Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification

Assaf Eisenman, Stanford University; Asaf Cidon, Stanford University and Barracuda Networks; Evgenya Pergament and Or Haimovich, Stanford University; Ryan Stutsman, University of Utah; Mohammad Alizadeh, MIT CSAIL; Sachin Katti, Stanford University

https://www.usenix.org/conference/nsdi19/presentation/eisenman

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19) is sponsored by

Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification

Assaf Eisenman[1], Asaf Cidon[1,2], Evgenya Pergament[1], Or Haimovich[1], Ryan Stutsman[3], Mohammad Alizadeh[4], and Sachin Katti[1]

[1]Stanford University, [2]Barracuda Networks, [3]University of Utah, [4]MIT CSAIL

Abstract

As its price per bit drops, SSD is increasingly becoming the default storage medium for hot data in cloud application databases. Even though SSD's price per bit is more than 10× lower, and it provides sufficient performance (when accessed over a network) compared to DRAM, the durability of flash has limited its adoption in write-heavy use cases, such as key-value caching. This is because key-value caches need to frequently insert, update and evict small objects. This causes excessive writes and erasures on flash storage, which significantly shortens the lifetime of flash. We present Flashield, a hybrid key-value cache that uses DRAM as a "filter" to control and limit writes to SSD. Flashield performs lightweight machine learning admission control to predict which objects are likely to be read frequently without getting updated; these objects, which are prime candidates to be stored on SSD, are written to SSD in large chunks sequentially. In order to efficiently utilize the cache's available memory, we design a novel in-memory index for the variable-sized objects stored on flash that requires only 4 bytes per object in DRAM. We describe Flashield's design and implementation, and evaluate it on real-world traces from a widely used caching service, Memcachier. Compared to state-of-the-art systems that suffer a write amplification of 2.5× or more, Flashield maintains a median write amplification of 0.5× (since many filtered objects are never written to flash at all), without any loss of hit rate or throughput.

1 Introduction

Flash has an order of magnitude lower cost per bit compared to DRAM. Consequently, it has become the preferred storage medium for hot data that requires high throughput and low latency. For example Google [36] and Facebook [30] use it for storing photos, and databases like LevelDB [5] and RocksDB [9] are deployed on top of flash.

Key-value caches are an essential layer in modern web scale applications, and are widely used by almost all web services, including Facebook, Twitter and Airbnb. Large web service providers run their own key-value cache clusters, while smaller providers often utilize caching-as-a-service solutions like Amazon ElastiCache [1] and Memcachier [7].

	SSD+DRAM		DRAM only	
	Count	Cost	Count	Cost
Dell 2×10 core server with 256 GB DRAM	1	$7,700	17	$130,900
Samsung 1 TB enterprise SSD	4	$4,800	0	0
Total		$12,500		$130,900

Table 1: The cost of a hybrid cache server with combined capacity of 4.25 TB, versus the cost of multiple DRAM-only cache servers with the same aggregate capacity. SSD's superior cost per bit results in a 10× lower total cost of ownership for a hybrid cache server.

However, due to its limited endurance under writes, flash is typically not used for key-value caches like Memcached [6] and Redis [8]. This is all the more perplexing since these caches are typically deployed in a dedicated remote cluster [31] or remote data center [1, 7] or with client-side batching [31]. As a result, client-observed accesses times can be hundreds of microseconds to milliseconds, so flash would only increase delays by a small fraction when compared to using DRAM.

Furthermore, since the performance of caches is primarily determined by the amount of memory capacity they provide [13, 14], and the cost per bit of SSD is more than 10× lower than DRAM, flash promises significant financial benefits compared to DRAM. Table 1 demonstrates that the cost difference between DRAM-only cache and hybrid cache, both with 4.25 TB capacity, is more than 10×. The Total Cost of Ownership (TCO) difference would be even greater due to power costs, since flash consumes significantly less power than DRAM, and can be powered down when there are fewer requests without requiring re-warming the cache.

The reason flash has not been widely adopted as a key-value cache is that cache workloads wear out flash drives very quickly. These workloads typically consist of small objects, some of which need to be frequently updated [10, 31]. But, modern flash chips within SSDs can only be written a

few thousand times per location over their lifetime.

Further, SSDs suffer from write amplification (WA). That is, for each byte written by the application (e.g., the key-value cache), several more bytes are written to the flash at the device level. WA occurs because flash pages are physically grouped in large blocks. Pages must be erased before they can be overwritten, but that can only be done in the granularity of blocks. The result is that over time, these large blocks typically contain a mix of valid pages and pages whose contents have been invalidated. Any valid pages must be copied to other flash blocks before a block can be erased. This garbage collection process creates device-level write amplification (DLWA) that can increase the amount of data written to flash by orders of magnitude. Modern SSDs exacerbate this by striping many flash blocks together (512 MB worth or more) to increase sequential write performance (§2.1, [38]).

To minimize the number of flash writes, SSD storage systems are constrained to writing data in large contiguous chunks. This forces a second-order form of write amplification, which is unique to caches, that we name *cache level write amplification* (CLWA). CLWA occurs when the cache is forced to relocate objects to avoid DLWA. For example, when a hot object occupies the same flash block as many items that are ready for eviction, the cache faces a choice. It can evict the hot object with the cold objects, or it can rewrite the hot object as part of a new, large write. Therefore, in existing SSD cache designs, objects get re-written multiple times into flash.

To deal with this problem, the state-of-the-art system, RIPQ [38], proposes to store hot and cold objects together on flash, by inserting them in different physical regions. However, efficient data placement on flash is not sufficient to protect against high CLWA, and in fact, may further increase CLWA in certain scenarios. For example, consider an application, in which a large number of objects are infrequently accessed (or frequently updated). Since RIPQ admits all objects (hot and cold) into flash, infrequently accessed objects will get inserted into a "cold" insertion point, and will typically get evicted before it is accessed again. Therefore, these objects can get inserted and evicted multiple times. We show that under such workloads, RIPQ suffers from a CLWA of up to 150 (§5), which means it will wear out flash devices too quickly for many applications.

The flash reliability problem will become even greater over time, since as flash density increases, its durability will continue to decrease [20]. In particular, the next generation of flash technology (QLC), can endure *30× fewer writes* than the existing technology (TLC) [3, 29, 32].

We present Flashield, a novel hybrid key-value cache that uses both DRAM and SSDs. Our contribution is a novel caching strategy that significantly extends the lifetime of SSDs, such that it is comparable to DRAM by controlling and minimizing the number of writes to flash. Our main observation is that not all objects entering the cache are good candidates for placement in SSD. In particular, the cache should avoid writing objects to flash that will be updated or that will not be read in the near future. However, when objects first enter the cache, it does not know which objects are good candidates for SSD and which are not.

Therefore, a key idea in Flashield's design is that objects inserted into the cache always spend a period of time in DRAM, during which the cache learns whether they are good candidates for flash storage. If they indeed prove themselves as flash-worthy, Flashield will move them into flash. If not, they are never moved into flash, which reduces the resulting write amplification. Since the flash layer is considerably larger than DRAM (e.g., 10× larger), objects moved to flash on average will remain in the cache much longer than those that stay in DRAM.

To dynamically decide which objects are flash-worthy under varying workloads, we implement the admission control algorithm using machine-learning based Support Vector Machine (SVM) classification. We train a different classifier for each application in the cache. To train the classifiers, we design a lightweight sampling technique that uniformly samples objects over time, collecting statistics about the number of past reads and updates. The classifier predicts whether an object will be read more than n times in the future without getting updated, which is used to determine its suitability for storage on flash. We term this metric *flashiness*.

