RIPQ: Advanced Photo Caching on Flash for Facebook[*]

Linpeng Tang[*], Qi Huang[†⋆], Wyatt Lloyd[‡⋆], Sanjeev Kumar[⋆], Kai Li[*]

[*]Princeton University, [†]Cornell University, [‡] University of Southern California, [⋆]Facebook Inc.

Abstract

Facebook uses flash devices extensively in its photo-caching stack. The key design challenge for an efficient photo cache on flash at Facebook is its workload: many small random writes are generated by inserting cache-missed content, or updating cache-hit content for advanced caching algorithms. The Flash Translation Layer on flash devices performs poorly with such a workload, lowering throughput and decreasing device lifespan. Existing coping strategies under-utilize the space on flash devices, sacrificing cache capacity, or are limited to simple caching algorithms like FIFO, sacrificing hit ratios.

We overcome these limitations with the novel Restricted Insertion Priority Queue (RIPQ) framework that supports advanced caching algorithms with large cache sizes, high throughput, and long device lifespan. RIPQ aggregates small random writes, co-locates similarly prioritized content, and lazily moves updated content to further reduce device overhead. We show that two families of advanced caching algorithms, Segmented-LRU and Greedy-Dual-Size-Frequency, can be easily implemented with RIPQ. Our evaluation on Facebook's photo trace shows that these algorithms running on RIPQ increase hit ratios up to ~20% over the current FIFO system, incur low overhead, and achieve high throughput.

1 Introduction

Facebook has a deep and distributed photo-caching stack to decrease photo delivery latency and backend load. This stack uses flash for its capacity advantage over DRAM and higher I/O performance than magnetic disks.

A recent study [22] shows that Facebook's photo caching hit ratios could be significantly improved with more advanced caching algorithms, i.e., the Segmented-LRU family of algorithms. However, naive implementations of these algorithms perform poorly on flash. For example, Quadruple-Segmented-LRU, which achieved ~70% hit ratio, generates a large number of small random writes for inserting missed content (~30% misses) and updating hit content (~70% hits). Such a random write heavy workload would cause frequent garbage collections at the Flash Translation Layer (FTL) inside modern NAND flash devices—especially when the write size is small—resulting in high write amplification, decreased throughput, and shortened device lifespan [37].

Existing approaches to mitigate this problem often reserve a significant portion of device space for the FTL (over-provisioning), hence reducing garbage collection frequency. However, over-provisioning also decreases available cache capacity. As a result, Facebook previously only used a FIFO caching policy that sacrifices the algorithmic advantages to maximize caching capacity and avoid small random writes.

Our goal is to design a flash cache that supports advanced caching algorithms for high hit ratios, uses most of the caching capacity of flash, and does not cause small random writes. To achieve this, we design and implement the novel Restricted Insertion Priority Queue (RIPQ) framework that efficiently approximates a priority queue on flash. RIPQ presents programmers with the interface of a priority queue, which our experience and prior work show to be a convenient abstraction for implementing advanced caching algorithms [14, 45].

The key challenge and novelty of RIPQ is how to translate and approximate updates to the (exact) priority queue into a flash-friendly workload. RIPQ aggregates small random writes in memory, and only issues aligned large writes through a restricted number of insertion points on flash to prevent FTL garbage collection and excessive memory buffering. Objects in cache with similar priorities are co-located among these insertion points. This largely preserves the fidelity of advanced caching algorithms on top of RIPQ. RIPQ also lazily moves content with an updated priority only when it is about to be evicted, further reducing overhead without harming the fidelity. As a result, RIPQ approximates the priority queue abstraction with high fidelity, and only performs consolidated large aligned writes on flash with low write amplification.

We also present the Single Insertion Priority Queue (SIPQ) framework that approximates a priority queue with a single insertion point. SIPQ is designed for memory-constrained environments and

[*]Technical report for the same paper published in FAST'15.

enables the use of simple algorithms like LRU, but is not suited to support more advanced algorithms.

RIPQ and SIPQ have applicability beyond Facebook's photo caches. They should enable the use of advanced caching algorithms for static-content caching—i.e., read-only caching—on flash in general, such as in Netflix's flash-based video caches [38].

We evaluate RIPQ and SIPQ by implementing two families of advanced caching algorithms, Segmented-LRU (SLRU) [27] and Greedy-Dual-Size-Frequency (GDSF) [16], with them and testing their performance on traces obtained from two layers of Facebook's photo-caching stack: the Origin cache co-located with backend storage, and the Edge cache spread across the world directly serving photos to the users. Our evaluation shows that both families of algorithms achieve substantially higher hit ratios with RIPQ and SIPQ. For example, GDSF algorithms with RIPQ increase hit ratio in the Origin cache by 17-18%, resulting in a 23-28% reduction in I/O Operations Per Second (IOPS) to the backend.

The contributions of this paper include:

- A flash performance study that identifies a significant increase in the minimum size for max-throughput random writes and motivates the design of RIPQ.
- The design and implementation of RIPQ, our primary contribution. RIPQ is a framework for implementing advanced caching algorithms on flash with high space utilization, high throughput, and long device lifespan.
- The design and implementation of SIPQ, an upgrade from FIFO in memory constrained environments.
- An evaluation on Facebook photo traces that demonstrates advanced caching algorithms on RIPQ (and LRU on SIPQ) can be implemented with high fidelity, high throughput, and low device overhead.

2 Background & Motivation

Facebook's photo-serving stack, shown in Figure 1, includes two caching layers: an Edge cache layer and an Origin cache. At each cache site, individual photo objects are hashed to different caching machines according to their URI. Each caching machine then functions as an independent cache for its subset of objects.[1]

The *Edge cache layer* includes many independent caches spread around the globe at Internet Points of Presence (POP). The main objective of the Edge caching layer—in addition to decreasing latency for users—is decreasing the traffic sent to Facebook's datacenters, so the metric for evaluating its effectiveness is byte-wise hit ratio. The *Origin cache* is a single cache distributed across Facebook's datacenters that sits behind the Edge cache. Its main objective is decreasing requests to Facebook's disk-based storage backends, so the metric for its effectiveness is object-wise hit ratio. Facing high request rates for a large set of objects, both the Edge and Origin caches are equipped with flash drives.

This work is motivated by the finding that SLRU, an advanced caching algorithm, can increase the byte-wise and object-wise hit ratios in the Facebook stack by up to 14% [22]. However, two factors confound naive implementations of advanced caching algorithm on flash. First, the best algorithm for workloads at different cache sites varies. For example, since Huang et al. [22], we have found that GDSF achieves an even higher object-wise hit ratio than SLRU in the Origin cache by favoring smaller objects (see Section 6.2), but SLRU still achieves the highest byte-wise hit ratio at the Edge cache. Therefore, a unified framework for many caching

[1] Though the stack was originally designed to serve photos, now it handles videos, attachments, and other static binary objects as well. We use "objects" to refer to all targets of the cache in the text.

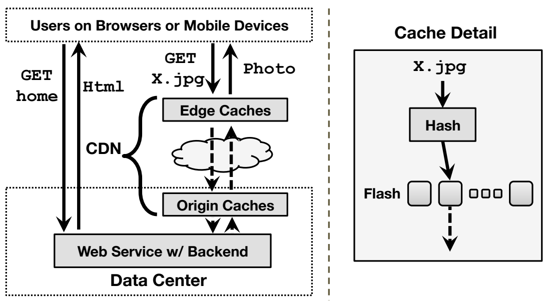


Figure 1: Facebook photo-serving stack. Requests are directed through two layers of caches. Each cache hashes objects to a flash equipped server.

Device	Model A	Model B	Model C
Capacity	670GiB	150GiB	~1.8TiB
Interface	PCI-E	SATA	PCI-E
Seq Write Perf	590MiB/s	160MiB/s	970MiB/s
Rand Write Perf	76MiB/s	19MiB/s	140MiB/s
Read Perf	790MiB/s	260MiB/s	1500MiB/s
Max-Throughput Write Size	512MiB	256MiB	512MiB

Table 1: Flash performance summary. Read and write sizes are 128KiB. Max-Throughput Write Size is the smallest power-of-2 size that achieves sustained maximum throughput at maximum capacity.

algorithms can greatly reduce the engineering effort and hasten the deployment of new caching policies. Second, flash-based hardware has unique performance characteristics that often require software customization. In particular, a naive implementation of advanced caching algorithms may generate a large number of small random writes on flash, by inserting missed content or updating hit content. The next section demonstrates that modern flash devices perform poorly under such workloads.

3 Flash Performance Study

This section presents a study of modern flash devices that motivates our designs. The study focuses on write workloads that stress the FTL on the devices because write throughput was the bottleneck that prevented Facebook from deploying advanced caching algorithms. Even for a read-only cache, writes are a significant part of the workload as missed content is inserted with a write. At Facebook, even with the benefits of advanced caching algorithms, the maximum hit ratio is ~70%, which results in at least 30% of accesses being writes.

Previous studies [19, 37] have shown that small random writes are harmful for flash. In particular, Min et al. [37] shows that at high space utilization, i.e., 90%, random write size must be larger than 16 MB or 32 MB to reach peak throughput on three representative SSDs in 2012, with capacities ranging between 32 GB and 64 GB. To update our understanding to current flash devices, we study the performance characteristics on three flash cards, and their specifications and major metrics are listed in Table 1. All three devices are recent models from major vendors,[2] and A and C are currently deployed in Facebook photo caches.

3.1 Random Write Experiments

This subsection presents experiments that explore the trade-off space between write size and device over-provisioning on random write performance. In these experiments we used different sizes to partition the device and then perform aligned random writes of that size under varying space utilizations. We use the flash drive as a raw block device to avoid filesystem overheads. Before each run we use `blkdiscard` to clear the existing data, and then repeatedly pick a random aligned location to perform write/overwrite. We write to the device with 4 times the data of its total capacity before reporting the final stabilized throughput. In each experiment, the initial throughput is always high, but as the device becomes full, the garbage collector kicks in, causing FTL write amplification and dramatic drop in throughput.

