu's write volume is generally significantly lower than RIPQ's at the same miss ratio, with reductions of up to up to 30% (e.g., Figure 4.4e at 2% miss ratio difference).



(a) Trace1 byte miss ratio

(b) Trace2 byte miss ratio

(c) Trace3 byte miss ratio

Figure 4.5: Byte miss ratio vs write volume Pareto frontier for Nabu and RIPQ for a 1TiB cache. Trace4 and Trace5 are not shown; byte and object miss ratio for those traces are equivalent because all objects are the same size. Nabu is at or near the Pareto frontier of byte miss ratio and write volume.

4.4.2 Byte miss ratio

In terms of byte miss ratio, Nabu typically achieves comparable or better performance than RIPQ (Figure 4.5). RIPQ achieves lower write volumes for some miss ratios—and slightly better minimum miss ratios—than Nabu in the variable-sized object traces, i.e., Trace1, Trace2, and Trace3. These traces have a high proportion of objects that only get accessed once, and SLRU variants are especially effective at evicting such objects early (here we use SLRU with 2, 3, and 4 segments [34, 66]). Future work will investigate how to improve Nabu's performance for byte miss ratio for such traces.

4.4.3 Discussion

Nabu's better object miss ratio versus write volume tradeoff compared to RIPQ is due to several factors. One is Nabu's flexible container erasure strategy, which allows Nabu to achieve low write volume without compromising object miss ratios. Nabu balances the goal of evicting objects unlikely to contribute to hits with the goal of keeping write volume low at each container erasure. RIPQ, on the other hand, is constrained with regard to the order it can erase containers; it erases containers from the tail of the queue. RIPQ can be optimized to skip erasing containers that will generate high copy-forward write volume [80] (4.1.2). This optimization significantly reduces copy-forward write volume and hence total write volume in both our simulations and Facebook's production deployments of RIPQ [80]. However, because skipped containers get promoted to the head of the queue and RIPQ does not explicitly track object values, this strategy can result in worse miss ratios as low-value objects are kept in the cache longer. We observed increasingly worse object miss ratios in our simulations as RIPQ's copy-forward ratio threshold was decreased (i.e., it more aggressively skips containers).

A factor contributing to Nabu's low object miss ratios may be that Nabu's expiry function may overestimate expiration times for objects valued highly by the caching algorithm. When the algorithm optimizes for object miss ratio, this means smaller objects are retained for longer and Nabu aggressively clears larger objects from the cache. This behavior has little penalty, even if those objects do not end up accruing hits, because small objects do not add much to write volume if copied forward and take up little excess cache capacity. In fact, Nabu is rewarded, because keeping smaller objects for longer lets it achieve lower object miss ratios. On the other hand, when the algorithm optimizes for byte miss ratio, it treats large and small objects the same. Nabu's expiry function may then overestimate expiration times for *large* objects and retain them for too long. Large objects naturally contribute more to copy-forward write volume and occupy more cache capacity, making it difficult for Nabu to keep write volume and byte miss ratios low. This hypothesis is supported by the fact that Nabu's best runs from an object miss ratio perspective typically get significantly worse byte miss ratios than RIPQ's best runs for object miss ratio.

This property of Nabu may not be the whole explanation for its modest performance in the byte miss ratio/write volume tradeoff. Nabu outperforms RIPQ for Trace4 and Trace5, which have same-size objects, so overvaluing small objects may not fully explain the observed behavior. The traces where Nabu performs worse for byte miss ratio—Trace1, Trace2, and Trace3—have a high proportion of objects accessed only once. RIPQ handles these traces well because the SLRU variants

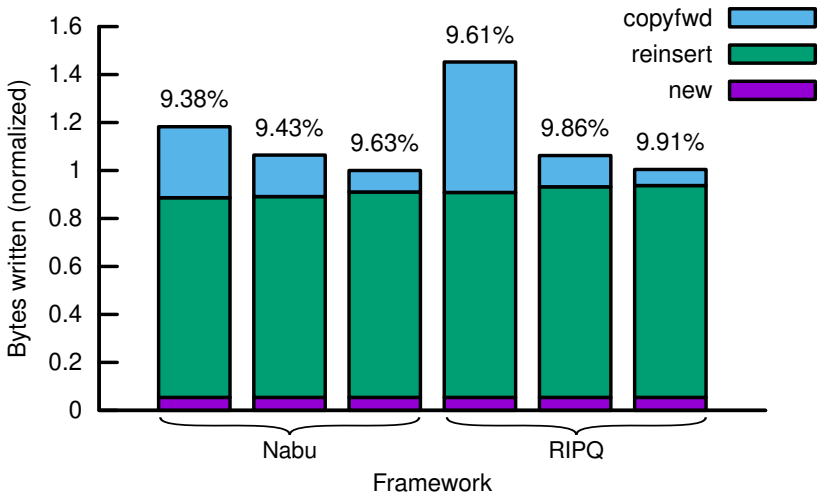


Figure 4.6: Breakdown of bytes written by type for a selection of configurations in Nabu and RIPQ, for the warmed-up Trace5 trace. Nabu's `thresh` increases from left to right, while RIPQ's θ decreases from left to right. The height of each bar is the total write volume, normalized to the minimum RIPQ write volume for these configurations. Byte miss ratios for each run are shown above each bar. Compulsory misses (labeled "new") are object bytes written to the cache for the first time. Reinsertion write volume ("reinsert") indicates objects evicted and then reinserted. Copy-forward write volume ("copyfwd") indicates objects copied forward during capacity reclaim. Nabu achieves lower total write volume than RIPQ at the same miss ratio on this trace by keeping copy-forward write volume low. Overall, copy-forward write volume is a significant contributor to write volume for both frameworks and accounts for the biggest difference in write volume across configurations.

are designed to clear single-access objects from the cache quickly. Nabu's performance in byte miss ratio might be improved with strategies to emulate SLRU's behavior.

4.5 Write volume is influenced by copy-forward write volume and misses

Figure 4.6 shows the contributions of three types of writes to the total write volume of Nabu and RIPQ for selected configurations of Trace5. Nabu configurations are shown with increasingly strict `thresh` (score thresholds) from left to right, while RIPQ configurations are shown with decreasing θ (more restricted copy-forward) from left to right (Section 4.1). The height of each bar shows the total write volume for that run; absolute byte miss ratios are shown above each bar. Writes from new insertions (labeled "new"), i.e., the first time an object is seen by the cache, are shown in purple. New writes are the same for a given trace regardless of the framework, parameters, or cache size. Reinsertion write volume, or writes from missed objects reinserted into the cache, are in teal (labeled "reinsert"). Finally, bytes copied forward, i.e., copy-forward write volume, are shown

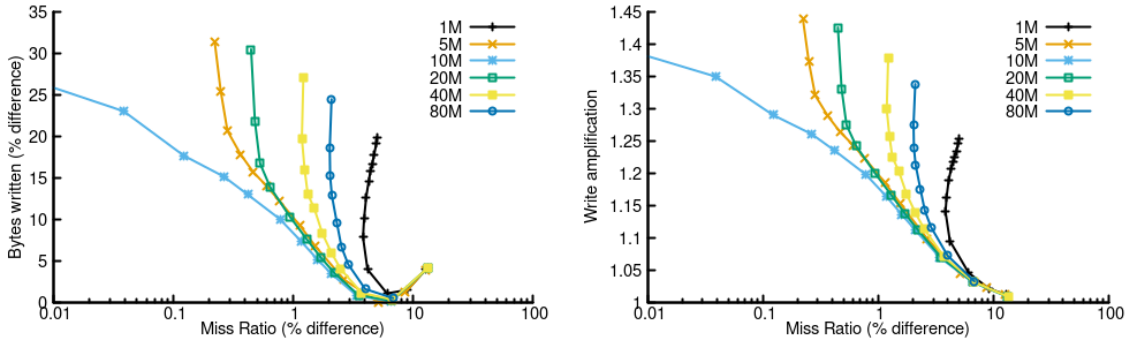
in blue (labeled "copyfwd").

This plot shows the impact of Nabu's `thresh` and RIPQ's θ on total write volume. The reinsertion write volume varies little across configurations because the differences in byte miss ratio are relatively small; the minimum byte miss ratio in the plot is only around 5% lower than the maximum. Write amplification varies more dramatically. Though a more permissive `thresh` or θ increases miss ratios (in this case byte miss ratio), the higher copy-forward write volume can exceed the write volume reduction from reinsertion write volume because more objects get copied forward. The plot shows an instance of the two frameworks having approximately the same byte miss ratio, 9.6%. In this case, Nabu achieves around 30% lower total write volume due to its low copy-forward write volume. For other bar pairs with similar write volumes, e.g., Nabu's 9.43% bar and RIPQ's 9.86% bar, Nabu's lower reinsertion write volume (from lower miss ratios) is cancelled out by a higher copy-forward write volume, showing how complex factors contribute to the central miss ratio/write volume tradeoff.

Comparing across Nabu configurations, the plot shows how some copy-forward write volume can be traded off for a lower miss ratio, but there are diminishing returns as objects compete for cache space as `thresh` increases. For instance, from Nabu's third to second bars (9.63% and 9.43% byte miss ratio, respectively), the miss ratio declines by 2% because `thresh` decreases, keeping objects in the cache longer. Write volume increases by 7% as `thresh` decreases, since more objects are being copied forward. However, from the second to the first bar (9.43% and 9.38% byte miss ratio, respectively), the miss ratio declines by only 0.5%. Write volume increases by 12% due to copy-forward write volume as the cache copies more objects forward. Though reinsertion write volume dominates the write volume for this trace, the significant contribution of copy-forward write volume to the total write volume indicates that reducing copy-forwards is a promising way to improve Nabu's device endurance impact further.

4.6 Sensitivity Analysis

Nabu's performance is influenced by the parameters outlined in Section 4.1.1 and Figure 4.1. Nabu's parameters can be tuned to find the best settings for a given environment and workload, given the system's priorities (i.e., low miss ratio or low write volume). Tuning parameters for a given service can be done in a trace-driven manner, where Nabu's performance with different parameter settings is evaluated on a recent trace and the production parameters are adjusted to reflect the best performance [24]. This process would be repeated periodically to ensure Nabu's parameters are compatible with current workload patterns.



