Kangaroo: Theory and Practice of Caching Billions of Tiny Objects on Flash

SARA MCALLISTER and BENJAMIN BERG, Carnegie Mellon University
JULIAN TUTUNCU-MACIAS, Goldman Sachs
JUNCHENG YANG, Carnegie Mellon University
SATHYA GUNASEKAR and JIMMY LU, Meta
DANIEL S. BERGER, Microsoft Research and University of Washington
NATHAN BECKMANN and GREGORY R. GANGER, Carnegie Mellon University

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

We present Kangaroo, a new flash-cache design that optimizes both DRAM usage and flash writes to maximize cache performance while minimizing cost. Kangaroo combines a large, set-associative cache with a small, log-structured cache. The set-associative cache requires minimal DRAM, while the log-structured cache minimizes Kangaroo's flash writes. Experiments using traces from Meta and Twitter show that Kangaroo achieves DRAM usage close to the best prior DRAM-optimized design, flash writes close to the best prior write-optimized design, and miss ratios better than both. Kangaroo's design is Pareto-optimal across a range of allowed write rates, DRAM sizes, and flash sizes, reducing misses by 29% over the state of the art. These results are corroborated by analytical models presented herein and with a test deployment of Kangaroo in a production flash cache at Meta.

CCS Concepts: • **Information systems** → **Information retrieval**; *Flash memory*; *Social networks;*

Additional Key Words and Phrases: Flash, caching, tiny objects

ACM Reference format:
Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2022. Kangaroo: Theory and Practice of Caching Billions of Tiny Objects on Flash. *ACM Trans. Storage* 18, 3, Article 21 (August 2022), 33 pages.
https://doi.org/10.1145/3542928

Sara McAllister is supported by an NDSEG Fellowship, Benjamin Berg and Juncheng Yang by Facebook Graduate Fellowships, and Nathan Beckmann by a Google Research Scholar Award. This work was supported by NSF-CMMI-1938909, NSF-CSR-1763701, and a 2020 Google Faculty Award.
Authors' addresses: S. McAllister, B. Berg, J. Yang, N. Beckmann, and G. R. Ganger, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15232; J. Tutuncu-Macias, Goldman Sachs, 200 West St., New York City, NY 10282; S. Gunasekar and J. Lu, Meta, 1 Hacker Way, Menlo Park, CA 94025; D. S. Berger, Microsoft Research and University of Washington, Microsoft Building 99, 14820 NE 36th Street, Redmond, Washington, 98052.

1553-3077/2022/08-ART21
https://doi.org/10.1145/3542928

1 INTRODUCTION

Many web services require fast, cheap access to billions of tiny objects, each a few hundred bytes or less. Examples include social networks like Facebook or LinkedIn [16, 24, 69], microblogging services like Twitter [72, 73], ecommerce [18], and emerging sensing applications in the Internet of Things [37, 45, 46]. Given the societal importance of such applications, there is a strong need to cache tiny objects at high performance and low cost (i.e., capital and operational expense).

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

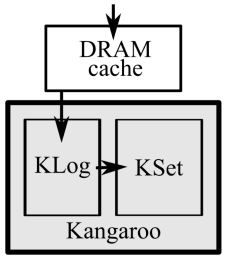
Flash's main challenge is its limited write endurance; i.e., flash can only be written so many times before wearing out. Wearout is especially problematic for tiny objects because flash can be read and written only at multi-KB granularity. For example, writing a 100 B object may require writing a 4 KB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

The problem. Prior flash-cache designs either use too much DRAM or write flash too much. *Log-structured caches* write objects to flash sequentially and keep an index (typically in DRAM) that tracks where objects are located on flash [19, 34, 47, 61, 62, 65]. By writing objects sequentially and batching many insertions into each flash write, log-structured caches greatly reduce the excess bytes written to flash. However, tracking billions of tiny objects requires a large index, and even a heavily optimized index needs large amounts of DRAM [34]. *Set-associative caches* operate by hashing objects' keys into distinct "sets," much like CPU caches [16, 24, 53]. These designs do not require a DRAM index because an object's possible locations are implied by its key. However, set-associative caches write many excess bytes to flash. Admitting a single small object to the cache requires re-writing an entire set, significantly amplifying the number of bytes written to the flash device.

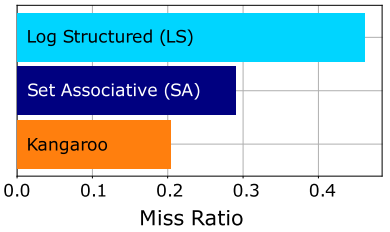
Our solution: Kangaroo. We introduce Kangaroo, a new flash-cache design optimized for billions of tiny objects. The key insight is that existing cache designs each address half of the problem, and they can be combined to overcome each other's weaknesses while amplifying their strengths.

Kangaroo adopts a hierarchical design to achieve the best of both log-structured and set-associative caches (Figure 1(a)). To avoid a large DRAM index, Kangaroo organizes the bulk of cache capacity as a set-associative cache, called *KSet*. To reduce flash writes, Kangaroo places a small (e.g., 5% of flash) log-structured cache, called *KLog*, in front of KSet. KLog buffers many objects, looking for objects that map to the same set in KSet (i.e., hash collisions), so that each flash write to KSet can insert multiple objects. Our insight is that even a small log will yield many hash collisions, so only a small amount of extra DRAM (for KLog's index) is needed to significantly reduce flash writes (in KSet).

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



(a) Overview.



(b) Kangaroo reduces misses by 29%.

Fig. 1. (a) High-level illustration of Kangaroo's design. (b) Miss ratio achieved on a production trace from Meta by different flash-cache designs on a 1.9 TB drive with a budget of 16 GB DRAM and three device-writes per day. Prior designs are constrained by either DRAM or flash writes, whereas Kangaroo's design balances these constraints to reduce misses by 29%.

Summary of results. We implement Kangaroo as a module in CacheLib [16] (cachelib.org). We evaluate Kangaroo by replaying production traces on real systems and in simulation for sensitivity studies. Prior designs are limited by DRAM usage or flash write rate, whereas Kangaroo optimizes for both constraints. For example, under typical DRAM and flash-write budgets, Kangaroo reduces misses by 29% on a production trace from Meta (Figure 1(b)), lowering miss ratio from 0.29 to 0.20. Moreover, in simulation, we show that Kangaroo scales well with flash capacity, performs well with different DRAM and flash-write budgets, and handles different access patterns well. We break down Kangaroo's techniques to see how much each contributes. Finally, we show that Kangaroo's benefits hold up in the real world through a test deployment at Meta.

Contributions. This paper contributes the following:

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

2 BACKGROUND AND RELATED WORK

This section discusses the important class of applications relying on billions of tiny objects, why flash is needed to cache them and the challenges flash brings, and the shortcomings of existing flash-cache designs.

2.1 Tiny Objects are Important and Numerous

Tiny objects are prevalent in many large-scale systems:

- At Meta, small objects are prevalent in the Facebook social graph. For example, the average social-graph edge size is under 100 B. Across edges, nodes, and other objects, the average

object size is less than 700 B [16, 24]. This has led to the development of a dedicated flash caching system for small objects [16].

- At Twitter, tweets are limited to 280 B, and the average tweet is fewer than 33 characters [55]. Due to the massive and growing number of tweets, Twitter seeks a cost-effective caching solution [74].
- At Microsoft Azure, sensor updates from IoT devices in Azure Streaming Analytics are a growing use case. Before an update can be processed (e.g., to trigger a real-time action), the server must fetch metadata (the sensor's unit of measurement, geolocation, owner, etc.) with an average size of 300 B. For efficiency and availability, it caches the most popular metadata [37]. Another use case arises in search advertising, where Azure caches predictions and other results [45, 46].

Each of these systems accesses billions of objects that are each significantly less than the 4 KB minimum write granularity of block-storage devices. For example, Facebook logs 1.5 billion users daily [9] and just friendship connections alone account for hundreds to thousands of edges per user [24, 66]. Twitter logs over 500 million new tweets per day and serves over 190 million daily users [10]. While IoT update frequencies and ad impressions are not publicly available, the number of connected devices is estimated to have surpassed 50 billion in 2020 [32], and the average person was estimated to see 5,000 ads every day as early as 2007 [63].

2.2 Caching Tiny Objects in Flash is Hard

While individual objects in the above applications are tiny, application working sets on individual servers still add up to TBs of data. To reduce throughput demands on back-end data-management systems, applications rely on large-scale, cost-efficient caches, as a single caching server can replace tens of backend servers [16]. Unfortunately, as described below, current caching systems are inefficient for tiny objects. There is therefore a need for caching systems optimized specifically for large numbers of tiny objects.

Why not just use memory? DRAM is expensive, both in terms of acquisition and power cost per bit. This makes traditional DRAM caches hard to scale, particularly with data sizes increasing exponentially [4]. DRAM capacity is also often limited due to operational concerns. Data-center operators generally provision a limited number of server configurations to reduce management complexity. It is not possible in some real deployments to fine-tune server configuration for caching [64]. Moreover, DRAM is often in high demand, so all applications are encouraged to minimize DRAM usage. For example, the trend in recent years at Meta is towards less DRAM and more flash per server [16, 64].

Why flash? Flash currently provides the best combination of performance and cost among memory and storage technologies, and is thus the technology of choice for most large-scale caches [16, 21, 22, 34, 62]. It is persistent, cheaper and more power-efficient than DRAM, and much faster than mechanical disks. While flash-based caches do use DRAM for metadata and "hot" objects, the bulk of the cache capacity is flash to reduce end-to-end system cost.

Challenges in flash caching. Flash presents many problems not present in DRAM caches. A big one is that flash has *limited write endurance*, which means there is a limit on the number of writes before the flash device wears out and must be replaced [23, 40, 44]. Without care, caches can quickly wear out flash devices as they rapidly admit and evict objects [16, 34]. Hence, many existing flash caches over-provision capacity, suffering more misses in order to slow wearout [16, 22]. New flash technologies, such as multi-layer QLC (four bits per cell) and PLC (five bits per cell) [27], increase capacity and decrease cost but significantly reduce write endurance.

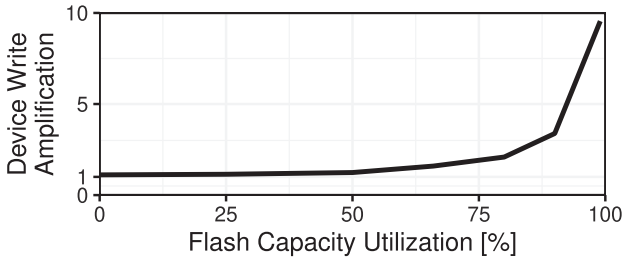


Fig. 2. The effect of flash over-provisioning on device-level write amplification (DLWA) of random writes of various sizes. DLWA increases as over-provisioning decreases.