[2] Vendor/model omitted due to confidentiality agreements.

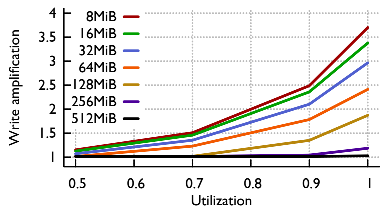
During garbage collection, the FTL often writes more data to the physical device than what is issued by the host, and the byte-wise ratio between these two write sizes is the *FTL write amplification* [21]. Figure 2a and Figure 2b show the FTL write amplification and device throughput for the random write experiments conducted on the flash drive Model A. The figures illustrate that as writes become smaller or space utilization increases, write throughput dramatically decreases and FTL write amplification increases. For example, 8 MiB random writes at 90% device utilization achieve only 160 MiB/s, a ~3.7x reduction from the maximum 590 MiB/s. We also experimented with mixed read-write workloads and the same performance trend holds. Specifically, with a 50% read and 50% write workload, 8 MiB random writes at 90% utilization lead to a ~2.3x throughput reduction. High FTL write amplification also reduces device lifespan, and as the erasure cycle continues to decrease for large capacity flash cards, the effects of small random writes become worse over time [10, 39].

Similar throughput results on flash drive Model B are shown in Figure 2c. However, its FTL write amplification is not available due to the lack of monitoring tools for physical writes on the device. Our experiments on flash drive Model C (details elided due to space limitations) agree with Model A and B results as well. Because of the low throughput under high utilization with small write size, more than 1000 device hours are spent in total to produce the data points in Figure 2.

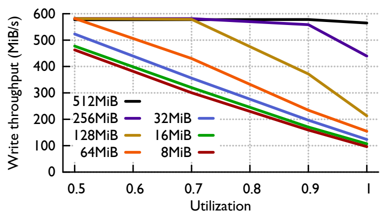
While our findings agree with the previous study [37] in general, we are surprised to find that under 90% device utilization, the minimum write size to achieve peak random write throughput has reached 256 MiB to 512 MiB. This large write size is necessary because modern flash hardware consists of many parallel NAND flash chips [8] and the aggregated erase block size across all parallel chips can add up to hundreds of megabytes. Communications with vendor engineers confirmed this hypothesis. This constraint informs RIPQ's design, which only issues large aligned writes to achieve low write amplification and high throughput.

3.2 Sequential Write Experiment

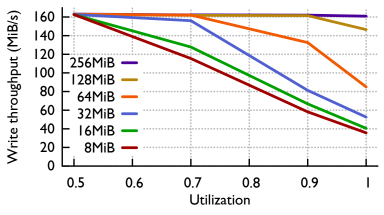
A common method to achieve sustained high write throughput on flash is to issue sequential writes. The FTL can effectively aggregate sequential writes to parallel erase blocks [31], and on deletes and over-



(a) Write amplification for Model A



(b) Throughput for Model A



(c) Throughput for Model B

Figure 2: Random write experiment on Model A and Model B.

writes all the parallel blocks can be erased together without writing back any still-valid data. As a result, the FTL write amplification can be low or even avoided entirely. To confirm this, we also performed sequential write experiments to the same three flash devices. We observed sustained high performance for all write sizes above 128KiB as reported in Table 1.[3] This result motivates the design of SIPQ, which only issues sequential writes.

4 RIPQ

This section describes the design and implementation of the RIPQ framework. We show how it approximates the priority queue abstraction on flash devices, present its implementation details, and then demonstrate that it efficiently supports advanced caching algorithms.

4.1 Priority Queue Abstraction

Our experience and previous studies [14, 45] have shown that a *Priority Queue* is a general abstraction that naturally supports various advanced caching policies. RIPQ provides that abstraction by maintaining content in its internal approximate priority queue, and allowing cache operations through three primitives:

- insert(x, p): insert a new object x with priority value p.
- increase(x, p): increase the priority value of x to p.
- delete-min(): delete the object with the lowest priority.

The priority value of an object represents its utility to the caching algorithm. On a hit, *increase* is called to adjust the priority of the accessed object. As the name suggests, RIPQ limits priority adjustment to increase only. This constraint simplifies the design of RIPQ and still allows almost all caching algorithms to be implemented. On a miss, *insert* is called to add the accessed object. *Delete-min* is implicitly called to remove the object with the minimum priority value when a cache eviction is triggered by insertion. Figure 3 shows the architecture of a caching solution implemented with the priority queue abstraction, where RIPQ's components are highlighted in gray. These components are crucial to avoid a small-random-writes workload, which can be generated by a naive implementation of priority queue. RIPQ's internal mechanisms are further discussed in Section 4.2.

Absolute/Relative Priority Queue Cache designers using RIPQ can specify the priority of their content based on access time, access frequency, size, and many other factors depending on the caching policy. Although traditional priority queues typically use *absolute* priority values that remain fixed over time, RIPQ operates on a different *relative* priority value interface. In a relative priority queue, an object's priority is a number in the $[0, 1]$ range representing the position of the object relative to the rest of the queue. For example, if an object i has a relative priority of 0.2, then 20% of the objects in queue have lower priority values than i and their positions are closer to the tail.

The relative priority of an object is explicitly changed when *increase* is called on it. The relative priority of an object is also *implicitly decreased* as other objects are inserted closer to the head of the queue. For instance, if an object j is inserted with a priority of 0.3, then all objects with priorities $\leqslant 0.3$ will be pushed towards the tail and their priority value implicitly decreased.

Many algorithms, including the SLRU family, can be easily implemented with the relative priority queue interface. Others, including the GDSF family, require an absolute priority interface. To support these algorithms RIPQ translates from absolutes priorities to relative priorities, as we explain in Section 4.3.

[3]Write amplification is low for tiny sequential writes, but they attain lower throughput as they are bound by IOPS instead of bandwidth.

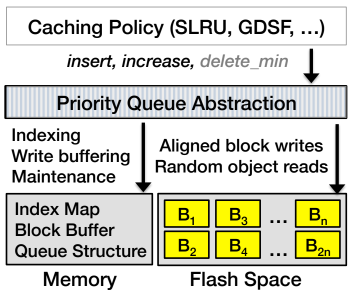


Figure 3: Advanced caching policies with RIPQ.

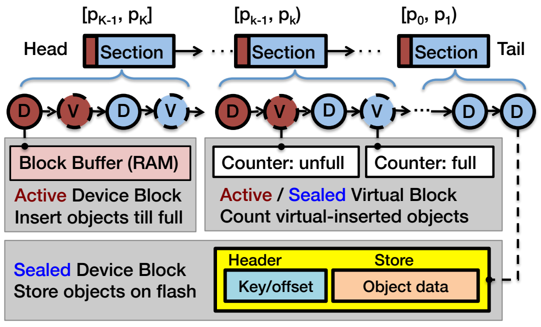


Figure 4: Overall structure of RIPQ.

4.2 Overall Design

RIPQ is a framework that converts priority queue operations into a flash-friendly workload with large writes. Figure 4 gives a detailed illustration of the RIPQ components highlighted in Figure 3, excluding the *Index Map*.

Index Map The *Index Map* is an in-memory hash table which associates all objects' keys with their metadata, including their locations in RAM or flash, sizes, and block IDs. The block structure is explained next.

In our system each entry is ~20 bytes, and RIPQ adds 2 bytes to store the *virtual block ID* of an object. Considering the capacity of the flash card and the average object size, there are about 50 million objects in one caching machine and the index is ~1GiB in total.

Queue Structure The major *Queue Structure* of RIPQ is composed of K sections that are in turn composed of blocks. *Sections* define the insertion points into the queue and a *block* is the unit of data written to flash. The relative priority value range is split into the K intervals corresponding to the sections: $[1, p_{K-1}], \ldots, (p_k, p_{k-1}], \ldots, (p_1, 0]$.[4] When an object is inserted into the queue with priority p, it is placed in the head of the section whose range contains p. For example, in a queue with sections corresponding to $[1, 0.7]$, $(0.7, 0.3]$ and $(0.1, 0]$, an object with priority value 0.5 would be inserted to the head of second section. Similar to relative priority queues, when an object is inserted to a queue of N objects, any object in the same or lower sections with priority q is *implicitly demoted* from priority q to $\frac{qN}{N+1}$. Implicit demotion captures the dynamics of many caching algorithms, including SLRU and GDSF: as new objects are inserted to the queue, the priority of an old object gradually decreases and it is eventually evicted from the cache when its priority reaches 0.

RIPQ **approximates** the priority queue abstraction because its design restricts where data can be inserted. The insertion point count, K, represents the key design trade-off in RIPQ between insertion accuracy and memory consumption. Each section has size $O(\frac{1}{K})$, so larger Ks result in smaller sections and thus higher insertion accuracy. However, because each active block is buffered in RAM until it is full and flushed to flash, the memory consumption of RIPQ is proportional to K. In practice we set $K = 20$ and with a 256MiB block size this translates to a moderate memory footprint of 5GiB. At the same time, our experiments show $K = 20$ ensures that RIPQ achieves hit ratios similar to the exact algorithm.

Device and Virtual Blocks As shown in Figure 4, each section includes one active device block, one active virtual block, and an ordered list of sealed device/virtual blocks. An *active device block* accepts insertions of new objects and buffers them in memory, i.e, the *Block Buffer*. When full it is sealed, flushed to flash, and transitions into a *sealed device block*. To avoid duplicating data on flash RIPQ **lazily updates** the location of an object when its priority is increased, and uses *virtual blocks* to track where an object would have been moved. The *active virtual block* at the head of each section accepts *virtually-updated* objects with increased priorities. When the active device block for a section is sealed, RIPQ also transitions the active virtual block into a *sealed virtual block*. Virtual update is an in-memory only operation, which sets the virtual block ID for the object in the *Index Map*, increases the size counter for the target virtual block, and decreases the size counter of the object's original block.

[4]We have inverted the notation of intervals from `[low, high)` to `(high, low]` to make it consistent with the priority order in the figures.

