TBF: A Memory-Efficient Replacement Policy for Flash-based Caches

Cristian Ungureanu, Biplob Debnath, Stephen Rago, and Akshat Aranya
NEC Laboratories America
{cristian,biplob,sar,aranya}@nec-labs.com

***Abstract*—The performance and capacity characteristics of flash storage make it attractive to use as a cache. Recency-based cache replacement policies rely on an in-memory full index, typically a B-tree or a hash table, that maps each object to its recency information. Even though the recency information itself may take very little space, the full index for a cache holding N keys requires at least $\log N$ bits *per key*. This metadata overhead is undesirably high when used for very large flash-based caches, such as key–value stores with billions of objects.**

To solve this problem, we propose a new RAM-frugal cache replacement policy that approximates the least-recently-used (LRU) policy. It uses two in-memory Bloom sub-filters (TBF) for maintaining the recency information and leverages an on-flash key–value store to cache objects. TBF requires only one byte of RAM per cached object, making it suitable for implementing very large flash-based caches. We evaluate TBF through simulation on traces from several block stores and key–value stores, as well as evaluate it using the Yahoo! Cloud Serving Benchmark in a real system implementation. Evaluation results show that TBF achieves cache hit rate and operations per second comparable to those of LRU in spite of its much smaller memory requirements.

I. Introduction

Caching has been a very effective technique for dealing with the performance gap between CPU and disk storage. While caches have traditionally been set-up in RAM, a new storage technology is changing the landscape: flash memory based solid state drives (SSDs). SSDs have been used to accelerate access to filesystems [1], [2], [3] and DBMSs [4], [5]. In this paper we propose a new cache replacement policy for very large disk-based databases. It is particularly well-suited for use with flash-based caches for noSQL databases [6] implemented using key–value stores. Applications using them usually have enough locality to benefit from caching.

In some cases it is possible to use RAM as cache: *Memcached* [7], an in-RAM key–value store cache, lists Wikipedia, Twitter, YouTube, and others among its users. In other cases, the desired size of the cache is too large to fit in the available RAM, or is too expensive to do so. SSDs have higher random I/O performance than hard disk, higher capacity (density), are less expensive, and use less power per bit than DRAM — thus, making them attractive to use as an additional, or even only, cache layer.

To accelerate access to an on-disk key–value store, it is convenient to organize the on-flash cache also as a key–value store. To implement this cache, we can leverage existing flash-based key–value store implementations, which are especially optimized for the physical characteristics (i.e., asymmetric read and write performance, wear out, etc.) of flash memory. However, we note that these implementations are difficult to use as a caching layer to a slower store: some of them do not support caching operations; some support caching operations but link the caching replacement policy with the underlying data organization on flash; and some require a substantial amount of RAM for the data structures necessary to implement a cache replacement policy. For example, memory-efficient SkimpyStash [8] and SILT [9] do not support caching operations; HashCache [10] implements a local LRU for a small set of keys (corresponding to hash buckets), but does not support a global LRU (for the entire key space), and it uses 7 bytes of in-RAM metadata per object, which makes it expensive for large caches; FlashStore [11] needs a log-structured data layout and uses 6 bytes of RAM per key–value pair, while FlashCache [2] uses 24 bytes of RAM per pair.

In this paper, our goal is two-fold: first, we want a caching policy that is memory-efficient (has low per-object memory overhead); second, we want the caching policy to be agnostic to the organization of data on SSD, freeing the user to choose their preferred key–value store design.

Over the last decades numerous caching algorithms have been introduced; among them, algorithms based on recency, such as LRU (least-recently-used) and its variants (for example, CLOCK [12]) have been preferred in practice due to their simplicity [2], [4], [5], [7], [10], [13], [14]. Here, we focus on recency-based caching policies that are memory efficient.

Recency-based caching algorithms typically use two in-memory data structures: an *access data structure* that maintains the recency information, and an *index* that maps an object's key to its associated recency information. The index is usually implemented as a hash table or a B-tree. For the access data structure, LRU uses two pointers per object to maintain a doubly-linked list, which is used to move an accessed object into the *most recently used* position; eviction victims are selected from the *least recently used* position in the list. CLOCK [12] uses one bit per object, which is set to '1' when the object is accessed. In CLOCK, the objects are maintained conceptually in a unidirectional circular list, with a "hand" moving over the list in order to find a candidate for eviction. For every such candidate, CLOCK checks the associated bit: if the bit is currently set to '0', the candidate is evicted from cache; otherwise, the bit is reset to '0', and the CLOCK hand moves to the next candidate. This step is repeated until the CLOCK hand encounters a candidate with the bit set to '0'.

TABLE I: Memory usage in bits per cached object for various recency-based caching policies

Caching Algorithm	Memory Requirement Per Cached Object (in bits)			
	Access Info			
	Info Proper	Full Index	Key-Value Store	Total
LRU	$2lg(N)$	$\geq lg(N)$	shares full index	$\geq 3lg(N)$
CLOCK	1	$\geq lg(N)$	shares full index	$\geq lg(N)$
Our Algorithm	8	0	varying (key–value store specific)	8 + key–value store specific

TABLE II: Memory usage of a 2 TB cache containing varying sized objects

Caching Policy	Metadata Overhead	Object Size			
		1 MB	4 KB	1 KB	256 B
LRU	24 bytes	48 MB	12 GB	48 GB	192 GB
CLOCK	8 bytes	16 MB	4 GB	16 GB	64 GB
Our Algorithm	1 byte	2 MB	0.5 GB	2 GB	8 GB

Using the same data structures for large flash-based caches is unattractive because they would use too much memory. Table I shows the *minimum* memory usage for LRU and CLOCK, as well as for our new cache replacement policy. We assume that the index is built using a data structure that has at least one occurrence of each key (e.g. a B-tree or a hash table); if there are N distinct keys, each key and each pointer in LRU's doubly-linked list must be at least $\lg(N)$ bits long. The access information for CLOCK is 1 bit, and we assume that the clock order can be maintained with no additional space (for example, it is somehow derived from the full index). In practice, the requirements are likely higher than the minimum above: the linked list pointers are likely to be either 4 or 8 bytes; the keys themselves might be larger — 16 bytes if produced through MD5 hashing, or 20 bytes if SHA-1 hashing; the indexing structure might have some per-key overhead as well (e.g. B-trees).

Table II shows conservative estimates of metadata requirements for a 2 TB cache containing varying sized objects under several caching algorithms. We assume that keys and the pointers in LRU are 8 bytes, the index is a hash table with open addressing and a load factor of 100%. Thus, ignoring the one bit for the access info, CLOCK uses 8 bytes per object, while LRU uses 24 bytes. As expected, the metadata size is not a big problem in cases when the objects are larger, as in the case of ZFS (64 KB blocks) [1] or FlashCache (4-16 KB blocks) [2]. But consider the case of using an Amazon Web Services "high-I/O" instance [15] that uses its 2 TB SSD as cache for a larger store of 256 byte objects; such instances have only 60.5 GB of RAM, which is insufficient for using LRU or CLOCK.

A possible alternative is to keep the access information on SSD. This would work even with key–value store implementations that do not keep the entire index in memory. However, this is unattractive because it results in too many flash writes, which is detrimental to both performance and flash longevity. Consider the case of a workload consisting of 20% writes and 80% reads, having a cache hit rate of 75%. The *cache misses* cause a higher number of writes to flash, since they lead to new objects being inserted into the cache; thus, even when the access information is kept in memory, 40% of operations cause flash writes (assuming write-through, all application writes plus the read misses). Keeping the access information on flash exacerbates this problem by requiring writes to flash even for *read hits*.

To avoid the problems caused by keeping a full index in memory, or the access metadata on flash, we propose a novel caching policy that maintains the access information in an in-memory data structure, based on Bloom filters [16], that does not require an in-memory index. Our policy is key–value store agnostic, allowing it to work with any existing key–value store implementation that provides a *traversal operation* (a way to iterate over the keys in some implementation-defined order). A traversal operation is trivial to provide, and most key–value stores do so; we use it during eviction to select a good victim. In Section II, we will discuss in more detail the issues arising from concurrency between insertion and traversal operations, and the added I/O cost of replacing the in-memory traversal with an on-flash key–value store traversal.