The second main idea in Flashield's design is its novel DRAM-based lookup index for variable-length objects stored on flash that requires less than 4 bytes of DRAM per object. This is more than 5× less than RIPQ, which consumes 22 bytes per object. Since the flash layer's capacity is much larger than the DRAM's, a naïve lookup index for objects stored on flash would consume the entire capacity of the DRAM. Our index consumes a relatively small amount of memory by not storing the location of the objects and their corresponding keys. Instead, for each object stored on flash, the index contains a pointer to a region in the flash where the object is stored, and it stores an additional 4 bits that specify a hash function on the object key that indicates the insertion point of the object in its region on flash. The index leverages bloom filters to indicate whether the object resides on flash or not without storing full keys in DRAM. On average, Flashield's lookup index only requires 1.03 reads from the SSD to return an object stored on it.

We implement Flashield in C and evaluate its performance on a set of real-world applications that use Memcachier [7], a popular cloud-based caching service, using week-long traces. We show that compared with RIPQ [38], Flashield reduces write amplification by a median of 5× and an average of 16×, and the index size by more than 5×, while maintaining the same average hit rates. We show that when objects are read from SSD, Flashield's read latency and throughput is close to the SSD's latency and throughput, and when objects are written to the cache or read from DRAM,

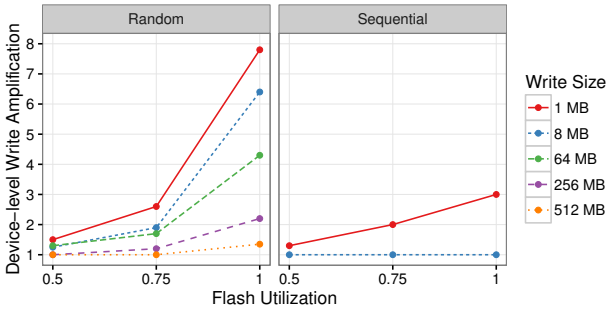

Figure 1: Device-level write amplification after writing 4 TB randomly and sequentially using different write sizes.

its latency and throughput are similar to that of DRAM-based caches like Memcached.

This paper makes three main contributions:

1. Flashield is the first SSD storage system that explicitly uses DRAM as an admission control filter for deciding which objects to insert into flash.
2. Flashield's novel in-memory lookup index for flash takes up less than 4 bytes per object in DRAM, without sacrificing flash write amplifiction and read amplification.
3. Flashield is the first key-value cache that uses a machine-learning based admission control algorithm and lightweight temporal sampling to predict which objects will be good candidates for flash.

As new generations of flash technology can tolerate even fewer writes [3,20,29,32], our dynamic admission control to flash can be extended to other systems beyond caches, such as flash databases and file systems.

2 The Problem

Designing an SSD-based cache requires solving two conflicting challenges. SSDs perform poorly and wear out quickly unless writes are large and sequential. This conflicts with the characteristics of cache workloads. Caches store small objects with highly variable lifetimes; this drives caches to prefer small random I/O for writes which will wear flash drives out quickly.

The lifetime of an SSD is defined by flash device manufacturers as the amount of time before a device has a non-negligible probability of producing uncorrectable read errors (e.g., a probability of 10^{-15} of encountering a corrupt bit). The lifetime of an SSD depends on several factors, including the number of writes and erasures (termed program-erase cycles), the average time between refresh cycles of the SSD cells, the cell technology, the error correction code and more. The typical lifetime of a flash cell is between 3-5 years assuming it is written 3-5 times a day on average.

The key metric for device wear is write amplification. Many write patterns force the SSD to perform additional writes to flash in order to reorganize data. Write amplification is the ratio of the bytes written to flash chips compared to the bytes sent to the SSD by the application. A write amplification of 1 means each byte written by the application caused a one byte write to flash. A write amplification of 10 means each byte written by the application caused an extra 9 bytes of data to be reorganized and rewritten to flash.

2.1 Device-level Write Amplification

Device-level write amplification (DLWA) is write amplification that is caused by the internal reorganization of the SSD. The main source of DLWA comes from the size of the unit of flash reuse. Flash is read and written in small (˜8 KB) pages. However, pages cannot be rewritten without first being erased. Erasure happens at a granularity of groups of several pages called blocks (˜256 KB). The mismatch between the page size (or object sizes) and the erase unit size induces write amplification when the device is at high utilization.

For example, when an application overwrites the contents of a page, the SSD writes it to a different, fresh block and maintains a relocation mapping called the Flash Translation Layer (FTL). The original block cannot be erased yet, because the other pages in the same block may still be live. When the flash chips are completely occupied, the SSD must erase blocks in order to make room for newly written pages. If there are no blocks where all of the pages have been superseded by more recently written data, then live pages from mutiple blocks must be consolidated into a single flash block.

This consolidation or *garbage collection* is the source of DLWA. If a device is at 90% occupancy, its DLWA can be very high. Figure 1 shows DLWA under sequential and random writes. The measurements were taken on a 480 GB Intel 535 Series SSD using SMART, a system for monitoring the internal formation of the device. For each data point, 4 TB of randomly generated data is written either randomly or sequentially to the raw logical block addresses of the device with varying buffer sizes. Specifically, in the random workload the logical block space is broken into contiguous fixed buffer-sized regions; each write overwrites one of the regions at random with a full buffer of random data. The sequential workload is circular; regions are overwritten in order of their logical block addresses, looping back to the start of the device as needed. For both patterns, we varied the device space utilization by limiting writes to a smaller portion of the logical block addresses.

The results show that random, aligned 1 MB flash writes experience a nearly 8× DLWA. This is surprising, since flash erase blocks are smaller than 1 MB. The reason for this write amplification is because SSDs are increasingly optimized for high write bandwidth. Each flash package within an SSD is accessed via a relatively slow link (50-90 MB/s today); SSDs stripe large sequential writes across many flash packages in parallel to get high write bandwidth. This effectively fuses erase blocks from several packages into one logical erase block. A 1 MB random write marks a large region of pages as ready for erase, but that region is striped across several erase units that still contain mostly live pages. Others have

Avg Object Size	Read / Write / Update %	Unread Writes %
257 B	90.0% / 9.5% / 0.5%	60.6%

Table 2: Statistics of the 20 applications with the most requests in the week-long Memcachier trace.

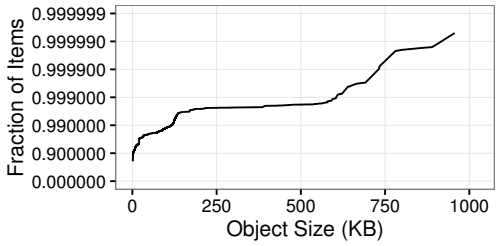

Figure 2: CDF of the object sizes written to memory by the top 20 applications in the Memcachier trace.

corroborated this effect as well [38].

There are two ways to combat this effect. The first is to write in units of $B \cdot W$ where B is the erase block size and W is how many blocks the SSD stripes writes across. Our results show that a cache would have to write in blocks of 512 MB in order to eliminate DLWA. The second approach is to write the device sequentially, in FIFO-order at all times. This works because each $B \cdot W$ written produces one completely empty $B \cdot W$ unit, even if writes are issued in units smaller than $B \cdot W$. Figure 1 shows that 8 MB sequential writes also eliminate DLWA.

This means our cache is extremely constrained in how it writes data to flash. To minimize DLWA the cache must write objects in large blocks or sequentially. In either case, this gives the cache little control on precisely *which* objects should be replaced on flash.

2.2 Cache-level Write Amplification

Writing to flash in large *segments* (contiguous chunks of data) is a necessary but not sufficient condition for minimizing overall write amplification. The main side effect of writing in large segments is *cache-level write amplification* (CLWA). CLWA occurs when objects that were removed from the SSD are re-written to it by the cache eviction policy. If the size of the segments (MBs) is significantly larger than the size of objects (bytes or KBs), it is difficult to guarantee that high-ranking objects in the cache will always be stored physically separate from low-ranked objects or objects that contain old values. Therefore, when a segment that has many low-ranked objects is erased from the cache, it may also inadvertently erase some high-ranking objects.