Exacerbating the endurance issue, flash drives suffer from *write amplification*. Write amplification occurs when the number of bytes written to the underlying flash exceeds the number of bytes of data originally written. Write amplification is expressed as a multiplier of the number of bytes written, and thus a value of of 1× is minimal, indicating no extra writes. Flash devices suffer from both device-level write amplification and application-level write amplification [40].

Device-level write amplification (DLWA) [34, 47, 65] occurs when the **flash translation layer (FTL)** writes more flash pages than asked for by the storage application (e.g., file system, database, or cache). Current flash drives implement the age-old block-storage interface, wherein hosts read and write logical blocks in a numerical **logical-block address (LBA)** namespace. Generally, the internal flash-page size and the external logical-block size are the same, with 4 KB being common, even though the flash device can only erase pages in much larger (e.g., 256 MB) "erase blocks". Most DLWA is caused by cleaning activity that copies live pages elsewhere before an erase block is cleared.

Generally speaking, DLWA worsens as more of the raw flash capacity is utilized and as access patterns consist more of small, random writes. A common approach to reduce DLWA is *over-provisioning*, i.e., only exposing a fraction of the raw flash capacity in the LBA namespace, so that cleaning tends to find fewer live pages in victim erase blocks [16, 22]. Figure 2 shows DLWA vs. utilized capacity for random 4 KB writes to a 1.9 TB flash drive. As expected, DLWA significantly increases as over-provisioning decreases, from ≈1× at 50% utilization to ≈10× at 100% utilization.

***Application-level write amplification* (ALWA)** occurs when the storage application re-writes some of its own data as part of its storage management. One form of this is akin to FTL cleaning, such as cleaning in log-structured file systems [44, 57] or compaction in log-structured merge trees [6, 8]. Another form is caused by having to write an entire logical block. To write a smaller amount of data, the application must read the block, install the new data, and then write the entire block [52]. For example, installing 1 KB of new data into a 4 KB logical block involves rewriting the other 3 KB, giving ALWA of 4×. Ideally, the unmodified data in the block would not have been rewritten.

Why caching tiny objects is hard. The size of tiny objects makes caching them on flash challenging. Tracking billions of tiny objects individually in large storage devices can require huge metadata structures [34], which either require a huge amount of DRAM, additional flash writes (if the index lives on flash), or both. To amortize tracking metadata, one could group many tiny objects into a larger, long-lived "meta-object". This can be inefficient, however, if individual objects in the meta-object are accessed in disparate patterns.

Tiny objects are also a major challenge for write amplification. Traditional cache designs (i.e., for DRAM caches) freely re-write objects in place, leading to small, random writes, i.e., the worst case for DLWA. Since tiny objects are much smaller than a logical block, re-writing them in place

would additionally involve substantial ALWA — 40× for a 100 B object in a 4 KB logical block — which is *multiplicative* with DLWA. Grouping tiny objects into larger meta-objects, as mentioned above, shifts ALWA from logical blocks to meta-objects but does not address the problem.

2.3 Current Approaches and Related Work

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

Key-value stores: Flash-efficient key-value stores have been developed and demonstrated [8, 33, 48, 56, 70], and it is tempting to consider them when a cache is needed. But key-value stores generally assume that deletion is rare and that stored values must be kept until told otherwise. In contrast, caches delete items frequently and at their own discretion (i.e., every time an item is evicted). With frequent deletions, key-value stores experience severe write amplification, much lower effective capacity, or both [15, 22, 34, 65, 70].

As a concrete example, consider SILT [48], the key-value store that comes closest to Kangaroo in its high-level design. Like Kangaroo, SILT uses a multi-tier flash design to balance memory index size vs. write amplification. Unfortunately, SILT's design is poorly suited to caching. For example, SILT's two main layers, which hold >99% of entries, are immutable. Because those layers are immutable, DELETE operations are logged and do not immediately reclaim space. Thus, cache evictions result in holes (i.e., reduced cache capacity) until the next compaction (merge and re-sort) occurs. One can reduce the lost cache capacity with more frequent compactions, but at a large penalty to performance and ALWA.

Similar issues with DELETEs affect most key-value stores, often with this same trade-off between compaction frequency and holes in immutable data structures. One may be able to reduce these overheads somewhat by coordinating eviction with compaction operations, but this is not trivial and not how these systems were designed. For instance, Netflix used RocksDB [8] as a flash cache and had to over-provision by 67% due to this issue [22]. Some key-value stores reduce ALWA by making reads less efficient [49, 56, 70], but do not sidestep the fundamental challenge of DELETEs wasting capacity. In contrast, flash caches have the freedom to evict objects when convenient. This lets flash caches co-design data structures and policies so that DELETEs are efficient and minimal space is wasted.

Log-structured caches: To reduce write amplification, many flash caches employ a log structure on flash with an index in DRAM to track objects' locations [19, 34, 47, 61, 62, 65]. While this solution often works well for larger objects, it requires prohibitively large amounts of DRAM for tiny objects, as the index must keep one entry per object. The index can spill onto flash [70], but spilling adds flash reads for lookups and flash writes to update the index as objects are admitted and evicted.

Even Flashield [34], a recent log-structured cache design for small objects, faces DRAM problems for larger flash devices. After optimizing its DRAM usage, Flashield needs 20 bits per object for indexing plus approximately 10 bits per object for Bloom filters. Thus, Flashield would need 75 GB of DRAM to track 2 TB of 100 B objects. In fact, Flashield's DRAM usage is much higher than this because it relies on an in-memory cache to decide which objects to write to flash. The DRAM cache must grow with flash capacity or else prediction accuracy will suffer, leading to more misses.

Thus, the total DRAM required for a log-structured cache can quickly exceed the amount available and significantly increase system cost and power. Technology trends will make these problems worse over time, since cost per bit continues to decrease faster for flash than for DRAM [27, 71].

Set-associative flash caches: Metadata to locate objects on flash can be reduced by restricting their possible locations [53]. Meta's CacheLib [16] implements such a design for small objects

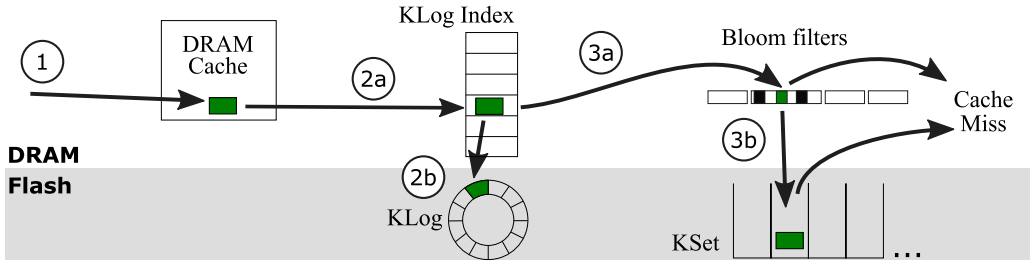


Fig. 3. Lookups in Kangaroo first check a tiny DRAM cache; then KLog, a small on-flash log-structured cache with an in-DRAM index; and finally KSet, a large on-flash set-associative cache. Kangaroo uses little DRAM because KLog is small and KSet has no DRAM index.

(<2 KB), e.g., items in the social graph [24]. CacheLib's **"small-object cache" (SOC)** is a set-associative cache with variable-size objects, using a hash function to map each object to a specific 4 KB set (i.e., a flash page). With this scheme, SOC requires no index and only ≈3 bits of DRAM per object for per-set Bloom filters.

Although more DRAM-efficient, set-associative designs suffer from excessive write rates. Inserting a new object into a set means rewriting an entire flash page, most of which is unchanged, incurring 40× ALWA for a 100 B object and 4 KB page as discussed above. In addition, flash writes are a worst case for DLWA: small and random (Figure 2). The multiplicative nature of ALWA and DLWA compounds the harmful effect on device lifetimes.

Set-associative flash caches limit their flash write-rate through two main techniques. To reduce DLWA, set-associative flash caches are often massively over-provisioned. For example, CacheLib's SOC is run in production with over half of the flash device empty [16]. That is, the cache requires more than *twice* the physical flash to provide a given cache capacity. Additionally, to limit ALWA, CacheLib's SOC employs a pre-flash admission policy [16, 34] that rejects a fraction of objects before they are written to flash. Unfortunately, both techniques reduce the cache's achievable hit ratio.

Summary: Prior work does not adequately address how to cache tiny objects in flash at low cost. Log-structured caches require too much DRAM, and set-associative caches add too much write amplification.

3 KANGAROO OVERVIEW AND MOTIVATION

Kangaroo is a new flash-cache design optimized for billions of tiny objects. Kangaroo aims to maximize hit ratio while minimizing DRAM usage and flash writes. Like some key-value stores [26, 48, 50], Kangaroo adopts a hierarchical design, split across memory and flash. Figure 3 depicts the two layers in Kangaroo's design: *(i)* KLog, a log-structured flash cache and *(ii)* KSet, a set-associative flash cache; as well as a DRAM cache that sits in front of Kangaroo.

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

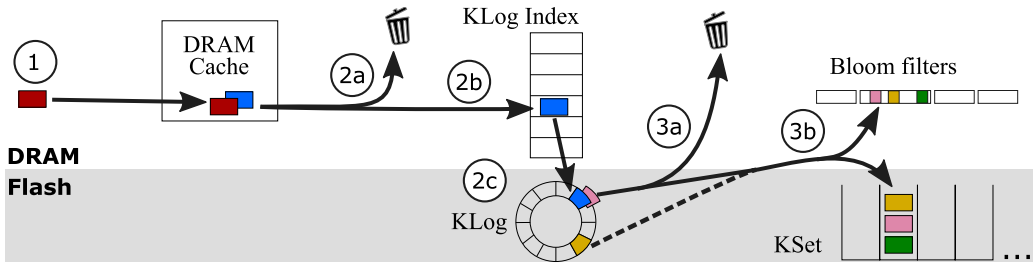


Fig. 4. Objects are first inserted into the tiny DRAM cache, then appended to KLog, and finally moved — along with all other objects mapping to the same set — to KSet. Kangaroo significantly reduces write amplification because KLog is written sequentially and each write to KSet inserts multiple objects.