A Bloom filter [16] is a space-efficient probabilistic data structure that supports *insert* and *membership* operations on a set. It consists of an array of m bits, initially all set to 0. There are k independent hash functions defined, each hashing a set element to one of the m bit positions. To insert an element, we first calculate all k bit positions, and set all them to 1. To test element membership in the set we check all relevant k bit positions; if any of these k bits are 0, then the element is not in this set. A nice survey of Bloom filters is given in [17]

The standard Bloom filter data structure does not provide a *delete* operation. However, our access data structure needs to account for evicted objects in some way. We have used two implementations that provide such functionality: one uses a $\underline{\text{B}}$loom $\underline{\text{f}}$ilter with $\underline{\text{d}}$eletion (BFD), in which we use a Bloom filter augmented with a deletion operation invoked at each eviction; the other uses $\underline{\text{t}}$wo $\underline{\text{B}}$loom sub-$\underline{\text{f}}$ilters (TBF) and periodically discards (zeroes-out) one sub-filter. Since both variants are essentially based on Bloom filters, we will informally refer to the access data structure as a *Bloom filter*, but keep in mind that it supports forgetting old recency information, an operation that we informally call "deletion" for both implementations. Although Bloom filters have false positives, and our addition of a deletion operations also introduces false negatives, they are tolerable for mapping objects to their access information. We'll briefly explain the intuition why false positives and false negatives might be tolerable for an accessed data structure; a detailed description of BFD and TBF implementations is given in Section II.

A false positive leads to an object being considered accessed recently even though it was not. This pollutes the cache, but as long as the false positive rate (FPR) is low, this is not a big concern; for example, an FPR of 1% means that at worst 1% of the cache is wasted, which is not likely to have a big impact on the cache hit rate.

False negatives are potentially more problematic. The TBF design does not require a deletion operation, and does not suffer from false negatives. However, in BFD a false negative arises due to Bloom filter hash function collisions during deletion, and causes a recently-accessed object to be considered *not* recently accessed, and incorrectly evicted. However, between the time an object has some of its bits reset because of collision with an object being evicted, and the time this object shows up in the traversal there is a chance that the incorrect bits have been "repaired" by intervening accesses; the higher the frequency of access to an object, the higher its chances of repair after incorrect deletion.

Both TBF and BFD implementations use one byte of RAM per cached object, resulting in the memory overheads shown in Table II. Note that this is separate from the amount of RAM used by the key–value store, and in particular its index. However, most key–value stores do *not* require having the entire index in memory; they either cache some portion of the index [18], or employ some other scheme that limits the amount of memory used [8], [9].

Our new policy resembles the CLOCK algorithm. We determine cache hit or miss by consulting the on-flash key–value store. Whenever an object is accessed, CLOCK sets the corresponding recency bit to '1'; similarly, we add that object's key to the set maintained by the Bloom filter. To select a victim for eviction, we use the traversal operation provided by the key–value store; this traversal approximates the CLOCK "hand" movement. For every candidate key, we look up its recency information in the Bloom filter; if the key *was not* recently accessed (does not belong to the set) we evict that object by removing it from the flash key–value store. If the key *was* recently accessed, we keep it in the cache, but "delete" it from the Bloom filter.

Note that keeping approximate information for *access* does not mean that key *lookups* are also approximate. Lookup is implemented by the key–value store, and is assumed to be exact; thus, a key that is present in the cache will never be considered a miss. The access information is only used to decide *evictions*, when the cache is full and a new object has to be inserted in it (during a new write or a read cache miss).

We evaluate the TBF and BFD policies, comparing them to recency-based policies LRU and CLOCK, as well as with RANDOM (a policy of evicting objects in random order). The comparison is done through simulation on traces from several block stores and key–value stores, as well as by building a real-prototype which executes the Yahoo! Cloud Serving Benchmark [19] on two Berkeley DB [18] key–value stores: one instance on disk (acts as main database) and one on flash (acts as cache). Our evaluation results show that TBF achieves performance comparable to that of LRU and CLOCK policies while using only one byte of memory per cached object.

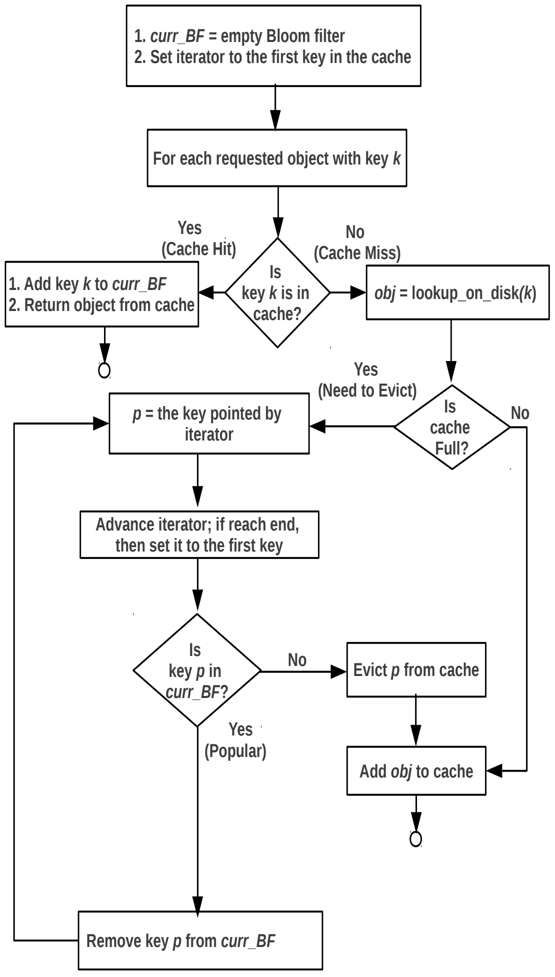
The rest of the paper is organized as follows: Section II describes our new policy, illustrating the design with simulation results; Section III presents an evaluation of the policy in a real system implementation using BerkeleyDB and Yahoo! Cloud Serving Benchmark; Section IV presents related work; Section V concludes the paper.

II. Memory-efficient Caching Policy

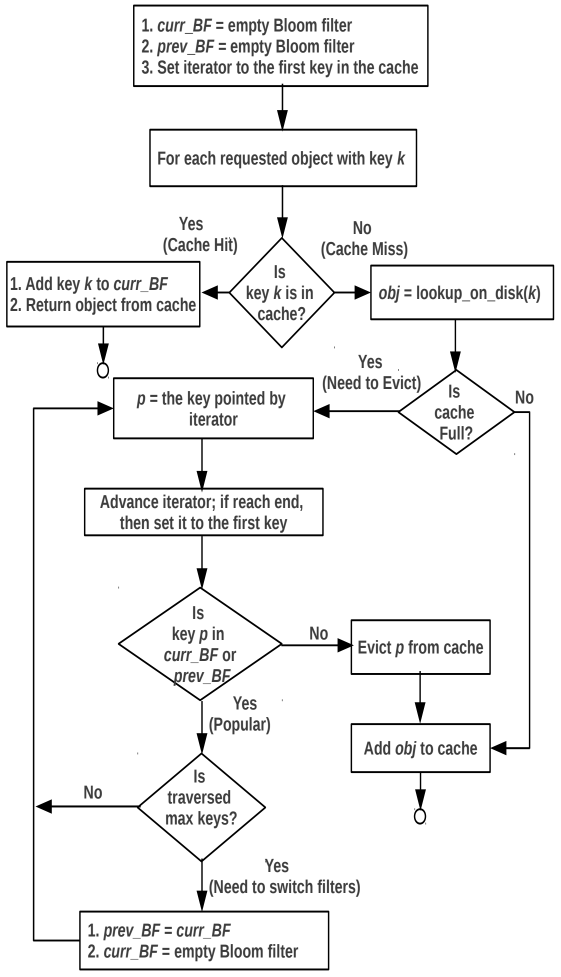
Recency-based caching algorithms need an *access data structure* that tracks accesses to objects. In-memory caching algorithms can maintain this data structure by leveraging the in-memory index they use to locate the objects. For example, the CLOCK algorithm [12] uses an extra bit (variations exist that use several bits), to be kept together with the key of each object in the cache. However, for very large caches on flash keeping an in-memory index is prohibitive, so maintaining the access information needs to be done some other way. We propose two data structures that can be used to implement recency-based policies approximating CLOCK (itself an approximation of LRU), but without an expensive in-memory index. The first is simply a Bloom filter with deletion, called BFD for short; the second, which we call TBF, uses two Bloom sub-filters (regular, without deletion).