Table 2 presents general statistics of a week-long trace of Memcachier, a commercial Memcached service provider [13, 14], and Figure 2 presents the distribution of the sizes of objects written in the trace. The figure demonstrates that object sizes vary widely, and in general they are very small: the average size of objects written to the cache is 257 bytes, and 80.67% of objects are smaller than 1 KB. Therefore, even with a segment size of 8 MB using sequential writes, which is the the smallest possible segment size that does not incur extra write amplification, each segment will contain on average over 32,000 unique objects.

	Hit Rate	CLWA
Victim Cache	69.72%	4.00
RIPQ	70.59%	2.59

Table 3: Hit rate and cache-level write amplification of RIPQ and the victim cache policy under the entire Memcachier trace.

In addition, 60.6% of writes (and 5.8% of all requests) are unread writes, which means they are never read after they are written, and 0.5% of all requests are updates. Both unread writes and updates contribute to write amplification. Ideally, unread writes should not be written to the cache. In the case of updates, to reclaim the space of an object after it was updated, the cache needs to erase and rewrite the object.

RIPQ [38] represents the state-of-the-art in minimizing CLWA; it is an SSD-based photo cache that minimizes CLWA by inserting objects that were read k times in the past together [1]. When objects are first inserted into the cache, they are buffered in memory, and periodically they are moved into flash together as a segment with other objects that have been read the same number of times. The idea is that objects that were read k times in the past might share a similar future eviction rank. For example, an object that was read once is stored on flash in the same segment with other objects that were read once. Segments that contain objects that have been read fewer times will be evicted faster than segments with objects that have been read many times.

RIPQ works for photos, which are large and immutable, but it breaks down on web cache workloads where values are small and updated more frequently. To illustrate, we simulated the CLWA of RIPQ (the RIPQ implementation is not publicly available) with the Memcachier traces using a segmented LRU with 8 queues. We also compared it with a victim cache policy, a naïve approach where the SSD simply serves as an L2 cache (i.e., every object evicted from DRAM is written to SSD). This policy is used by TAO [11], Facebook's graph data store, which leverages a limited amount of flash as a victim cache for data stored in DRAM. The simulation assigns the same amount of memory for each application in the trace, with a ratio of DRAM to SSD of 1:7.

The results are presented in Table 3 and show that, while RIPQ considerably improves upon victim cache, it still suffers from a very high CLWA. Note that the victim cache would suffer from an even greater total WA, because it also suffers from DLWA (since it does not write to flash in large segments). RIPQ suffers from CLWA for two reasons. First, RIPQ has no admission policy and it writes *all* incoming objects to flash; even unread objects or objects that are frequently updated. Second, when the frequency of reads of a

[1] Non-cache SSD key-value systems that store data persistently [5, 9, 25, 27] are not affected by CLWA, because they do not evict objects (all data fits in the database)

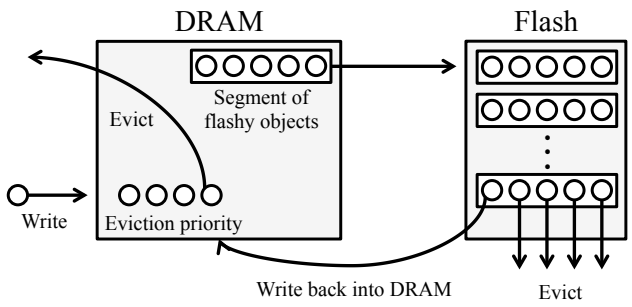

Figure 3: Lifetime of an object in Flashield. Objects always enter into DRAM. Objects that are a good fit for flash (*flashy* objects) are aggregated and moved into flash as a segment. The decision of whether to evict objects from DRAM or flash is based on a global eviction priority.

certain object changes, it creates additional writes. For example, if an object was read twice over a period of time after it was written, it is grouped with other objects that were read twice on flash. However, if it was read five more times, RIPQ needs to rewrite it to group it with other higher ranking objects. Since the objects are much smaller than the segment size, and there is a relatively high ratio of writes in the trace, RIPQ struggles to guarantee that objects that have been read around the same time will be stored in the same segment.

These results give two clues on how a cache should exploit DRAM differently to minimize CLWA for web cache workloads. First, not every object inserted into the cache by the application is a good candidate to be stored on SSD. For example, objects that are updated soon after they are first written or objects that have a low likelihood of being read in the future. However, the occurrence of such objects varies widely across different applications. For example, in some applications of the Memcachier trace, more than half of written objects are never read again, and in some applications, a vast majority of objects are read many times and should be written to the cache. Second, due to the disparity between the segment size and the object size, it is difficult to guarantee that objects that were similarly ranked by the eviction policy will be stored in physically adjacent regions on SSD.

Both of these insights motivate Flashield, a cache that successfully minimizes CLWA with no DWLA.

3 Design

The design goal of Flashield is to minimize cache-level and device-level write amplification, while maintaining comparable hit rate. The key insights of Flashield's design are to use DRAM as a filter, which prevents moving objects into flash that will be soon thereafter evicted or updated, and to maintain an efficient in-memory index which retains low write and read amplification.

Figure 3 illustrates the lifetime of an object in Flashield. Objects are first always written to DRAM. After the object is read for the first time, Flashield starts collecting features that describe its performance. These contain information about when and how many times the object has been read and updated. An object may be evicted from DRAM by Flashield's eviction algorithm.

Periodically, Flashield moves a segment (e.g., 512 MB) composed of many DRAM objects into flash. Flashield uses a machine learning classifier to rank objects based on their features. If an object passes a rank threshold, it will be considered as a candidate to move to flash. The candidates to flash are then ranked based on their score, which determines the order they are moved by Flashield into flash. This order is important when there are more flashy candidates than can fit in a single segment. After it gets moved to flash, an object will live in the cache for a relatively long duration. It will get moved out of flash once its segment is erased from flash, in FIFO order. At that point, the object will be evicted if it is low in terms of eviction priority, or it will get re-inserted into DRAM if it has a high eviction priority. Once the object is re-inserted into DRAM, it will have to prove itself again as flash worthy before it is re-written to flash. For more details, see §4.3.

In Flashield, DRAM serves three purposes. First, it is used as a filter to decide which objects should be inserted into SSD. Second, it stores the metadata for looking up and evicting objects on flash. Third, it serves as a caching layer for objects before they are moved to SSD and for objects that are not candidates for SSD.

3.1 DRAM as a Filter

In Flashield, DRAM serves as a proving ground for moving objects into flash. When objects are first written into DRAM, Flashield does not have a-priori knowledge whether they will be good candidates for flash. Furthermore, applications have unique workloads, so their access patterns need to be learned individually.

A strawman approach for determining which objects are flash-worthy is to rank them based on simple metrics like recency or frequency, as done by standard cache replacement polices like LRU or LFU. However, it is difficult to set a single threshold for flash-worthiness that will work for all applications. For example, the system can define a frequency-based threshold, requiring that an object will be read more than once before it enters flash. However, for some applications, such a threshold proves too stringent where the access patterns are long and reduces the hit rate due to premature evictions. It can also be too lenient for other applications, in which objects would be unnecessarily written to flash. Even for a single application, such a threshold is a heuristic that would have to be manually tuned (see the example described below and depicted in Table 4).

Instead of using a one-size-fits-all approach, machine learning can be used as a way to dynamically learn which objects are flash-worthy for each individual application.

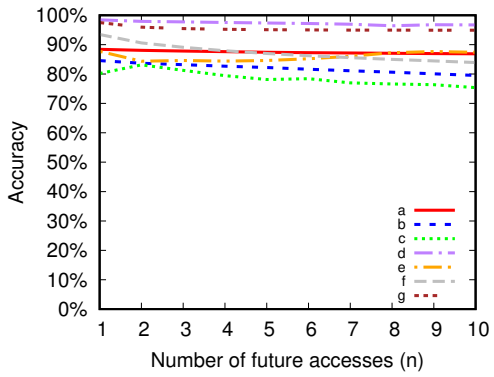

Figure 4: Accuracy of SVM classifier in different Memcachier applications, for predicting whether an object will be accessed at least n times in the future without updates.