(a) Lifetime cap impact on write volume

(b) Lifetime cap impact on copy-forward write volume

Figure 4.7: Byte miss ratio and copy-forward write volume sensitivity to the lifetime cap, `lcap`, for Trace5. Axes show the percent difference from the lowest value for that metric, with the miss ratio difference on a log scale. Each line corresponds to a lifetime setting expressed in timesteps (where M is millions). Each point on the line corresponds to a different score threshold, with scores becoming more strict moving from left to right. Setting `lcap` correctly for a trace (here, 10M timesteps) is important for both low miss ratio and low write volume.

Figures 4.7, 4.8, 4.9, 4.10 and 4.11 show how the score threshold, lifetime cap, ϕ, open container count, copy-forward filter, and the size penalty influence byte miss ratio, write volume, and copy-forward write volume. All results except for Figure 4.8 are for Trace5. Objects in this trace are all the same size, so byte and object miss ratios are equivalent. The size penalty parameter (Figure 4.8) depends on object size, so we show sensitivity to it using Trace1.

The lines in each plot use the same configuration except the parameter of interest is varied. The default parameters correspond to a configuration at or near the Pareto frontier of Nabu's performance for the given trace. Each line is generated by varying the score threshold, `thresh` (becoming stricter/higher from left to right). Write volumes and miss ratios are shown as a percent difference over the lowest value for each metric, with miss ratios on a log scale to amplify differences among the parameters.

The analysis shows that Nabu's performance is particularly sensitive to the lifetime cap, `lcap`, and the score threshold, `thresh`. The size penalty, `szmod`, allows Nabu to tune performance along a spectrum optimizing for byte miss ratio and lower write volume at one end and object miss ratio at the other end. Other parameters enable a system using Nabu to further optimize performance, but Nabu is not very sensitive to different values.

4.6.1 Sensitivity to lifetime cap, `lcap`

Figure 4.7 shows how Nabu's performance changes as the lifetime cap and score threshold change for Trace5. The lifetime cap, `lcap`, is an expiry function parameter that limits the maximum

expiration time an object can be assigned at the current time (Section 4.1.1). This cap drives the minimum achievable miss ratio by influencing how long objects stay in the cache. However, it is trace-dependent; for this trace, the best results were with `lcap` set to 10M timesteps, because the trace has relatively short request interarrival times. If `lcap` is too small, e.g., at 1M in Figure 4.7, the cache cannot keep useful objects in the cache long enough for those objects to get hit. The high write volume at that `lcap` compared to the other values is a result of the increased reinsertion write volume from the high byte miss ratio (Figure 4.7a).

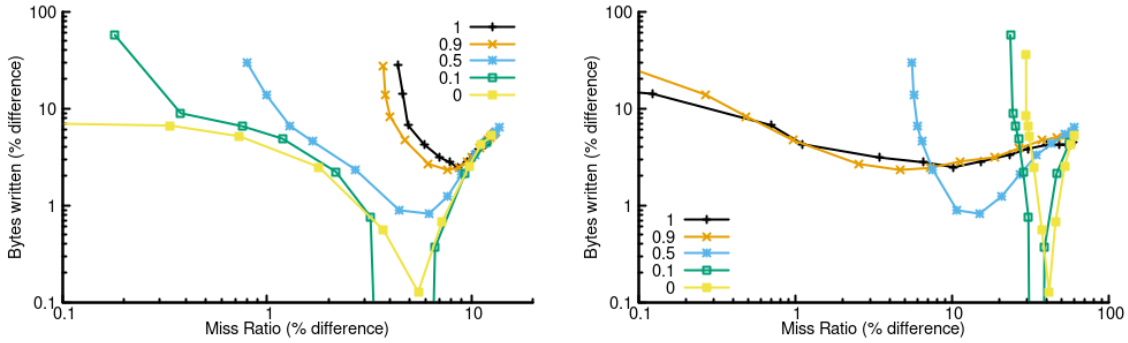
On the other hand, as `lcap` is increased beyond its best setting of 10M, the minimum achievable miss ratio increases as longer-lived objects compete more for cache capacity, forcing container erasures to occur more frequently. Many objects are also long-lived at this setting, resulting in excessive copy-forwards (Figure 4.7b).

4.6.2 Sensitivity to score threshold, `thresh`

The score threshold, `thresh`, is shown by individual points on each line in this section's plots (i.e., Figures 4.7, 4.8, 4.9, 4.10 and 4.11). `thresh` is an expiry function parameter: it controls how expiration times are calculated by setting the score threshold below which an object is considered low-value enough to evict. `thresh` controls where Nabu performs in the miss ratio/write volume tradeoff (Section 4.1.1). As Figure 4.7 shows, changing `thresh` can result in a miss ratio decrease of up to 6% along the tradeoff frontier (i.e., for `lcap` = 10M). On the other hand, it results in an increase in write volume of over 25% between the lowest achievable write volume and the write volume at the lowest miss ratio (Figure 4.7a). At the highest `thresh` (rightmost points), copy-forward write volume is 1×, since nothing is being copied forward (see, e.g., Figure 4.7b). However, the high byte miss ratios induce reinsertion write volume, exceeding the benefit of low copy-forward write volume (Figure 4.7a). Write volumes decline as lower score thresholds contribute to better miss ratios, until excessive writes due to copy-forward write volume cause total write volume to increase again.

4.6.3 Sensitivity to size penalty, `szmod`

The size penalty, `szmod`, is an expiry function parameter. It controls the contribution of size to an object's expiration time. In doing so, it allows tunability along the spectrum of optimizing for byte miss ratio and write volume versus optimizing for object miss ratio (Section 4.1.1). Figure 4.8a shows that Nabu achieves the best byte miss ratio and write volume for this trace when size is fully discounted when computing an object's expiration time, i.e., when `szmod` is set to 0. On the



(a) Size modifier impact on byte miss ratio and write volume

(b) Size modifier impact on object miss ratio and write volume

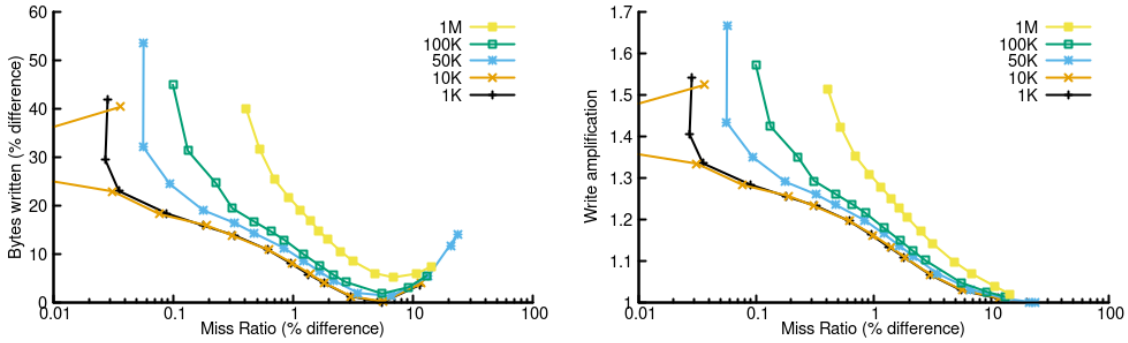
Figure 4.8: Byte and object miss ratio vs. write volume sensitivity to the size penalty, `szmod`, in the Trace1 trace, which has variable-size objects. Axes show the percent difference from the lowest value for that metric; both axes are shown on a log scale. Each point on the line corresponds to a different score threshold, with scores becoming more strict moving from left to right. The `szmod` parameter tunes performance along a spectrum of optimizing for byte miss ratio and write volume, versus optimizing for object miss ratio.

other hand, Figure 4.8b shows that the object miss ratio is best when size is heavily weighted in the expiration time calculation, i.e., when `szmod` is set to 0.9 or 1. Because setting `szmod` to a high value achieves a worse byte miss ratio, this results in a higher overall write volume. Specifically, by setting `szmod` to 1, Nabu achieves up to a 27% decrease in minimum object miss ratio compared to when `szmod` = 0. However, that configuration supports a minimum write volume 30% higher than when `szmod` = 0.

This parameter is useful when a system using Nabu is targeting a specific write volume and wants to optimize for object miss ratio, or vice versa. For instance, say a system is targeting write volume at +1% bytes written over the minimum. With `szmod` = 0 (i.e., a size-unaware expiration time function), Nabu can only achieve an object miss ratio 35% higher than the minimum Nabu is capable of (Figure 4.8b). With `szmod` = 1 (i.e., a size-aware expiration time function), Nabu cannot achieve the target write volume. However, with `szmod` = 0.5, Nabu hits the target write volume with an object miss ratio only 10% higher than Nabu's minimum.

4.6.4 Sensitivity to ϕ

ϕ is a grouping policy parameter controlling when a new container is opened (Section 3.3). Figure 4.9 shows the effect of a larger ϕ: with ϕ = 1M, the grouping policy is less aggressive about opening containers for objects with widely varying expiration times. The less effective groupings lead to higher copy-forward write volume (Figure 4.9b). There is also a downstream effect of making it



(a) ϕ impact on write volume

(b) ϕ impact on copy-forward write volume

Figure 4.9: Byte miss ratio and copy-forward write volume sensitivity to ϕ in the Trace5 trace. Axes show the percent difference from the lowest value for that metric, with the miss ratio difference on a log scale. Each line corresponds to a setting of ϕ expressed in timesteps. Each point on the line corresponds to a different score threshold, with scores becoming more strict moving from left to right. Setting ϕ too high causes the grouping policy initialize new clusters too conservatively, leading to high copy-forward write volume as objects with widely-varying expiration times are grouped together.