BFD – Bloom Filter with Deletion: A flow chart for BFD operation is shown in Figure 1(a). When an object with key k is requested, we look for it in the key–value store; if found, we *mark* it by inserting k into the Bloom filter. If not found, a victim object is selected for eviction, and the object with key k is inserted into the cache. Similar to CLOCK, variations exist in which a newly inserted object is either marked or not. To find a victim, we iterate over the objects in the cache until we find an unmarked object; since we lack an in-memory index, we rely on the key–value store to provide the iteration capability. We do not require knowledge of the order over which the keys are iterated, but we require that any object appear only once during a traversal of the entire set (an object that has been removed and re-inserted may appear more than once). This property is provided by most key–value stores. Thus, the iterator plays the role of *clock hand* in the CLOCK algorithm. As the hand moves, CLOCK un-marks the object; the counterpart in BFD is removing that key from the set.

Removal of a key may be accomplished by choosing at random a subset of the functions used by the Bloom filter and resetting the bits at the corresponding hash-values, where the cardinality of the subset is between 1 and the maximum number of functions used. Note that this introduces *false negatives*: removing an element p might cause some other element q to be removed because of collisions between q's hash-values and hash-values in the subset chosen for resetting during p's removal. With the exception of these false negatives,


1. curr_BF = empty Bloom filter
2. Set iterator to the first key in the cache
For each requested object with key k
Yes (Cache Hit)
No (Cache Miss)
Is key k is in cache?
1. Add key k to curr_BF
2. Return object from cache
obj = lookup_on_disk(k)
Yes (Need to Evict)
Is cache Full?
No
p = the key pointed by iterator
Advance iterator; if reach end, then set it to the first key
Is key p in curr_BF?
No
Evict p from cache
Yes (Popular)
Add obj to cache
Remove key p from curr_BF

(a) One Bloom Filter with Deletion (BFD)


1. curr_BF = empty Bloom filter
2. prev_BF = empty Bloom filter
3. Set iterator to the first key in the cache
For each requested object with key k
Yes (Cache Hit)
No (Cache Miss)
Is key k is in cache?
1. Add key k to curr_BF
2. Return object from cache
obj = lookup_on_disk(k)
Yes (Need to Evict)
Is cache Full?
No
p = the key pointed by iterator
Advance iterator; if reach end, then set it to the first key
Is key p in curr_BF or prev_BF
No
Evict p from cache
Yes (Popular)
Add obj to cache
No
Is traversed max keys?
Yes (Need to switch filters)
1. prev_BF = curr_BF
2. curr_BF = empty Bloom filter

(b) Two Bloom sub-Filters (TBF)

Fig. 1: Flow chart diagram for the BFD and TBF caching policies

a key returned by the traversal but not found in the BFD corresponds to an object that is in the cache but has not been accessed recently, and thus is a good choice for eviction.

The false positive behavior of standard Bloom filters is understood very well and we know how to size them to obtain a low *false positive rate*; however, introducing a deletion operation not only introduces false negatives, but also change the calculation for false positives. For example, if only a strict subset of positions is reset, then some bits remain set to one, increasing the false positive rate. We will discuss in more detail the impact of false negatives in Section II-D.

TBF – Two Bloom sub-Filters: While BFD allows us to remove a key from the Bloom filter once the object has been evicted from the cache, we note that this removal need not be immediate; an object that is considered for eviction will only be considered again after all other objects have been considered. Until then, its marked or unmarked status is irrelevant. This suggests a solution in which we use two Bloom sub-filters and we drop many elements in bulk by resetting an entire sub-filter.

A flow chart for TBF operation is shown in Figure 1(b). In TBF we maintain two Bloom sub-filters: one current, cur_BF, and one previous, $prev\_BF$. The current filter is used to mark any keys that are cache hits; to evict, we search (by traversing the key–value store) for keys that are *not marked in either filter*. Periodically, we *flip* the filters: we discard $prev\_BF$, logically replace the previous filter with the current filter, and clear (empty) the current Bloom sub-filter.

When should we do a flip? One possibility is to keep marking elements in the current Bloom sub-filter until the number of bits set to zero equals the number of bits set to one (in some sense this is when the sub-filter is “full” — a Bloom filter sized to accommodate n objects will have roughly equal number of zero and one bits after n distinct insertions). However, note that we mark objects by inserting them in the current Bloom sub-filter when a key is accessed. A workload that has high locality of reference will lead to a very slow accumulation of ones in the Bloom filter. This

TABLE III: Traces used for simulation

Trace	Description
YCSB-Zipf	Zipf distribution with 95% reads
YCSB-latest	"Latest" distribution
YCSB-uniform	Uniform distribution
Financial	Two traces from UMass
Microsoft	`prxy volume1` trace from MSR

has the undesirable effect of keeping rarely-accessed objects marked in the Bloom filter together with the frequently-accessed objects for a long time.

Since the Bloom filter is not itself the cache, but rather keeps only access information, it need not be kept until it is full. Rather, the point is to provide information whether an object has been accessed recently. Because of this, we choose another option, closer in spirit to the CLOCK algorithm (which we intend to emulate using less memory): flip after we traverse a number of objects equal to the cache size (in objects). This allows TBF to approximate a useful property of the CLOCK policy: an accessed object survives in the cache for at least one full traversal (all other keys must be considered for eviction before this object is evicted). In fact, TBF "remembers" access information from between one and two full periods, somewhat similar to a counting-CLOCK variation (with a maximum count of two) [20].

A. Differences from CLOCK

While BFD and TBF behave similar to CLOCK, there are three important differences. The first is that the traversal potentially requires the key–value store on flash do I/O operations. In practice, we want to impose an upper bound on the amount of time it takes to find a victim and evict it to bring the newly-accessed object into the cache. Stopping the traversal before an unmarked object is found potentially results in poor eviction decisions. We will discuss the impact of the key–value store traversal on the caching policy in more detail later.

The second difference is that CLOCK inserts new objects "behind" the hand; thus, a newly inserted object is only considered for eviction after all other objects in the cache at the time of its insertion are considered. However, our traversal order is decided by the key–value store implementation and this property might not be satisfied. For example, a log-structured key–value store [11] will provide this property, while a key–value store with a B-tree index, such as [18], will not. Note that newly inserted objects might be encountered during the traversal before all the old objects are visited. Because of this, a complete traversal might visit a number of objects *higher than the cache size*. In such a case, cur_BF might be populated with more keys than intended. This provides another reason for our choice to replace $prev\_BF$ with cur_BF after a fixed number of objects is traversed.

The third difference comes from the false positive property of Bloom filters, which affects both TBF and BFD. Additionally, BFD has false negatives. Both false positives and false negatives lead to potentially poor decisions for eviction.

We now discuss in more detail each of these differences from CLOCK. To assess the impact of each difference quantitatively, we have performed simulations using the traces shown in Table III. YCSB is a trace generated by the Yahoo! Cloud Serving Benchmark [19]; we modified the benchmark to log the keys that are requested, and replayed the trace for simulation. We tested three different workloads: Zipf, uniform, and latest distributions, all with a mixture of 95% read operations and 5% update (read-modify-write) operations, for a disk-based key–value store of 200 million objects. Financial traces are taken from the UMass Trace Repository [21]. These block-level traces represent OLTP applications running at two large financial institutions. We have also used a trace collected at Microsoft Research Cambridge Lab [22]; we have chosen `prxy volume1`, a block-level trace that has relatively large mean read and write request rates per second.