Insertions follow a similar procedure to reads, as shown in Figure 4. (1) Newly inserted items are first written to the DRAM cache. This likely pushes some objects out of the DRAM cache, where they are either (2a) dropped by KLog's pre-flash admission policy or (2b) added to KLog's DRAM index and (2c) appended to KLog's flash log (after buffering in DRAM to batch many insertions into a single flash write). Likewise, inserting objects to KLog will push other objects out of KLog, which are either (3a) dropped by another admission policy or (3b) inserted into KSet. Insertions to KSet operate somewhat differently than in a conventional cache. For any object moved from KLog to KSet, Kangaroo moves *all objects in KLog that map to the same set* to KSet, no matter where they are in the log. Doing this amortizes flash writes in KSet, significantly reducing Kangaroo's ALWA.

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

On top of this basic design, Kangaroo introduces three techniques to minimize DRAM usage, minimize flash writes, and reduce cache misses. *(i)* Kangaroo's *partitioned index* for KLog can efficiently find all objects in KLog mapping to the same set in KSet, and is split into many independent partitions to minimize DRAM usage. *(ii)* Kangaroo's *threshold admission* policy between KLog and KSet only admits objects to KSet when at least n objects in KLog map to the same set, reducing ALWA by $\geq n\times$. *(iii)* Kangaroo's *"RRIParoo" eviction* improves hit ratio in KSet by approximating RRIP [41], a state-of-the-art eviction policy, while only using a single bit of DRAM per object.

4 THEORETICAL FOUNDATIONS OF KANGAROO

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

Assumptions. For tractability, this analysis makes several simplifying assumptions that do not hold in our design (Section 5), implementation (Section 6), or evaluation (Section 7). We assume the **independent reference model (IRM)**, in which each object has a fixed reference popularity, drawn independently from a known object probability distribution. However, we make no assumptions about the object popularity distribution; our model holds across any popularity distribution

Table 1. Key Variables in the Markov Model

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

(uniform, Zipfian, etc.). We also assume that all objects are fixed-size and that KSet uses FIFO eviction. Similar assumptions are common in prior cache models [11, 17, 19, 30, 31, 38, 42, 54, 58].

We model a cache consisting of two layers: a log-structured cache and a set-associative cache, called KLog and KSet as in Kangaroo — but note that the model simplifies Kangaroo's operation significantly. We assume that an object is first admitted to KLog. Once KLog fills up, it flushes all objects to KSet. KLog and KSet may employ an admission policy that drops objects instead of admitting them, as described below. Our goals are *(i)* to compute the miss probability and flash writes per cache access and *(ii)* to show that Kangaroo improves miss ratio for a given write rate vs. the baseline set-associative cache.

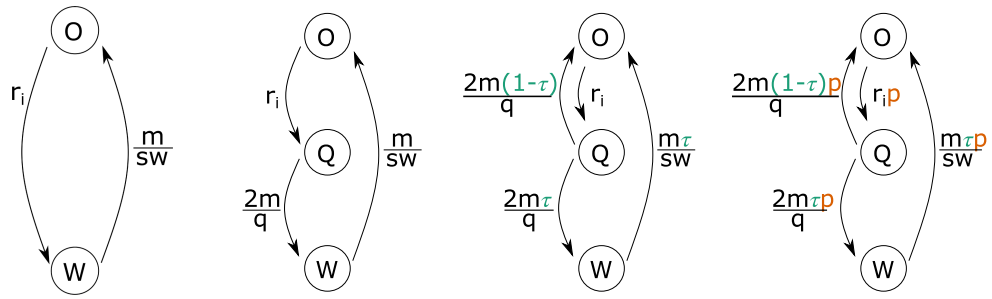
Modeling approach. We model how a single object moves through KLog and KSet. Figure 5 shows our simple continuous-time Markov model, which has three states: an object can be out-of-cache (O), in KLog (Q), or in KSet (W). To compute the miss probability m, we need to know each object's probability of being requested, which is fixed according to the IRM, and the probability that it is out-of-cache (in state O). To find the latter, we need to know each state's stationary probabilities, π, i.e., the likelihood of an object being in a given state once the cache reaches its steady-state behavior. To compute these probabilities and to find flash write rate, we require the transition rates between states, e.g., how often an object transitions $O \rightarrow Q$ when an object is admitted to KLog. Table 1 summarizes the key variables in the model.

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

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

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

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



(a) Baseline. *(b) + KLog.* *(c) + Threshold.* *(d) + Probabilistic admission.*

Fig. 5. The continuous Markov model for Kangaroo's basic cache design (a) with no log, (b) with no admission policies, (c) with Kangaroo's threshold admission before KSet, and (d) with probabilistic admission before KLog.

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

4.1 Baseline Set-associative Cache

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

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

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

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

$$1 = \pi_{i,O} + \pi_{i,W} \tag{2}$$

$$r_i \, \pi_{i,O} = \frac{m}{s \, w} \pi_{i,W} \tag{3}$$

which means that the stationary probabilities are:

$$\pi_{i,O} = \frac{m}{m + s \, w \, r_i} \tag{4}$$

$$\pi_{i,W} = \frac{s \, w \, r_i}{m + s \, w \, r_i} \tag{5}$$

Miss ratio: The miss ratio is computed by summing the probability that an object will miss when it is requested for all objects. Object i is requested with probability r_i and misses when it is

out-of-cache with probability $\pi_{i,O}$. Hence, the overall miss ratio ratio m is:

$$m_{\text{baseline}} = \sum_i r_i \, \pi_{i,O} = \sum_i \frac{m \, r_i}{m + s \, w \, r_i} \tag{6}$$

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

Flash writes: To compute the flash write rate, we assign a write-cost to each edge in Figure 5. For the baseline set-associative cache, each transition $O \rightarrow W$ re-writes the entire set, and so the transition has a write-cost of w. Transitions $W \rightarrow O$ do not write anything to flash, and so they have no write-cost. The flash write rate f is the average bytes written to flash on each access; that is, we compute write rate in logical time. In the baseline design, this is:

$$f_{\text{baseline}} = \sum_i r_i \cdot \pi_{i,O} \cdot w = w \cdot m_{\text{baseline}} \tag{7}$$

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

$$\text{ALWA}_{\text{baseline}} = \frac{f_{\text{baseline}}}{m_{\text{baseline}}} = w, \tag{8}$$

which matches our expectations for set-associative designs, since w is just the size of each set (Section 2.3).

4.2 Add KLog, No Admission Policies

Next we add KLog, a log-structured cache, in front of KSet, as shown in Figure 5(b). KLog's operation in our simple model is to buffer objects until full, and then flush the log's contents to KSet.

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

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

Stationary probabilities:

$$\pi_{i,O} = \frac{2m}{q \, r_i + 2m + 2s \, w \, r_i} \approx \frac{m}{m + s \, w \, r_i} \tag{9}$$

$$\pi_{i,Q} = \frac{q \, r_i}{q \, r_i + 2m + 2s \, w \, r_i} \tag{10}$$

$$\pi_{i,W} = \frac{2 \, s \, w \, r_i}{q \, r_i + 2m + 2s \, w \, r_i} \tag{11}$$

The approximation for $\pi_{i,O}$ holds when $q \ll sw$ (i.e., when KLog is much smaller than KSet). We find that Equation (9) is the same as Equation (4), demonstrating that adding KLog does not significantly affect the probability an object is out-of-cache, so long as KLog is small.

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

$$m_{\text{KLog-only}} = \sum_i r_i \, \pi_{i,O} \tag{12}$$

$$= \sum_i r_i \times \frac{2m}{qr_i + 2m + 2swr_i} \tag{13}$$

$$\approx \sum_i r_i \times \frac{m}{m + swr_i} \tag{14}$$

$$= m_{\text{baseline}} \tag{15}$$

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

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

$$f_{\text{KLog-only}} = \sum_i r_i \cdot \pi_{i,O} \cdot 1 + \frac{2m}{q} \cdot \pi_{i,Q} \cdot \frac{w}{\mathbb{E}[X|X \geq 1]} \tag{16}$$

Which means every object suffers write amplification of:

$$\text{ALWA}_{\text{KLog-only}} = 1 + \frac{w}{\mathbb{E}[X|X \geq 1]} \tag{17}$$

Deriving Equation (17) in detail:

$$f_{\text{KLog-only}} = \sum_i \frac{r_i \cdot 2m \cdot 1 + \frac{2m}{q} \cdot qr_i \cdot \frac{w}{\mathbb{E}[X|X \geq 1]}}{qr_i + 2m + 2swr_i}$$

$$= \left(1 + \frac{w}{\mathbb{E}[X|X \geq 1]}\right) \times \left(\sum_i \frac{2mr_i}{qr_i + 2m + 2swr_i}\right)$$

$$= \left(1 + \frac{w}{\mathbb{E}[X|X \geq 1]}\right) \times m_{\text{KLog-only}}$$

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

4.3 Add Threshold Admission Before KSet

Next, we consider the impact of adding Kangaroo's threshold admission policy (Section 5.3), which only admits objects to a set in KSet if at least n objects map to that set. For instance, a threshold of 2 means that if KLog has only one object mapping to a set, then that object is discarded (evicted) instead of being inserted into KSet. To represent threshold admission, we add an edge in the Markov model (Figure 5(c)) back from $Q \to O$, denoting the discarded objects.

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

objects are admitted to KSet if there are greater than n:

$$\tau(n) = \frac{\overline{F}_X(n)}{\overline{F}_X(1)}, \tag{18}$$

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

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

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

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

$$1 = \pi_{i,O} + \pi_{i,Q} + \pi_{i,W} \tag{19}$$

$$r_i\,\pi_{i,O} = \frac{m\,\tau}{s\,w}\,\pi_{i,W} + \frac{2\,m\,(1-\tau)}{q}\,\pi_{i,Q} \tag{20}$$

$$\frac{m\,\tau}{s\,w}\,\pi_{i,W} = \frac{2\,m\,\tau}{q}\,\pi_{i,Q} \tag{21}$$

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

$$m_{\text{threshold}} = m_{\text{KLog-only}} \approx m_{\text{baseline}} \tag{22}$$

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

$$f_{\text{threshold}} = \sum_i r_i \cdot \pi_{i,O} \cdot 1 + \frac{2\,m\,\tau}{q} \cdot \pi_{i,Q} \cdot \frac{w}{\mathbb{E}\left[X | X \geq n\right]} \tag{23}$$

$$\text{ALWA}_{\text{threshold}} = 1 + \frac{w}{\mathbb{E}\left[X | X \geq n\right]} \cdot \tau \tag{24}$$

This formula is derived similar to Equation (17) above. Note that the ALWA can be easily read off from Figure 5 at a glance by "following the write loop" from O back to O, adding up write costs for each edge and scaling them by their transition rate relative to KLog-only, e.g., by a factor of τ for the second term.