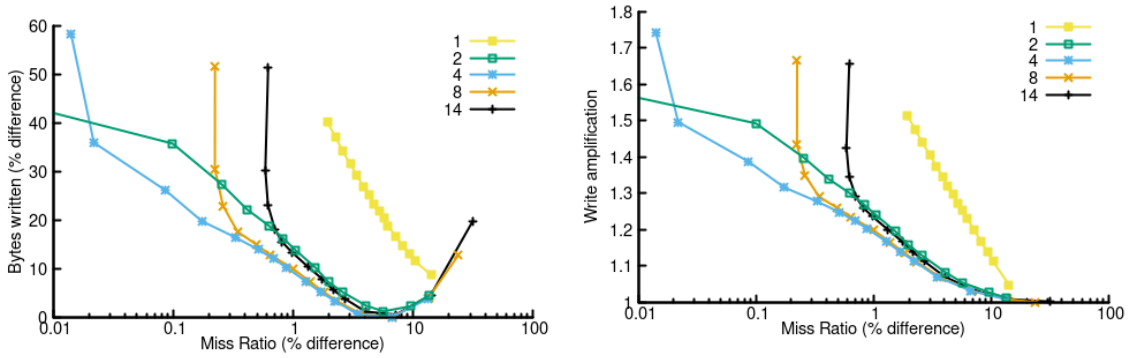
more difficult to effectively choose a container to erase, since highly-varying expiration times result in extremely heterogeneous containers. In particular, it is less likely to find a container with mostly expired objects, so it is harder to find a container to erase that clearly makes a good tradeoff between clearing low-value objects and keeping copy-forward write volume low.

Nabu's miss ratio is not very sensitive to different values of ϕ for this trace; the miss ratio difference between the worst and best settings, ϕ = 1M and ϕ = 10K, respectively, is less than 1% despite the ϕ values spanning two orders of magnitude (Figure 4.9a). On the other hand, as expected, the write volume difference can be more extreme: comparing the write volume at ϕ = 1M's lowest miss ratio to the write volume at ϕ = 10K for the same miss ratio, ϕ = 1M does 22% more writes than the better configuration.

4.6.5 Sensitivity to open container count, k

Like ϕ, the open container count, k, is a grouping policy parameter (Section 3.3). It influences performance in two ways. First, more open containers may support lower copy-forward write volume as the grouping policy can more effectively group objects with similar expiration times. Figure 4.10b supports this: with k = 1, copy-forward write volume increases at a much faster rate than other configurations. Setting k too small has a similar effect, and for similar reasons, as having ϕ set too high (Section 4.6.4).

Other values of k support lower copy-forward write volume with a more gradual increase as the score threshold increases. Second, with a larger k, the amount of device capacity allocated for



(a) Open container impact on write volume

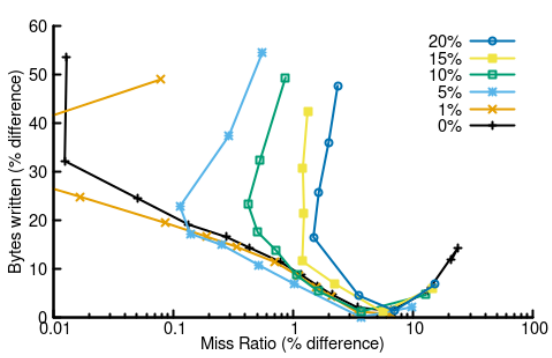
(b) Open container impact on copy-forward write volume

Figure 4.10: Byte miss ratio and copy-forward write volume sensitivity to the open container count, k, in the Trace5 trace. Axes show the percent difference from the lowest value for that metric, with the miss ratio difference on a log scale. Each line corresponds to a different setting of k. Each point on the line corresponds to a different score threshold, with scores becoming more strict moving from left to right. Having too few open containers leads to high copy-forward write volume because objects are grouped together regardless of expiration time. Additional open containers allow better separation of objects with different expiration times, supporting lower copy-forward write volume, but Nabu is not very sensitive to different settings of $k > 2$ for this trace.

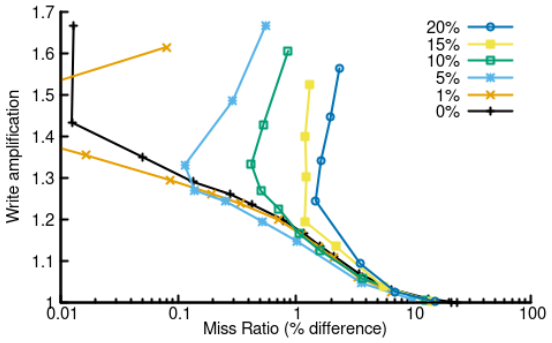
scratch space increases (Section 4.1.1). As such, once there are enough open containers for grouping with low copy-forward write volume, additional open containers may slightly hurt performance since there is less capacity available for caching. For instance, in this trace, $k = 4$ best balances these factors, but the difference from other settings of $k > 2$ in write volume and miss ratio is negligible (Figure 4.10a).

4.6.6 Sensitivity to copy-forward filter, `cffilter`

The copy-forward filter, `cffilter`, is a copy-forward policy parameter. It sets a configurable threshold on what gets copied forward when a container is erased (Section 3.5). Here, we express `cffilter` as a percentage of `lcap`. Intuitively, any object that has less remaining lifetime than this value is evicted. Filtering copy-forwards can significantly influence write volume (Figure 4.11a). On one hand, it can reduce copy-forward write volume, but on the other it may increase reinsertion write volume. Comparing the 0% setting with the 1% setting, filtering some copy-forwards reduces copy-forward write volume (and hence write volume) as the score thresholds increase (Figure 4.11b). However, the effect on miss ratio of further increasing the filter value is significant, since copy-forward filtering can often result in evicting valuable objects that must be re-inserted later.



(a) Copy-forward filter impact on write volume



(b) Copy-forward filter impact on copy-forward write volume

Figure 4.11: Byte miss ratio and copy-forward write volume sensitivity to the `cffilter` in the Trace5 trace. Axes show the percent difference from the lowest value for that metric, with the miss ratio difference on a log scale. Each line corresponds to a different filter setting, expressed as a percentage of `lcap`. Each point on the line corresponds to a different score threshold, with scores becoming more strict moving from left to right. Filtering copy-forwards can reduce write volume, but often at the expense of higher miss ratios.

4.7 Takeaways

In terms of cache performance and endurance, Nabu is usually the better choice compared to RIPQ: for a given miss ratio, Nabu's endurance impact on the SSD is typically lower than RIPQ's. In practice, Nabu's lower write volume means that Nabu can support a longer device lifespan for cheaper devices with lower endurance budgets, e.g., SSDs with multi-level cells (MLC, TLC, etc.). Lower object miss ratios mean Nabu could support more efficient systems with better user-visible performance. For instance, in a CDN, Nabu's lower object miss ratios would translate to lower user-experienced latencies, because fewer requests need to be retrieved from distant datacenters. In a storge tier, lower object miss ratios would translate to proportionally fewer hard disk accesses on storage servers. Since hard disks are typically access-rate-bound (i.e., IOPS-bound) [50], fewer disk accesses may mean fewer disks are needed to serve the same workload.

However, Nabu could be strengthened by improving the tradeoff achieves between byte miss ratio and write volume (Section 4.4). Nabu could also potentially lower its total write volume in all settings by more aggressively targeting copy-forward write volume, which can contribute significantly to total write volume especially at lower miss ratios (Section 4.5).

Chapter 5

Related Work

5.1 Flash translation and abstraction layers

The flash translation layer (FTL) is firmware in an SSD's controller that manages the device's physical flash. It carries out tasks like ensuring that erase blocks accumulate wear evenly and that bad blocks are not used for storing data, as well as translating logical addresses to physical addresses. In conventional SSDs, it exposes the familiar block interface to the host; in Zoned Namespaces SSDs, it exposes the zoned interface [10, 27, 64]. FTLs implement two of the three policies that flash caches do: object grouping (where objects are flash pages) and container erasure selection (where a container is a flash erase block). Unlike in flash caches, the FTL cannot modify the copy-forward policy: all valid flash pages must be copied forward to preserve the stored data. FTLs are not designed to handle data that can be arbitrarily erased as in a flash cache, so they cannot manage the tradeoff between miss ratios and endurance. However, many of the ideas in FTL design are applicable to the flash caching setting.

Flash abstraction layers are host-side software that interact with the SSD on behalf of applications. Flash-specific storage systems like file systems and key-value stores, as well as flash caching frameworks, are special cases of flash abstraction layers. Flash abstraction layers often handle object grouping and container erasure selection, since the flash abstraction layer may have access to application information that improves the effectiveness of these policies for reducing write amplification. Like FTLs, flash abstraction layers that implement storage systems cannot modify the copy-forward policy, since they have to preserve application data. A flash abstraction layer typically issues large, contiguous writes to the SSD to avoid write amplification at the device level. It may

alternatively leverage SSD streams [56] or Zoned Namespaces FTLs to control how data is written to the device [9, 10, 53, 66, 73].

Grouping data together on flash by estimating similarities in invalidation time is a key technique for reducing write amplification in SSDs [9, 15, 18, 19, 27, 36, 37, 39, 47, 53, 59, 70, 72, 73]. While many FTLs and flash abstraction layers use data write frequency or other features as a proxy for estimated invalidation time, some directly predict data invalidation time [15, 70, 72]. Predicting invalidation time is analogous to Nabu's assigning a expiration time to each object, and grouping objects by their expiration times. In fact, expiration times are a prediction of when a naïve caching algorithm implementation would invalidate (i.e., evict) an object. However, unlike with FTLs and generic storage systems, Nabu ultimately decides when an object gets erased (evicted), giving it an additional lever of control over write amplification.

5.1.1 Grouping data on flash

Several systems uses streaming k-means clustering to group pages into erase blocks [36, 72, 78]. Streaming k-means clustering groups one item at a time into one of k clusters. Each cluster has a *centroid* which is the average of all items, and the algorithm places an item into the cluster whose centroid the item is closest to (by some distance measure). In the flash caching setting, deciding when to initialize a container (i.e., start a new cluster) based on this algorithm is difficult: a cluster centroid does not correspond to any actual items, and the average may be a poor representation of a cluster's contents. This property made it difficult to strike the right balance between conservatively and proactively initializing new containers (Section 3.3). The algorithm defined in Csirik et al. uses actual items to define a cluster's bounds. Nabu bases its clustering strategy on this algorithm because it gives Nabu more control over when new containers are initialized [21] (Figure 8.2).