Algorithm	**Interface Used**	**On Miss**	**On Hit**
Segmented-L LRU	Relative Priority Queue	$\texttt{insert}(x, \frac{1}{L})$	$\texttt{increase}(x, \frac{\min(1,(1+\lceil p \cdot L \rceil)}{L}))$
Greedy-Dual-Size-Frequency L	Absolute Priority Queue	$\texttt{insert}(x, \text{Lowest} + \frac{c(x)}{s(x)})$	$\texttt{increase}(x, \text{Lowest} + c(x)\frac{\min(L,n(x))}{s(x)})$

Table 2: SLRU and GDSF with the priority queue interface provided by RIPQ.

All objects associated with a *sealed device block* are stored in a contiguous space on flash. Within each block, a header records all object keys and their offsets in the data following the header. As mentioned earlier, an updated object is marked with its target virtual block ID within the *Index Map*. Upon eviction of a sealed device block, the block header is examined to determine all objects in the block. The objects are looked up in the *Index Map* to see if their virtual block ID is set, i.e., their priority was increased after insertion. If so, RIPQ *reinserts* the objects to the priorities represented by their virtual blocks. The objects move into active device blocks and their corresponding virtual objects are deleted. Because the updated object will not be written until the old object is about to be evicted, RIPQ maintains at most one copy of each object and duplication is avoided. In addition, lazy updates also allow RIPQ to coalesce all the priority updates to an object between its insertion and reinsertion.

Device blocks occupy a large buffer in RAM (active) or a large contiguous space on flash (sealed). In contrast, virtual blocks resides only in memory and are very small. Each virtual block includes only metadata, e.g., its unique ID, the count of objects in it, and the total byte size of those objects.

Naive Design One naive design of a priority queue on flash would be to fix an object's location on flash until it is evicted. This design avoids any writes to flash on priority update but does not align the location of an objects with its priority. As a result the space of evicted objects on flash would be non-contiguous and the FTL would have to coalesce the scattered objects by copying them forward to reuse the space, resulting in significant FTL write amplification. RIPQ avoids this issue by grouping objects of similar priorities into large blocks and performing writes and evictions on the block level, and by using lazy updates to avoid writes on update.

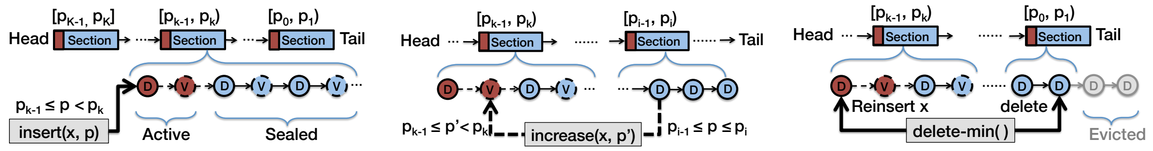
4.3 Implementing Caching Algorithms

To demonstrate the flexibility of RIPQ, we implemented two families of advanced caching algorithms for evaluation: Segmented-LRU [27], and Greedy-Dual-Size-Frequency [16], both of which yield major caching performance improvement for Facebook photo workload. A summary of the implementation is shown in Table 2.

Segmented-LRU Segmented-L LRU (S-L-LRU) maintains L LRU caches of equal size. On a miss, an object is inserted to the head of the L-th LRU cache. On a hit, an object is promoted to the head of the previous LRU cache, i.e., if it is in sub-cache l, it will be promoted to the head of the $\max(l-1, 1)$-th LRU cache. An object evicted from the l-th cache will go to the head of the $(l+1)$-th cache, and objects evicted from the last cache are evicted from the whole cache. This algorithm was demonstrated to provide significant cache hit ratio improvements for the Facebook Edge and Origin caches [22].

Implementing this family of caching algorithms is straightforward with the relative priority queue interface. On a miss, the object is inserted with priority value $\frac{1}{L}$. On a hit, RIPQ finds the previous priority, p, of the accessed object, meaning it is currently in the $\lceil (1-p) \cdot L \rceil$-th queue. It is then promoted to the head of the next queue, with the new priority $\min(1, \frac{1+\lceil (1-p) \cdot L \rceil}{L})$. With the relative priority queue abstraction, the priority of an object is automatically decreased when an object is inserted/updated to a higher priority. When an object is inserted at the head of the l-th LRU cache, all objects in l-th to L-th (the last) caches will be demoted, and the objects at the end of these caches will be either demoted to the next cache or evicted if it is at the end of the last cache—the dynamics of SLRU are exactly captured by relative priority queue interface.

Greedy-Dual-Size-Frequency The Greedy-Dual-Size algorithm [14] provides a principled way to trade-off increased object-wise hit ratio with decreased byte-wise hit ratio by favoring smaller objects. It achieves even higher object-wise hit ratio for Origin cache than SLRU (Section 2), and is favored for that use case as the main purpose of Origin cache is to protect backend storage from excessive IO requests. Greedy-Dual-Size-Frequency [16] (GDSF) improves GDS by taking frequency into consideration. In GDSF, we update the priority of an object x to be $\texttt{Lowest} + c(x) \cdot \frac{n}{s(x)}$ upon its n-th access since it was inserted to the cache, where $c(x)$ is the programmer-defined penalty for a miss on x, `Lowest` is the lowest priority value in the current



(a) Insertion (b) Increase (c) Delete-min

Figure 5: Insertion, and increase, and delete-min operations in RIPQ.

priority queue, and $s(x)$ is the size of the object. We use a variant of GDSF that caps the maximum value of the frequency of an object to L. L is similar to the number of segments in SLRU. It prevents the priority value of a frequently accessed object from blowing up and adapts better to dynamic workloads. The update rule of our variant of GDSF algorithm is thus $p(x) \leftarrow \texttt{Lowest} + c(x) \cdot \frac{\min(L,n)}{s(x)}$. Because we are maximizing object-wise hit ratio we set $c(x) = 1$ for all objects. GDSF uses the absolute priority queue interface.

Limitations RIPQ also supports many other advanced caching algorithms like LFU, LRFU [29], LRU-k [40], LIRS [25], SIZE [6], but there are a few notable exceptions that are not implementable with a single RIPQ, e.g., MQ [48] and ARC [35]. These algorithms involve multiple queues and thus cannot be implemented with one RIPQ. Extending our design to support them with multiple RIPQs coexisting on the same hardware is one of our future directions. A harder limitation comes from the update interface, which only allows increasing priority values. Algorithms that decrease the priority of an object on its access, such as MRU [17], cannot be implemented with RIPQ. MRU was designed to cope with scans over large data sets and does not apply to our use case.

RIPQ does not support delete/overwrite operation because such operations are not needed for static content such as photos. But, they are necessary for a general-purpose read-write cache and adding support for them is also one of our future directions.

4.4 Implementation of Basic Operations

RIPQ implements the three operations of a regular priority queue with the data structures described above.

Insert(x, p) RIPQ inserts the object to the active device block of section k that contains p, i.e., $p_k > p \geqslant p_{k-1}$.[5] The write will be buffered until that active block is sealed. Figure 5a shows an insertion.

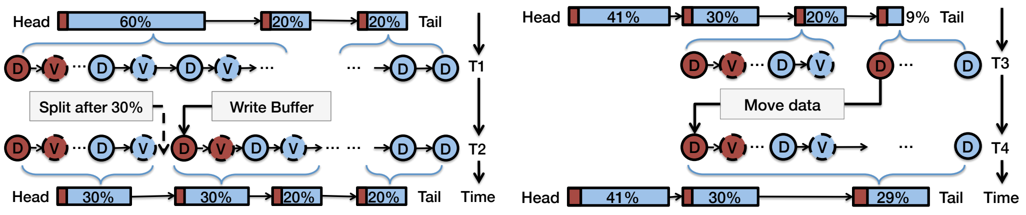
Increase(x, p) RIPQ avoids moving object x that is already resident in a device block in the queue. Instead, RIPQ virtually inserts x into the active virtual block of section k that contains p, i.e., $p_k > p \geqslant p_{k-1}$, and logically removes it from its current location. Because we remember the virtual block ID in the object entry in the indexing hash table, these steps are simply implemented by setting/resetting the virtual block ID of the object entry, and updating the size counters of the blocks and sections accordingly. No read/write to flash is performed during this operation. Figure 5b shows an update.

Delete-min() We maintain a few reserved blocks on flash for flushing the RAM buffers of device blocks when they are sealed.[6] When the number of reserved blocks falls below this threshold, the `Delete-min()` operation is called implicitly to free up the space on flash. As shown in Figure 5c, the lowest-priority block in queue is evicted from queue during the operation. However, because some of the objects in that blocks might have been updated to higher places in the queue, they need to be reinserted to maintain their correct priorities. The reinsertion (1) reads out all the keys of the objects in that block from the block header, (2) queries the index structure to find whether an object, x, has a virtual location, and if it has one, (3) finds the corresponding section, k, of that virtual block and copies the data to the active device block of that section in RAM, and (4) finally sets the virtual block field in the index entry to be empty. We call this whole process *materialization* of the virtual update.

These reinsertions help preserve caching algorithm fidelity, but cause additional writes to flash. These additional writes cause *implementation write amplification*, which is the byte-wise ratio of host-issued

[5]A minor modification when $k = K$ is $1 = p_K \geqslant p \geqslant p_{K-1}$.

[6]It is not a critical parameter and we used 10 in our evaluation.



(a) Section split process **(b) Section merge process**

Figure 6: RIPQ internal operations.

writes to those required to inserted cache misses. RIPQ can explicitly trade lower caching algorithm fidelity for lower write amplification by skipping materialization of the virtual objects whose priority is smaller than a given threshold, e.g., in the last 5% of the queue. This threshold is the *logical occupancy parameter* θ ($0 < \theta < 1$).

Internal operations RIPQ must have neither too many nor too few insertion points: too few leads to low accuracy, and too many leads to high memory usage. To avoid these situations RIPQ splits a section when it grows too large and merges consecutive sections when their total size is too small. This is similar to how B-tree [12] splits/merges nodes to control the size of the nodes and the depth of the tree.