3.2 Flashiness

We define *flashiness* as a metric that predicts whether an object will be a good fit for flash. An object that has a high flashiness score is an object that meets two criteria. First, it is an object that will be accessed n times in the near future (where n is a configurable parameter). This guarantees that it will not be evicted by the cache's eviction function. Second, it needs to be immutable in the near future, since updating an object in SSD requires an additional write.

Note that the threshold n, the number of times an object will be read in the future, can be used by the system to indicate how sensitive it is to write amplification. If the system is very sensitive to write amplification, it can set n to a relatively high number, which will ensure that Flashield will only move objects into flash that it predicts will be read many times in the future. On the other hand, if the system is more sensitive to hit rate, n will be set as a low number. In addition, Flashield allows the operator to set a fixed limit on the flash write rate to maintain a certain target lifetime.

Both of the above flashiness criteria can be captured by predicting the number of times an object will be read in the near future (e.g., one hour), and omitting objects that are predicted to be updated during this preiod.

Flashield uses a binary classifier using Support Vector Machine (SVM) to predict flashiness, by collecting two features: (1) number of past reads and (2) number of past updates. Figure 4 provides the accuracy of the classifier on different applications from the Memcachier traces, with variable n values. Accuracy is defined as $\frac{tp+tn}{tp+tn+fp+fn}$, where tp is true positives, tn is true negatives, fp is false positives, and fn is false negatives. The classifier tries to predict whether an object will be accessed at least n times in the future without being updated, using a training time of one hour.

The accuracy of the prediction varies among the different applications (from 75% to 99%), due to their varying workloads. In addition, the accuracy generally decreases as n increases. This is because as n increases, the classifier is trying to predict more rare events, of which it has observed fewer training data points. For example, there are more objects that have been read more than once in the following hour, than objects that have been been read five times or more.

App	a	b	c	d	e	f	g
Num Accesses	5	4	5	2	4	4	6

Table 4: The threshold of the number of past accesses that predict whether an object will be accessed 5 times or more in the next hour.

To demonstrate why machine learning is more effective than having a fixed threshold of the number of past accesses for determining flashiness, consider the following example. We trained a simple classifier across the applications from the trace, which tries to predict flashiness with $n = 5$, utilizing a single feature (number of past reads), using a decision tree with a depth of 1. Table 4 presents the thresholds that the decision tree chose for each application, which would provide the highest prediction accuracy, based on its training samples. The results demonstrate that there is no one single static threshold that would be optimal for all applications. This also shows that it is difficult to determine what this threshold would be a-priori. For example, for application d, only two reads occurring in the past is sufficient to predict that it will be read 5 more times or more in the future.

3.3 Flashiness Design Discussion

We experimented with several different features related to the number and frequency of reads and updates. We found that the only features that were impactful in the prediction and capture past information on reads and update are: (1) number of past reads and (2) number of past updates.

To our surprise, we found that across all the applications we measured, features related to recency (e.g., time between reads, time since the last read) had no positive impact on predictions, and in fact, in some instances reduced classifier accuracy. This supports our design choice to decouple the flashiness metric, which is based on number and type of past accesses, from the eviction policy, which is typically based on recency (e.g., LRU or one of its derivatives, see §3.4).

In addition, we experimented with several different classification algorithms. Initially, we tried predicting this number directly using a logistic regression. We ran this classifier on the Memcachier trace and found the prediction was highly inaccurate. After trying different features and classifiers, we found it is difficult to accurately predict exactly how many times an object will be accessed in the future, which is why we use binary classification, which predicts whether the number of future reads is above n. We also tried using a different binary classifier, decision trees, which provided very similar accuracy to SVM. We decided to use SVM, because they provide a continuous score, which is used to provide a global flashiness rank for objects. With decision trees, the range of the score is limited to the number of leaves.

3.4 DRAM as an Index for Flash

Unlike log-structured merge trees (LSM), Flashield stores the index in DRAM (both for objects in DRAM and in flash). This allows Flashield to service requests at much lower latency, since the index is read from DRAM. More importantly, storing the index on flash requires LSMs to constantly update the index when objects get updated, which creates a large number of writes [24, 27, 39]. When the index is on DRAM, it is trivial to update it. However, since Flashield uses DRAM also as an admission control layer, we must ensure that indexes will consume a minimal amount of space on DRAM.

Similar to Memcached, Flashield stores its index in a hash-table to enable efficient lookups. A naïve index would contain the identity of the keys stored in flash, the location of the values, and their position in an eviction queue. However, such an index would be prohibitively expensive. If we take an example of a 6 TB flash device with an average object size of 257 bytes (equal to the average object size of the top 20 applications in the Memcachier trace), storing a hash of the key for each object that avoids collisions requires at least 8 bytes, storing the exact location of each object would be 43 bits, and keeping a pointer to a position in a queue would be 4-8 bytes. Storing 17 bytes per object on DRAM would require 406 GB of DRAM. This would take up (or exceed) all of the DRAM of a high end server. In RIPQ, for example, each in-memory index entry is 22 bytes. We design a novel in-memory lookup index for variable-sized objects that uses less than 4 bytes per object, without incurring additional flash write amplification.

Identities of keys. Rather than storing the identities of keys in the index, Flashield keeps them only in the flash device, as part of the object metadata. In order to identify hash collisions in the lookup hash-table, Flashield compares the key from flash. To limit the number of flash reads during key lookup and avoid complex table expansions, Flashield utilizes a multiple-choice hash-table without chains. During lookup, pre-defined hash functions are used one by one, such that if the key is not found, the next hash function is used. If all hash functions are used and the key was still not found then Flashield returns a miss. Similarly if a collision happens during insertion, the key is re-hashed with the next hash function to map it to another entry in the lookup table. If all hash functions are used and there is still a collision, the last collided object is evicted to make space for the new key.

To reduce the number of excess reads from the flash in case of hash collisions, Flashield utilizes an in-memory bloom filter for each segment, which indicates whether a key is stored in the segment. We decided to use a bloom filter per segment, rather than a global bloom filter, to eliminate the need of the bloom filter to support deletions (since each segment is immutable). We use bloom filters with a false positive rate of 1%. For the Memcachier trace, this translates to an average of 1.03 accesses to flash for every hit in the flash and an extra memory overhead of 10 bits per item.

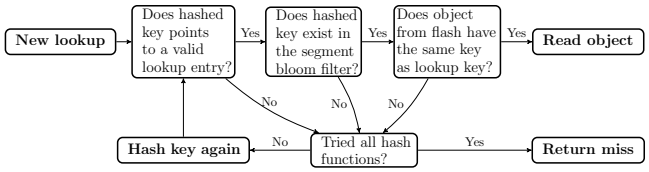

Figure 5: Algorithm for determining if an object exists in flash.

Object location. Instead of directly storing the location of the SSD object, the index contains two separate fields: segment number and the ID of a predefined hash function. The segment number represents a contiguous segment in flash where the object is stored. Hashing the object's key using the predefined hash function provides the offset of the object within the segment. Using a hash function to indicate the object location in the segment may reduce flash utilization, because it limits the number of possible positions for placing an object within a segment. Note that these hash functions are orthogonal to the hash functions used for the hash-table lookup. We chose to utilize 16 pre-defined hash functions (i.e., up to 16 possible positions for an object) since increasing the number of hash functions beyond that provided negligible improvement in the flash utilization. We explore the flash utilization in §5.3. Note that since data is written to flash sequentially, segment sizes of 8 MB or larger achieves minimal DLWA. We use 512 MB segments in order to reduce the indexing overhead.