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

4.4 Add Probabilistic Admission Before KLog

The above techniques — KLog and threshold admission — are Kangaroo's main tricks to reduce flash writes. However, the design thus far always has write amplification at least 1× because all objects are admitted to KLog. It is possible to achieve write amplification below 1× by adding an admission policy in front of KLog. We now consider the effect of adding a probabilistic admission policy that drops objects with a probability p before they are admitted to KLog, as shown in Figure 5(d).

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

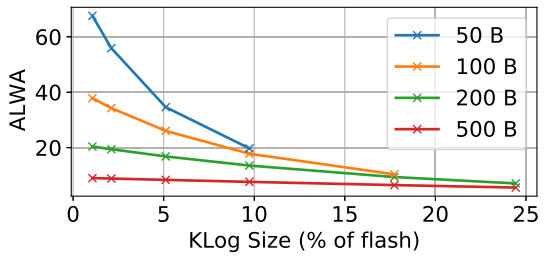


Fig. 6. Modeled ALWA for Kangaroo with different percentages of the flash cache devoted to KLog, assuming 4 KB sets, 100% probabilistic admission, and no admission threshold.

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

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

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

$$f_{\text{Kangaroo}} = \sum_i p\, r_i \cdot \pi_{i,O} \cdot 1 + \frac{2\,m\,p\,\tau}{q} \cdot \pi_{i,Q} \cdot \frac{w}{\mathbb{E}\left[X|X \geq n\right]} \tag{25}$$

$$\text{ALWA}_{\text{Kangaroo}} = p\left(1 + \frac{w}{\mathbb{E}\left[X|X \geq n\right]} \cdot \tau\right) \tag{26}$$

This equation is Theorem 1, as expected.

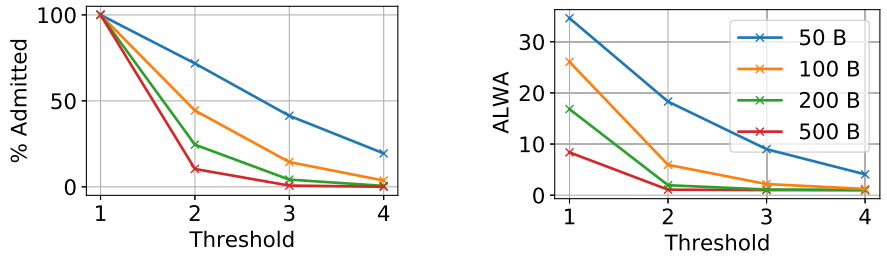
4.5 Modeling Results

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

Figure 6 shows the effect of KLog's size on ALWA without an admission policy. As Kangaroo devotes more space to KLog, its ALWA greatly decreases. For instance, doubling the percent of the cache dedicated to KLog from 2.5% to 5% decreases ALWA by up to 38%. Since the overall amount of flash dedicated to KLog is small, there is only a small increase in DRAM overhead for this configuration of Kangaroo. Thus, even a small KLog greatly decreases ALWA.

Figure 7 shows the effect of thresholding on ALWA and KSet's admission probability for different object sizes using Theorem 1, keeping KLog at 5% of cache size. With no thresholding ($n = 1$), no objects are rejected; but as the threshold increases more objects are rejected (Figure 7(a)). Also, since more objects fit in the KLog when objects are smaller, smaller objects are more likely to be admitted. Thresholding significantly reduces ALWA (Figure 7(b)). Importantly, the ALWA savings are larger than the fraction of objects rejected, unlike purely probabilistic admission. For instance, with 100 B objects, threshold $n = 2$ admits 44.4% of objects, but its write rate is only 22.8% of the write rate with threshold $n = 1$.

The ALWA results are consistent with results in our experiments (Section 7), giving us confidence when using the model to explore the ALWA design space.



(a) Percent of objects admitted. *(b) ALWA.*

Fig. 7. Modeled (a) admission percentage and (b) ALWA for Kangaroo with different threshold values and object sizes, assuming 4 KB sets and KLog w/ 5% of capacity.

5 KANGAROO DESIGN

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

5.1 Pre-flash Admission to KLog

Like previous flash caches, Kangaroo may not admit all objects evicted from the DRAM cache [16, 28, 29, 34–36]. It has a pre-flash admission policy that can be configured to randomly admit objects to KLog with probability p, decreasing Kangaroo's write rate proportionally without additional DRAM overhead. Compared to prior designs, Kangaroo can afford to admit a larger fraction of objects to flash than prior flash caches due to its low ALWA; in fact, except at very low write budgets, Kangaroo admits almost all objects to KLog.

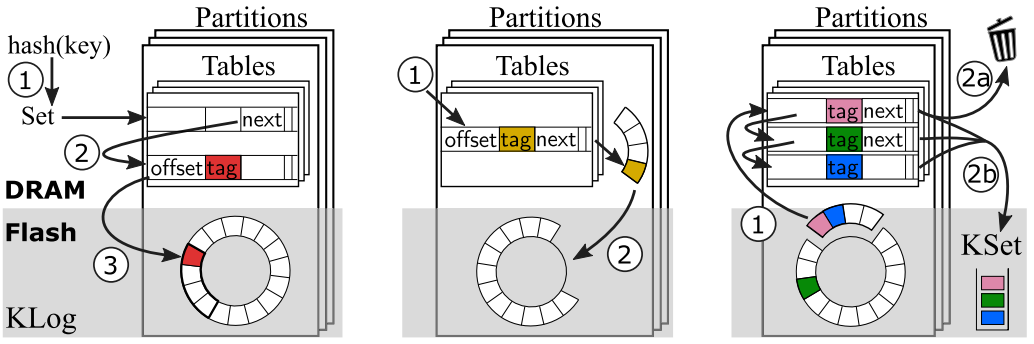
5.2 KLog

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

Operation. Like other log-structured caches, KLog writes objects to a circular buffer on flash in large batches and tracks objects via an index kept in DRAM. To support ENUMERATE-SET efficiently, KLog's index is implemented as a hash table using separate chaining. Each index entry contains an `offset` to locate the object in the flash log, a `tag` (partial hash of the object's key), a `next-pointer` to the next entry in the chain (for collision resolution), eviction-policy metadata (described in Section 5.4), and a `valid` bit.

LOOKUP: To look up a key (Figure 8(a)), ① KLog determines which bucket it belongs to by *computing the object's set* in KSet. ② KLog traverses index entries in this bucket, ignoring `invalid` entries, until a `tag` matches a hash of the key. If there is no matching `tag`, KLog returns a miss. ③ KLog reads the flash page at `offset` in the log. After confirming a full key match, KLog returns the data and updates eviction-policy metadata.

INSERT: To insert an object (Figure 8(b)), ① KLog creates an index entry, adds it to the bucket corresponding to the key's set in KSet, and appends the object to an in-DRAM buffer. The on-flash circular log is broken into many *segments*, one of which is buffered in DRAM at a time. ② Once the segment buffer is full, it is written to flash.



(a) KLog lookup. *(b) KLog insertion.* *(c) KLog eviction.*

Fig. 8. Overview of KLog operations.

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

Internal KLog structure. As depicted in Figure 8, KLog is structured internally as multiple *partitions*. Each partition is an independent log-structured cache with its own flash log and DRAM index. Moreover, each partition's index is split into multiple *tables*, each an independent hash table.

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

Reducing DRAM usage in KLog. Table 2 breaks down Kangaroo's DRAM usage per object vs. a naïve log-structured design as a standalone cache ("Naïve Log-Only") and as a drop-in replacement for KLog ("Naïve Kangaroo").

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

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

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

In a naïve cache using LRU eviction, each entry keeps a pointer to adjacent entries in the LRU list. This requires $2{\cdot}\log_2(LogSize/ObjectSize)$ bits. In contrast, Kangaroo's RRIParoo policy (Section 5.4) is based on RRIP [41] and only needs 3 b per object in KLog (and even less in KSet).

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

Table 2. Breakdown of DRAM Per Object for a 2 TB Cache, Comparing Kangaroo to a Naïve Log-structured Cache and Kangaroo with a Naïve Log Index

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

Bucket and LRU overhead assume 200 B objects.

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

All told, KLog's partitioned structure reduces the per-object metadata from 190 b to 48 b per object, a 3.96× savings vs. the naïve design. Compared to prior index designs, KLog's index uses slightly more DRAM per object than the state-of-the-art (30 b per object in Flashield [34]), but it supports Enumerate-Set and has fewer false positives. Most importantly, *KLog only tracks ≈5% of objects in Kangaroo, so indexing overheads are just 3.2 b per object.* Adding KSet's DRAM overhead gives a total of 7.0 b per object, a 4.3× improvement over the state-of-the-art.

5.3 KLog → KSet: Minimizing Flash Writes

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

Moving objects from KLog to KSet. A background thread keeps one segment free in each log partition. This thread flushes segments from the on-flash log in FIFO order, moving objects from KLog to KSet as shown in Figure 8(c). For each victim object in the flushed segment, this thread ① calls Enumerate-Set to find all other objects in KLog that should be moved with it; (2a) if there are not enough objects to move (see below), the victim object is dropped or, if popular, is re-admitted to KLog; (2b) otherwise, the victim object and all other objects returned by Enumerate-Set are moved from KLog to KSet in a single flash write.

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

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

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

5.4 KSet

KSet's role is to minimize the overall DRAM overhead of the cache. KSet employs a set-associative cache design similar to CacheLib's Small Object Cache [16]. This design splits the cache into many *sets*, each holding multiple objects; by default, each set is 4 KB, matching flash's read and write granularity. KSet maps an object to a set by hashing its key. Since each object is restricted to a small number of locations (i.e., one set), an index is not required. Instead, to look up a key, KSet simply reads the entire set off flash and scans it for the requested key.

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

RRIParoo: Usage-based eviction without a DRAM index. Usage-based eviction policies can significantly improve miss ratio, effectively doubling the cache size (or more) without actually adding any resources [12, 19, 20, 41, 62]. Unfortunately, implementing these policies on set-associative flash caches is hard, as such policies involve per-object metadata that must be updated whenever an object is accessed. Since KSet has no DRAM index to store metadata and cannot update on-flash metadata without worsening ALWA, it is not obvious how to implement a usage-based eviction policy. For these reasons, most flash caches use FIFO eviction [2, 5, 7, 8, 16, 25, 37, 39, 60, 67], which keeps no per-object state. Unfortunately, FIFO significantly increases miss ratio because popular objects continually cycle out of the cache.