A parameter α controls the number of sections of RIPQ in a principled way. α is in $(0, 1)$ and determines the average size of sections. RIPQ splits a section when its relative size—i.e., a ratio based on the object count or byte size—has reached 2α. For example, if $\alpha = 0.3$ then a section of $[0.4, 1.0]$ would be split to two sections of $[0.4, 0.7)$ and $[0.7, 1.0]$ respectively, shown in Figure 6a. RIPQ merges two consecutive sections if the sum of their sizes is smaller than α, shown in Figure 6b. These operations ensure there are at most $\lceil \frac{2}{\alpha} \rceil$ sections, and that each section is no larger than 2α.

No data is moved on flash for a split or merge. Splitting a section creates a new active device block with a write buffer and a new active virtual block. Merging two sections combines their two active device blocks: the write buffer of one is copied into the write buffer of the other. Splitting happens often and is how new sections are added to queue as objects in the section at the tail are evicted block-by-block. Merging is rare because it requires the total size of two consecutive sections to shrink from 2α (α is the size of a new section after a split) to α to trigger a merge. The amortized complexity of a merge per operation provided by the priority queue API is only $O(\frac{1}{\alpha M})$, where M is the number of blocks.

Supporting Absolute Priorities Caching algorithms such as LFU, SIZE [6], and Greedy-Dual-Size[14] require the use of absolute priority values when performing insertion and update. RIPQ supports absolute priorities with a mapping data structure that translates them to relative priorities. The data structure maintains a dynamic histogram that supports insertion/deletion of absolute priority values, and when given an absolute priorities return approximate quantiles, which are used as the internal relative priority values.

The histogram consists of a set of bins, and we merge/split bins dynamically based on their relative sizes, similar to the way we merge/split sections in RIPQ. We can afford to use more bins than sections for this dynamic histogram and achieve higher accuracy of the translation, e.g., $\kappa = 100$ bins while RIPQ only uses $K = 20$ sections, because the bins only contains absolute priority values and do not require a large dedicated RAM buffer as the sections do. Consistent sampling of keys to insert priority values to the histogram can be further applied to reduce its memory consumption and insertion/update cost.

4.5 Other Design Considerations

Parameters Table 3 describes the parameters of RIPQ and the value chosen for our implementation. The *block size* B is chosen to surpass the threshold for a sustained high write throughput for random writes, and the *number of blocks* M is calculated directly based on cache capacity. The number of blocks affects the memory consumption of RIPQ, but this is dominated by the size of the write buffers for active blocks and the indexing structure. The number of active blocks equals the number of *insertion points* K in the queue. The *average section size* α is used by the split and merge operations to bound the memory consumption and approximation error of RIPQ.

Durability Durability is not a requirement for our static-content caching use case, but not having to

Parameter	Symbol	Our Value	Description and Goal
Block Size	B	256MiB	To satisfy the sustained high random write throughput.
Number of Blocks	M	2400	Flash caching capacity divided by the block size.
Average Section Size	α	0.05	To bound the number of sections $\leqslant \lceil 2/\alpha \rceil$ and the size of each section $\leqslant 2\alpha$, trade-off parameter for insertion accuracy and RAM buffer usage.
Insertion Points	K	20	Same as the number of sections, controlled by α and proportional to RAM buffer usage.
Logical Occupancy	θ	0	Avoid reinsertion of items that will soon be permanently evicted.

Table 3: Key parameters of RIPQ for a 670GiB flash drive currently deployed in Facebook.

refill the entire cache after a power loss is a plus. Fortunately, because the keys and locations of the objects are stored in the headers of the on-flash device blocks, all objects that have been saved to flash can be recovered, except for those in the RAM buffers. The ordering of blocks/sections can be periodically flushed to flash as well and then used to recover the priorities of the objects.

4.6 Theoretical Analysis

RIPQ is a practical approximate priority queue for implementing caching algorithms on flash, but enjoys some good theoretical properties as well. In an appendix [44] we omit due to space constraints we show RIPQ can simulate a LRU cache *faithfully* with 4α of additional space: if $\alpha = 0.05$, this would mean RIPQ-based LRU with 20% additional space would include all the objects in an exact LRU cache. In general RIPQ with adjusted insertion points can simulate a S-L- LRU cache with $4L\alpha$ of additional space. We also show the number of writes to the flash is $\leqslant I + U$, where I is the number of inserts and U is the number of updates.

Using K sections/insertion points, the complexity of finding the approximate insertion/update point takes $O(K)$, and the amortized complexity of split/merge internal operations is $O(1)$, so the amortized complexity of RIPQ is only $O(K)$. If we arrange the sections in a red-black tree, it can be further reduced to $O(\log K)$. In comparison to this, with N objects, an exact implementation of priority queues using red-black tree would take $O(\log N)$ per operation, and a Fibonacci heap takes $O(\log N)$ per delete-min operation. ($K \ll N$, K is typically 20, N is typically 50 million). The computational complexity of these exact, tree and heap based data structures are not ideal for a high performance system. In contrast, RIPQ hits the sweet spot with fast operations and high fidelity, in terms of both theoretical analysis and empirical hit ratios.

5 SIPQ

RIPQ's buffering for large writes creates a moderate memory footprint, e.g., 5 GiB DRAM for 20 insertion points with 256 MiB block size in our implementation. This is not an issue for servers at Facebook, which are equipped with 144 GiB of RAM, but limits the use of RIPQ in memory-constrained environments. To cope with this issue, we propose the simpler Single Insertion Priority Queue (SIPQ) framework.

SIPQ uses flash as a cyclic queue and only sequentially writes to the device for high write throughput with minimal buffering. When the cache is full, SIPQ reclaims device space following the same sequential order. In contrast to RIPQ, SIPQ maintains an exact priority queue of the *keys* of the cached objects in memory and does not co-locate similarly prioritized objects physically due to the single insertion limit on flash. The drawback of this approach is that reclaiming device space may incur many reinsertions for SIPQ in order to preserve its priority accuracy. Similar to RIPQ, these reinsertions constitute the implementation write amplification of SIPQ.

To reduce the implementation write amplification, SIPQ only includes the keys of a portion of all the cached objects in the in-memory priority queue, referred to as the *virtual cache*, and will only reinsert evicted objects that are in this cache. All on-flash capacity is referred to as the *physical cache* and the ratio between the total byte size of objects in the virtual cache to the size of the physical cache is controlled by a *logical occupancy parameter* θ ($0 < \theta < 1$). Because only objects in the virtual cache are reinserted when they are about to be evicted from the physical cache, θ provides a trade-off between priority fidelity and implementation write amplification: the larger θ, the more objects are in the virtual cache and the higher fidelity SIPQ has relative to the exact caching algorithm, and on the other hand the more likely evicted objects will need to be reinserted and thus higher write amplification caused by SIPQ. For $\theta = 1$, SIPQ implements an exact priority queue for all cached data on flash, but incurs high write amplification for reinsertions. For $\theta = 0$, SIPQ deteriorates to FIFO with no priority enforcement. For θ in between, SIPQ performs additional writes compared to FIFO but also delivers part of the improvement of more advanced caching algorithms. In our evaluation, we find that SIPQ provides a good trade-off point for Segmented-LRU algorithms with $\theta = 0.5$, but does not perform well for more complex algorithms like GDSF. Therefore, with limited improvement at almost no additional device overhead, SIPQ can serve as a simple upgrade for FIFO when memory is tight.

6 Evaluation

We compare RIPQ, SIPQ, and Facebook's current solution, FIFO, to answer three key questions:

1. What is the impact of RIPQ and SIPQ's approximations of caching algorithms on hit ratios, i.e., what is the effect on algorithm fidelity?
2. What is the write amplification caused by RIPQ and SIPQ versus FIFO?
3. What throughput can RIPQ and SIPQ achieve?
4. How does the hit-ratio of RIPQ change as we vary the number of insertion points?

6.1 Experimental Setup

Implementation We implemented RIPQ and SIPQ with 1600 and 600 lines of C++ code, respectively, using the Intel TBB library [5] for the object index and the C++11 thread library [1] for the concurrency mechanisms. Both the relative and absolute priority interfaces (enabled by an adaptive histogram translation) are supported in our prototypes.

Hardware Environment Experiments are run on servers equipped with a Model A 670GiB flash device and 144GiB DRAM space. All flash devices are configured with 90% space utilization, leaving the remaining 10% for the FTL.

Framework Parameters RIPQ uses a 256MiB block size to achieve high write throughput based on our performance study of Model A flash in Section 3. It uses $\alpha = 0.05$, i.e., 20 sections, to provide a good trade-off between the fidelity to the implemented algorithms and the total DRAM space RIPQ uses for buffering: 256MiB $\times$ 20 = 5GiB, which is moderate for a typical server.

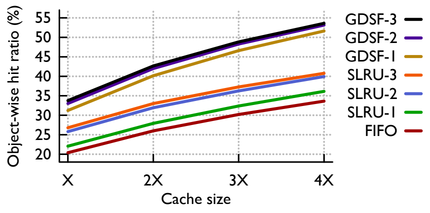
SIPQ also uses the 256MiB block size to keep the number of blocks on flash the same as RIPQ. Because SIPQ only issues sequential writes, its buffering size could be further shrunk without adverse effects. Two *logical occupancy* values for SIPQ are used in evaluation: 0.5, and 0.9, each representing a different trade-off between the approximation fidelity to the exact algorithm and implementation write amplification. Later, these two settings are noted as SIPQ-0.5 and SIPQ-0.9, respectively.

Caching Algorithms Two families of advanced caching algorithms, Segmented-LRU (SLRU) [27] and Greedy-Dual-Size-Frequency (GDSF) [16], are evaluated on RIPQ and SIPQ. For Segmented-LRU, we vary the number of segments from 1 to 3, and report their results as SLRU-1, SLRU-2, and SLRU-3, respectively. We similarly set L from 1 to 3 for Greedy-Dual-Size-Frequency, denoted as GDSF-1, GDSF-2, and GDSF-3. Description of these algorithms and their implementations on top of the priority queue interface are explained in Section 4.3. Results of 4 segments or more for SLRU and $L > 4$ for GDSF are not included due to their marginal differences in the caching performance.