Chakraborttii and Litz group flash pages by predicted invalidation times [15]. Because predicted invalidation times show the same time-varying nature as expiration times (i.e., they drift into the past over time), their design sends all pages to a group whose pages' invalidation times are in the past to fill that group quickly. This strategy is similar to Nabu's strategy of adjusting the function to compute the distance between an object and a cluster to quickly fill containers whose bounds have drifted into the past (Section 3.3). Nabu integrates this strategy into an online clustering algorithm, while Chakraborrtii and Litz's design relies on distributions of page invalidation times. Clustering algorithms react faster to workload changes and require less tracking metadata than distribution-based approaches to grouping because they do not rely on historical workload patterns.

5.1.2 Using cost/benefit analysis for container erasure

Many FTLs and flash-based storage applications do a cost/benefit analysis to select a container (i.e., erase block, segment, or contiguous region of data) to erase [18, 19, 35, 37, 38, 47, 48]. The cost/benefit technique was first proposed for selecting a log segment to clean in log-structured filesystems [55]. For an FTL or storage application, the cost/benefit analysis generally tries to avoid erasing containers that will not free up much capacity (equivalently, which will generate a lot of write amplification), as well as containers with valid objects likely to be invalidated in the near future. In Nabu, the cost/benefit analysis also accounts for write amplification, but it balances this cost with the benefit of erasing containers with older expired objects, since such objects are unlikely to contribute to hits.

Nabu reduces the cost of container erasures by maintaining a ranking of containers by erasure benefit. This ranking is reused for a fixed number of container erasures before being refreshed by re-running and re-ranking containers (Section 3.4). This approach is similar to one used in Nagel et al. to reduce the overhead of garbage collection in FTLs [48].

5.2 Flash caching frameworks

Flash caching frameworks handle caching on flash on behalf of applications. RIPQ [66] and Pannier [41] are the flash caching frameworks closest to the goals and target environments of Nabu. Like Nabu, RIPQ and Pannier directly address the tradeoff between endurance and miss ratios for objects written to the device; they are agnostic to admission filtering or higher-level caches.

5.2.1 RIPQ

RIPQ is the state-of-the-art flash caching framework [66]. RIPQ's grouping, container erasure, and copy-forward policies are designed to approximate a priority queue. RIPQ supports caching algorithms that can be implemented with priority queues, such as GDSF [17] and SLRU [34]. On insertion, each object is assigned a relative priority based on its score. Objects are grouped at insertion time into open containers by this priority. The priority queue is divided into a small number of segments, and each open container is at the head of one queue segment. Each open container represents an insertion point into the priority queue. When an object is reaccessed, its new score is translated to a priority. The new priority is marked by pointing the object to a "virtual" placeholder container. The object will be physically moved when its container is erased.

RIPQ orders containers by their position in the priority queue. RIPQ's container erasure policy erases the tail container in the queue to reclaim capacity. Any object that points to a virtual container (i.e., was accessed since insertion) is copied forward into the open container at the head of its virtual container's queue segment.

RIPQ groups objects by their scores, translated to relative priorities [66]. However, unlike expiration times, scores on their own are only useful for ranking objects relative to one another to identify the worst object in the cache (Section 3.2.1). They cannot be used in a straightforward manner to determine whether an object can be evicted. Using scores thus poses challenges for erasing containers when the worst object in the cache (by score) is located in a container that would generate high write amplification if erased.

Our simulations showed that, when RIPQ chooses a different container to erase than the worst one in the cache to avoid excessive write amplification [80], miss ratios suffer. RIPQ orders containers by the relative scores of their contents. The tail container in this ordering contains the lowest-scoring objects and, under RIPQ's default configuration, it is the next container to erase [66]. RIPQ's production configuration may skip tail containers that would generate excessive copy-forward write volume if they were erased at that time [80]. Such containers get promoted to the head of the container queue and the next tail container is evaluated for erasure.

This strategy is problematic for a couple of reasons. First, because RIPQ values objects relative to one another, there is no straightforward way to identify objects in the new tail container that should be kept in the cache. RIPQ only copies forward objects that have been accessed recently. This problem is especially dire when many tail containers must be skipped to find a suitable erasure candidate, because useful objects may not have had the opportunity to get hit. Second, RIPQ does not explicitly track object scores unless an object has been accessed [66]. For all other objects, its score is that of its container. When a container is skipped and promoted, every object in the container gets promoted. Low-value objects may thus be kept longer than they should, increasing competition for cache capacity for new and higher-value objects.

At the same time, we observed that this optimization can reduce RIPQ's write volume, showing the value of leveraging object-level information—in this case, an explicit accounting of the reinsertion write volume generated by erasing a container—when making container erasure decisions. It also highlights how flexibility in container erasure decisions can improve system performance. Nabu uses this insight as a central design principle of its container erasure policy, rather than an optimization, leading to lower overall endurance impact without sacrificing miss ratios compared to RIPQ.

5.2.2 Pannier

Pannier's driving design principle is to proactively remove cold objects (i.e., those without recent/frequent accesses) and invalid objects (i.e., deleted or updated objects) [41]. In addition to hot and cold queues of containers, Pannier maintains a third "survival queue" of all containers ordered by an elastic notion of age they call *survival time*. A container's survival time increases when an object in the container is accessed, and decreases when an object is invalidated. Pannier's container erasure policy first checks whether the survival queue's top container has run out its survival time clock. If so, that container is erased. Otherwise the tail container in the cold queue is erased. In both cases, any objects with enough accesses are copied forward into either the hot or cold queues.

Pannier's container erasure policy only broadly incorporates per-object information by adjusting the survival time according to object accesses. It does not directly account for write amplification or miss ratios when choosing a container. Nabu, on the other hand, directly targets the miss ratio/endurance tradeoff in its container erasure policy's cost/benefit analysis. Furthermore, Pannier does not account for object sizes when assigning value to objects. As such, it is likely to perform poorly for workloads where objects vary in size and object miss ratio is important, a common scenario in web service workloads.

However, Pannier is designed to handle workloads where objects get updated and deleted, which neither Nabu nor RIPQ are designed to handle. We suspect that Nabu's use of expiration times and its container erasure policy that explicitly accounts for object-level information will make it straightforward to handle object updates and deletions. It is an interesting avenue for future work that could broaden the use cases of Nabu (Section 6.1).

5.2.3 Other flash caching frameworks

Flashtier implements an erase block selection policy that considers the valid page count of each container [58]. Flashtier erases the entire block, i.e., its copy-forward policy is to copy forward nothing. Unlike Nabu, Flashtier does not account for object value when selecting a block, leading to potentially high-value objects being erased.

Many other flash caching frameworks use techniques which are orthogonal and potentially complementary to Nabu. Admission filtering preserves endurance by reducing writes to flash [5, 23, 29, 32, 52, 81]. Deduplication and compression reduce flash writes by reducing the capacity needed to store objects [40, 43, 58]. These systems could benefit from careful management of object grouping, container erasure, and copy-forward provided by Nabu. Kangaroo [45] targets caching very small

objects on flash. Kangaroo and Nabu are complementary.

Grouping by eviction time in the offline setting Cheng et al. design an offline algorithm that establishes a lower bound on the write volume of caching workload on flash for a given miss ratio [16]. This work has a similar aim as Belady's offline MIN algorithm, which finds the optimal miss ratio for workloads with same-size objects [4].

The algorithm works by finding the time at which Belady's MIN would evict each object, then implements admission, grouping, container erasure, and copy-forward policies that use this information to keep write amplification low. One way it reduces write volume is by using object eviction times to group objects into containers, similar to how Nabu uses expiration times for grouping. However, its grouping policy fills a write buffer with objects, then sorts the objects and flushes them to flash in container-sized chunks when the buffer is full. Nabu instead uses clustering to immediately assign an object to a container; this strategy avoids the overhead of sorting objects and re-buffering them according to expiration time. Nabu's grouping policy also requires no DRAM buffering on devices where the host can control data placement, e.g., ZNS SSDs [10].

Cheng et al.'s algorithm is offline. It is thus able to find the true eviction time of each object for an optimal caching algorithm. These strategies are impossible in the online setting; Nabu currently predicts the earliest possible eviction time for algorithms which are known to have good performance in practice.

5.3 Time-to-live and grouping-based caching

5.3.1 TTL-based caching

Time-to-live (TTL)-based caching is commonly used to control data staleness, e.g., in web caches and DNS [2, 20, 33, 67, 76]. Objects are assigned TTLs when they are retrieved from their backing store. If the object's TTL has passed when the object is requested from the cache, it is considered a miss and the most up-to-date version of the object is retrieved from the backing store. TTLs are similar to expiration times in that an object can be evicted when its TTL has expired. Unlike TTLs used for controlling data staleness, Nabu's expiration times are predictions of when a caching algorithm would evict the object, and hence are better tailored to achieving low miss ratios.

TTLs are also often used for modeling simple caching algorithms like LRU and FIFO [7, 25]. Cached objects are assigned TTLs, which in the case of LRU may be extended when an object is hit. An object can be evicted from the cache when its TTL expires. This approach is similar to predicting

earliest possible eviction times for new and cached objects in Nabu. In this dissertation, we identify the concept of earliest possible eviction times as useful inputs to the cache management strategies implemented by flash caching frameworks. We additionally identify a method for computing earliest possible eviction times for more sophisticated algorithms that incorporate hit counts and object sizes (Section 3.2).

5.3.2 Grouping-based caching

Making caching decisions over groups of objects instead of individual objects, i.e., analogous to container erasure policies, has been studied in the in-memory caching setting [75, 77]. Similar to Nabu, GL-Cache [75] does a cost/benefit analysis to find the best group of objects to erase from a miss ratio perspective. Unlike GL-Cache, Nabu's container erasure policy explicitly accounts for the impact on write volume in its cost/benefit analysis. GL-Cache's in-memory setting means the cost of keeping objects from erased groups is lower, compared to copying objects forward in the flash setting. Thus, GL-Cache keeps a fixed fraction of objects from each container, while Nabu keeps only those objects that have not expired. For a similar reason, Nabu groups objects into containers based on objects' earliest possible eviction times, whereas GL-Cache groups objects by insertion time. Nabu's more nuanced grouping and copy-forward policies keep write volume low to avoid impacting device endurance.