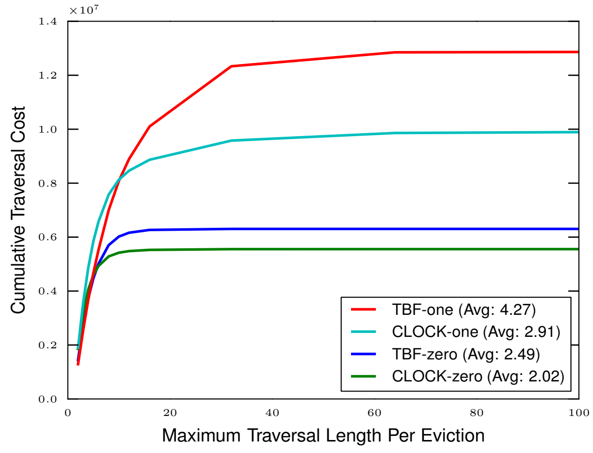
B. Key-Value Store Traversal

There are two main concerns regarding the traversal. First, unlike the case when the entire set of keys is stored in memory, iterating over the key–value store on flash incurs an I/O cost. We would like to keep this cost low. Second, it is possible that the key–value store traversal encounters a long sequence of marked objects. At an extreme, it is possible for all objects to be accessed between two traversals. We would like to keep the cost of traversal bounded, even in the worst case, to avoid unpredictable latencies.

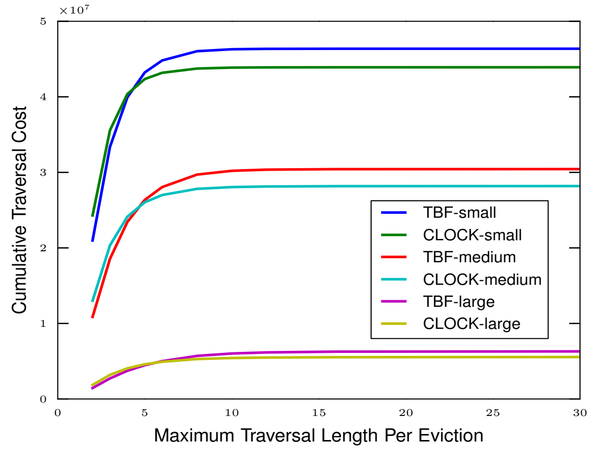
A simple, practical scheme is to limit the amount of time spent searching for a victim to the amount of time it takes to service the cache miss from disk. The number of keys traversed during this time varies not only with the types of flash and disk devices, but also with the internal organization of the key–value store on flash. A key–value store that has an index separate from the data — for example, a B-tree index — will bring into memory, on average, many keys with just one I/O operation. A key–value store that keeps data and metadata together — for example, a hash table organization — might bring into memory just one key per I/O operation. Even in such a case, however, we can traverse many keys on flash in the time it takes to perform one disk I/O operation.

On average, the number of keys that have to be traversed to find a victim depends on whether the objects newly inserted into the cache are marked. For in-memory caches, both variations are used. They offer the following trade-offs: if inserted unmarked, an object that is not subsequently accessed will be evicted more quickly (allowing other, potentially useful, objects to remain in cache longer). However, an object that has a reuse distance (defined as the cardinality of the set of items accessed in between the two accesses to the object) smaller than the cache size can still be evicted before being reused, whereas it would have been kept if marked on insertion.

In our experiments we have simulated both approaches. Figure 2(a) shows the effect of the choice of initial bit in the *cost of traversal*, which is defined as the total number of keys traversed for cache eviction during a simulation run using the large cache size from Figure 2(b). The y-axis shows the



(a) Initial bit



(b) Cache size

Fig. 2: Effect of initial bit and cache size on traversal cost. The y-axis shows the cumulative number of objects visited for each traversal limit.

cumulative number of objects visited for traversals up to the length given by the value on the x-axis (similar to a cumulative distribution function, but not normalized to 1), while the average number of objects visited per eviction is shown in the legend box. For example, we see that for CLOCK-one (objects marked on insertion), about 75% of traversals found an unmarked object in less than 10 visits.

Marking objects on insertion increases the average number of keys visited by the traversal by 44–71%. It also causes longer stretches of marked objects. This leads to a higher number of poor evictions (where we have to evict a marked object). The experiment shows that long stretches actually occur in CLOCK, when objects are marked on insertion. For TBF the situation is even worse, since the amount of false positives causes longer stretches than in the case of CLOCK. Note that in our case the traversal causes I/O to flash. Since having a higher number of traversals and a higher number of poor evictions are both undesirable, *we choose to leave objects unmarked on insertion*.

Figure 2(b) shows the effect of cache size (and hence, cache hit ratio) on the cost of traversal. The exact sizes of "small", "medium", and "large" are given in Table IV. The keys accessed are drawn from total number of 200 million keys; new entries are inserted unmarked when using TBF. Larger cache sizes achieve higher cache hit rates; this increases the traversal *per eviction*, since more clock bits are marked but decreases the number of evictions; overall, the *total* traversal cost is lower. Even though it doesn't increase the total traversal cost, a high limit to the number of candidates visited might require too much time to find a victim. By inserting new objects as unmarked, TBF reduces the probability of encountering stretches of marked objects longer than a given maximum during traversal; for example, a maximum of 10 leads to only 0.05% undesirable evictions. In such cases, when all traversed objects were accessed, TBF uses the two filters to determine whether the access happened recently (item is in cur_BF) or not (item is only in $prev\_BF$), thus doing better than a random eviction.

C. Insertion Order

CLOCK considers objects for eviction in the order in which they were inserted into the cache, thus guaranteeing that a newly inserted object will only be considered for eviction after all other objects in the cache are considered. In our case, the keys are not in memory, and both the traversal and the positions at which keys are inserted are imposed by the key–value store. Thus, this property might not hold (although there are cases where it might, such as a log-structured key–value store that provides the traversal in log order).

For uncorrelated traversal and insertion orders, a newly-inserted object is considered for eviction after half of the objects in the cache have been traversed, on average. We could guarantee that all other objects are considered at least once before a newly-inserted object by marking the objects on insertion, but as discussed previously, this increases the amount of keys visited by the traversal. If the I/O cost for traversal is not high, as in the case of key–value stores that have an index separated from the data where the key traversal can be done with little I/O, marking the objects on insertion might be an attractive option.

D. False Positives and False Negatives

The existence of false negatives and false positives in identifying marked objects is not a correctness issue. Bloom filter lookup is *not* used to determine whether an object is in cache — that is left to the key–value store; rather, it is used to decide evictions. As such, incorrect access information can lead to a poor eviction decision. Whether this is tolerable depends on the impact on the cache hit ratio.

A false negative arises when the traversal encounters an object that has been accessed since the last traversal, but its filter lookup fails (returns not-marked), leading to the object's incorrect eviction. We expect the penalty for false negatives

TABLE IV: Cache sizes used for simulation

	Cache Size (in number of objects)		
Workload	Small	Medium	Large
All YCSB	1,000,000	2,000,000	4,000,000
Financial	1,000	10,000	100,000
MSR proxy	4,000	20,000	40,000

will be small in practice. The intuition is that a frequently-accessed object, even if removed by mistake (through collision), will likely be accessed again before it is visited by the traversal, while the eviction of infrequently-accessed objects may not be too detrimental anyway. For an incorrect eviction to occur, the following conjunction of events has to happen: the object, o_1, is accessed at some time t_1; some other object o_2 with which o_1 collides is traversed at time $t_2 > t_1$ before the traversal reaches o_1 again; at least one of the bit positions on which o_2 collides with o_1 is actually among those that are reset; and there are no other accesses to o_1 before the traversal encounters it again. For frequently-accessed objects, we expect these conditions to be rarely met.

A false positive arises when the traversal encounters an object that was not accessed since the last traversal, but the Bloom filter has all the bits corresponding to that key's hash functions set to 1. In addition to the reason a standard Bloom filter (SBF) has false positives, a Bloom filter with deletion (BFD) might have additional false positives if the deletion operation does not reset all the bits. The lower the false positive rate, FPR, the lower the fraction of objects kept in memory by mistake, which pollute the cache. The first component of the FPR, originating from the nature of an SBF, can be kept low with only a few bits per object; for example, to achieve an $FPR < 0.01$, an SBF requires only 10 bits per object when using 5 hash functions. We will discuss later how large the second component of the FPR is for BFD, but this component can be reduced to zero if we reset *all* the bit positions during the remove operation. Note that an object causing pollution does not necessarily remain in the cache forever: in BFD, the traversal resets the bits of the objects even if not evicted.