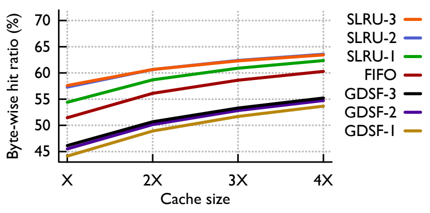
Facebook Photo Trace Two sets of 15-day sampled traces collected within the Facebook photo-serving stack are used for evaluation, one from the Origin cache, and the other from a large Edge cache facility. The Origin trace contains over 4 billion requests and 100TB worth of data, and the Edge trace contains over 600 million requests and 26TB worth of data. To emulate the effect of different total cache capacities in Origin/Edge with the same space utilization of the experiment device and thus controlling for the effect of FTL, both traces are further down sampled through hashing: we randomly sample $\frac{1}{2}$, $\frac{1}{3}$, and $\frac{1}{4}$ of the cache key space of the original trace for each experiment to emulate the effect of increasing the total caching capacity to $2X$, $3X$, and $4X$. We report experimental results at $2X$ because it closely matches our production configurations. For all evaluation runs, we use the first 10-day trace to warm up the cache and measure performance during the next 5 days. Because both the working set and the cache size are very large, it takes hours to fill up the cache and days for the hit ratio to stabilize.

6.2 Experimental Results

This section presents our experimental results regarding the algorithm fidelity, write amplification, and throughput of RIPQ and SIPQ with the Facebook photo trace. We also include the hit ratio, write amplification and throughput achieved by Facebook's existing FIFO solution as a baseline. For different cache sites, only their target hit ratio metrics are reported, i.e., object-wise hit ratio for the Origin trace and byte-wise hit ratio for the Edge trace. Exact algorithm hit ratios are obtained via



(a) Object-wise hit ratios on Origin trace.



(b) Byte-wise hit ratios on Edge trace.

Figure 7: Exact algorithm hit ratios on Facebook trace.

simulations as the baseline to judge the approximation fidelity of implementations on top of RIPQ and SIPQ.

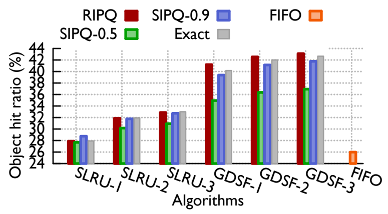
Performance of Exact Algorithms We first investigate hit ratios achieved by the exact caching algorithms to determine the gains of a fully accurate implementation. Results are shown in Figure 7.

For object-wise hit ratio on the Origin trace, Figure 7a shows that GDSF family outperforms SLRU and FIFO by a large margin. At $2X$ cache size, GDSF-3 increases the hit ratio over FIFO by 17%, which translates a to a 23% reduction of backend IOPS. For byte-wise hit ratio on the Edge trace, Figure 7b shows that SLRU is the best option: at $2X$ cache size, SLRU-3 improves the hit ratio over FIFO by 4.5%, which results in a bandwidth reduction between Edge and Origin by 10%. GDSF performs poorly on the byte-wise metric because it down weights large photos. Because different algorithms perform best at different sites with different performance metrics, flexible frameworks such as RIPQ make it easy to optimize caching policies with minimal engineering effort.

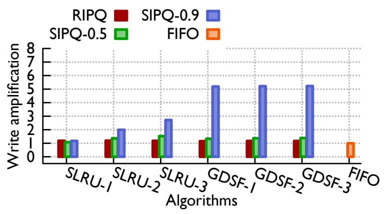
Approximation Fidelity Exact algorithms yield considerable gains in our simulation, but are also challenging to implement on flash. RIPQ and SIPQ make it simple to implement the algorithms on flash, but do so by approximating the algorithms. To quantify the effects of this approximation we ran experiments presented in Figures 8a and 8d. These figures present the hit ratios of different exact algorithms (in simulations) and their approximate implementations on flash with RIPQ, SIPQ-0.5, and SIPQ-0.9 (in experiments) at $2X$ cache size setup from Figure 7. The implementation of FIFO is the same as the exact algorithm, so we only report one number. In general, if the hit ratio of an implementation is similar to the exact algorithm the framework provides high fidelity.

RIPQ consistently achieves high approximation fidelities for the SLRU family, and its hit ratios are less than 0.2% different for object-wise/byte-wise metric compared to the exact algorithm results on Origin/Edge trace. For the GDSF family, RIPQ's algorithm fidelity becomes lower as the algorithm complexity increases. The greatest "infidelity" seen for RIPQ is a 5% difference on the Edge trace for GDSF-1. Interestingly, for the GDSF family, the infidelity generated by RIPQ improves byte-wise hit ratio—the largest infidelity was a 5% improvement on byte-wise hit-ratio compared to the exact algorithm. The large gain on byte-wise hit ratio can be explained by the fact that the exact GDSF algorithm is designed to trade byte-wise hit ratio for object-wise hit ratio through favoring small objects, and its RIPQ approximation shifts this trade-off back towards a better byte-wise hit-ratio. Not shown in the figures (due to space limitation) is that RIPQ-based GDSF family incurs about 1% reduction in *object-wise* hit ratio. Overall, RIPQ achieves high algorithm fidelity on both families of caching algorithms that perform the best in our evaluation.

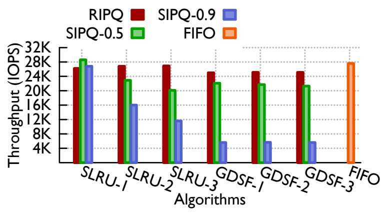
SIPQ also has high fidelity when the occupancy parameter is set to 0.9, which means 90% of the caching capacity is managed by the exact algorithm. SIPQ-0.5, despite only half of the cache capacity being managed by the exact algorithm, still achieves a relatively high fidelity for SLRU algorithms: it creates a 0.24%-2.8% object-wise hit ratio reduction on Origin, and 0.3%-0.9% byte-wise hit ratio reduction on Edge. These algorithms tend to put new and recently accessed objects towards the head of the queue, which is similar to the way SIPQ inserts and reinserts objects at the head of the cyclic queue on flash. However, SIPQ-0.5 provides low fidelity for the GDSF family, causing object-wise hit ratio to decrease on Origin and byte-wise hit ratio to increase on Edge. Within these algorithms, objects may have diverse priority values due to their size differences even if they enter the cache at the same time, and SIPQ's single insertion point design results in a poor approximation.



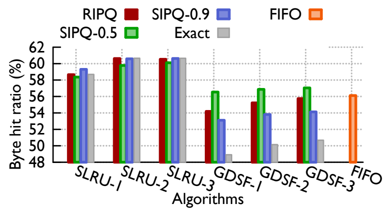
(a) Object-wise hit ratio (Origin)



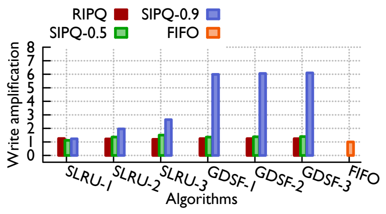
(b) Write amplification (Origin)



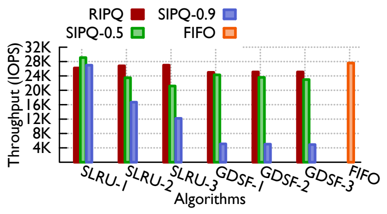
(c) IOPS throughput (Origin)



(d) Byte-wise hit ratio (Edge)



(e) Write amplification (Edge)



(f) IOPS throughput (Edge)

Figure 8: Performance of RIPQ, SIPQ, and FIFO on Origin and Edge.

Write Amplification Figure 8b and 8e further show the combined write amplification (i.e., $FTL \times implementation$) of different frameworks. RIPQ consistently achieves the lowest write amplification, with an exception for SLRU-1 where SIPQ-0.5 has the lowest value for both traces. This is because SLRU-1 (LRU) only inserts to one location at the queue head, which works well with SIPQ, and the logical occupancy 0.5 further reduces the reinsertion overhead. Overall, the write amplification of RIPQ is largely stable regardless of the complexity of the caching algorithms, ranging from 1.18 to 1.25 for the SLRU family, and from 1.15 to 1.25 for the GDSF family.

SIPQ-0.5 achieves moderately low write amplifications but with lower fidelity for complex algorithms. Its write amplification also increases with the algorithm complexity. For SLRU, the write implementation for SIPQ-0.5 rises from 1.08 for SLRU-1 to 1.52 for SLRU-3 on Origin, and from 1.11 to 1.50 on Edge. For GDSF, the value ranges from 1.33 for GDSF-1 to 1.37 to GDSF-3 on Origin, and from 1.36 to 1.39 on Edge. Results for SIPQ-0.9 observe a similar trend for each family of algorithms, but with a much higher write amplification value for GDSF around 5-6.

Cache Throughput Throughput results are shown in Figure 8c and 8f. RIPQ and SIPQ-0.5 consistently achieve over 20 000 requests per second (rps) on both traces, but SIPQ-0.9 has considerably lower throughput, especially for the GDSF family of algorithms. FIFO has slightly higher throughput than RIPQ based SLRU, although the latter has higher byte hit ratio and correspondingly fewer writes from misses.

This performance is highly related to the write amplification results because in all three frameworks (1) workloads are write-heavy with below 63% hit ratios, and our experiments are mainly write-bounded with a sustained write-throughput around 530 MiB/sec, (2) write amplification proportionally consumes the write throughput, which further throttles the overall throughput. This is why SIPQ-0.9 often with the highest write amplification has the lowest throughput, and also why RIPQ based SLRU has lower throughput than FIFO. However, RIPQ/SIPQ-0.5 still provides high performance for our use case, with RIPQ paticularly achieving over 24 000 rps on both traces. The less than 3 000 rps lower throughput comparing to FIFO is well worth the hit-ratio improvement that results in backend IOPS reduction and bandwidth reduction between Edge and Origin.