Chapter 6

Future work

Two avenues for future work are particularly compelling. One is exploring Nabu's ability to support object deletions and updates. Another is exploring how Nabu, and flash caching frameworks more generically, could leverage learning for better flash caching.

6.1 Supporting object deletions and updates

Nabu's current design supports read-only workloads (i.e., cached objects are immutable and the application never explicitly issues a delete operation to the cache). This type of workload is common in web services. For instance, media such as videos and photos are commonly stored as immutable objects [50, 62, 66]. However, other important workloads such as filesystems and key-value stores include object deletions and updates [13, 14, 51]. Updating or deleting an object at the application layer means the cached version must be *invalidated* in the cache, since it will never be returned to clients. Because of flash's constraints, a deleted object cannot be immediately erased. Similarly, an updated object cannot be overwritten in place; if it should be cached, it gets inserted as a new object. Nabu would be more broadly useful if it could handle such workloads.

The challenge in handling updates/deletions is that invalidated objects (i.e., the original version of the object that got updated/deleted) should be cleared from the cache as soon as possible. An invalidated object takes up cache capacity, but unlike a low-value but still valid object, it will never contribute to hits. However, invalidation adds an extra degree of unpredictability to cached objects' behavior: while an object's eviction time is later than its predicted earliest possible eviction time (in the absence of copy-forward filtering), invalidation means an object may need to be erased before the earliest possible eviction time.

This problem could be attacked from a number of angles. One approach would make expiration times invalidation-aware, so that updates/deletions could be supported with Nabu's existing cache management policies. This strategy might involve assigning objects an expiration time that balances anticipated update/deletion times with earliest possible eviction times. Another approach would augment the policies themselves to be invalidation-aware. That is, instead of combining anticipated update/deletion times with earliest possible eviction times into a single value that is used in place of expiration times, the framework could handle these concerns separately in Nabu's policies. For instance, the grouping policy could use two-dimensional clustering—or a different approach all together—to group together objects which are most similar across both dimensions. This strategy avoids having to decide how to combine the two concerns into a single value by which objects are grouped. The container erasure policy could handle invalidated objects as distinct from valid (expired or unexpired) objects. The container erasure policy's cost/benefit analysis could then be redesigned to consider invalidations separately from valid objects, e.g., by assigning a special benefit to erasing invalidated objects.

These approaches rely on some method of predicting when objects will be invalidated. Predicting invalidation times for flash pages or writes has been investigated to improve flash translation layer and flash storage system designs [15, 70, 72] (Chapter 5). Other work estimates data update frequency more coarsely (e.g., a few categories representing more- or less-frequently updated data) [9, 18, 19, 36, 37, 39, 47, 53, 59, 73]. An important part of this work would involve looking into these approaches to identify if any of them are appropriate for cache objects. In particular, it is unclear if using predicted invalidation times is necessary in our setting, or if coarser representations of data update frequency are enough.

6.2 Incorporating more learning into flash caching

Nabu supports algorithms where objects are assigned scores, and an object's score decays in a predictable way over time in the absence of accesses to that object (Section 3.2). Though such algorithms are useful, novel algorithms are constantly being created to achieve lower miss ratios. There are now a number of algorithms that use machine learning to achieve excellent miss ratios for CDN workloads and other workloads of interest [6, 54, 62, 69, 74, 75]. An interesting avenue for future work would look into using such algorithms in Nabu.

The main challenge is that it is not clear how to compute an earliest possible eviction time for objects under these algorithms. For instance, LRB's scoring function returns the predicted time

of an object's next read [62]. LeCaR and CACHEUS track multiple algorithms' performance on a workload and make eviction decisions based on each algorithm's track record on the workload to date; the algorithm used to evict a given object is not known at insertion time [54, 69]. None of these approaches outputs information at object insertion time that could be straightforwardly converted to earliest possible eviction times. However, the excellent miss ratios of these algorithms merit further investigation to adapt them to Nabu.

Using existing machine learning-based caching algorithms might help improve miss ratios in Nabu. However, it would be worthwhile to investigate whether machine learning could help make better decisions with regard to write volume, as well. For instance, could learning help make earliest possible eviction time predictions that are more accurate, and would that accuracy help Nabu reduce write volume and/or improve miss ratios? Could learning directly predict eviction times, and if so, to what extent would that improve Nabu's performance? Nabu uses clustering in its grouping policy (Section 3.3). To what extent could a different learning-based strategy help make better grouping decisions? How could learning be applied to container erasure or copy-forward decisions?

One possible source of insight is GL-Cache, a cache that uses machine learning to find the best group of objects to evict, instead of the best individual object to evict [75]. By making predictions about groups instead of individual objects, GL-Cache is more efficient and in some cases can achieve lower miss ratios than caching algorithms that make predictions about individual objects. GL-Cache's group eviction policy does not account for write volume, as it was not designed for the flash setting. Its policy for grouping objects also does not work in Nabu's setting. It groups objects by write time, which is equivalent in Nabu to using a single open container, a configuration that worked poorly in our experiments (Section 4.6). However, GL-Cache uses learning to account for an object's potential future usefulness in its group selection strategy, while Nabu uses predicted earliest possible eviction times. It would be interesting future work to compare GL-Cache's learning strategy to predicted earliest possible eviction times in the flash caching setting and determine how best to combine it with write volume awareness.

Chapter 7

Conclusion

Caches are crucial for resource efficiency and user-visible performance of web services. The large working set sizes of modern web service workloads demand high-capacity caches to achieve low miss ratios. Flash-based SSDs meet this need by providing sufficient performance and high capacity at low cost. However, SSDs are hindered by their low endurance: the underlying flash wears out with each write, eventually leading to the SSD's failure. However, achieving low miss ratios involves continually inserting new objects while keeping some useful objects in the cache, leading to high write rates and often high write amplification. The goal of low miss ratios is thus in tension with the goal of preserving SSD lifespan.

In this dissertation, we describe a novel method of caching objects on flash that leverages earliest possible eviction times. Earliest eviction times are useful in the flash caching setting because they are a basis for grouping objects that will be evicted together if they do not receive more hits, contributing to low write amplification. Earliest eviction times support container erasure decisions that balance the goals of low write amplification and low miss ratios. Finally, earliest possible eviction times clearly express which objects can be evicted at a given time, and hence which objects are good candidates for erasure. We provide a method for predicting earliest possible eviction times for a useful class of caching algorithm.

This dissertation demonstrates the usefulness of earliest possible eviction times with Nabu, a flash caching framework based on expiration times, or predicted earliest possible eviction times. Nabu uses expiration times to support cache management decisions that achieve low miss ratios and low write amplification. In particular, Nabu's design includes a policy for erasing containers that runs a cost/benefit analysis using expiration times as the input. Nabu selects the container to erase that best

balances the benefit of erasing low-value objects with the cost of copying a container's useful objects forward. Because expiration times express an object's value and evictability in absolute terms—i.e., without requiring expensive comparisons among objects—they support a container erasure policy that can select any container in the cache to erase. Nabu also designs a novel clustering algorithm for grouping objects into containers based on their expiration times. This grouping policy supports low write amplification when containers are erased by making it more likely that most objects in a container can be evicted at the same time.

Our evaluation of Nabu on five CDN traces shows that expiration times and Nabu's cache management policy designs push out the Pareto frontier of the object miss ratio and write volume tradeoff. Nabu can achieve up to 20% fewer object misses than RIPQ for the same volume of flash writes. When comparing performance at the same object miss ratios, Nabu does up to 30% fewer flash writes than RIPQ. Nabu's lower object miss ratios may translate to improved user-visible performance as more requests are served from a cache than from slower backend storage, or lower hard disk provisioning requirements as hard disks are shielded from excess load by a cache. With this improved performance, Nabu's lower write volume supports longer SSD lifespans or the ability to use cheaper devices for the same SSD lifespan.

Nabu performs at or near the Pareto frontier of the byte miss ratio/write volume tradeoff. We believe there is room for improvement in Nabu's performance in byte miss ratio. In particular, we observed that Nabu's performance relative to RIPQ is worse for traces which have a high proportion of single-access objects. We plan to further explore how Nabu could better handle such workloads.

This dissertation also outlines two avenues for future work. First, Nabu is currently designed for read-only workloads, i.e., where objects are immutable. This is a common workload for web service caches [50, 62, 66]. However, other important workloads like filesystem access workloads include object deletions and updates [13, 14, 51]. Exploring how Nabu could support updates and deletions in addition to reads, while maintaining low miss ratios and write volume, is interesting future work. We also propose investigating how learning could be incorporated into Nabu. Predicting earliest possible eviction times is challenging for learning-based algorithms, but such algorithms have shown excellent miss ratio performance. Future work could look into how to support those algorithms in Nabu. In general, the role of learning in making flash caching decisions merits further investigation.

Chapter 8

Appendix

8.1 Grouping Policy Clustering Algorithm

Figures 8.1 and 8.2 give pseudocode for Nabu's clustering algorithm. A description of the grouping policy, including a high-level overview of this grouping algorithm, can be found in Section 3.3.

```
Function Dist(c: Container, o: Object):
    if o.expts ≥ c.lower_bound and o.expts ≤ c.upper_bound then
        return 0;
    else
        if o.expts < c.lower_bound then
            δ ← c.lower_bound − o.expts;
        else
            δ ← o.expts − c.upper_bound;
        end
    end
    if c.upper_bound is in the past then
        δ ← δ ÷ (event_clock − c.upper_bound);
    end
    return δ;
```

Figure 8.1: Function to compute an object's distance to a cluster/container.

```
Function GroupObject(o: Object):
    c, update_bounds ← FindBestContainer(o);
    c.AddObject(o, update_bounds);
Function FindBestContainer(o: Object):
    while true do
        δ_min ← ∞;
        c_min ← ∅;
        for c : open containers do
            δ ← Dist(c, o);
            if c is uinitialized then
                c_free ← c;
            else
                if δ < c_min then
                    δ_min ← δ;
                    c_min ← c;
                    if δ = 0 then
                        break;
                    end
                end
            end
        end
        if δ_min > φ and c_free ≠ ∅ then
            c_best ← c_free;
            update_bounds ← True;
            break;
        end
        if o.size + c.size > c.capacity then
            close c, replace with free container;
        else
            c_ret ← c_min;
            if δ_min > phi then
                update_bounds ← False;
            else
                update_bounds ← True;
            end
            break;
        end
    end
    return c_best, update_bounds;
Function Container::AddObject(o: Object, update_bounds: Bool):
    if update_bounds then
        this.upper_bound ← max(o.expts, this.upper_bound);
        this.lower_bound ← min(o.expts, this.lower_bound);
    end
    ... ;                                          /* Additional bookkeeping */
```

Figure 8.2: Pseudocode for the grouping policy's clustering algorithm, `FindBestContainer`, and the parent function that calls it to group an object into a container, `GroupObject`. `Dist` is described in Figure 8.1.