Eviction policy. To avoid the overhead of maintaining a full eviction queue composed of a doubly-linked list of pointers, Flashield uses the CLOCK algorithm [16], similar to other memory key-value caches [18]. CLOCK approximates the LRU policy, so to evaluate its impact we ran the top 5 applications in the Memcachier trace in a simulation and compared the results between CLOCK and LRU. The results show that by keeping just two bits per object for CLOCK timestamps, the hit rate decreases by an average of only 0.1% compared to LRU.

Figure 5 summarizes Flashield's lookup process. The lookup key is first hashed to find the corresponding entry ID in the lookup hash-table, which provides the segment ID. Then, Flashield performs a key lookup in the segment's bloom filter. If the key is found in the bloom filter, Flashield reads the object from the segment on flash. Since the bloom filter may cause a false positive, if the object that was read from flash does not have the same key as the object which is being looked up, the key will be hashed again and Flashield will look it up again in the lookup hash-table. Similarly, if the key is not found in the bloom filter, the key is hashed again and Flashield performs another lookup in the lookup hash-table. Flashield will attempt to lookup an object using all the configured hash functions (16 by default) until the object is found. If the object is not found after all attempts, the object does not exist in flash and Flashield returns a miss.

The hash-table entry format is summarized in Figure 6.

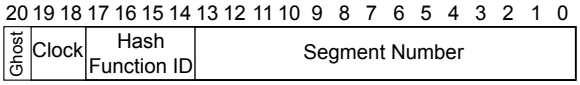

Figure 6: Hash-table entry format for objects stored on flash.

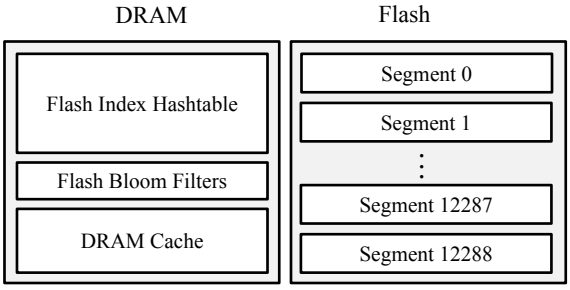

Figure 7: Flashield's architecture. The flash index is an in-memory hash table. The bloom filters provide fast lookups for object existence in flash, and the rest of the DRAM is a cache. Most of the cache objects are stored on flash in segments.

The index contains an extra bit (*ghost*), that indicates whether the object is scheduled for deletion from flash. We describe the purpose of this flag at §4.3.

4 Implementation

This section presents the implementation of Flashield. We implemented Flashield in C from scratch, except for the transport, dispatch, request processing, and the hash table for DRAM objects, which are borrowed from Memcached 1.4.15. Flashield has four main functions: read, write, move data to flash and evict. Figure 7 depicts the high level components of Flashield's architecture. It supports the generic Memcached protocol, so applications that deploy Memcached can transparently utilize Flashield.

For reads, Flashield first checks whether the object exists in the hash table for DRAM objects, which is based on Memcached's hash table. If not, it checks whether the object exists in flash using a separate hash table for flash objects. If the object exists either in DRAM or flash, Flashield returns it, otherwise the request is counted as a miss. Incoming writes and updates are always stored in DRAM first. In the case of updates, the updated object is stored in DRAM, and the old version is invalidated. Flashield always maintains free space in the size of a segment in DRAM for incoming writes.

Flashield uses a configurable number of worker threads that process the client requests in parallel. To maintain enough free space on DRAM, Flashield uses a dedicated cleaner thread that works in the background and is not on the critical path for normal request (read/write) processing. In addition, Flashield let the operator configure a flash write limit to guarantee a certain target lifetime. When the free space on DRAM drops below a segment size, if there are enough objects that meet a threshold for their flashiness score and the flash write rate limit was not reached, the cleaner copies them into a segment buffer. When the buffer is full, the cleaner writes the segment to flash and then frees the space the objects occupied in DRAM. Objects are moved to flash in an order based on their flashiness score. When the SSD is full, the cleaner will remove the last segment from flash based on FIFO order.

For eviction, Flashield maintains a global priority rank for all objects, whether they are stored in DRAM or flash. Objects are evicted from Flashield based on this global priority. By default the priority is an approximation of LRU using CLOCK. If the next object for eviction is in DRAM, Flashield simply evicts it. If the next object for eviction is in flash, Flashield marks it as a *ghost* object, and it will be evicted when its segment is removed from flash. Note that the movement of data from DRAM into flash is decoupled from eviction. They are conducted in parallel and use different metrics to rank objects. Objects that are moved between the flash and DRAM always keep their global priority ranking. When there are not enough objects in DRAM that meet a threshold for their flashiness score, or the flash write rate reached its limit, the cleaner will evict items from DRAM to maintain sufficient free space.

The rest of the section describes in detail how Flashield moves objects into flash, and the implementation of Flashield's classifier and eviction algorithm.

4.1 Writing Objects to Flash

Flashield constructs a flash-bound segment in DRAM, by greedily trying to find space for the objects in the segment one-by-one. The output bits of the pre-determined hash functions provide different possible insertion points in the segment for each object. Flashield first assembles a group of objects that need to be moved to flash based on the their flashiness. It then tries to insert the objects from this group based on their size. Larger objects go first, because they require more contiguous space than smaller objects. In this process, some objects will not have available space in the segment. Flashield skips these objects and tries to insert them again next time it creates a new segment. We evaluate the resulting segment utilization in § 5.3.

4.2 Classifier Implementation

Flashield's flashiness score is computed based on two features for each object. Since these features depend on information across multiple object accesses, the features for an object are only generated after an object has been read at least once. If an object has never been read, its flashiness score is automatically equal to zero.

Flashield periodically trains a separate classifier for each application. For the commercial traces we used, we found that a training period of one hour at the beginning of the trace was sufficient.

The naïve way to train the classifier would be to update the features at each access to the DRAM. However, this approach may oversample certain objects, which can create an unbalanced classifier. For example, if a small set of objects account for 99% of all accesses, multiple sets of features would be created for these objects, and the flashiness esti-

mation would be biased towards popular objects.

To tackle this problem, we implemented a sampling technique that generates a single sample for each object, chosen uniformly over all of its accesses during the training period. Instead of updating the features at each object access, Flashield does it only with a probability of $\frac{1}{n}$, where n is the number of times the object was read and updated so far.

To illustrate this sampling technique, consider the following example. Suppose an object was written for the first time, and then read. Its feature vector is: $[1,0]$ (number of past reads, number of past updates). Since the number of reads and updates is equal to 1, the feature vector generated by its first read will be the feature we use for training at a probability of 1. If the object is updated (feature vector is now: $[1,1]$), Flashield will keep the second set of features with a probability of $\frac{1}{2}$, since the number of reads and updates is equal to 2. This is equal to uniformly sampling the features from the first or second access. Each subsequent access will be sampled at a uniform probability of $\frac{1}{n}$, and the probability of prior accesses to be sampled will also be uniform.

After collecting the samples for an hour, we measure the number of times each of the objects is hit in the subsequent hour. This number is used as the target function for the training. After these two periods, Flashield trains the classifier using these training samples and labels.

4.3 Eviction

Flashield uses the CLOCK algorithm to rank objects for eviction. Instead of keeping precise priority rank, each object has only two CLOCK bits in its hash table entry that signify priority. In order to approximate LRU, when the object is read, its bits are all set to 1. MFU (Most Frequently Used) is approximated by incrementing the bits by 1 at each read.