Sensitivity Analysis on Number of Insertion Points Figure 9 shows the effect of varying the number of insertion points in RIPQ on approximation accuracy. The number of insertion points, K, is roughly inversely proportional to α, so we vary K to be approximately $2, 4, 8, 16$, and 32, by varying α from $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$ to $\frac{1}{32}$. We measure approximation accuracy empirically through the object-wise hit-ratios of RIPQ based SLRU-3 and GDSF-3 on the origin trace with 2X cache size.

When $K \approx 2$ ($\alpha = \frac{1}{2}$), a section in RIPQ can

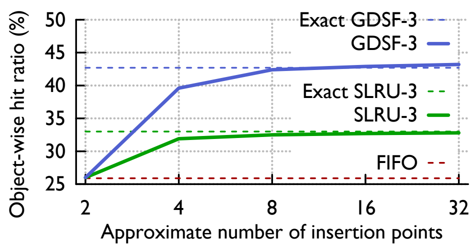


Figure 9: Object-wise hit ratios sensitivity on approximate number of insertion points.

grow to the size of the entire queue before it splits. In this case RIPQ effectively degenerates to FIFO with equivalent hit-ratios. SLRU-3 hit ratio saturates quickly when $K \gtrsim 4$, while GDSF-3 reaches its highest performance only when $K \gtrsim 8$. GDSF-3 uses many more insertion points in an exact priority queue than SLRU-3 and RIPQ thus need more insertion points to effectively colocate content with similar priorities.

7 Related Work

To the best of our knowledge, there is no existing work that provides a flexible framework for efficiently implementing advanced caching algorithms on flash. However, related work can be found in several heavily-researched fields: Flash-based Caching, RAM-based Advanced Caching, Flash-based Storage, Flash Performance Studies, and Priority Queues.

Flash-based Caching Solutions Flash devices have been applied in various caching solutions for their large capacities and high I/O performance [2, 7, 9, 23, 24, 28, 32, 36, 39, 42, 46]. To avoid their poor handling of small random write workloads, previous studies either use sequential eviction akin to FIFO [7], or only perform coarse-grained caching policies at the unit of large blocks [23, 32, 46]. Similarly, SIPQ and RIPQ also achieve high write throughputs and low device overheads on flash through sequential writes and large aligned writes, respectively. In addition, they allow efficient implementations of advanced caching policies at a fine-grained object unit, and our experience show that photo caches built on top of RIPQ and SIPQ yield significant performance gains at Facebook. While our work mainly focuses on the support of *eviction* part of caching operations, techniques like *selective insertions* on misses [23, 46] are orthogonal to RIPQ and can be applied to further reduce the data written to flash.[7]

RAM-based Advanced Caching Caching has been an important research topic since the early days of computer science and many algorithms have been proposed to better capture the characteristics of different workloads. Some well-known features include recency (LRU, MRU [17]), frequency (LFU [34]), inter-reference time (LIRS [25]), and size (SIZE [6]). There have also been a plethora of more advanced algorithms that consider multiple features, such as Multi-Queue [48] and Segmented LRU (SLRU) [27] for both recency and frequency, Greedy-Dual [47] and its variants like Greedy-Dual-Size [14] and Greedy-Dual-Size-Frequency [16] (GDSF) using a more general method to compose the expected miss penalty and minimize it.

While more advanced algorithms can potentially yield significant performance improvements, such as SLRU and GDSF for Facebook photo workload, a gap still remains for efficient implementations on top of flash devices because most algorithms are hardware-agnostic: they implicitly assume data can be moved and overwritten with little overhead. Such assumptions do not hold on flash due to its asymmetric performance for reads and writes and the performance deterioration caused by its internal garbage collection.

Our work, RIPQ and SIPQ, bridges this gap. They provide a priority queue interface to allow easy implementation of many advanced caching algorithms, providing similar caching performance while generating flash-friendly workloads.

Flash-based Store Many flash-based storage systems, especially key-value stores have been recently proposed to work efficiently on flash hardware. Systems such as FAWN-KV [11], SILT [33], LevelDB [3], and RocksDB [4] group write operations from an upper layer and only flush to the device using sequential writes. However, they are designed for read-heavy workloads and other performance/application metrics such as memory footprints and range-query efficiencies. As a result, these systems make trade-offs such as conducting on-flash data sorting and merges, that yield high device overhead for write-heavy workloads. We have experimented with using RocksDB as an on-flash photo store for our application, but found it to have excessively high write amplification (~5 even when we allocated 50% of the flash space to garbage collection). In contrast, RIPQ and SIPQ are specifically optimized for

[7]We tried such techniques on our traces, but found the hit ratio dropped because of the long-tail accesses for social network photos.

a (random) write-heavy workload and only support caching-required interfaces, and as a result have low write amplification.

Study on Flash Performance and Interface While flash hardware itself is also an important topic, works that study the application perceived performance and interface are more related to our work. For instance, previous research [13, 26, 37, 43] that reports the random write performance deterioration on flash helps verify our observations in the flash performance study.

Systematic approaches to mitigate this specific problem have also been previously proposed at different levels, such as separating the treatment of cold and hot data in the FTL by LAST [30], and the similar technique in filesystem by SFS [37]. These approaches work well for skewed write workloads where only a small subset of the data is hot and updated often, and thus can be grouped together for garbage collection with lower overhead. In RIPQ, cached contents are explicitly tagged with priority values that indicate their hotness, and are co-located within the same device block if their priority values are close. In a sense, such priorities provide a *prior* for identifying content hotness.

While RIPQ (and SIPQ) runs on unmodified commercial flash hardware, recent studies [32, 41] which co-design flash software/hardware could further benefit RIPQ by reducing its memory consumption.

Priority Queue Both RIPQ and SIPQ rely on the priority queue abstract data type and the design of priority queues with different performance characteristics have been a classic topic in theoretical computer science as well [15, 18, 20]. Instead of building an exact priority queue, RIPQ uses an approximation to trade algorithm fidelity for flash-aware optimization.

8 Conclusion

Flash memory, with its large capacity, high IOPS, and complex performance characteristics, poses new opportunities and challenges for caching. In this paper we present two frameworks, RIPQ and SIPQ, that implement approximate priority queues efficiently on flash. On top of them, advanced caching algorithms can be easily, flexibly, and efficiently implemented, as we demonstrate for the use case of a flash-based photo cache at Facebook. RIPQ achieves high fidelity and low write amplification for both tested SLRU and GDSF algorithms. SIPQ is a simpler design, requires less memory and still achieves good results for simple algorithms like LRU. Experiments on both the Facebook Edge and Origin traces show that RIPQ can improve hit ratios by up to ~20% over the current FIFO system, reducing bandwidth consumption between the Edge and Origin, and reducing I/O operations to backend storage.

References

[1] C++11 Thread Support Library. http://en.cppreference.com/w/cpp/thread, 2014.

[2] Flashcache at Facebook: From 2010 to 2013 and beyond. http://tinyurl.com/oljloxb, 2014.

[3] LevelDB, A fast and lightweight key/value database library by Google. https://code.google.com/p/leveldb, 2014.

[4] RocksDB, A persistent key-value store for fast storage environments. http://rocksdb.org, 2014.

[5] Intel Thread Building Blocks. https://www.threadingbuildingblocks.org, 2014.

[6] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. Williams. Removal policies in network caches for World-Wide Web documents. In *ACM SIGCOMM Computer Communication Review*, 1996.

[7] A. Aghayev and P. Desnoyers. Log-structured cache: trading hit-rate for storage performance (and winning) in mobile devices. In *Proc. Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.

[8] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. USENIX Annual Technical Conference (ATC)*, 2008.

[9] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and E. Schrock. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *Proc. USENIX Annual Technical Conference (ATC)*, 2013.

[10] D. G. Andersen and S. Swanson. Rethinking flash in the data center. *IEEE micro*, 2010.

[11] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[12] R. Bayer and E. McCreight. *Organization and maintenance of large ordered indexes.* Springer, 2002.

[13] L. Bouganim, B. r Jnsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2009.

[14] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proc. USENIX Symposium on Internet Technologies and Systems (USITS)*, 1997.

[15] B. Chazelle. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM (JACM)*, 2000.

[16] L. Cherkasova and G. Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *High-Performance Computing and Networking*, 2001.

[17] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1986.

[18] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 1987.

[19] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[20] J. E. Hopcroft. Data structures and algorithms. *AddisonWeely*, 1983.

[21] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proc. International Systems and Storage Conference (SYSTOR)*, 2009.

[22] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An Analysis of Facebook Photo Caching. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[23] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with Lazy Adaptive Replacement. In *Proc. IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2013.

[24] D. Jiang, Y. Che, J. Xiong, and X. Ma. uCache: A Utility-Aware Multilevel SSD Cache Management Policy. In *Proc. IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC)*, 2013.

[25] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Performance Evaluation Review*, 2002.

[26] K. Kant. Data center evolution: A tutorial on state of the art, issues, and challenges. *Computer Networks*, 2009.

[27] R. Karedla, J. S. Love, and B. G. Wherry.

Caching strategies to improve disk system performance. *IEEE Computer*, 1994.

[28] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2008.

[29] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.*, 2001.

[30] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 2008.

[31] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.

[32] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *Proc. USENIX Annual Technical Conference (ATC)*, 2014.

[33] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. ACM Symposium on Operating Systems Principles*, 2011.

[34] S. Maffeis. Cache management algorithms for flexible filesystems. *ACM SIGMETRICS Performance Evaluation Review*, 1993.

[35] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[36] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu. vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In *Proc. USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2014.

[37] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[38] Netflix. Netflix Open Connect. https://www.netflix.com/openconnect, 2014.

[39] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Improving performance and lifetime of the SSD RAID-based host cache through a log-structured approach. In *Proc. Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.

[40] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1993.

[41] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: software-defined flash for web-scale internet storage systems. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[42] M. Saxena, M. M. Swift, and Y. Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proc. ACM European Conference on Computer Systems (EuroSys)*, 2012.