8.2 Nabu and RIPQ performance relative to FIFO

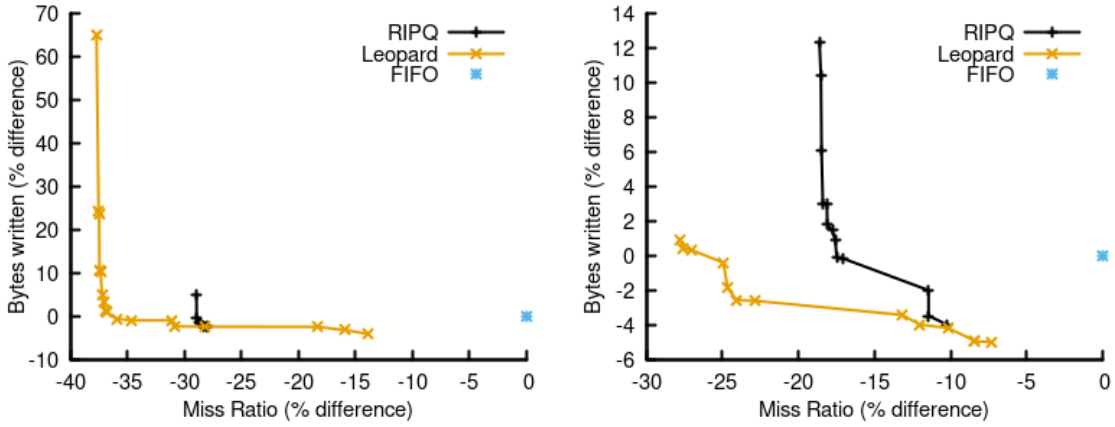
Figures 8.4 and 8.5 show the performance of Nabu and RIPQ relative to FIFO. Section 8.2 shows the absolute values used to compute the percent differences from RIPQ. These results demonstrate that although FIFO does not incur any write amplification, it often has higher write volume at much higher miss ratios than more sophisticated flash caching strategies. The high miss ratios cause reinsertion write volume as the frequent misses are rewritten into the cache. Writes from reinsertion write volume typically exceed the benefit of eliminating copy-forward write volume.

Trace	WV (TiB)	Object MR	Byte MR
Trace1	76.6	20.8%	23.9%
Trace2	22.2	9.5%	7.9%
Trace3	37.0	10.9%	12.9%
Trace4	504.7	20.1%	–
Trace5	219.4	11.2%	–

Figure 8.3: Absolute values used to compute percent differences from FIFO minimums in Figures 8.4 and 8.5. *WV* is the write volume in terabytes; *MR* is miss ratio.

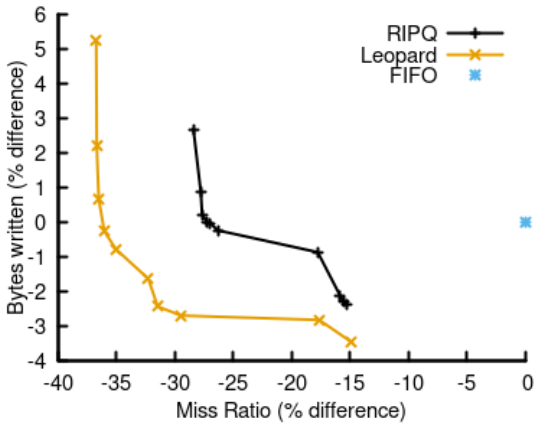
8.3 Nabu and RIPQ performance for 2TiB cache

Figures 8.7 and 8.8 show the performance of Nabu and RIPQ relative to FIFO. Section 8.3 shows the absolute values used to compute the percent differences from RIPQ. This section shows results for Nabu and RIPQ for a 2TiB cache. The results show that the conclusions made for a 1TiB cache still hold, though RIPQ achieves relatively better performance compared to RIPQ for experiments optimizing byte miss ratio as compared to the 1TiB cache.

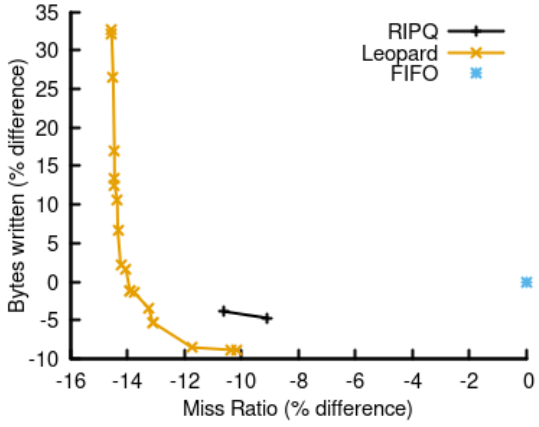


(a) Trace1 object miss ratio

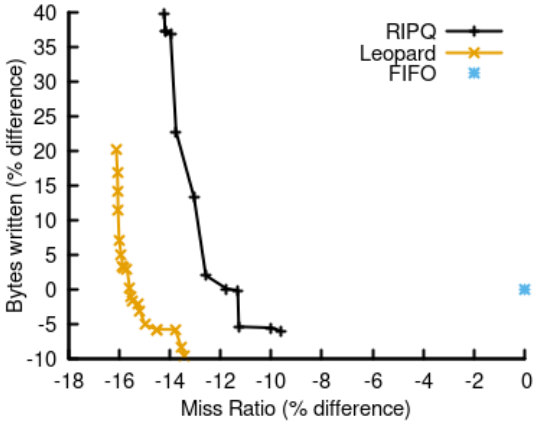
(b) Trace2 object miss ratio



(c) Trace3 object miss ratio

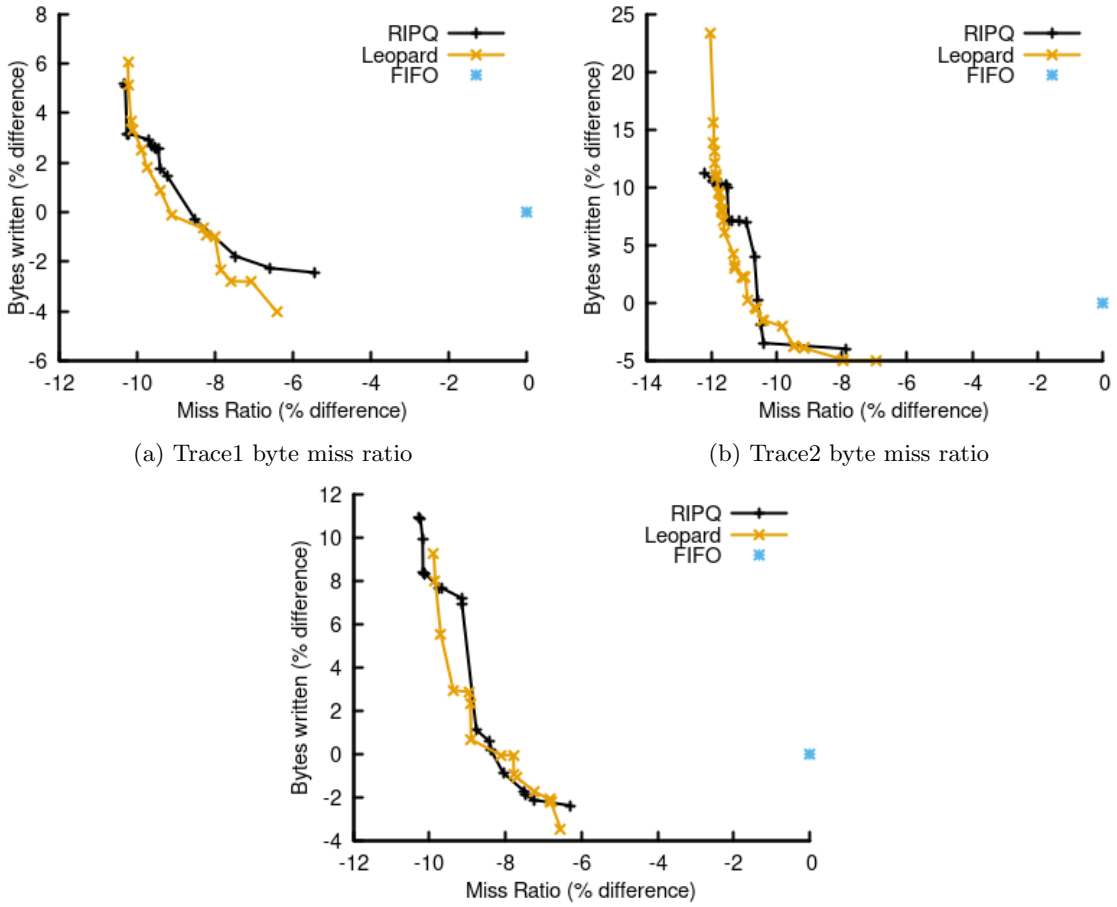


(d) Trace4 object miss ratio



(e) Trace5 object miss ratio

Figure 8.4: Object miss ratio vs write volume for Nabu, RIPQ, and FIFO for a 1TiB cache (Nabu and RIPQ results are identical to Figure 4.4). Results are shown as percent differences from FIFO. The absolute values used to compute these differences are listed in Section 8.2. FIFO, a common choice of caching algorithm for flash caches, performs poorly in miss ratio, causing it to also generate high write volume despite copying no objects forward.