Our goal is to provide a good approximation of CLOCK with as few bits as possible. A Bloom filter's false positives depends of the ratio $\frac{m}{n}$, where m is the number of bits, and n is the number of objects inserted into the Bloom filter, and k, the number of hash functions used. Usually, a Bloom filter is sized based on n, the number of keys it needs to accommodate. However, in our case, n does not represent the total number of objects in the cache but rather the number of *marked objects*, which is not known *a priori* (it depends on the workload's locality). From the traversal experiments with CLOCK (i.e., Figure 2(a)) we can see that, even at high cache hit rates, the traversal visits on average 2.02 objects before finding an unmarked one; this implies that only about half the objects in the cache are marked (for that workload).

We aim for a false positive rate in the single digits, and choose to use 4 bits per cached object; depending on the actual distribution of hits, this corresponds to between 4 and 8 bits per marked object. The optimal number of hash functions

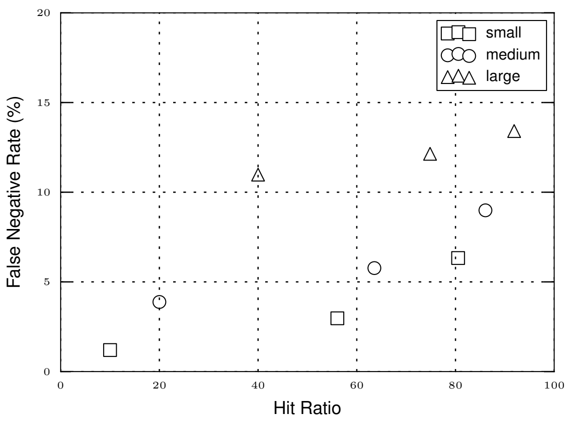


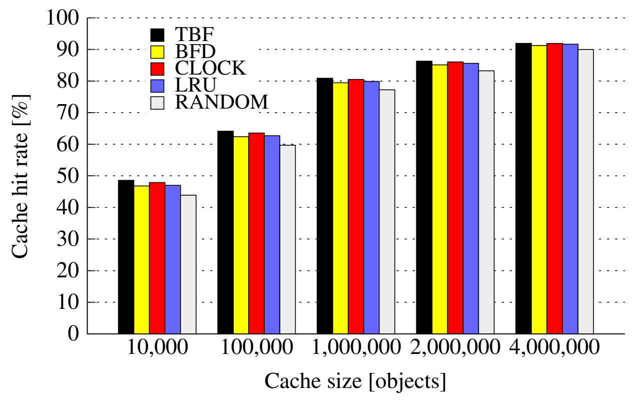
Fig. 3: BFD false negative rate for YCSB workloads

to use depends on $\frac{m}{n}$. In practice, we expect the ratio to be higher than 4, although not quite reaching 8 because the false positives feedback into the algorithm by increasing the number of apparently marked objects, which decreases $\frac{m}{n}$ by increasing n. To strike a balance, we have chosen to use 3 hash functions, a number that should work well over the range of likely $\frac{m}{n}$ ratios. Referring again to Figure 2(a), we see that the Bloom filter false positives actually increases somewhat the ratio of marked objects in the cache (from 2.02 to 2.49 in the experiment with the large cache size).

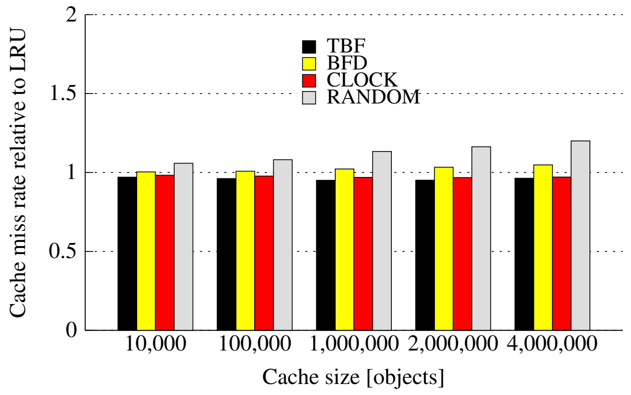
Both TBF and BFD aim to behave like CLOCK, but deviate from it because of the false positives, and in the case of BFD, the false negatives. To quantify how much they deviate, we run the CLOCK algorithm while maintaining the TBF and BFD data structures in parallel, and count how many times the algorithms take different decisions. However, if we take independent eviction decisions the two sets of cache objects will diverge, in which case the deviations will no longer be due just to false positives or false negatives. To avoid such behavior, we evict based on CLOCK for both algorithms; although it is only an approximation it should still give us some idea of how large the deviation is.

Figure 3 shows the false negative rates of BFD at various cache sizes for YCSB workloads. We observe that the false negative rate is correlated with the cache hit rate for each distribution, but not across distributions. In all cases it remains below 15%; as explained earlier, we expect that this should not affect the cache hit ratio too much. For comparison, false positive rates reached a maximum of 0.26% among all the evaluated workloads (not shown in the figure).

TBF has a small amount of positive deviations at all cache sizes, and no negative deviations. Of the positive deviations, some are due to Bloom filter collisions, but others are due to the fact that TBF preserves access information beyond one full traversal; CLOCK maintains access information for an object for exactly one traversal. TBF keeps the object marked for longer: an unused object will survive in cur_BF up to one full traversal, and will survive in $prev\_BF$ for another full traversal. Thus, TBF occasionally keeps in memory objects with a reuse distance longer than the cache size. For traces



(a) Cache hit rates



(b) Cache miss rates relative to LRU

Fig. 4: Caching behavior for YCSB workload with '"Latest" distribution

in which the distribution of the object's reuse distance does not have an abrupt drop exactly at the cache size, we expect TBF to perform slightly better than CLOCK. Note that more objects marked causes the traversal to visit more objects until an unmarked object is found, and thus the hand moves "faster".

E. Simulation Results

Figure 4(a) shows a comparison of the cache hit ratios for the YCSB benchmark with "latest" distribution using various caching policies. We notice that in absolute terms all caching policies give results within a 3% range. How important is a difference of a few percentage points? The higher the cache hit rate, the more important even a small difference is, because the system's performance is determined by the number of requests that have to be serviced from the slow device (disk). To highlight this difference, in Figure 4(b) we show the cache miss rate normalized to the miss rate of LRU.

We notice that TBF does better than all the others. BFD is worse than LRU when the cache hit rate is high, but better when the rate is low. CLOCK is better than LRU, and RANDOM does the worst. We also notice that both BFD and RANDOM become relatively worse as the cache hit rates increases. Intuitively, this happens because both algorithms have the potential to evict even objects that are accessed frequently (BFD does that because of the false negatives introduced by deletion). The higher the cache hit rate, the more expensive it is to make such poor choices.

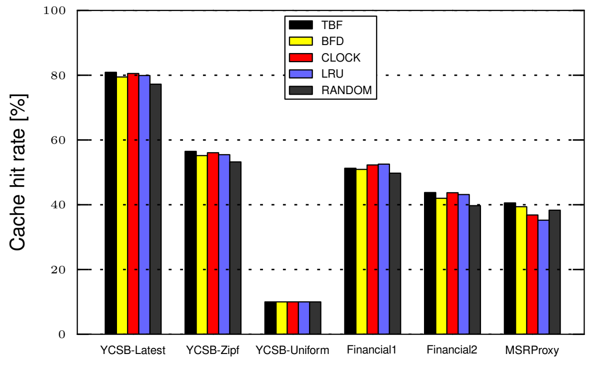
Figure 5 shows a comparison of the various caching policies for all the traces. Both TBF and BFD use one byte per object: TBF uses four bits per object for both its Bloom sub-filters and three hash functions, while BFD uses eight bits per object for its sole Bloom filter and five hash functions. Note that different workloads have different working set sizes. To show a reasonable comparison among the workloads, we chose three different cache sizes for each workload (the sizes are shown in Table IV).