[43] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *Proc. International Workshop on Data Management on New Hardware*, 2009.

[44] L. Tang. RIPQ Appendix: Theoretical Analysis of RIPQ. http://www.cs.princeton.edu/~linpengt/publications/ripq-appendix.pdf, 2014.

[45] R. P. Wooster and M. Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 1997.

[46] J. Yang, N. Plasson, G. Gillis, and N. Talagala. Hec: improving endurance of high performance flash-based cache devices. In *Proc. International Systems and Storage Conference (SYSTOR)*, 2013.

[47] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 1994.

[48] Y. Zhou, J. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX Annual Technical Conference (ATC)*, 2001.

A Theoretical Analysis of RIPQ

Overview and Notations

For simplicity we assume the objects are of unit sizes and we want to approximate an exact priority queue cache of size n_c with K fixed insertion points. Note that Segmented LRU-K lies in this class.

Our goal is to construct a RIPQ cache simulating the exact priority queue cache so at any time the objects in the exact priority queue cache will always be a subset of the objects in the approximate RIPQ cache. Of course we have to pay some price with the approximation, and this is reflected in the additional space used by the RIPQ cache. We will bound the additional space needed by RIPQ and show that it is proportional to the average section size and the number of insertion points K.

Object Location Use X to denote the set of all the objects in the workload, then we define

$$\mathcal{L}^{(t)}(x) : X \longmapsto (-\infty, n_c]$$

to be the location of object x in the cache at time t. The object at the queue head will have location n_c, the next object $n_c - 1$, etc. The objects can have non-positive locations if we use more space than the exact cache.

Imagine we run an exact priority queue cache (abbreviated to exact cache from now on) side by side, then each object would have a location in the exact cache as well, we call that its **ideal object location**, denoted as $\mathcal{L}^{*(t)}(x)$.

Block Location RIPQ consists of a ordered set of physical/virtual blocks, partitioned into sections, with a active device block and a active virtual block at the head of each section. We use $b_1, b_2, \cdots, b_m$ to denote the chain of blocks in the queue, with b_1 the block at the queue tail and b_m the block at the queue head. We overload the notation $\mathcal{L}$ and define the location of block b to be the maximal location of objects in that block:

$$\mathcal{L}(b) \triangleq \max_{x \in b} \mathcal{L}(x)$$

We call b_j above b_i and write $b_j > b_i$ if $j > i$, and correspondingly for b_i below b_j.

Approximation Space Overhead As mentioned before our goal is to keep all the objects in the exact cache in our approximate cache as well, and we are allowed to use some additional space. Now the lowest location of the objects in the exact cache is $\min_{x:\mathcal{L}^{*(t)}(x) \geqslant 1} \mathcal{L}^{(t)}(x)$, and if we are going to keep this object in the approximate cache, the additional space we use is

$$\max_{x:\mathcal{L}^{*(t)}(x) \geqslant 1} -\mathcal{L}^{(t)}(x) + 1$$

We will prove it grows linearly with the expected section size α and the number of insertion points K.

Block Max Ideal Location is the maximal ideal location of objects in that block, i.e.

$$\gamma^{(t)}(b) \triangleq \max_{x:x \in^{(t)} b} \mathcal{L}^{*(t)}(x)$$

where $x \in^{(t)} b$ means x is in block b at time t.

Insertion Points of the Exact Cache We use $0 < q_1 < \cdots < q_K = n_c$ to denote K insertion points of the exact cache. If we are running Segmented-K LRU then $q_k = \lceil k/K \times n_c \rceil$. Note that we can only insert or update an object to these locations in the queue. Inserting x to q_k causes the locations of objects *lower than or equal to* q_k to decrease by 1, while objects of higher locations retain their current locations.

$$\mathcal{L}^{*(t+1)}(y) = \begin{cases} \mathcal{L}^{*(t)}(y) & \text{if } \mathcal{L}^{*(t)}(y) > q_k \\ q_k & \text{if } y = x \\ \mathcal{L}^{*(t)}(y) - 1 & \text{if } \mathcal{L}^{*(t)}(y) \leqslant q_k \end{cases}$$

If we update an object from L_1 to L_2 ($L_1 < L_2$) and $L_2 = q_k$, then

$$\mathcal{L}^{*(t+1)}(y) = \begin{cases} \mathcal{L}^{*(t)}(y) - 1 & \text{if } L_1 < \mathcal{L}^{*(t)}(y) \leqslant L_2 \\ q_k & \text{if } y = x \\ \mathcal{L}^{*(t)}(y) & \text{otherwise} \end{cases}$$

Algorithm

Now we describe the variant of the RIPQ algorithm that simulates an exact priority queue cache with provable, bounded space overhead. Define the adjusted insertion point $\mathcal{I}_k = q_k - 4(K-k)\alpha$, find the section containing $\mathcal{I}_k$ and insert/update q_k objects into that section. Or equivalently, find the *lowest* active physical/virtual block above or equal to $\mathcal{I}_k$ and insert/update the object there.

Note that the insertion points are adjusted in RIPQ such that the space between two insertion points q_{k-1}, q_k increases from $q_k - q_{k-1}$ to $q_k - q_{k-1} + 4\alpha$. The additional 4α space is needed because of the approximate insertion error and the error from materialization of virtual insertions, so the location of an object in RIPQ might be lower than its ideal location.

If the insertion points are not adjusted to allow for this error, then we lose any reasonable theoretical guarantees of approximation error: imagine inserting many new objects to q_k, and some object with a *higher ideal location than* q_k will be pushed towards the tail of the queue and eventually evicted, while its ideal location is actually higher than all the objects newly inserted to q_k (there can be $q_k n_c$ such objects in the exact cache), so the approximation space overhead will be at least q_k.

For simplicity we assume the blocks are all unit-size for now, so the block seal and section combine/split operations are all no-ops for RIPQ structure, and we omit the effects of these operations in the statement of the main theorem and the proof. We show the additional overhead brought by a larger block size is actually quite small in the following text.

Main Theorem

For any block b at any time, the maximal block ideal location satisfies

$$\gamma_k(b) \leqslant \mathcal{L}(b) + 4(K - k + 1)\alpha,$$

or the ideal location of any k-insertion object won't be larger than the location of its block plus $4(K - k + 1)\alpha$.

Because $1 \leqslant k \leqslant K$, it follows that $\gamma(b) \leqslant \mathcal{L}(b) + 4K\alpha$. If $\mathcal{L}(b) \leqslant -4K\alpha$, then $\gamma(b) \leqslant 0$, so all the objects in that block will already have been evicted by the exact cache. This means the simulation space overhead of RIPQ $\leqslant 4K\alpha$.

Note that for an exact cache, $\gamma_k(b) = \mathcal{L}(b)$, the additional $4(K-k+1)\alpha$ is from the error of the approximate insertion and materialization operation. The maximum section size is 2α, and both operations incur an approximation error of $O(\alpha)$ per insertion point and it accumulates for each insertion point from queue head to queue tail.

Analysis and Proof

We prove by induction a stronger version of the main theorem. At any time t, for any block b and any object x in b, we are going to show by induction that

$$\mathcal{L}^{*(t)}(x) + c^{(t)}(x) \leqslant \mathcal{L}^{(t)}(b) + 4(K - k)\alpha + \mathcal{M}^{(t)}(b)$$

$c(x)$ and $\mathcal{M}(b)$ is *account variables* associated with x and b respectively, used in amortized analysis. By its definition $c(x)$ is guaranteed to be non-negative and we are going to show $\mathcal{M}(b) \leqslant 2\alpha$ so the main theorem follows.

- $c(x)$: we add 1 to $c(x)$ when we insert y to the queue with a higher ideal location than x but with a lower location than x in RIPQ, and subtract it by 1 whenever y is later updated to a higher location in RIPQ. $c(x)$ by definition is guaranteed to be non-negative.
- $\mathcal{M}(b)$: it records the number of over-b materializations: the materializations "flying over" b since it "crossed" the last insertion point, i.e. say x is materialized and moved from b_i to b_j, then if $b_i \leqslant b < b_j$ we add 1 to $\mathcal{M}(b)$. If later x is updated to a higher location we subtract $\mathcal{M}(b)$ by 1 as well. And if currently $\mathcal{I}_{k-1} < \mathcal{L}(b) \leqslant \mathcal{I}_k$, then we account for these operations *only since* $\mathcal{L}(b) \leqslant \mathcal{I}_k$, and we call it the *$\mathcal{I}_k$-phase* for block b. Because the location of any block is non-increasing over time, its phases are well defined.

Lemma The location of any block is non-increasing over time.

Proof We consider all the different operations on RIPQ.

1. Insertion. If we insert x to block b_i, then for any $b < b_i$, its location decreases by 1, while for any block $b \geqslant b_i$, its location remains the same.
2. Update. Assume object x is moved from b_i to b_j during update. Because we only allow increasing priority, $b_i < b_j$, so for any block b in between, i.e. $b_i \leqslant b < b_j$, its location decreases by 1, while for any other block its location remains the same.
3. Materialization. During materialization an object is moved to a higehr location in RIPQ, so the case is similar to update.

Proof of the main theorem by induction Assume

$$\mathcal{L}^{*(t)}(x) + c^{(t)}(x) \leqslant \mathcal{L}^{(t)}(b) + 4(K - k)\alpha + \mathcal{M}^{(t)}(b)$$

we are going to show it holds for time $t + 1$ as well by considering all the operations that can happen at $t + 1$.

- Insertion.
  Assume x is inserted to q_k in the exact cache, and block b_i in RIPQ cache. By the insertion algorithm and the 2α-bounded size of each section

$$\begin{aligned}\mathcal{L}^{(t)}(b_j) + 4(K - k)\alpha - 2\alpha &< q_k \\ &\leqslant \mathcal{L}^{(t)}(b_j) + 4(K - k)\alpha\end{aligned}$$

For any block $b > b_i$, their location remains the same, while for any object y in those blocks there are two cases: (1) its ideal location remains the same (2) its ideal location decreases by 1 while $c(y)$ increases by 1, so the conclusion holds.