(a) Trace1 byte miss ratio

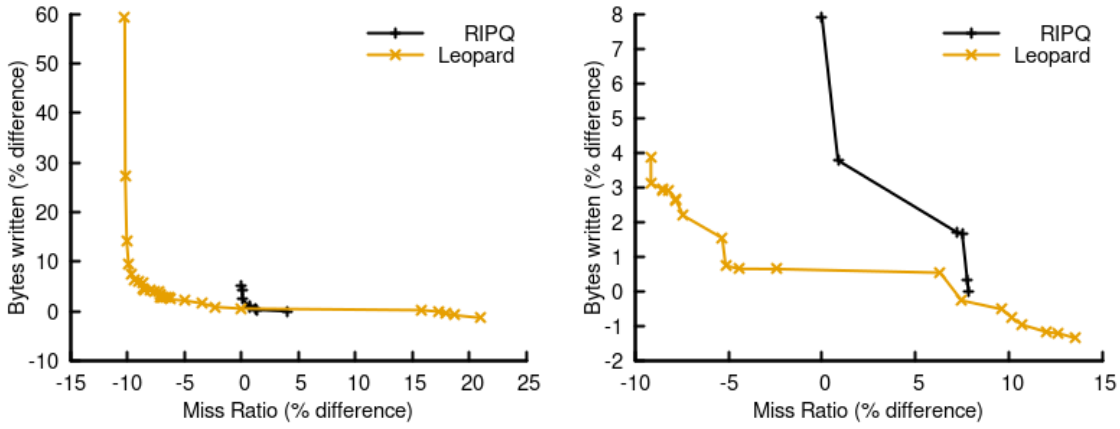
(b) Trace2 byte miss ratio

(c) Trace3 byte miss ratio

Figure 8.5: .
Byte miss ratio vs write volume Pareto frontier for Nabu, RIPQ, and FIFO for a 1TiB cache (Nabu and RIPQ results are identical to Figure 4.4). For Trace4 and Trace5, byte and object miss ratio are equivalent because all objects are the same size. Results are shown as percent differences from FIFO. The absolute values used to compute these differences are listed in Section 8.2. FIFO, a common choice of caching algorithm for flash caches, performs poorly in miss ratio, causing it to also generate high write volume despite copying no objects forward.

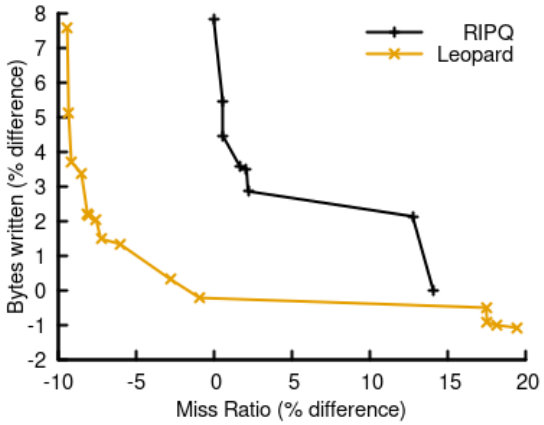
Trace	WV (TiB)	Object MR	Byte MR
Trace1	65.1	12.7%	18.2%
Trace2	19.1	6.9%	6.1%
Trace3	31.6	6.8%	9.9%
Trace4	412.9	15.3%	–
Trace5	150.2	6.9 %	–

Figure 8.6: Absolute values used to compute percent differences from RIPQ minimums in Figures 8.7 and 8.8. *WV* is the write volume in terabytes; *MR* is miss ratio.

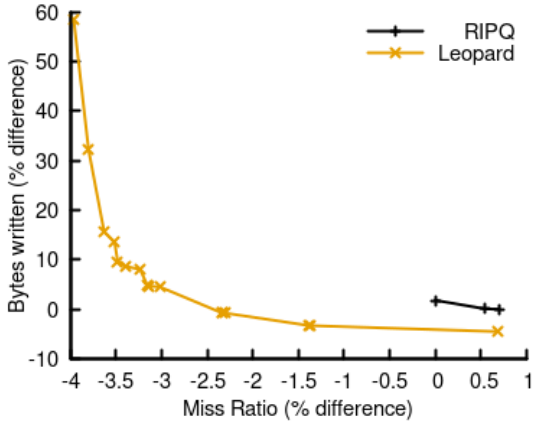


(a) Trace1 object miss ratio

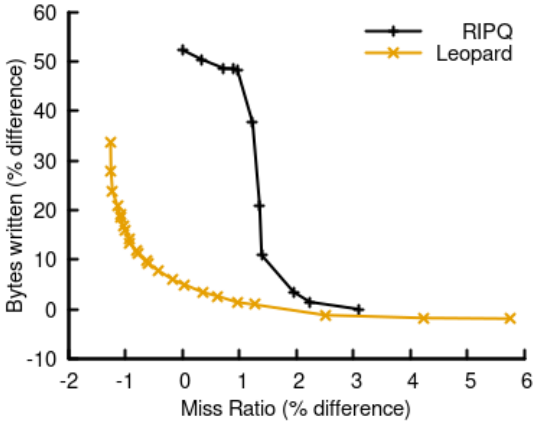
(b) Trace2 object miss ratio



(c) Trace3 object miss ratio

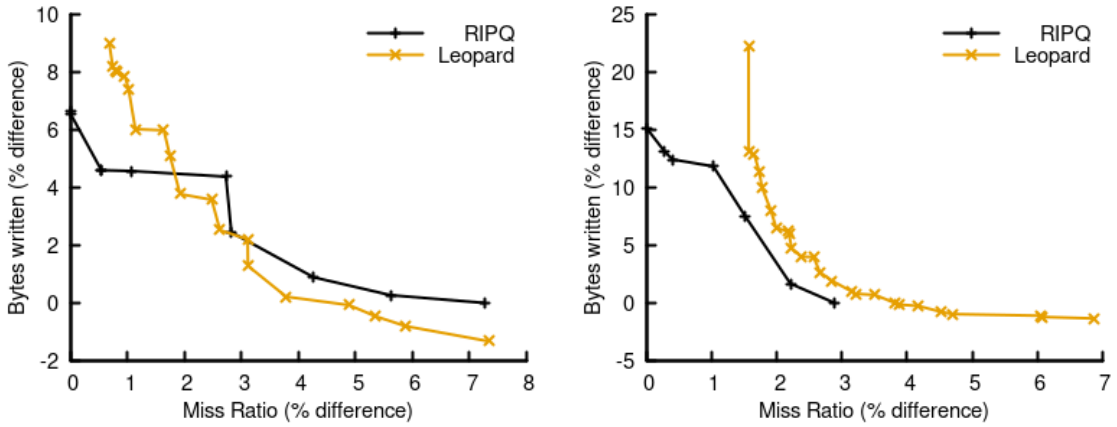


(d) Trace4 object miss ratio



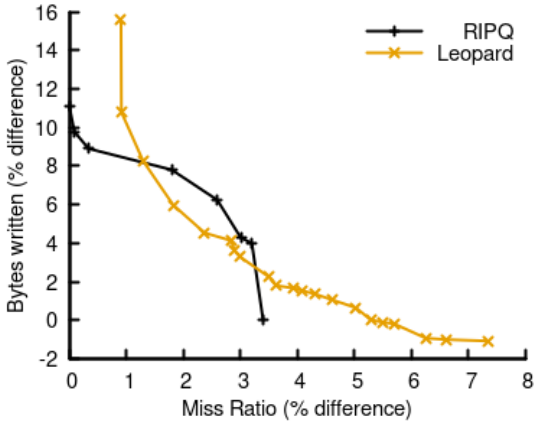
(e) Trace5 object miss ratio

Figure 8.7: Object miss ratio vs write volume for Nabu and RIPQ for a 2TiB cache. Results are shown as percent differences from RIPQ. The absolute values used to compute these differences are listed in Section 8.3. Nabu pushes out the Pareto frontier of the object miss ratio/write volume tradeoff in five CDN workloads compared to RIPQ.



(a) Trace1 byte miss ratio

(b) Trace2 byte miss ratio



(c) Trace3 byte miss ratio

Figure 8.8: Byte miss ratio vs write volume Pareto frontier for Nabu and RIPQ for a 2TiB cache. For Trace4 and Trace5, byte and object miss ratio are equivalent because all objects are the same size. Results are shown as percent differences from RIPQ. The absolute values used to compute these differences are listed in Section 8.3. Nabu performs comparably or slightly worse than RIPQ in byte miss ratio on these traces, with nearly 2% higher minimum byte miss ratios than RIPQ.

Bibliography

[1] DRAMeXchange. `https://www.dramexchange.com`. Accessed: 2022-09-15.

[2] Omri Bahat and Armand M Makowski. Measuring consistency in TTL-based caches. *Performance Evaluation*, 62(1-4):439–455, 2005.

[3] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive TTL-based caching for content delivery. *IEEE/ACM Transactions on Networking*, 26(3):1063–1077, 2018.

[4] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[5] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Berkeley, CA, USA, 2020. USENIX Association.

[6] Daniel S Berger. Towards lightweight and robust machine learning for CDN caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets'18)*, New York, NY, USA, 2018. Association for Computing Machinery (ACM).

[7] Daniel S Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. Exact analysis of TTL cache networks. *Performance Evaluation*, 79:2–23, 2014.

[8] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, Berkeley, CA, USA, 2017. USENIX Association.

[9] Janki Bhimani, Zhengyu Yang, Jingpei Yang, Adnan Maruf, Ningfang Mi, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. Automatic stream identification to improve flash endurance in data centers. *ACM Transactions on Storage (TOS)*, 18(2):1–29, 2022.

[10] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC'21)*, Berkeley, CA, USA, 2021. USENIX Association.

[11] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, Berkeley, CA, USA, 2017. USENIX Association.

[12] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, Berkeley, CA, USA, 2010. USENIX Association.

[13] Dhruba Borthakur. Hdfs architecture guide. `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`. Accessed: 2023-04-23.

[14] Ceph. Ceph file system. `https://docs.ceph.com/en/quincy/cephfs/index.html`. Accessed: 2023-04-23.

[15] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Systems and Storage Conference (SYSTOR'21)*, New York, NY, USA, 2021. Association for Computing Machinery (ACM).

[16] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*, Berkeley, CA, USA, 2016. USENIX Association.

[17] Ludmila Cherkasova and Gianfranco Ciardo. Role of Aging, Frequency, and Size in Web Cache Replacement Policies. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking (HPCN'01)*, Berlin, Germany, 2001. Springer.