These results show that TBF indeed approximates CLOCK and LRU algorithms very well, outperforming them by up to 5% in the cache miss rate. BFD achieves lower rates than TBF in all cases, and lower than CLOCK and LRU in most cases, but still better than those achieved by RANDOM. We also note that RANDOM performs reasonably well; even though this is not a new observation [23], actual deployments of RANDOM are not common. Part of the reason is that in-memory caches hold both data and metadata, and since the space occupied by metadata is typically a small fraction of that occupied by data, there is little to gain by using RANDOM in order to save memory. (We like to note that for flash-based caches this equation changes, since data is stored on flash and not in memory.) Another point is that when the cache miss rate is very low, even a small difference has a big performance impact, as can be seen in the MSR trace. Given these results, and the fact that TBF does not have to deal with false negatives, we prefer it over BFD.

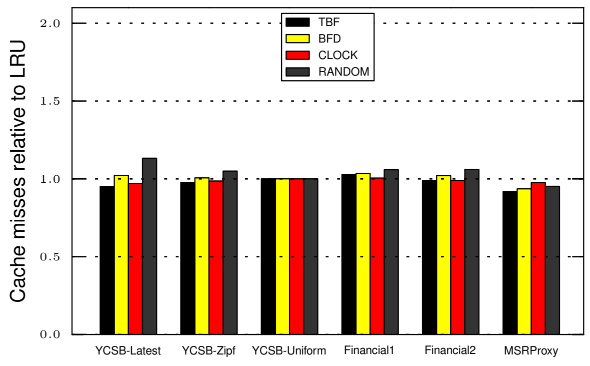
III. Evaluation

Simulating the various cache replacement algorithms allows us to compare cache hit ratios; although this ratio is a good indicator of performance, it does not account for real-life constraints that might make some approaches less viable than others. We complement our simulation results by evaluating each cache replacement policy using the Yahoo! Cloud Serving Benchmark [19]. YCSB generates a workload of requests to a key–value store, with the request keys conforming to selected distributions. We tested three different distributions: *Zipf*, *uniform*, and *latest*, all with a mixture of 95% read operations and 5% update (read-modify-write) operations.

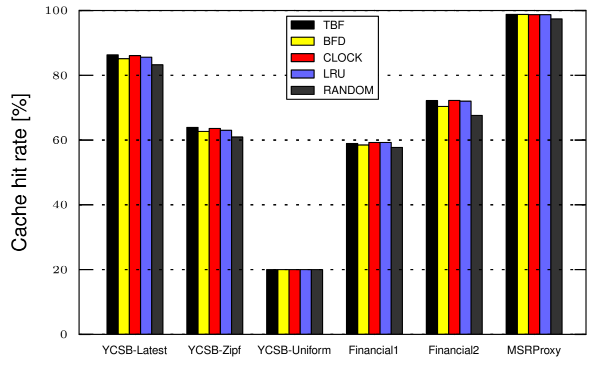
All experiments were run on an NEC Express5800 server with two quad-core Intel Xeon X5450 CPUs, two 1 TB 7200 RPM SATA disks in RAID-0 configuration, one 80 GB Fusion ioDrive PCIE X4 [24], and 8 GB of memory. Note that we use this particular setup only for illustration purposes; our



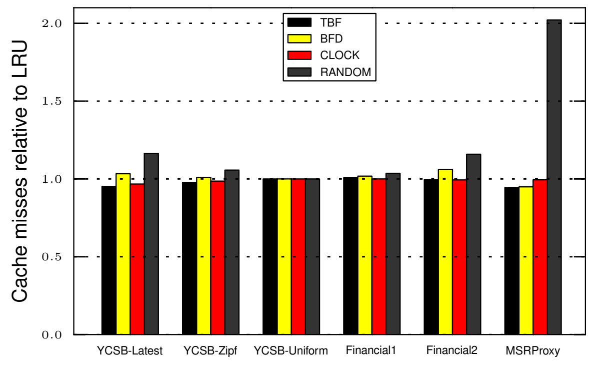
(a) Cache hit rate for "small" cache size



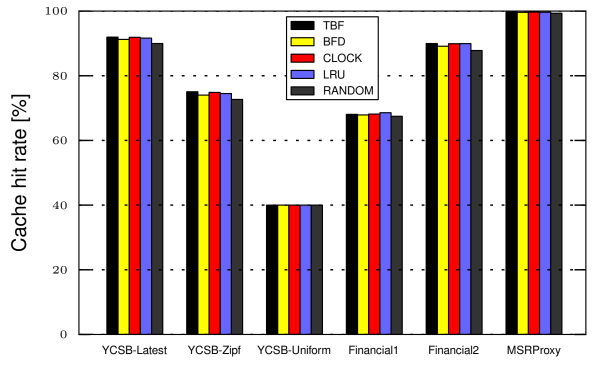
(b) Cache miss rate relative to LRU for "small" cache size



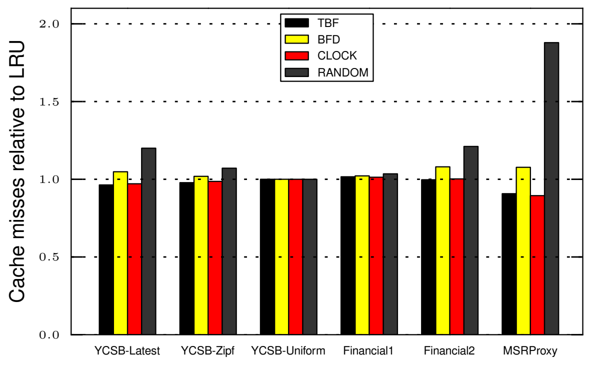
(c) Cache hit rate for "medium" cache size



(d) Cache miss rate relative to LRU for "medium" cache size



(e) Cache hit rate for "large" cache size



(f) Cache miss rate relative to LRU for "large" cache size

Fig. 5: Cache behavior for various workloads

algorithms are agnostic to flash devices as well as RAID configurations. The filesystem used for both the disks and the SSD is XFS [25]. Figure 6 shows the setup for this benchmark. To manage the disk storage, we used Berkeley DB [18] (BDB), which provides a key–value store interface. For simplicity, we also used BDB to manage the SSD storage acting as the cache. Both instances were configured to use the log-structured data layout; this layout offers the best performance for SSD storage, as reported in recent work [8], [11]. Our adapter module interposes between YCSB and the on-disk store, and implements the caching policy. The adapter can be configured to use either a write-back or a write-through policy.

When a record is read, the adapter checks the SSD cache. If the record is found, the read is satisfied by the cached copy. If the record isn't in the cache, the adapter reads the record from the database stored on the hard disk, after which it inserts it in the cache. If the cache is full, the adapter asks the replacement policy to select a victim to be evicted from the cache. The adapted handles record writes by writing to the SSD cache, and under the write-through policy, also writing to the database on hard disk. Under the write-back policy, the write to hard disk is deferred until the record is evicted from the cache. All experiments in this paper were performed with the write-through policy.

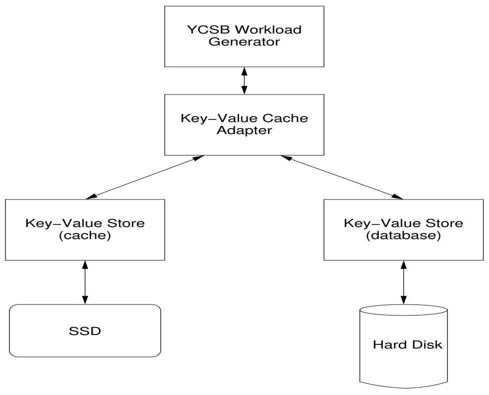


Fig. 6: Key-Value cache prototype architecture

TABLE V: Memory usage for various caching policies

Cache Policy	Memory Usage (GB)		
	Cache Policy	Java Heap	Total OS
LRU	3.05	7.0	8.0
CLOCK	1.5	5.8	6.8
TBF	0.15	4.1	5.1
RANDOM	0	3.8	4.8
NONE (no caching)	0	3.8	4.8

The on-disk key–value store was populated with 1.5 billion 256-byte objects (key–value pairs). The flash cache was sized to accommodate 10% of that (150 million objects). We measure the performance of each replacement algorithm with the cache warm. Before every test, we completely remove the cache stored on the SSD, as well as unmount and remount the file system containing the database on hard disk.