For any block $b < b_i$, its location decreases by 1. Assume $y \in b$, (1) if $y \in \mathcal{I}_{k'}$ for some $k' \leqslant k$, then its ideal location is definitely lower than q_k and so decreases by 1, (2) if $y \in \mathcal{I}_{k'}$ for some $k' > k$, then by induction assumption

$$\begin{aligned}\mathcal{L}^{(*)}(y) + c(y) &\leqslant \mathcal{L}(b) + 4(K-k')\alpha + \mathcal{M}(b)\\ &\leqslant \mathcal{L}(b_i) + 4(K-k)\alpha - 2\alpha + \mathcal{M}(b_i) \leqslant q_k\end{aligned}$$

The second inequality holds because $\mathcal{M}(b) \leqslant 2\alpha$, and the third inequality holds by definition of the adjusted insertion location of q_k.

In both cases $\mathcal{L}^*(y) < q_k$, so the ideal location of y decrease by 1 and the conclusion holds.

- Update.
  Assume x is updated to q_k in the exact cache and moved from b_i to b_j ($b_i < b_j$) in RIPQ cache.
  For any block b above b_j, its location remains the same. For any object $y \in b$, similar to the insertion case, either its ideal location doesn't change, or its ideal location decreases by 1 but its account $c(y)$ increases by 1, so the inequality holds for both cases.
  For any block below b_i, its location doesn't change, and the ideal location of any object in that block doesn't increase, so does its account variable, so the inequality holds as well.
  We only need to consider the blocks between b_i and b_j. Assume $b_i \leqslant b < b_j$, then its location decreases by 1. Assume $y \in b$, there are two cases (1) $\mathcal{L}^{*(t)}(y) < \mathcal{L}^{*(t)}(x)$, then y's ideal location remains the same, but x has previously added 1 to y's account variable and we use it now (decrease $c(y)$ by 1), (2) $\mathcal{L}^{*(t)}(y) > \mathcal{L}^{*(t)}(x)$, and this is the same as the insertion case.
- Materialization.
  Assume x is materialized from b_i to b_j. During this process no object's ideal location changes, and for any block above b_j or below b_i, its location doesn't change either and the conclusion holds. For any block b such that $b_i \leqslant b < b_j$, its location decreases by 1 while its materialization account variable $\mathcal{M}(b)$ increases by 1, so the conclusion still holds. Note we will reduce $c(y)$ if $\mathcal{L}^{*(t)}(y) < \mathcal{L}^{*(t)}(x)$ as well, but that's OK because x must have previously added 1 to $c(y)$.

So far all operations that can happen to RIPQ at time $t+1$, the inequality still holds.

Proof of $\mathcal{M}(b) \leqslant 2\alpha$ Recall that $\mathcal{M}(b)$ is the number of materializations that flew over b and haven't been updated and moved out of the section ever since in its current phase. We further divide b's current ($\mathcal{I}_k$) phase into two parts:

- Since the beginning of the phase, and all the time while *its section hasn't crossed* $\mathcal{I}_k$, i.e. $\mathcal{L}(s(b)) \geqslant \mathcal{I}_k$. We define the number of over-b materializations in this period to be $\mathcal{M}_0(b)$.
- Since its section has crossed $\mathcal{I}_k$. We define the number of over-b materializations in this period to be $\mathcal{M}_1(b)$.

So $\mathcal{M}(b) = \mathcal{M}_0(b) + \mathcal{M}_1(b)$.

We further divide $\mathcal{M}_0(b)$ into two parts: (1) $\mathcal{M}'_0(b)$: number of over-b materializations before its section crossed $\mathcal{I}_k$ *that hadn't been separated from b's section when it had just crossed* $\mathcal{I}_k$. (2) $\mathcal{M}''_0(b)$: $\mathcal{M}_0(b) - \mathcal{M}'_0(b)$, i.e., the subset of $\mathcal{M}_0(b)$ that has been separated from b's section (resulting from splits) when it just crossed $\mathcal{I}_k$.

Let's focus on the state of of RIPQ when b's section has just crossed $\mathcal{I}_k$. This event can take place for two reasons (1) a downward split for b's section (2) insertion/update of objects to higher locations in the queue. Define "`past`" to be the number of objects above b but below $\mathcal{I}_k$

$$\texttt{past} = |\{x : \mathcal{L}(x) > \mathcal{L}(b) \wedge \mathcal{L}(x) \leqslant \mathcal{I}_k\}|\,.$$

Define "`below`" to be the number of objects in the same section but with a lower location than b

$$\texttt{below} = |\{x : x \in s(b) \wedge \mathcal{L}(x) \leqslant \mathcal{L}(b)\}|\,.$$

Because just before b's section crossed $\mathcal{I}_k$, its size was smaller than 2α, and it was intersecting with $\mathcal{I}_k$,

$$\texttt{past} + \texttt{below} \leqslant |s(b)| + 1 \leqslant 2\alpha.$$

The following claims also hold true during b's current phase:

- The number over-b materializations before its section crossed $\mathcal{I}_k$ is at most `past`:
  $$\mathcal{M}_0(b) \leqslant \texttt{past}.$$
  Proof: Each over-b materialization will decrease b's location by 1, so $\mathcal{M}_0(b) \leqslant \mathcal{I}_k - \mathcal{L}(b) \leqslant$ `past`.
- If there are merges for b's section after it crossed $\mathcal{I}_k$, then
  $$\mathcal{M}'_0(b) + \mathcal{M}_1(b) \leqslant \alpha.$$

Proof: at the time of the merge the size of b's section is α, and until $\mathcal{I}_{k-1}$ starts to insert new objects into b's section its size will be no larger than α. $\mathcal{M}_0'(b)+\mathcal{M}_1(b)$ together is the number of over-b materializations that has remained in the same section after its section crossed $\mathcal{I}_k$, and the total number will be bounded by the section size until new objects are inserted to the section. When the section starts intersecting with $\mathcal{I}_{k-1}$, the new objects will be inserted/updated to higher locations than b directly, so won't transit to over-b materializations. If another split happens then the section might *merge in* new objects and thus have new over-b materializations. However, for the split to happen at least α new objects need to be inserted to the section such that its size can reach 2α. By then b would have crossed $\mathcal{I}_{k-1}$, so the next phase would have started.

- If there is no merge for b's section after it crossed $\mathcal{I}_k$, then

$$\mathcal{M}_1(b) \leqslant \texttt{below}.$$

Proof: if there is no merge, then the over-b materializations can only come from objects in the same section but below b.

Now we show $\mathcal{M}(b) \leqslant 2\alpha$ in two cases:

- Case 1: $\texttt{past} + \texttt{below} \leqslant \alpha$.
If there is no merge after b's section crossed $\mathcal{I}_k$, then $\mathcal{M}_1(b) \leqslant \texttt{below}$, and we also have $\mathcal{M}_0(b) \leqslant \texttt{past}$, so

$$\mathcal{M}(b) = \mathcal{M}_0(b)+\mathcal{M}_1(b) \leqslant \texttt{past}+\texttt{below} \leqslant \alpha$$

If there are merges, then $\mathcal{M}_1(b) \leqslant \mathcal{M}_0'(b) + \mathcal{M}_1(b) \leqslant \alpha$, and $\mathcal{M}_0(b) \leqslant \texttt{past} \leqslant \alpha$, so

$$\mathcal{M}(b) = \mathcal{M}_0(b) + \mathcal{M}_1(b) \leqslant 2\alpha$$

- Case 2: $\alpha < \texttt{past} + \texttt{below} \leqslant 2\alpha$.
If there is no merge after b's section crossed $\mathcal{I}_k$, then by the same argument as in Case 1

$$\mathcal{M}(b) = \mathcal{M}_0(b)+\mathcal{M}_1(b) \leqslant \texttt{past}+\texttt{below} \leqslant 2\alpha.$$

If there are merges, then $\mathcal{M}_0'(b) + \mathcal{M}_1(b) \leqslant \alpha$. And because the size of b's section just before it crossed $\mathcal{I}_k$ is at least $\texttt{past} + \texttt{below} > \alpha$, there is at most one previous downward split for that section since b's current phase began: each downward split would decrease the location of a section by α, so if there are two then b's section would have crossed $\mathcal{I}_k$ at that time. Each split will separate out α objects from the section, so $\mathcal{M}_0''(b) \leqslant \alpha$, and

$$\mathcal{M}(b) = \mathcal{M}_0''(b) + \mathcal{M}_0'(b) + \mathcal{M}_1(b) \leqslant 2\alpha.$$

Larger block sizes

If the block size is larger than 1, then the structure of RIPQ would change in (1) block seal (2) section merge operations.

When we seal a block, its location will decrease by at most B, the maximum block size. Note that each block can only be sealed once before it is evicted.

When we merge two sections s_{i+1}, s_i (s_{i+1} above s_i), we move the objects in the active device block of s_i to the head of s_{i+1}, so the location of any block in s_{i+1} decreases by at most B. We call such merge a "downward merge" for s_{i+1} and our goal is to bound the decrease in block location resulting from these downward merges.

Consider a stable state of RIPQ (no pending merging/splitting/sealing operations) and three consecutive sections s_i, s_{i-1}, s_{i-2}. Then $s_{i-1} + s_{i-2} \geqslant \alpha$, otherwise the two sections would have merged. For s_i to perform two downward merges, it would have at least merged to s_{i-2}, and thus its minimal object location would decrease by at least α. Meanwhile the decrease of block location from merging $\leqslant 2B$. The ratio between the two is $2B/\alpha$.

In conclusion, the additional space amplification between two insertion points from a larger block size is bounded by $n_{\mathcal{I}} \times 2B/\alpha + B$ ($n_{\mathcal{I}}$ is the number of objects between two consecutive insertion points).

For TB-scale flash, we expect to have thousands of blocks while only tens of sections for one RIPQ instance, so the space overhead brought by block size is on the order of 1/100 and relatively small compared to the overhead brought by approximate insertion and materialization.