Kangaroo introduces RRIParoo, a new technique to efficiently support usage-based eviction policies in flash caches. Specifically, RRIParoo implements RRIP [41], a state-of-the-art eviction policy originally proposed for processor caches, while using only ≈1 bit of DRAM per object and without any additional flash writes.

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

RRIParoo's key ideas. There are two ideas to support RRIP in KSet. First, rather than tracking all of RRIP's predictions in a DRAM index, RRIParoo stores the eviction metadata in flash and keeps only a small portion of it in DRAM. Second, to reduce DRAM metadata to a single bit, we observe that RRIP only updates predictions upon eviction (incrementing predictions towards FAR) and when an object is accessed (promoting to NEAR). Our insight is that, so long as KSet tracks which objects are accessed, *promotions can be deferred to eviction time*, so that *all* updates to on-flash RRIParoo

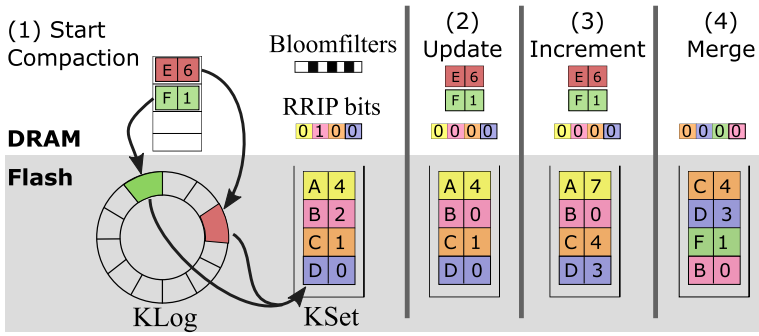


Fig. 9. RRIParoo implements RRIP eviction with only ≈1 b in DRAM per object and no additional flash writes.

metadata are only made at eviction, when the set is being re-written anyway. Hence, since KSet can track whether an object has been accessed using only single bit in DRAM, KSet achieves the hit-ratio of a state-of-the-art eviction policy with one-third of the DRAM usage (1 b vs. 3 b).

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

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

Example: Figure 9 illustrates this procedure, showing how a set is re-written in KSet. ① We start when KLog flushes a segment containing object **F**, which maps to a set in KSet. KLog finds a second object, **E**, elsewhere in the log that also maps to this set. Meanwhile, the set contains objects **A**, **B**, **C**, and **D** with the RRIP predictions shown on flash, and **B** has received a hit since the set was last re-written, as indicated by bits in DRAM. ② Since **B** received a hit, we promote its RRIP prediction to NEAR and clear the bits in DRAM. ③ Since no object is at FAR, we increment all objects' predictions by 3, whereupon object **A** reaches FAR. ④ Finally, we fill up the set by merging objects in DRAM and flash in prediction-order until the set is full. The set now contains **B**, **F**, **D**, and **C**; **A** was evicted, and **E** stays in KLog for now since its KLog segment was not flushed. (The set on flash is only written once, after the above procedure completes.)

DRAM usage. As shown in Table 2, KSet needs up to 4 bits in DRAM per object: one for RRIParoo and three for the Bloom filters. Combined with the DRAM usage of KLog that contains about 5% of objects, Kangaroo needs ≈7.0 b per object, 4.3× less than Flashield [34]. Moreover, the 1 b per object DRAM overhead for RRIParoo can be lowered by tracking fewer objects in each set. Taken to the extreme, this would cause the eviction policy to decay to FIFO, but it allows Kangaroo to adapt to use less DRAM if desired.

6 EXPERIMENTAL METHODOLOGY

This section describes the experimental methodology that we use to evaluate Kangaroo in Section 7.

Table 3. Kangaroo's Default Parameters

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

6.1 Kangaroo Implementation and Parameterization

We implement Kangaroo in C++ as a module within the CacheLib caching library [16]. Table 3 describes Kangaroo's default parameters; we evaluate sensitivity to these parameters in Section 7.3.

6.2 Comparisons

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

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

Except where noted, all experiments are configured to stay within 16 GB of DRAM (all-inclusive — DRAM cache, index, etc.); 62.5 MB/s flash writes, as measured directly from the device (i.e., including DLWA); and 100 K requests/s, similar to what is achieved by flash caches in production [16, 22]. To mimic a memory-constrained system, we limit LS's flash capacity to the maximum allowed by a 16 GB index assuming 30 b/object, the best reported in the literature [34], but also grant LS an *additional* 16 GB for its DRAM cache. Note that this is optimistic for LS, as DRAM is LS's main constraint. We use this variant as we were unable to compare to state-of-the-art systems: the source code of Flashield [34] is not available, and we were unable to run FASTER [25] as a cache. All systems use CacheLib's probabilistic pre-flash admission policy.

6.3 Simulation

To explore a wide range of parameters and constraints, we implemented a trace-driven cache simulator for Kangaroo. The simulator measures miss ratio and application-level write rate. We estimate device-level write amplification based on our results in Section 2, using a best-fit exponential curve to the DLWA of random, 4 KB writes for SA and Kangaroo, and assuming a DLWA of 1× (no write amplification) for LS. Note that this is pessimistic for Kangaroo, since writes to KLog incur less DLWA than SA. Comparing the results with our experimental data shows the simulator to be accurate within 10%. The simulator does not implement some features of CacheLib including promotion to the memory cache, which can affect miss ratios, but we have found it able to give a good indication of how the full system would perform as parameters change.

Table 4. Key Parameters in Scaling Methodology

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

6.4 Workloads

Our experiments use sampled 7-day traces from Meta [16] and Twitter [73]. These traces have average object sizes of 291 B and 271 B, respectively. For systems experiments, we scale the Meta trace to achieve 100 K reqs/s by running it 3× concurrently in different key spaces.

The simulator results use sampled-down traces, and we scale-up the measurements to a full-server equivalent based on the server's flash capacity and desired load as described below, Section 6.6. We use 1% of the keys for the Meta trace and 10% of the keys for the Twitter trace. Unless otherwise noted, we report numbers for the last day of requests, allowing the cache to warm up and display steady-state behavior.

6.5 Metrics

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

6.6 Scaling Traces

Our scaling methodology allows us explore a wide range of system parameters in simulation. This methodology builds on prior analysis of scaling caches [13, 14, 43, 51, 59, 68], and Table 4 summarizes its key parameters.

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

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

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

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

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

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

$$R_m = \ell \cdot R_o, \tag{27}$$

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

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

$$\ell_{max} = R_{max}/R_o. \tag{28}$$

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

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

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

$$W_m \propto m_m(S_m) \cdot R_m. \tag{29}$$

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

The maximum cache size S_{max} is determined from the flash-write constraint, D_{max}. Specifically, we multiply the application-level flash write rate W_m by the estimated DLWA to get the device-level write rate D_m. We then sweep the flash cache size S_m to find the sizes that stay within the constraint. Increasing cache size has two competing effects on write rate: larger caches generally have fewer misses, leading to fewer insertions, but they also suffer higher DLWA, increasing the cost of each insertion. As a result, the maximum size usually lies on the "knee" of the DLWA curve (see Figure 2), though which size hits the knee depends on the cache design (via ALWA), the load factor (via R_m), and the trace itself (via m_m).

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

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

$$k = R_s/R_o \tag{30}$$

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

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

$$S_s = k \cdot S_m. \tag{31}$$

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

$$W_m = W_s/k. \tag{32}$$

However, simulated miss ratio does not change

$$m_m(S_m) = m_m(S_s/k) = m_s(S_s). \tag{33}$$

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

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

$$Q_s = \frac{Q_m S_s}{S_m} \tag{34}$$

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

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

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

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

$$S_m = \frac{Q_m S_s}{Q_s}. \tag{35}$$

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

$$\ell = \frac{R_m}{R_s} k = \frac{S_m}{S_s} k, \tag{36}$$

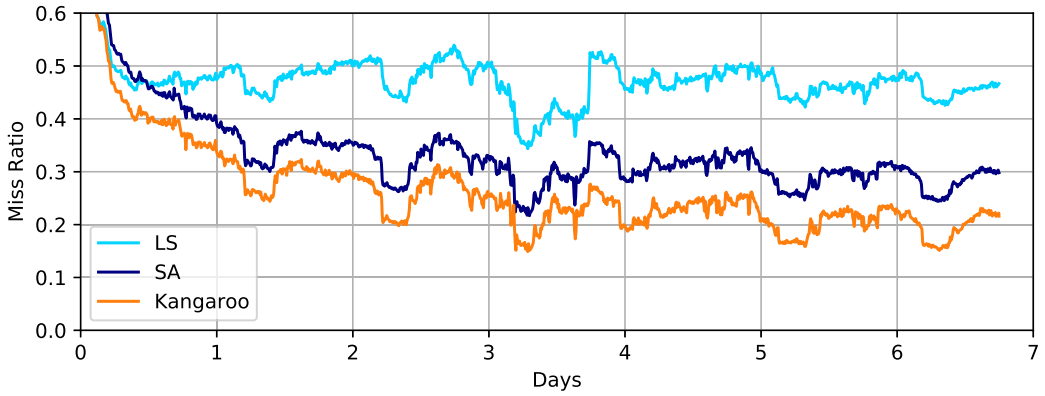


Fig. 10. Miss ratio for all three systems over a 7-day Meta trace. All systems are run with 16 GB DRAM, a 1.9 TB drive, and with write rates less than 62.5 MB/s.

yielding a modeled request rate of

$$R_m = \frac{S_m}{S_s} R_s \tag{37}$$

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

$$D_m = \text{DLWA}(S_m) \cdot \frac{W_s}{k}. \tag{38}$$

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

7 EVALUATION

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

7.1 Main Result: Kangaroo Significantly Reduces Misses vs. Prior Cache Designs Under Realistic Constraints

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

Miss ratio: Figure 10 shows that Kangaroo reduces cache misses by 29% vs. SA and by 56% vs. LS. This is because *Kangaroo makes effective use of both limited DRAM and flash writes,* whereas prior designs are hampered by one or the other. Specifically, SA is limited primarily by its high write rate, which forces it to admit a lower percentage of objects to flash and to over-provision flash to reduce device-level write amplification. SA uses only 81% of flash capacity. Similarly, LS is limited by the reach of its DRAM index. LS warms up as quickly as Kangaroo until it runs out of indexable