[18] Mei-Ling Chiang, Chen-Lon Cheng, and Chun-Hung Wu. A new FTL-based flash memory management scheme with fast cleaning mechanism. In *Proceedings of the 2008 International*

Conference on Embedded Software and Systems (ICESS'08), Washington, DC, USA, 2008. IEEE Computer Society.

[19] 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.

[20] Edith Cohen, Eran Halperin, and Haim Kaplan. Performance aspects of distributed caches using TTL-based consistency. *Theoretical computer science*, 331(1):73–96, 2005.

[21] János Csirik, Leah Epstein, Csanád Imreh, and Asaf Levin. Online clustering with variable sized clusters. *Algorithmica*, 65(2):251–274, 2013.

[22] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[23] 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 *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, Berkeley, CA, USA, 2019. USENIX Association.

[24] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and Flexibility in Distribution of Hot Content. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, Berkeley, CA, USA, 2022. USENIX Association.

[25] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Performance evaluation of hierarchical TTL-based cache networks. *Computer Networks*, 65:212–231, 2014.

[26] Bryan Harris and Nihat Altiparmak. Ultra-low latency SSDs' impact on overall energy efficiency. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*, Berkeley, CA, USA, 2020. USENIX Association.

[27] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*, New York, NY, USA, 2017. Association for Computing Machinery (ACM).

[28] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, New York, NY, USA, 2013. Association for Computing Machinery (ACM).

[29] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving Flash-based Disk Cache with Lazy Adaptive Replacement. *ACM Transactions on Storage (TOS)*, 12(2):1–24, 2016.

[30] SK hynix. SK hynix Enterprise SSD. `https://product.skhynix.com/products/ssd/essd.go`. Accessed: 2023-04-07.

[31] Intel. Intel SSD D7-P5520 Series. `https://ark.intel.com/content/www/us/en/ark/products/213416/intel-ssd-d7p5520-series-1-92tb-2-5in-pcie-4-0-x4-3d4-tlc.html`. Accessed: 2022-09-15.

[32] Dejun Jiang, Yukun Che, Jin Xiong, and Xiaosong Ma. uCache: A utility-aware multilevel SSD cache management policy. In *Proceedings of the 2013 IEEE International Conference on High Performance Computing and Communications (HPCC) & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, Washington, DC, USA, 2013. IEEE Computer Society.

[33] Jaeyeon Jung, Arthur W Berger, and Hari Balakrishnan. Modeling ttl-based internet caches. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'03)*, Washington, DC, USA, 2003. IEEE Computer Society.

[34] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[35] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proceedings of the 1995 USENIX Annual Technical Conference (USENIX ATC'95)*, Berkeley, CA, USA, 1995. USENIX Association.

[36] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Joo Young Hwang, Jongyoul Lee, and Jihong Kim. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, Berkeley, CA, USA, 2019. USENIX Association.

[37] Kevin Kremer and André Brinkmann. FADaC: A self-adapting data classifier for flash memory. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*, New York, NY, USA, 2019. Association for Computing Machinery (ACM).

[38] Ohhoon Kwon, Kern Koh, Jaewoo Lee, and Hyokyung Bahn. Fegc: An efficient garbage collection scheme for flash memory based storage systems. *Journal of Systems and Software*, 84(9):1507–1523, 2011.

[39] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Berkeley, CA, USA, 2015. USENIX Association.

[40] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, Grant Wallace, Fangting Huang, Qing Liu, Gregory R Ganger, Phillip B Gibbons, et al. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*, Berkeley, CA, USA, 2014. USENIX Association.

[41] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: A Container-based Flash Cache for Compound Objects. In *Proceedings of the 16th Annual Middleware Conference (Middleware'15)*, New York, NY, USA, 2015. Association for Computing Machinery (ACM).

[42] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[43] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. CacheDedup: In-line deduplication for flash caching. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, Berkeley, CA, USA, 2016. USENIX Association.

[44] Sara McAllister. Kangaroo. `https://github.com/saramcallister/Kangaroo`, 2021.

[45] 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 *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*, New York, NY, USA, 2021. Association for Computing Machinery (ACM).

[46] Nimrod Megiddo and Dharmendra S Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, Berkeley, CA, USA, 2002. USENIX Association.

[47] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, Berkeley, CA, USA, 2012. USENIX Association.

[48] Lars Nagel, Tim Süß, Kevin Kremer, M Umar Hameed, Lingfang Zeng, and André Brinkmann. Time-efficient garbage collection in SSDs. *arXiv preprint arXiv:1807.09313*, 2018.

[49] The Editors of Encyclopaedia Britannica. Nabu. `https://www.britannica.com/topic/Nabu`. Accessed: 2022-09-23.

[50] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, Berkeley, CA, USA, 2021. USENIX Association.

[51] Meta Platforms. Rocksdb. `https://rocksdb.org/`. Accessed: 2023-04-23.

[52] Timothy Pritchett and Mithuna Thottethodi. SieveStore: a highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*, New York, NY, USA, 2010. Association for Computing Machinery (ACM).

[53] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. FStream: Managing flash streams in the file system. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*, Berkeley, CA, USA, 2018. USENIX Association.

[54] Liana V Rodriguez, Farzana Beente Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, Berkeley, CA, USA, 2021. USENIX Association.

[55] Mendel Rosenblum and John K Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[56] Samsung. Multi-Stream SSD Technology. `https://semiconductor.samsung.com/resources/brochure/Multi-Stream\%20SSD\%20Technology.pdf`. Accessed: 2023-04-22.

[57] Samsung. Samsung SSD PM9A3 Series. `https://download.semiconductor.samsung.com/resources/white-paper/PM9A3_SSD_Whitepaper_230327_2.pdf`. Accessed: 2023-04-07.

[58] Mohit Saxena, Michael M Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys'12)*, New York, NY, USA, 2012. Association for Computing Machinery (ACM).

[59] Mansour Shafaei, Peter Desnoyers, and Jim Fitzpatrick. Write amplification reduction in flash-based SSDs through extent-based temperature identification. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*, Berkeley, CA, USA, 2016. USENIX Association.

[60] Anton Shilov. Toshiba's 768Gb 3D QLC NAND Flash Memory: Matching TLC at 1000 P/E Cycles? `https://www.anandtech.com/show/11590/toshiba-768-gb-3d-qlc-nand-flash-memory-1000-p-e-cycles`. Accessed: 2022-04-16.

[61] Alan J Smith. Disk cache—miss ratio analysis and design considerations. *ACM Transactions on Computer Systems (TOCS)*, 3(3):161–203, 1985.

[62] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning Relaxed Belady for Content Distribution Network Caching. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation (NSDI'20)*, USA, 2020. USENIX Association.

[63] David Starobinski and David Tse. Probabilistic methods for web caching. *Performance Evaluation*, 46(2-3):125–137, 2001.

[64] Theano Stavrinos, Daniel S Berger, Ethan Katz-Bassett, and Wyatt Lloyd. Don't Be a Blockhead: Zoned Namespaces Make Work on Conventional SSDs Obsolete. In *The 18th Workshop on Hot Topics in Operating Systems (HotOS'21)*, New York, NY, USA, 2021. Association for Computing Machinery (ACM).

[65] Radu Stoica, Roman Pletka, Nikolas Ioannou, Nikolaos Papandreou, Sasa Tomic, and Haris Pozidis. Understanding the design trade-offs of hybrid flash controllers. In *Proceedings of the*

27th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'19), Washington, DC, USA, 2019. IEEE Computer Society.

[66] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Berkeley, CA, USA, 2015. USENIX Association.

[67] Xueyan Tang, Jianliang Xu, and Wang-Chien Lee. Analysis of TTL-based consistency in unstructured peer-to-peer networks. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1683–1694, 2008.

[68] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, et al. Tao: how Facebook serves the social graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*, New York, NY, USA, 2012. Association for Computing Machinery (ACM).

[69] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving Cache Replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'18)*, Berkeley, CA, USA, 2018. USENIX Association.

[70] Qiuping Wang, Jinhong Li, Patrick PC Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in {Log-Structured} Storage. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*, Berkeley, CA, USA, 2022. USENIX Association.

[71] Wikitech. Caching overview. `https://wikitech.wikimedia.org/wiki/Caching_overview`. Accessed: 2022-09-15.

[72] Jing Yang, Shuyi Pei, and Qing Yang. WARCIP: Write amplification reduction by clustering I/O pages. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*, New York, NY, USA, 2019. Association for Computing Machinery (ACM).

[73] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. AutoStream: Automatic stream management for multi-streamed SSDs. In *Proceedings of the 10th ACM*

International Systems and Storage Conference (SYSTOR'17), New York, NY, USA, 2017. Association for Computing Machinery (ACM).

[74] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: Mining Sporadic Associations for Cache Prefetching. In *Proceedings of the ACM Symposium on Cloud Computing 2017 (SoCC'17)*, New York, NY, USA, 2017. Association for Computing Machinery (ACM).

[75] Juncheng Yang, Ziming Mao, Yao Yue, and KV Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*, Berkeley, CA, USA, 2023. USENIX Association.

[76] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Berkeley, CA, USA, 2020. USENIX Association.

[77] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *Proceedings of the 18th USENIX Conference on Networked Systems Design and Implementation (NSDI'21)*, USA, 2021. USENIX Association.

[78] 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)*, Berkeley, CA, USA, 2019. USENIX Association.

[79] ZDNET. SSD vs HDD: What's the difference, and which should you buy? `https://www.zdnet.com/article/ssd-vs-hdd-whats-the-difference-and-which-should-you-buy/`. Accessed: 2022-09-15.

[80] Huapeng Zhou, Linpeng Tang, Qi Huang, and Wyatt Lloyd. The Evolution of Advanced Caching in the Facebook CDN. `https://research.facebook.com/blog/2016/04/the-evolution-of-advanced-caching-in-the-facebook-cdn/`, 2016. Accessed: 2022-09-1.

[81] Ke Zhou, Yu Zhang, Ping Huang, Hua Wang, Yongguang Ji, Bin Cheng, and Ying Liu. Efficient SSD Cache for Cloud Block Storage via Leveraging Block Reuse Distances. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2496–2509, 2020.