When a set operation inserts an object into the cache, it may trigger an eviction. On eviction, Flashield walks round-robin through each object entry in the index, decrementing its CLOCK value by one. It stops the walk when it reaches an entry that has a CLOCK value of zero. This object is chosen as the next victim for eviction. If the victim object is in DRAM, its space is freed and may be reused for the incoming value. In case there is sufficient space after freeing the victim, eviction stops, otherwise the process repeats as needed. If the object is in flash, Flashield cannot delete it immediately from flash, since fine-grained writes to the SSD would incur high DLWA. Instead, the entry is marked as a *ghost object*, which acts as a hint to the flash cleaning process. Later, when the on-flash segment that the object resides is about to be overwritten, the ghost object will not be preserved, effectively freeing the storage as part of the bulk flash cleaning process. Even so, a ghost object is still accessible if it is the most current value associated with a particular key, so long as the flash cleaning process has not yet overwritten its segment on flash. In a sense, ghost objects approximate the bottom of the global eviction rank (including both flash and DRAM); non-ghost objects, are considered to be at the top of the global eviction rank and we call them *hot objects*.

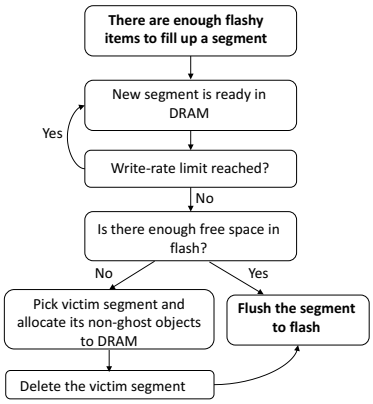

Figure 8: Flashield's process of allocating and deleting a segment to and from flash.

Flashield triggers a segment deletion once a new segment is allocated and ready to be moved from DRAM to flash, given that the flash is full and the configured write rate limit was not exceeded. The cleaner removes the last segment from flash in FIFO order. During segment erasure, its ghost objects are removed from the cache, while hot objects are re-inserted into the DRAM. Figure 8 summarizes this process.

Moving objects from flash back to DRAM will trigger evictions; left unchecked this can create two issues. First, hit rates could suffer if objects are prematurely evicted from DRAM without proving they are flashy. Second, if too many flashy objects are evicted it can contribute to write amplification. Flashield guards against this with a *hot data threshold (HDT)*, which ensures that in the limit enough objects can be discarded during cleaning to free up sufficient space on flash, without placing too much pressure on eviction. Without HDT, the cleaner could re-allocate low ranked objects, at the expense of higher ranked objects residing in the DRAM.

The HDT is defined as $DRAM + SSD \cdot hot$, where $DRAM$ is the available object storage in DRAM, SSD is the total size of the SSD, and hot is the percentage of SSD that is allocated for hot objects. Flashield strives to maintain the HDT, even when an incoming object has sufficient space in DRAM. To do so, whenever the amount of hot data exceeds the HDT, Flashield triggers a new eviction, which marks additional objects as ghost if they reside on flash. By default, hot is 70%, so about 30% of the objects on flash are ghost objects.

Ghost objects can still be accessed after they were marked as ghosts, since they are not immediately removed from flash. If a ghost object is accessed, it is not considered a ghost anymore and Flashield marks it as a hot object (the ghost bit is set to zero). Since Flashield always maintains the HDT, switching a ghost object from ghost to hot may trigger an eviction. To avoid unnecessary DRAM evictions, Flashield will not evict low ranking objects from DRAM in such case, but only walk through flash objects to mark ob-

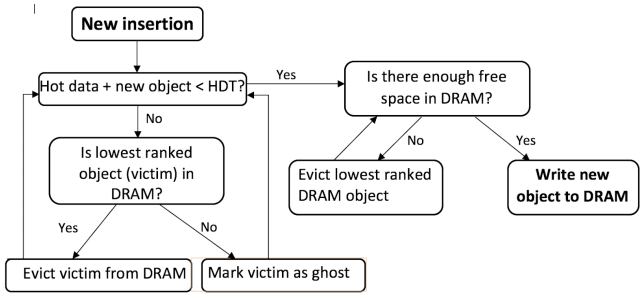

Figure 9: Flashield's eviction process.

jects as ghosts.

Although the cleaner is responsible for maintaining enough free space in DRAM (by allocating new segments to flash), in rare occasions the DRAM may not have enough free space to accommodate an incoming write. This may happen when the flash write rate limit is reached, or if the number of objects with flashiness score above the threshold is not enough for forming a new segment. In such scenario, Flashield will trigger a special eviction where it will walk through the DRAM objects only, and will evict low ranking objects from DRAM to accommodate the incoming write.

Figure 9 demonstrates Flashield's flow chart when a set operation inserts new object to the cache.

Delete operations in Flashield do not incur writes to flash. If the object is in DRAM, it is simply deleted. If it resides in flash, it is not immediately removed from flash, since that would incur DLWA. It is also not marked as a ghost, because ghost objects can still be accessed. Instead, Flashield deletes the object's lookup entry. During segment eviction, the cleaning process identifies deleted objects by comparing the segment ID in their corresponding lookup entry with the evicted segment ID, and will not preserve them. Building on that, Flashield handles update operations as a delete operation followed by a new insertion.

5 Evaluation

In this section we evaluate the end-to-end performance of Flashield compared to existing systems. Unfortunately, to the best of our knowledge, there are no public traces of large-scale key-value caches. We use real-world traces of an entire week, provided by Memcachier, a widely used Memcached service provider. Since the Memcachier traces are fairly sparse in terms of their request rate, we ran a set of synthetic microbenchmarks to stress the performance of the system to measure its throughput and latency.

5.1 End-to-end Performance

We compare the end-to-end hit rate and write amplification of Flashield to RIPQ and the victim cache policy, by rerunning real-world applications from the Memcachier traces. Since no public implementation of RIPQ is available [38], we are forced to run and compare a simulation of the three systems. Each one of the policies uses the same amount of memory that was allocated in the Memcachier trace, with a ratio of 1:7 of DRAM and SSD. We run Flashield with a threshold of one future read. In other words, objects that are predicted to have at least one future read are deemed sufficiently flash-worthy. Since Flashield utilizes a separate SVM for each application, we compare the results of individual applications. To run RIPQ with 8 insertion points, and therefore at least 8 different segments on flash, we only run applications that were allocated a sufficient amount of memory by Memcachier.

	Flashield		RIPQ		Victim Cache	
App	Hit %	CLWA	Hit %	CLWA	Hit %	CLWA
a	98.8%	5.8	98.5%	151.9	99.3%	4536.3
b	98.6%	2.8	98.8%	4.4	98.9%	21.7
c	83.1%	0.4	83.1%	2.9	93.3%	3.7
d	98.1%	0.2	98.7%	12.4	99.3%	34.0
e	96.0%	0.8	96.0%	1.6	96.2%	1.3
f	90.1%	0.2	91.3%	1.8	94.4%	2.4
g	97.3%	0.5	97.3%	1.4	97.4%	1.0

Table 5: Hit rates and CLWA of Flashield using a threshold of one future read, RIPQ and victim cache.

	Flashield 1		Flashield 10		Flashield 100	
App	Hit %	CLWA	Hit %	CLWA	Hit %	CLWA
a	98.8%	5.8	99.0%	9.2	98.9%	5.0
b	98.6%	2.8	98.6%	2.7	95.2%	0.0
c	83.1%	0.4	83.1%	0.4	83.0%	0.4
d	98.1%	0.2	98.1%	0.2	98.1%	0.2
e	96.0%	0.8	95.9%	0.7	95.9%	0.7
f	90.1%	0.2	85.5%	0.0	85.2%	0.0
g	97.3%	0.5	97.3%	0.5	97.3%	0.5

Table 6: Hit rates and CLWA of Flashield using a flashiness prediction threshold of 1, 10 and 100 future reads.

Table 5 presents the results comparing Flashield and RIPQ. The results show that Flashield achieves significantly lower CLWA than RIPQ and victim cache. The median CLWA of Flashield is 0.54, the median of RIPQ is 2.85 and the median of victim cache is 3.67. Even though Flashield uses a low threshold for flashiness of one future read, it still prevents a large number of writes that are not a good fit for SSD from being written to flash. Flashield and RIPQ have an almost identical hit rate. Both have a lower hit rate than victim cache, but victim cache suffers from significantly higher CLWA (and since it does not handle DLWA, also a much higher overall write amplification).