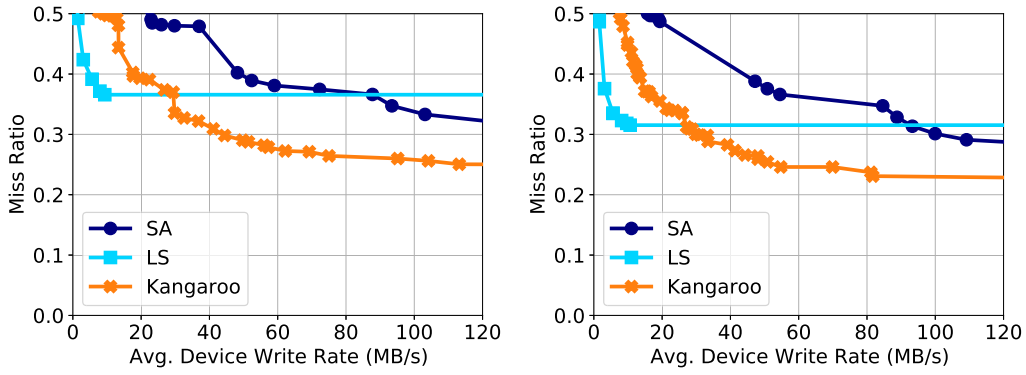


Fig. 11. Pareto curve of cache miss ratio at different device-level write rates under 16 GB DRAM and 2 TB flash capacity. At very low write rates, LS is best, but it is limited by DRAM from scaling to larger caches. Thus, for most write rates, Kangaroo outperforms both systems because it can take advantage of the entire cache capacity, has a lower write rate than SA, and has a better eviction policy than the other two systems.

flash capacity at 61% of device capacity, even though we provision LS extra DRAM for both an index and DRAM cache (Section 6.2).

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

Request latency and throughput: Kangaroo achieves reasonable throughput and tail latencies. When measuring flash cache performance without a backing store, Kangaroo's peak throughput is 158 K gets/s, LS's is 172 K gets/s, and SA's is 168 K gets/s. Kangaroo achieves 94% of SA's throughput and 91% of LS's throughput, and it is well above typical production request rates [16, 22, 69].

In any reasonable caching deployment, request tail latency will be set by cache misses as they fetch data from backend systems. However, for completeness and to show that Kangaroo has no performance pathologies, we present tail latency at peak throughput. Kangaroo's p99 latency is 736 µs, LS's is 229 µs, and SA's is 699 µs. All of these latencies are orders-of-magnitude less than typical SLAs [1, 3, 22, 75], which are set by backend databases or file systems. For instance, in production, the p99 latency in Facebook's social-graph cache is 51 ms and Twitter's is 8 ms, both orders-of-magnitude larger than Kangaroo's p99 latency. In addition, Kangaroo might reduce p99 latency in practice because its improved hit ratio reduces load on backend systems.

7.2 Kangaroo Performs Well as Constraints Change

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

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

Figure 11 shows the tradeoff between device-level write rate budget and miss ratio. At 62.5 MB/s (the default) on both the Meta and Twitter workloads, Kangaroo consistently performs better than

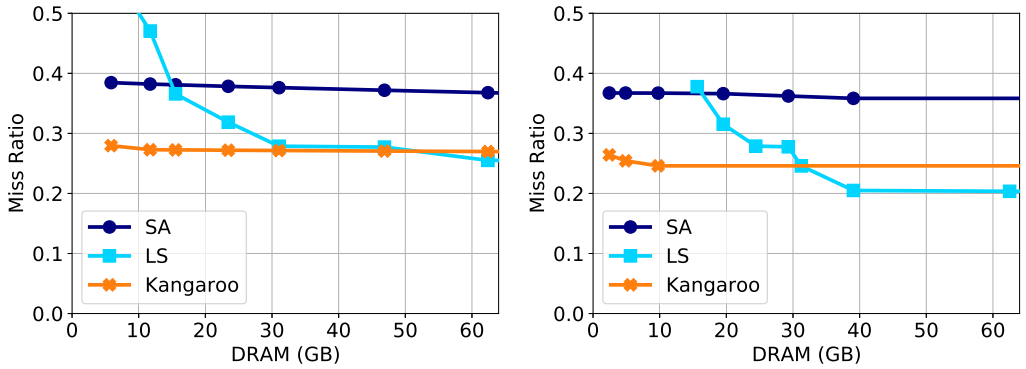


Fig. 12. Pareto curve of cache miss ratio as DRAM capacity varies from 5 to 64 GB. Flash capacity is fixed at 2 TB and device write rate is capped at 62.5 MB/s. The amount of DRAM does not greatly affect SA or Kangaroo, which are both write-rate-constrained, but has a huge effect on LS by increasing its cache size.

both SA and LS. At higher write budgets, Kangaroo continues to have lower miss ratio. In this range, SA suffers both due to its FIFO eviction policy and its higher ALWA, which shifts points farther right vs. similar Kangaroo configurations. LS is mostly constrained by DRAM capacity, which is why its achievable miss ratio does not change above 15 MB/s for both traces. However, at very low device-level write budgets, LS performs better than Kangaroo. This is because Kangaroo is designed to balance DLWA and DRAM capacity, whereas LS focuses only on DLWA. At extremely low write budgets, Kangaroo's higher DLWA (in KSet) forces it to admit fewer objects. (Kangaroo configurations where KLog holds a large fraction of objects, which we did not evaluate, would solve this problem.)

DRAM capacity. Over time, the typical ratio of DRAM to flash in data-center servers is decreasing in order to reduce cost [64]. Figure 12 compares miss ratios for DRAM capacities up to 64 GB, holding flash-device capacity fixed at 2 TB and device-write rate at 62.5 MB/s. DRAM capacity does not greatly affect SA. Larger DRAM capacity allows Kangaroo to use a larger log. Even so, both of these systems are mainly constrained by device-level write rate. By contrast, LS is very dependent on DRAM capacity. LS approaches, though does not reach, Kangaroo's miss ratio at 64 GB of DRAM on the Meta trace and at 40 GB on the Twitter trace. At this point, Kangaroo is constrained from reducing misses further by device write rate (see Figure 11).

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

Flash-device capacity. As stated in the previous section, the size of the flash device greatly impacts miss ratio and the significance of write-rate and DRAM constraints. As we look forward, flash devices are likely to increase while DRAM capacity is unlikely to grow much and may even shrink [64]. Figure 13 shows the miss ratio for each system as the device capacity changes. Each system can use as much of the device capacity as it desires while staying within 16 GB DRAM and 3 device writes per day.

Except at smaller flash capacities, Kangaroo is Pareto-optimal across device capacities. At smaller device capacities (<1.2 TB for the Meta trace and <1 TB for the Twitter trace), Kangaroo and SA are increasingly write-rate-limited while LS is *de*creasingly DRAM-limited. However, as flash capacity increases, LS is quickly constrained by DRAM capacity. In contrast, Kangaroo and

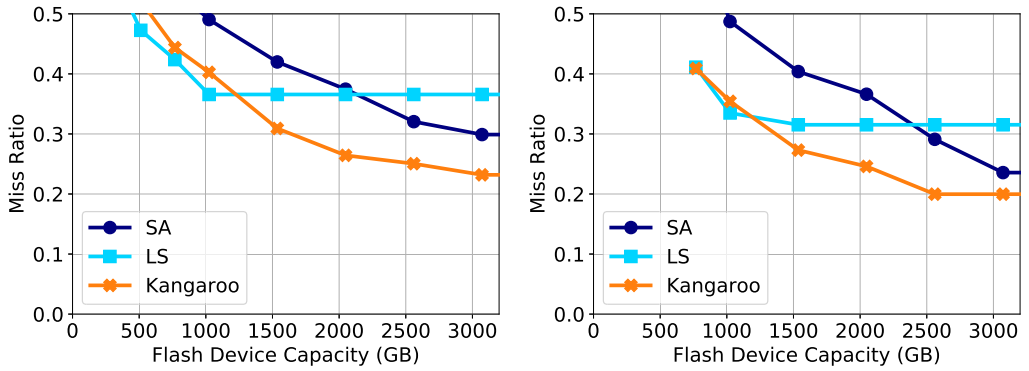


Fig. 13. Pareto curve of cache miss ratio at different device sizes. The DRAM capacity is limited to 16 GB and the device write rate to 3 device writes/day (e.g., 62.5 MB/s for a 2 TB drive).

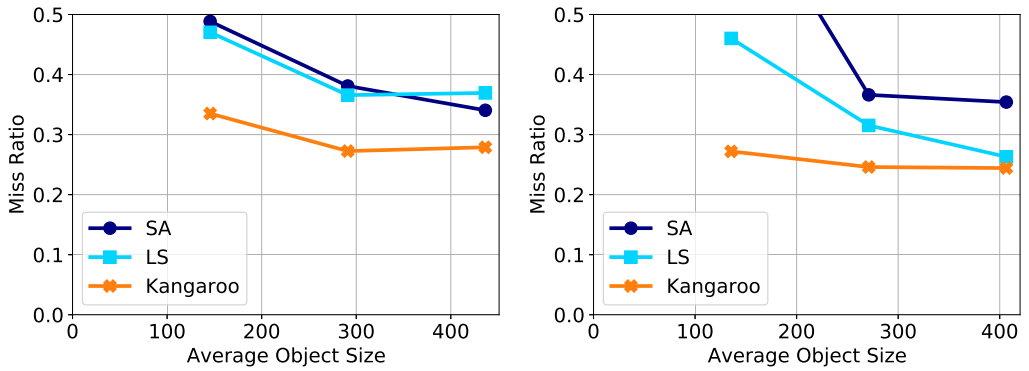


Fig. 14. Pareto curve of cache miss ratio vs. average object size. Object sizes are limited to [1 B, 2048 B]. The write rate is constrained to 62.5 MB/s for a 2 TB flash drive with 16 GB of DRAM.

SA both take advantage of the additional write budget and flash capacity. Kangaroo is consistently better than SA due to lower ALWA (allowing larger cache sizes) and RRIParoo.

Object size. The final feature that we study is object sizes. Figure 14 shows how miss ratio changes for each system as we artificially change the object sizes. For each object in the trace, we multiply its size by a scaling factor, but constrain the size to [1 B, 2 KB]. To study the impact of cache design as object sizes change, we keep the working-set size constant by scaling up the sampling rate (Section 6.6).