We are interested in finding out whether our algorithm achieves comparable performance with CLOCK and LRU while having much smaller memory requirements. To make such a comparison, we have booted-up the system with varying amounts of RAM to account for the differences between the algorithms. The on-disk key–value store, the on-flash key–value store, and the adapter implementing the caching logic run inside a Java Virtual Machine (JVM). For each experiment we configured the JVM with some base amount of memory plus whatever was calculated to be necessary for the caching policy, given the data structures that it uses. The total amount of RAM in the system was configured to be 1 GB more than that necessary for the JVM. The in-memory index for both LRU and CLOCK was implemented with a direct chaining hash table using a 150-million-entry array; keys are 8 bytes; both the bucket chains and the LRU's doubly-linked list use 4 byte pointers (integer indices into the array). Table V summarizes the amount of memory necessary for the various caching policies, as well as the JVM and the OS memory sizes used during experiments.

Table VI shows how the cache hit rates obtained through simulation compare with the average hit rates observed during the experiment. The difference between the simulated and observed rates are quite small (up to 0.07%). Such difference

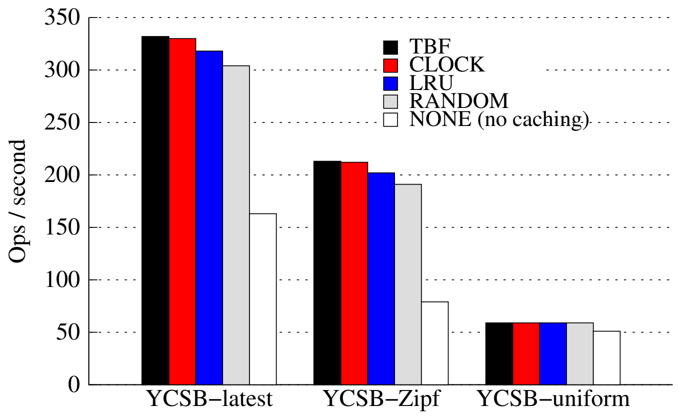


Fig. 7: Performance of various caching policies

can be attributed to two factors. One is that YCSB generates load from multiple threads, and thus the order in which the keys are requested varies from run to run. The other comes from the fact that the TBF and RANDOM caching algorithms rely on the key–value store traversal; the one assumed during simulation likely differs somewhat from the traversal actually executed by Berkeley DB. Since the differences are small, we have not investigated their cause in more depth.

Figure 7 shows the performance in YCSB operations per second. We make several observations. First, the performance of our algorithm is comparable with that of LRU and CLOCK in spite of the fact that it uses less memory. Second, for workloads that exhibit locality, such as YCSB-latest and YCSB-Zipf, TBF outperforms RANDOM (the other algorithm that is RAM-frugal).

We also observe that the performance of NONE differs across all workloads, despite the absence of a flash cache: it ranges from 51 operations/second for YCSB-uniform to 163 operations/second for YCSB-latest. We attribute this to caching in memory of metadata, both by Berkeley DB and by the underlying filesystem. The 51 operations/second achieved for the uniform distribution suggests that for each YCSB request the system executes about 6 I/Os to disk. Of these, only one is for data; the others are for metadata (Berkeley DB index and XFS file index). Workloads that exhibit more locality to data would also have metadata locality, in which case even small amounts of memory would improve performance.

The performance numbers also illustrate nicely that, as expected, the performance is not proportional to the cache hit rate, but rather is inversely proportional to the cache miss rate. Cache hits are served from the fast flash, while misses are served from the slow disk; therefore the miss rate dictates the overall performance. Increasing the cache hit rate from 77.7% to 84.9% (cache miss rates of 22.3% and 15.1% respectively) increases the performance from 213 to 332 operations/second. Thus, for example, even the small difference in cache hit rate between LRU and TBF for YCSB-Zipf, 77% versus 77.7% leads to a noticeable increase in performance.

As expected, the extra I/O to flash to perform traversal in search of good candidates for eviction does not significantly

TABLE VI: Comparison of cache hit rates, and the average number of key traversed during TBF to find good eviction victims

Workload	Cache Hit Rate (%)								Average Number of Keys Traversed by TBF
	TBF		CLOCK		LRU		RANDOM		
	Simulated	Observed	Simulated	Observed	Simulated	Observed	Simulated	Observed	
YCSB-Uniform	10.0	10.0	10.1	10.0	10.0	10.0	10.0	10.0	1.12
YCSB-Zipf	77.3	77.7	77.3	77.6	77.0	77.0	74.9	74.4	1.39
YCSB-Latest	84.4	84.9	84.4	84.8	84.1	84.1	82.2	81.5	1.51

affect performance. The average number of keys traversed is shown in Table VI. As expected, the higher the cache hit rate, the higher the average, as the traversal encounters more often keys that have been recently accessed; during our experiments there was no instance in which the traversal had to be stopped before finding a good victim.

IV. Related Work

In this section, we briefly review recent work related to our design and highlight differences with our approach.

A. Flash-based Key-Value Stores Supporting Caching Policies

Several key–value stores have been designed to act as a cache, using flash for backing storage. The primary difference with our work is that we seek to design a cache replacment algorithm that is both RAM-frugal and agnostic to key–value store implementation.

FlashStore [11] is a high-throughput, persistent key–value store that uses flash memory as a non-volatile cache between RAM and hard disk. It uses 6 bytes of RAM per key–value pair. To track recency information, FlashStore maintains an additional bit per key–value pair, which helps it to implement a CLOCK-like algorithm, based on the assumption that the on-flash key–value store is organized in a log-structured manner. In contrast, our design neither requires that a key–value store maintain in-RAM information per pair, nor does it need a log-structured data layout.

BufferHash [26] builds a content-addressable memory system using flash storage. It uses 4 bytes of RAM per key–value pair. BufferHash can only efficiently support a coarse-grained FIFO policy. However, it cannot implement LRU and other caching policies without incurring additional RAM and flash space overhead. For example, to implement an LRU policy, it reinserts a key, whenever a key is accessed and not present in memory; thus incurs additional RAM overhead as the same key is stored multiple times. Moreover, all the caching policy implementations are tied to BufferHash's internal data layout, so it is not directly portable to other key–value store designs.

HashCache [10] designs a memory-efficient LRU policy for web caches. On average, it needs 7 bytes of RAM space per key–value pair. However, the HashCache design is not applicable to key–value stores that do not use chaining-based hash table indexes.

FlashCache [2] is a Linux kernel module developed by Facebook that implements a block-level flash-based cache. It supports LRU and FIFO caching policies, while it needs 24 bytes of RAM per block stored on flash. Similary, FlashTier [27] takes approximately 8.4 bytes per block. In addition, recently there are some proposals to extend database buffer pools using SSDs [4], [5], [28], however they do not focus on improving the memory effciency of the underlying caching algorithms.

B. Flash-based Memory-Efficient Key-Value Stores

SkimpyStash [8] provides a RAM-efficient implementation of a flash-based key–value store, using on average, 1 ($\pm$ 0.5) bytes of RAM per key–value pair. It "moves" most of the pointers that locate each key–value pair from RAM to flash. It uses a hash table directory in RAM to index key–value pairs stored in a log-structure fashion on flash. Similarly, FAWN-DS [29] uses a memory-efficient hash index structure to organize key–value pairs on flash memory. On average, it uses 12 bytes of RAM per key–value pair. SILT [9], a recent improvement of FAWN-DS, reduces memory consumption to only 0.7 bytes per key–value pair by combining partial-key cuckoo hashing and entropy-coded tries with three different modular key–value store designs.