Table 6 compares Flashield with different flashiness prediction thresholds n. While the results vary from application to application, generally speaking, the higher the threshold the lower the CLWA and the lower the hit rate. Note that in some applications, such as in application a, this trade off does not hold, since we train the classifier individually on each application, and each application performs differently.

Table 7 depicts the results when we vary the ratio of DRAM and SSD, while keeping the total amount of memory constant for each application. The results show that if we reduce the amount of DRAM too much, the hit rate drops. This is due to the fact that when the DRAM is low, objects do not

App	DRAM 1:15 Hit %	DRAM 1:15 CLWA	DRAM 1:7 Hit %	DRAM 1:7 CLWA	DRAM 1:3 Hit %	DRAM 1:3 CLWA
a	99.0%	5.1	99.0%	4.6	99.0%	2.6
b	98.3%	3.1	98.6%	4.1	98.8%	4.9
c	81.4%	0.4	83.2%	0.4	92.7%	0.8
d	97.6%	1.2	98.4%	0.9	98.9%	2.2
e	95.7%	0.7	96.0%	0.8	96.2%	0.9
f	89.0%	0.2	91.0%	0.3	94.3%	0.4
g	97.2%	0.5	97.3%	0.5	97.3%	0.5

Table 7: Hit rates and CLWA of Flashield using a threshold of 1, with varying ratios of DRAM and SSD. The results use a smaller segment size (2 MB).

	Flashield			Memcached	
	SSD Hits	DRAM Hits	Misses	Hits	Misses
Throughput (IOPS)	150K	270K	239K	275K	287K
Latency (µs)	106	13.5	19	13	12

Table 8: Throughput and latency of SSD hits, DRAM hits and cache misses for Flashield and Memcached.

have sufficient time to prove themselves as flashy enough to be moved to SSD before they are evicted from DRAM. Note that we used a smaller segment size in these runs, in order to be able to display results for a 1:15 ratio of DRAM.

5.2 Microbenchmarks

We drive Flashield's implementation with microbenchmarks to stress the performance of the system, and compare its latency and throughput with Memcached. We use 4-core 3.4 GHz Intel Xeon E3-1230 v5 (with 8 total hardware threads), 32 GB of DDR4 DRAM at 2133 MHz with a 480 GB Intel 535 Series SSD. All experiments are compiled and run using the stock kernel, compiler, and libraries on Debian 8.4 AMD64. The microbenchmark requests are based on random keys, with an average object size of 257 bytes, which is the average object size of the top 20 application in the Memcachier trace. We disabled the operating system buffer cache to guarantee that SSD reads are routed directly to the SSD drive. Since the performance of SSD and DRAM is an order of magnitude different, we separately measured SSD and DRAM hits. Finally, we measured the latency and throughput of Memcached 1.4.15 as a baseline.

Table 8 presents the throughput and latency of the microbenchmark experiment. Note that in the case of both Memcachier and Facebook, Memcached is not CPU bound, but rather memory capacity bound [14, 15]. The latency and throughput of DRAM hits in Flashield are very similar to the latency and throughput of Memcached. While the average latency of SSD hits is significantly higher than DRAM, their latencies become similar when deploying over the network (network access times are typically 100 µs or more). The miss latency of Flashield is similar to the latency of DRAM hits, because all of Flashield's lookup indices are stored in DRAM, and the only case it needs to access flash in a miss is when one of the in-memory bloom filters returns a false positive. The write throughput and latency of Flashield were identical to Memcached, because writes always enter Flashield's DRAM.

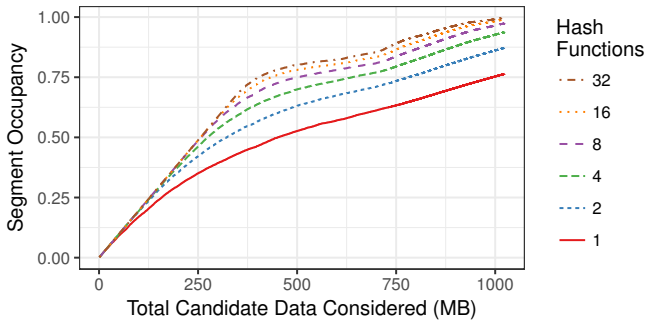

Figure 10: Utilization of a 512 MB segment on flash when Flashield tries to allocate space with a varying number of objects from the Memcachier trace. As Flashield tries to allocate more objects, it achieves higher utilization.

5.3 Utilization on Flash

When moving data from DRAM to flash, Flashield tries to allocate space for objects in different possible insertion points in the flash segment, using pre-defined hash functions. If none of the insertion point references to sufficient contigious free space for the object, Flashield skips the object and will try to insert it during the next segment allocation.

Figure 10 depicts the utilization of Flashield's flash allocation algorithm. To measure the utilization, we ran Flashield's allocation algorithm on the Memcachier traces with different number of hash functions over a segment size of 512 MB. The allocation greedily tries to allocate space to more data and measures the resulting utilization. Note that after the segment reaches about 60% utilization, its utilization curve gradient decreases, since when Flashield tries to allocate objects there is a higher probability of collisions with other existing objects in the segment. Using 16 hash functions, it takes about 1 GB of objects to reach a 99% utilization, and on average each object needs to be hashed 8.2 times until it finds an insertion point with enough space.

6 Related Work

There are two types of prior research. There are several prior SSD-based key-value caches for specific workloads (e.g., photo cache, graph database), but all of them suffer from low flash lifetime under a general-purpose key-value workload with small keys and variable objects without leveraging specialized hardware. There is also a large number of prior SSD-based persistent key-value stores. Unlike caches, persistent stores do not maintain an admission control and eviction policies and do not suffer from CLWA, hence their write amplification problems are less severe.

SSD-based Key Value Caches Facebook's flash-based photo cache evolved from McDipper [19] to BlockCache [2], and then to RIPQ [38], trying to improve hit rates while maintaining low write amplification. McDipper uses a simple FIFO policy, which causes it to suffer from low hit

rates. BlockCache improves cache hit rates by leveraging the SLRU policy which co-locates similarly prioritized content on flash, but incurs much higher write amplifcation than McDipper. RIPQ achieves even higher hit rates than BlockCache, while keeping its write amplication comparable to McDipper [2]. RIPQ performs insertions with priority-aware memory blocks, and uses virtual blocks to track the increased priority value when an item is accessed. However, in a general purpose key-value service like Memcachier, RIPQ suffers from more than 5× higher write amplification than Flashield, and up to 150× on specific applications. Furthermore, RIPQ's in-memory index map occupies 22 bytes per entry, consuming a very large amount of DRAM. Flashield's novel index requires less than 4 bytes of DRAM per object. TAO [11], Facebook's graph data store, uses a limited amount of flash as a victim cache for data stored in DRAM. Therefore, it suffers from a high rate of writes, because items which are not frequently accessed are written into flash and evicted soon after.

Twitter has explored SSD-based caching for its data center cache with Fatcache [21], a modified version of Memcached that buffers small writes and utilizes FIFO as an eviction policy. Flashield has better write amplification than Fatcache, since not all write requests are written to flash, and higher hit rates, because it uses eviction policies similar to LRU, which provide a higher hit rate than FIFO. Moreover, Fatcache's in-memory index requires 32 bytes (or more) per entry, which is 8× larger than Flashield.

A couple of systems try to support SSD-based caches by modifying the SSD's Flash Translation Layer (FTL). Duracache [26] tries to extend the life of the SSD cache, by dynamically increasing the flash device's error correction capabilities. Shen et al [37] allow the cache to directly map keys to the device itself, and remove the overhead of the flash garbage collector. Unlike these systems, Flashield addresses CLWA without any changes in the flash device.