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

Kangaroo is also affected as objects get smaller, but not as much as prior designs. KSet's ALWA increases with smaller objects, but less than SA. For example, as avg object size goes from 500 B to 50 B, Kangaroo's ALWA increases by 4×, while SA's ALWA increases by 10× (Figure 7). Similarly, KLog uses more DRAM with smaller objects, but, unlike LS, Kangaroo can reduce DRAM usage

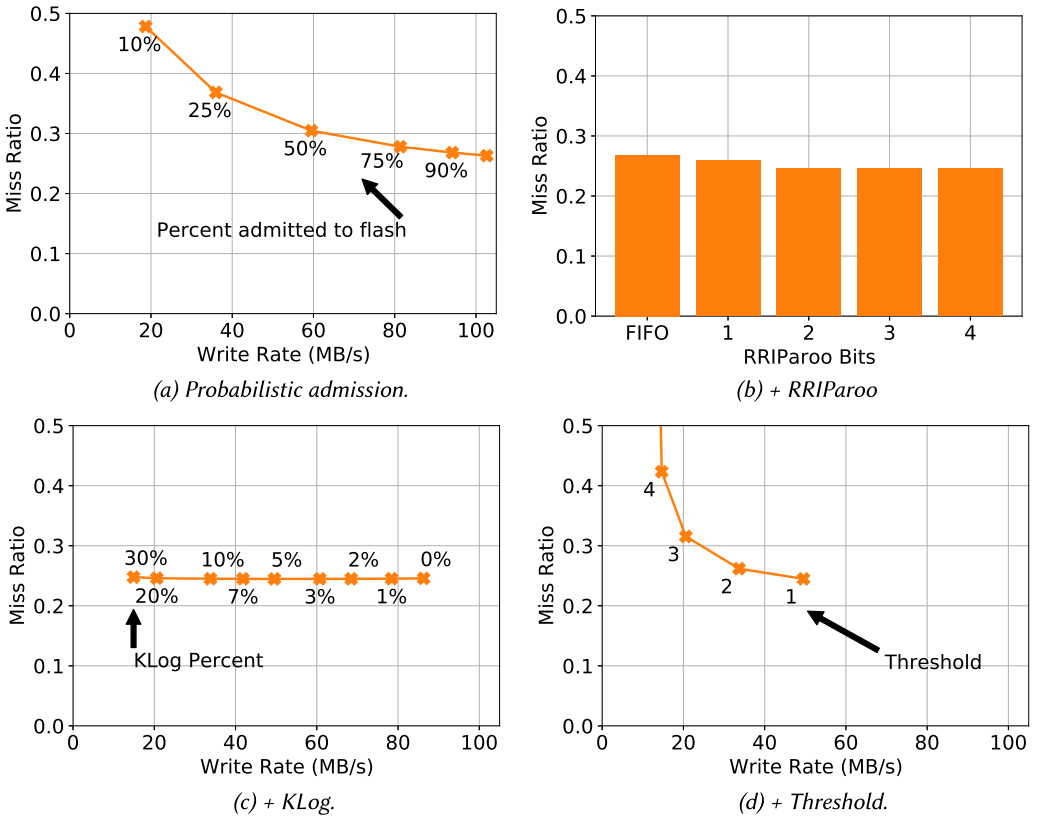


Fig. 15. Miss ratio vs. application-level write rate based on various design parameters in Kangaroo: (a) KLog admission probability, (b) RRIParoo metadata size, (c) KLog size (% of flash-device capacity), and (d) KSet admission threshold.

by decreasing KLog's size without decreasing overall cache size. The tradeoff is that ALWA increases slightly (see below). Kangaroo thus scales better than prior flash-cache designs as objects get smaller: on the Twitter trace, Kangaroo reduces misses by 7.1% vs. LS with 500 B avg object size and by 41% vs. LS with 50 B avg object size.

7.3 Parameter Sensitivity and Benefit Attribution

We now analyze how much each of Kangaroo's techniques contributes to Kangaroo's performance and how each should be parameterized. Figure 15 evaluates Kangaroo's sensitivity to four main parameters on the Meta trace: KLog admission probability, KSet eviction policy, KLog size, and KSet admission threshold. All setups use the full 2 TB device capacity and 16 GB of memory. We build up their contribution to miss ratio and application write rate from a basic set-associative cache with FIFO eviction.

Pre-flash admission probability. Figure 15(a) varies admission probability from 10% to 100%. As admission probability increases, write rate increases because more objects are written to flash. However, the miss ratio does not decrease linearly with admission probability. Rather, it has a smaller effect when the admission percentage is high than when the admission percentage is low. Since Kangaroo's other techniques significantly reduce ALWA, we use a pre-flash admission probability of 90%.

Number of RRIParoo bits. Figure 15(b) shows miss ratios for FIFO and RRIParoo with one to four bits. Although changing the eviction policy does slightly change the write rate (because there are fewer misses), we show only miss ratio for readability. RRIParoo with one bit suffers 3.4% fewer misses vs. FIFO, whereas RRIParoo with three bits suffers 8.4% fewer misses. Once RRIParoo uses four bits, the miss ratio increases slightly, a phenomenon also noticed in the original RRIP paper [41]. Since three-bit RRIParoo uses the same amount of DRAM as one-bit RRIParoo (Section 5.4), we use three bits because it achieves the best miss ratio.

KLog size. Figure 15(c) shows that, as KLog size increases, the flash write rate decreases significantly, but the miss ratio is unaffected (<.05% maximum difference). However, a bigger KLog needs more DRAM for its index. Thus, KLog should be as large enough to substantially reduce write amplification, but cannot exceed available DRAM nor prevent using a DRAM cache. Flash writes can be further reduced via admission policies or by over-provisioning flash capacity as needed. We use 5% of flash capacity for KLog.

KSet admission threshold. Figure 15(d) shows the impact of threshold admission to KSet. Thresholding reduces flash write rate up to 70.4% but increases misses by up to 72.9% at the most extreme. Note that these results assume rejected objects are re-admitted to KLog if they have been hit, since re-admission reduces misses without significantly impacting flash writes. We use a threshold of 2, which reduces flash writes by 32.0% while only increasing misses by 6.9%.

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

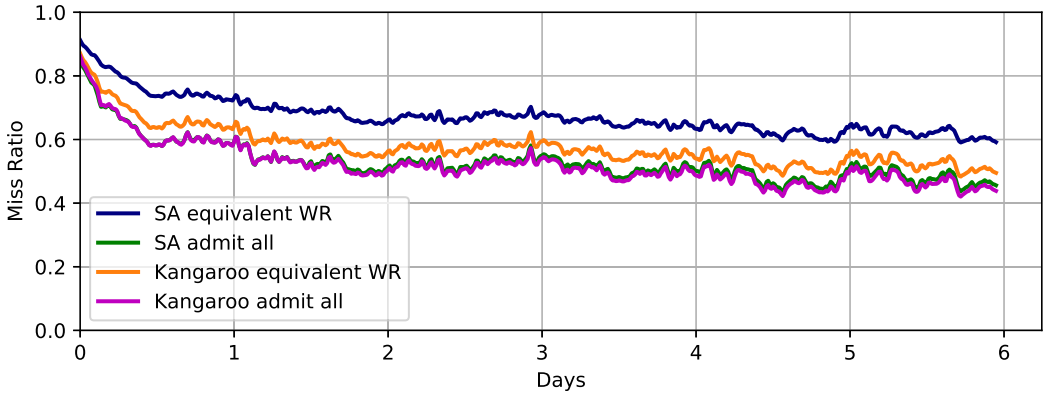
7.4 Production Deployment Test

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

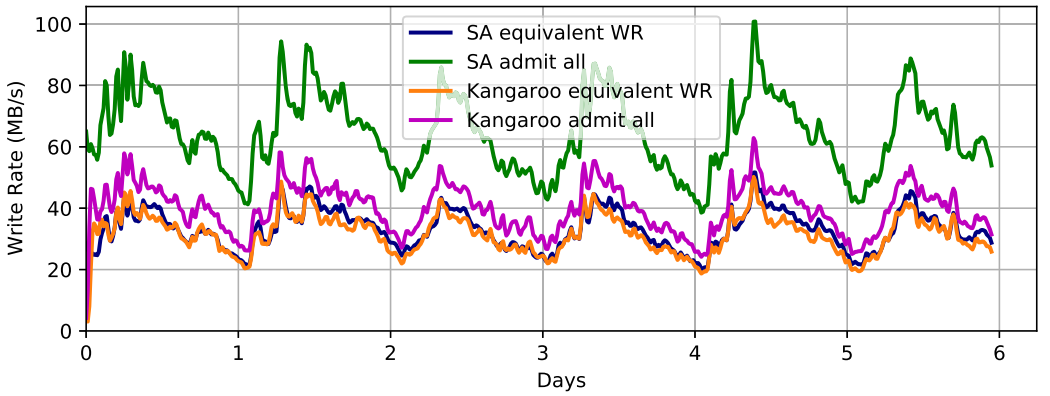
To find appropriate production configurations, we chose seven configurations for each system that performed well in simulation: four with probabilistic pre-flash admission and three with a **machine-learning (ML)** pre-flash admission policy. The first production deployment ran all probabilistic admission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

Figures 16(a) and 16(b) present results over a six-day request stream for configurations with similar write rates ("equivalent WA") as well as configurations that admit all objects to the flash cache ("admit-all"). Kangaroo reduces misses by 18% vs. SA in the equivalent-WA configurations, which both have similar write rates at ≈33 MB/s. The admit-all configurations achieve the best miss ratio for each system at the cost of additional flash writes. Here, Kangaroo reduces flash misses by 3% vs. SA while writing 38% less.

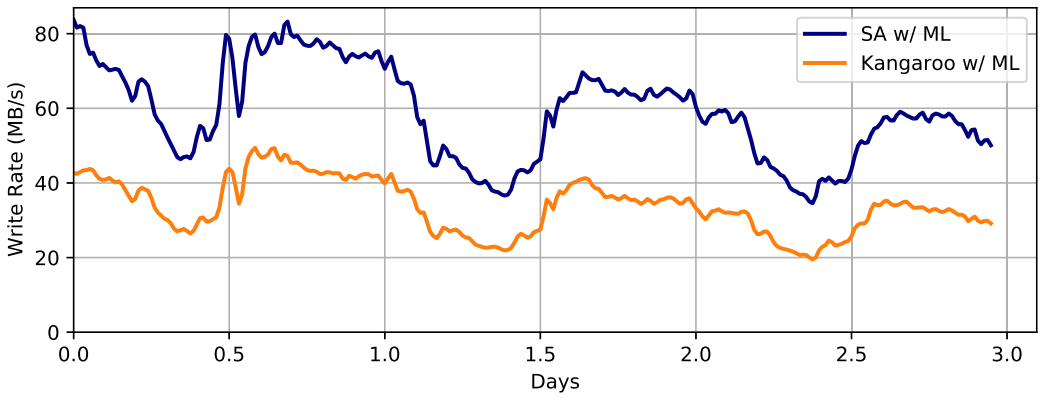
We also tested both systems with the ML pre-flash admission policy that Meta uses in production [16]. Figure 16(c) presents results over a three-day request stream. The trends are the same:



(a) Flash miss ratio.