Although SkimpyStash, FAWN, and SILT have been designed with memory-efficiency in mind, these designs make no provisions for use as a cache. Our approach is complementary to these key–value store designs. By leveraging the underlying hash table index and using two in-memory Bloom sub-filters, we can easily implement recency-based caching schemes over these RAM-frugal key–value stores.

C. Memory-Efficient Data Structures

In this paper, we use Bloom filters [17] to track recency information. We periodically delete some elements from the Bloom filter to forget past accesses. In our Bloom filter with deletion (BFD) design, the bits are reset randomly, in a way similar to [30]. Another Bloom filter variant that supports object deletion is the counting Bloom filter [31]. While it is possible to use counting Bloom filters, the resulting policy would resemble more least-frequently-used (LFU) than LRU, which was not our intent. The main problem with these two variants that support individual object deletion is that they introduce random false negatives, i.e., other objects are unmarked and are possibly evicted by mistake as a result, regardless of the recency of their last access. In addition, a counting Bloom filter increases memory consumption by four times compared to a standard Bloom filter [31].

To solve the false negative problem, we use a design consisting of two Bloom sub-filters (TBF). TBF has no false negatives, and it guarantees that data accessed with at least some recency will not be evicted by mistake. Recently, the technique of using two sub-filters has been used to track dynamic sets [32], [33]. However, these designs do not support key–value stores, as they are only focused on set membership functionality, where the items correspond to network flows in routers. In addition, these designs are intended for answering cache membership queries directly. In contrast, we use Bloom filters to answer *recency* questions and we leverage the key–value store to answer cache membership queries directly. As such, there are differences in the "goodness" criteria for the Bloom filter: we optimize the TBF design to emulate recency-based CLOCK policy [12]. As a result, we put a bound on the staleness of the access information by performing timely Bloom filter flips; the other two designs are optimized to *minimize flipping*, as this leads to spikes (temporary increases) in the cache miss rate.

Recently, SPOCA used a series of 17 Bloom filters to implement an aging effect [34]. In contrast, we use only two filters, which help to save memory. Psounis et al. [35] propose a sampling-based randomized cache replacement scheme for web caches that use utility-based functions for deciding victims for evictions. It avoids the cost of maintaining a priority queue data structure to choose a victim. However, it still requires an in-memory index for every key in the cache, as well as the related metadata used to compute the utility function. This scheme is not compatible with our approach as we do not maintain exact, one-to-one, key-to-metadata mapping information in memory.

V. Conclusion

In this paper, we propose an RAM-frugal policy that approximates recency-based LRU replacement policy for flash-based caches. To save RAM, we decouple the *access data structure* from the *index structure* of an LRU cache implementation. We use two in-RAM Bloom sub-filters (TBF) to keep track of recency information, while we do not maintain any in-RAM index; we leverage the underlying on-flash key–value store to cache objects, and iterate over cached objects to search for good victims.

In the TBF design, we periodically reset one Bloom sub-filter to forget past recency information, while retaining current recency information in the other sub-filter. This allows TBF to approximate an LRU policy very well. For each Bloom sub-filter, we recommend using $\frac{m}{n} = 4$ and $k = 3$, where m is the filter size in bits, n is the maximum number of objects tracked in the cache, and k is the number of hash functions. Therefore, in total TBF takes $4*2 = 8$ bits (one byte) of RAM space for each cached objects. Experimental results show that TBF algorithm gives similar (or higher) hit ratios compared to widely used LRU and CLOCK (an LRU variant) algorithms for a diverse set of workloads. In addition, implementation in a real prototype demonstrates that the TBF algorithm achieves similar (or higher) operations per second (OPS) compared to in-memory CLOCK and LRU implementations, while consuming 10 to 20 times less memory.

Acknowledgment

We thank Erik Kruus and the anonymous reviewers for their valuable feedback.

References

[1] B. Gregg, "ZFS L2ARC ," http://137.254.16.27/brendan/entry/test.

[2] M. Srinivasan, "Flashcache Released by FaceBook," https://github.com/facebook/flashcache/tree/master/doc/flashcache-doc.txt.

[3] "bcache: A Linux Kernel Block Layer Cache," http://bcache.evilpiepirate.org/.

[4] M. Canim, G. Mihaila, B. Bhattacharjee, K. Ross, and C. Lang, "SSD Bufferpool Extensions for Database Systems," *Proc. VLDB Endow.*, vol. 3, Sep 2010.

[5] J. Do, D. Zhang, J. Patel, D. DeWitt, J. Naughton, and A. Halverson, "Turbocharging DBMS Buffer Pool Using SSDs," in *SIGMOD*, 2011.

[6] "noSQL," http://www.wikipedia.org/wiki/Nosql.

[7] B. Fitzpatrick, "Distributed Caching with Memcached," *LINUX Journal*, vol. 124, Aug 2004.

[8] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: A RAM Space Skimpy Key-Value Store on Flash-based Storage," in *SIGMOD*, 2011.

[9] H. Lim, B. Fan, D. Andersen, and M. Kaminsky, "SILT: A Memory-Efficient, High-Performance Key-Value Store," in *SOSP*, 2011.

[10] A. Badam, K. Park, V. Pai, and L. Peterson, "HashCache: Cache Storage for the Next Billion," in *NSDI*, 2009.

[11] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High Throughput Persistent Key-Value Store," *Proc. VLDB Endow.*, vol. 3, Sep 2010.

[12] F. Corbato, "A Paging Experiment with the Multics System," in *MIT Project MAC Report MAC-M-384*, 1968.

[13] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory," in *USENIX ATC*, 2010.

[14] B. Zhu, K. Li, and H. Patterson, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System," in *FAST*, 2008.

[15] "AWS High I/O Instances," http://aws.amazon.com/ec2/instance-types/#highio-instances.

[16] B. Bloom, "Space/Time Tradeoffs in Hash Coding with Allowable Errors," *CACM*, vol. 13, no. 7, Jul. 1970.

[17] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," in *Internet Mathematics*, 2002.

[18] M. Olson, K. Bostic, and M. Seltzer, "Berkeley DB," in *USENIX ATC*, 1999.

[19] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *SoCC*, 2010.

[20] A. J. Smith, "Sequentiality and Prefetching in Database Systems," *TODS*, vol. 3, no. 3, Sep. 1978.

[21] "UMass Trace Repository," http://traces.cs.umass.edu/index.php/Storage/Storage.

[22] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," in *FAST*, 2008.

[23] R. Arpaci-Dusseau and A. Arpaci-Dusseau, "Operating Systems: Four Easy Pieces," http://pages.cs.wisc.edu/~remzi/OSFEP/.

[24] "Fusion's ioDrive," http://www.fusionio.com/platforms/iodrive/.

[25] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," in *USENIX ATC*, 1996.

[26] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath, "Cheap and Large CAMs for High Performance Data-Intensive Networked Systems," in *NSDI*, 2010.

[27] M. Saxena, M. M. Swift, and Y. Zhang, "FlashTier: A Lightweight, Consistent and Durable Storage Cache," in *EuroSys*, 2012.

[28] I. Koltsidas and S. Viglas, "Designing a Flash-Aware Two-Level Cache," in *ADBIS*, 2011.

[29] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A Fast Array of Wimpy Nodes," in *SOSP*, 2009.

[30] J. Lim and K. Shin, "Gradient-Ascending Routing via Footprints in Wireless Sensor Networks," in *RTSS*, 2005.

[31] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Transaction on Networking*, vol. 8, June 2000.

[32] F. Chang, W. Feng, and K. Li, "Approximate Caches for Packet Classification," in *INFOCOM*, 2004.

[33] M. Yoon, "Aging Bloom Filter with Two Active Buffers for Dynamic Sets," *IEEE TKDE*, vol. 22, Jan 2010.

[34] A. Chawla, B. Reed, K. Juhnkey, and G. Syed, "Semantics of Caching with SPOCA: A Stateless, Proportional, Optimally-Consistent Addressing Algorithm," in *USENIX ATC*, 2011.

[35] K. Psounis and B. Prabhakar, "Efficient Randomized Web-Cache Replacement Schemes Using Samples From Past Eviction Times," *Transaction on Networking*, vol. 10, no. 4, Aug 2002.