Other than key-value caches, there are several systems that utilize flash as a block-level cache for disk storage [4, 22, 23, 33, 35, 40]. Unlike Flashield, storage blocks in these systems are always written to flash, and are fixed-sized (typically kilobytes in size). For this reason, they use a naive (inefficient) in-memory index to map from block's key to a location in flash. These properties make them impractical for general purpose key-value workloads with a variable and on average small object sizes.

Cheng et al [12] present an offline analysis of the trade-off between write amplification and eviction policies in block-level caches. They generalize Belady's MIN algorithm to flash-based caches, and demonstrate that LRU-based eviction is far from the optimal oracle eviction policy. However, they do not provide an online algorithm and an implementation that reduces write amplification of SSD-based caches.

SSD-based Key Value Stores Since these systems are persistent stores, all objects must be eventually written to flash, and thus they do not maintain an admission control and eviction policies, which are necessary for cache systems like Flashield. Consequently, persistent key-value stores do not suffer from CLWA and its implications, so their lifetime constraints are less severe than in a cache workload. However, they still strive to minimize write amplification for performance, since they must still suffer write amplification costs to compact data and update their indexes.

Systems such as LevelDB [5] and RocksDB [9] store the entire dataset and index on flash using Log-structure Merge-trees (LSM), and buffer writes to flash in DRAM to avoid DLWA. To enable efficient lookups, LSM-trees continuously perform a background compaction process that sorts and rewrites key-value pairs to flash, creating a major write amplification, particularly for workloads like key-value caches. WiscKey [27] reduces write amplification by separating keys and values. Keys are kept sorted in the LSM-tree, while values are stored separately in a log, which is helpful for workloads with large value sizes. PebblesDB [34] aims to reduce write amplification during compaction by using Fragmented Log-Structured Merge Trees (FLSM), avoiding rewriting data in the same tree level. In addition, NVMKV [28] is a key-value store that relies on advanced FTL capabilities (advanced multi-block writes) to deliver higher performance and lower write amplification. SILT [25] is a flash key-value database that minimizes the index stored in memory by utilizing three basic key-value stores. Objects are inserted first to a write-optimized store, and then re-written and merged into increasingly more memory-efficient stores. The majority of the objects are stored in the most memory-efficient store, making the average index cost per key low. However, unlike Flashield, SILT is not optimized for write amplification, and assumes values are fixed-length.

7 Conclusions

SSD faces unique challenges to its adoption for key-value cache use cases, since the small object sizes and the frequent rate of evictions and updates create excessive writes and erasures. Flashield is the first key-value cache that uses DRAM as a filter for objects that are not ideal for SSD. Flashield profiles objects using lightweight machine learning, and dynamically learns and predicts which objects are the best fit for flash. It introduces a novel in-memory index for variable sized objects with an overhead of less than 4 bytes per object, without sacrificing the flash write and read amplifictions.

The ideas in this paper can be extended to other use cases. For example, non-volatile memory (NVM) faces durability challenges too, especially when used as a replacement for DRAM, and may also require an admission policy [17]. This is also the case in multi-tiered storage systems, where cheaper storage layers offer more capacity at the expense of decreased performance. Finally, dealing with the durability of flash becomes an ever more pressing issue, as its density increases (and its ability to tolerate writes decreases).

References

[1] Amazon Elasticache. aws.amazon.com/elasticache/.

[2] The evolution of advanced caching in the facebook cdn. https://research.fb.com/the-evolution-of-advanced-caching-in-the-facebook-cdn/.

[3] Facebook asks for QLC NAND, Toshiba answers with 100TB QLC SSDs with TSV, author = Alcorn, P, note = http://www.tomshardware.com/news/qlc-nand-ssd-toshiba-facebook,32451.html,.

[4] Flashcache. github.com/facebookarchive/flashcache.

[5] LevelDB. leveldb.org/.

[6] Memcached. memcached.org/.

[7] Memcachier. www.memcachier.com.

[8] Redis. http://redis.io/. 7/24/2015.

[9] RocksDB. rocksdb.org//.

[10] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.

[11] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 49–60.

[12] CHENG, Y., DOUGLIS, F., SHILANE, P., WALLACE, G., DESNOYERS, P., AND LI, K. Erasing belady's limitations: In search of flash cache offline optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 379–392.

[13] CIDON, A., EISENMAN, A., ALIZADEH, M., AND KATTI, S. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (Santa Clara, CA, July 2015), USENIX Association.

[14] CIDON, A., EISENMAN, A., ALIZADEH, M., AND KATTI, S. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, Mar. 2016), USENIX Association, pp. 379–392.

[15] CIDON, A., RUSHTON, D., RUMBLE, S. M., AND STUTSMAN, R. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 321–334.

[16] CORBATO, F. J. A paging experiment with the multics system. Tech. rep., DTIC Document, 1968.

[17] EISENMAN, A., GARDNER, D., ABDELRAHMAN, I., AXBOE, J., DONG, S., HAZELWOOD, K., PETERSEN, C., CIDON, A., AND KATTI, S. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 42:1–42:13.

[18] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 371–384.

[19] GARTRELL, A. Mcdipper: A key-value cache for flash storage. https://code.facebook.com/posts/223102601175603/mcdipper-a-key-value-cache-for-flash-storage/.

[20] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2012), FAST'12, USENIX Association, pp. 2–2.

[21] HOERNER, B., RAJASHEKHAR, M., YUE, Y., AND NYMEN, T. Fatcache. engineering.twitter.com/opensource/projects/fatcache.

[22] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 501–512.

[23] LI, C., SHILANE, P., DOUGLIS, F., AND WALLACE, G. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th Annual Middleware Conference* (New York, NY, USA, 2015), Middleware '15, ACM, pp. 50–62.

[24] LIM, H., ANDERSEN, D. G., AND KAMINSKY, M. Towards accurate and fast evaluation of multi-stage log-structured designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 149–166.

[25] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 1–13.

[26] LIU, R.-S., YANG, C.-L., LI, C.-H., AND CHEN, G.-Y. Duracache: A durable ssd cache using mlc nand flash. In *Proceedings of the 50th Annual Design Automation Conference* (New York, NY, USA, 2013), DAC '13, ACM, pp. 166:1–166:6.

[27] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 133–148.

[28] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (Philadelphia, PA, 2014), USENIX Association.

[29] MELLOR, C. Toshiba flashes 100TB QLC flash drive, may go on sale within months. really. http://www.theregister.co.uk/2016/08/10/toshiba_100tb_qlc_ssd//.

[30] MEZA, J., WU, Q., KUMAR, S., AND MUTLU, O. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2015), SIGMETRICS '15, ACM, pp. 177–190.

[31] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[32] OHSHIMA, S., AND TANAKA, Y. New 3D flash technologies offer both low cost and low power solutions. https://www.flashmemorysummit.com/English/Conference/Keynotes.html.

[33] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. Sdf: Software-defined flash for web-scale internet storage systems. *SIGARCH Comput. Archit. News 42*, 1 (Feb. 2014), 471–484.

[34] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium*

on Operating Systems Principles (SOSP '17) (Shanghai, China, October 2017).

[35] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. FlashTier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 267–280.

[36] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 67–80.

[37] SHEN, Z., CHEN, F., JIA, Y., AND SHAO, Z. Optimizing flash-based key-value cache systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (2016).

[38] TANG, L., HUANG, Q., LLOYD, W., KUMAR, S., AND LI, K. RIPQ: Advanced photo caching on flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 373–386.

[39] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, July 2015), USENIX Association, pp. 71–82.

[40] YANG, Q., AND REN, J. I-CASH: Intelligently coupled array of SSD and HDD. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture* (Washington, DC, USA, 2011), HPCA '11, IEEE Computer Society, pp. 278–289.