(b) Application flash write rate.



(c) ML admission.

Fig. 16. Results from two production test deployments of Kangaroo and SA, showing (a) flash miss ratio and (b) application flash write rate over time using pre-flash random admission and (c) application flash write rate over time using ML admission. With random admission at equivalent write-rate, Kangaroo reduces misses by 18% over SA. When Kangaroo and SA admit all objects, Kangaroo reduces write rate by 38%. With ML admission, Kangaroo reduces the write rate by 42.5%.

Kangaroo reduces application flash writes by 42.5% while achieving a similar miss ratio to SA. Kangaroo thus significantly outperforms SA, independent of pre-flash admission policy.

8 CONCLUSION

Kangaroo is a flash cache for billions of tiny objects that handles a wide range of DRAM and flash-write budgets. Kangaroo leverages prior log-structured and set-associative designs, together with new techniques, to achieve the best of both designs. Experiments using traces from Meta and Twitter show DRAM usage close to the best prior DRAM-optimized design, flash writes close to the best prior write-optimized design, and miss ratios better than either. Kangaroo has been implemented in CacheLib [16] and is open-sourced for use by the community. Kangaroo shows that flash caches can support tiny objects, an adversarial workload for DRAM usage and write amplification, while maintaining flash's cost advantage.

ACKNOWLEDGMENTS

We thank Meta, Twitter, and the members and companies of the PDL Consortium (Alibaba, Amazon, Datrium, Google, HPE, Hitachi, IBM, Intel, Meta, Microsoft, NetApp, Oracle, Pure Storage, Salesforce, Samsung, Seagate, Two Sigma and Western Digital) and VMware for their interest, insights, feedback, and support.

REFERENCES

[1] [n. d.]. Amazon DynamoDB. https://aws.amazon.com/dynamodb/features/. 5/5/21.
[2] [n. d.]. Apache Traffic Server. https://trafficserver.apache.org. Accessed: 2019-04-22.
[3] [n. d.]. Azure Cache for Redis. https://azure.microsoft.com/en-us/services/cache/#what-you-can-build. 5/5/21.
[4] [n. d.]. Big Data Statistics, Growth, and Facts 2020. https://saasscout.com/statistics/big-data-statistics/. 5/6/21.
[5] [n. d.]. Fatcache. https://github.com/twitter/fatcache.
[6] [n. d.]. LevelDB. https://github.com/google/leveldb.
[7] [n. d.]. Redis on Flash. https://docs.redislabs.com/latest/rs/concepts/memory-architecture/redis-flash/.
[8] [n. d.]. RocksDB. http://rocksdb.org.
[9] 2020. Facebook reports first quarter 2020 results. *investor.fb.com* (Apr. 2020).
[10] 2021. Twitter first quarter 2021 results. *investor.twitterinc.com* (May 2021).
[11] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. 1971. Principles of optimal page replacement. *J. ACM* (1971).
[12] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving hit rate by maximizing hit density. In *USENIX NSDI*.
[13] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A simple way to remove cliffs in cache performance. In *IEEE HPCA*.
[14] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *IEEE HPCA*.
[15] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. 2019. Small refinements to the DAM can have big consequences for data-structure design. In *ACM SPAA*.
[16] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory G. Ganger. 2020. The CacheLib caching engine: Design and experiences at scale. In *USENIX OSDI*.
[17] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical bounds on optimal caching with variable object sizes. In *ACM SIGMETRICS*.
[18] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *USENIX OSDI*.
[19] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. 2017. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *USENIX NSDI*.
[20] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC*.
[21] Netflix Technology Blog. 2016. Application Data Caching Using SSDs. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef.

[22] Netflix Technology Blog. 2018. Evolution of Application Data Caching: From RAM to SSD. https://netflixtechblog.com/evolution-of-application-data-caching-from-ram-to-ssd-a33d6fa7a690.
[23] Simona Boboila and Peter Desnoyers. 2010. Write endurance in flash drives: Measurements and analysis. In *USENIX FAST*.
[24] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*.
[25] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: An embedded concurrent key-value store for state management. *VLDB* (2018).
[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
[27] Andromachi Chatzieleftheriou, Ioan Stefanovici, Dushyanth Narayanan, Benn Thomsen, and Antony Rowstron. 2020. Could cloud storage be disrupted in the next decade? In *USENIX HotStorage*.
[28] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*.
[29] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. 2017. Memshare: A dynamic multi-tenant key-value cache. In *USENIX ATC*.
[30] Edward G. Coffman and Peter J. Denning. 1973. *Operating Systems Theory*. Prentice Hall Professional Technical Reference.
[31] Asit Dan and Don Towsley. 1990. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *ACM SIGMETRICS*.
[32] Gary Davis. 2018. 2020: Life with 50 billion connected devices. In *IEEE International Conference on Consumer Electronics*. 1–1.
[33] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *ACM SIGMOD*.
[34] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: A hybrid key-value cache that controls flash write amplification. In *USENIX NSDI*.
[35] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI*.
[36] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux Journal* 2004, 124 (2004), 5.
[37] Peter Freiling and Badrish Chandramouli. [n. d.]. Microsoft. Personal communication.
[38] Massimo Gallo, Bruno Kauffmann, Luca Muscariello, Alain Simonian, and Christian Tanguy. 2012. Performance evaluation of the random replacement policy for networks of caches. *SIGMETRICS Perform. Eval. Rev.* 40, 1 (June 2012), 395–396.
[39] Alex Gartrell, Mohan Srinivasan, Bryan Alger, and Kumar Sundararajan. [n. d.]. McDipper: A Key-value Cache for Flash Storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/.
[40] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The unwritten contract of solid state drives. In *ACM EuroSys*.
[41] Aamer Jaleel, Kevin Theobald, Simon Steely Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction. In *ISCA-37*.
[42] E. G. Coffman Jr. and Predrag Jelenković. 1999. Performance of the move-to-front algorithm with Markov-modulated request sequences. *Oper. Res. Lett.* 25, 3 (Oct. 1999), 109–118.
[43] Richard E. Kessler, Mark D. Hill, and David A. Wood. 1994. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. Comput.* 43, 6 (1994), 664–675.
[44] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage.
[45] Conglong Li, David G. Andersen, Qiang Fu, Sameh Elnikety, and Yuxiong He. 2017. Workload analysis and caching strategies for search advertising systems. In *ACM SoCC*.
[46] Conglong Li, David G. Andersen, Qiang Fu, Sameh Elnikety, and Yuxiong He. 2018. Better caching in search advertising systems with rapid refresh predictions. In *WWW*. 1875–1884.
[47] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. 2017. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage* 13, 3 (2017), 24.
[48] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *ACM SOSP*.
[49] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *USENIX FAST*.

[50] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. 2009. Modular data storage with Anvil. In *ACM SOSP*.
[51] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2016. Whirlpool: Improving dynamic cache management with static data classification. In *ASPLOS*.
[52] James O'Toole and Liuba Shrira. 1994. Opportunistic log: Efficient installation reads in a reliable storage server. In *USENIX OSDI*.
[53] David A. Patterson and John L. Hennessy. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[54] Ramtin Pedarsani, Mohammad Ali Maddah-Ali, and Urs Niesen. 2016. Online coded caching. *IEEE/ACM Transactions on Networking* (2016).
[55] Sara Perez. 2018. Twitter's doubling of character count from 140 to 280 had little impact on length of tweets. *Tech Crunch* (2018). Available at https://techcrunch.com/2018/10/30/twitters-doubling-of-character-count-from-140-to-280-had-little-impact-on-length-of-tweets/. 5/4/2021.
[56] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *ACM SOSP*.
[57] Mendel Rosenblum and John K. Ousterhout. 1991. The design and implementation of a log-structured file system. In *ACM SOSP*.
[58] Elisha J. Rosensweig, Jim Kurose, and Don Towsley. 2010. Approximate models for general cache networks. In *IEEE INFOCOM*.
[59] Rathijit Sen and David A. Wood. 2013. Reuse-based online models for caches. In *ACM SIGMETRICS*.
[60] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2016. Optimizing flash-based key-value cache systems. In *USENIX HotStorage*.
[61] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2018. Didacache: An integration of device and application for flash-based key-value caching. *ACM Transactions on Storage (TOS)* 14, 3 (2018), 1–32.
[62] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning relaxed Belady for content distribution network caching. In *USENIX NSDI*.
[63] Louise Story. 2007. Anywhere the eye can see, it's likely to see an ad. *The New York Times* 15, 1 (2007). Available at https://www.nytimes.com/2007/01/15/business/media/15everywhere.html. 9/6/2020.
[64] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A unified cluster management system for shared infrastructure. In *USENIX OSDI*.
[65] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced photo caching on flash for Facebook. In *USENIX FAST*.
[66] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. 2011. The anatomy of the Facebook social graph. *arXiv preprint arXiv:1111.4503* (2011).
[67] Francisco Velázquez, Kristian Lyngstøl, Tollef Fog Heen, and Jérôme Renard. 2016. *The Varnish Book for Varnish 4.0*. Varnish Software AS.
[68] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache modeling and optimization using miniature simulations. In *USENIX ATC*.
[69] Rui Wang, Christopher Conrad, and Sam Shah. 2013. Using set cover to optimize a large-scale low latency distributed graph. In *USENIX HotCloud*.
[70] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *USENIX ATC*.
[71] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, et al. 2016. Bluecache: A scalable distributed flash-based key-value store. *VLDB* 10, 4 (2016), 301–312.
[72] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at Twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.
[73] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *USENIX OSDI*.
[74] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2021. Segcache: A memory-efficient and scalable in-memory key-value cache for small objects. In *USENIX NSDI*.
[75] Yao Yue. 2020. Taming Tail Latency and Achieving Predictability. https://twitter.github.io/pelikan/2020/benchmark-adq.html.

Received February 2022; accepted May 2022