Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality

Yue Cheng
Virginia Tech

Fred Douglis
EMC

Philip Shilane
EMC

Michael Trachtman
EMC

Grant Wallace
EMC

Peter Desnoyers
Northeastern University

Kai Li
Princeton University

Abstract

NAND-based solid-state (flash) drives are known for providing better performance than magnetic disk drives, but they have limits on endurance, the number of times data can be erased and overwritten. Furthermore, the unit of erasure can be many times larger than the basic unit of I/O; this leads to complexity with respect to consolidating live data and erasing obsolete data. When flash drives are used as a cache for a larger, disk-based storage system, the choice of a cache replacement algorithm can make a significant difference in both performance and endurance. While there are many cache replacement algorithms, their effectiveness is hard to judge due to the lack of a baseline against which to compare them: Belady's MIN, the usual offline best-case algorithm, considers read hit ratio but not endurance.

We explore offline algorithms for flash caching in terms of both hit ratio and flash lifespan. We design and implement a multi-stage heuristic by synthesizing several techniques that manage data at the granularity of a flash erasure unit (which we call a container) to approximate the offline optimal algorithm. We find that simple techniques contribute most of the available erasure savings. Our evaluation shows that the container-optimized offline heuristic is able to provide the same optimal read hit ratio as MIN with 67% fewer flash erasures. More fundamentally, our investigation provides a useful approximate baseline for evaluating any online algorithm, highlighting the importance of comparing new policies for caching compound blocks in flash.

1 Introduction

Unlike magnetic disk drives, flash devices such as solid state drives (SSDs) transfer data in one unit but explicitly *erase* data in a larger unit before rewriting. This erasure step is time-consuming (relative to transfer speeds) and it also has implications for the endurance of the device, as the number of erasures of a given location in flash is limited. A common endurance metric is Erasures Per Block Per Day (EPBPD), a rate commonly guaranteed by flash manufacturers for a time period [33, 34].

SSDs typically provide a flash translation layer (FTL) within the device, which maps from logical block numbers to physical locations. A host can access individual file blocks that are kilobytes in size, and if some live blocks are physically located in the same erasure unit as data that can be recycled, the FTL will garbage collect (GC) by copying the live data and then erasing the previous location to make it available for new writes.

As an alternative to performing GC in the FTL, the host can group file blocks to match the erasure unit (also called *blocks* in flash terminology). While some research literature refers to these groupings as "blocks" (e.g., RIPQ [38]), there are many other names for it: write-evict unit [22], write unit [31], erase group unit [30], and *container* in our own recent work on online flash cache replacement [23]. Thus we use "container" to describe these groupings henceforth.

Containers are written in bulk, thus the FTL never sees partially dead containers it needs to GC. However, the host must do its own GC to salvage important data from containers before reusing them. The argument behind moving the functionality from the SSD into the host is that the host has better overall knowledge and can use the SSD more efficiently than the FTL [21].

Flash storage can be used for various purposes, including running a standalone file system [7, 16, 20, 24, 41] and acting as a cache for a larger disk-based file system [8, 9, 13, 19, 29, 32, 36, 42]. The latter is a somewhat more challenging problem than running a file system, because a cache has an additional degree of freedom: data can be stored in the cache or bypassed [15, 35], but a file system must store all data. In addition, the cache may have different goals for the flash storage: maximize hit rate, regardless of the effect on flash endurance; limit flash wear-out, and maximize the hit rate subject to that limit; or optimize for some other utility function that takes both performance and endurance into account [23].

Flash cache replacement solutions such as RIPQ [38] and Pannier [23] consider practical approaches given a historical sequence of file operations; i.e., they are "online" algorithms. Traditionally, researchers compare online algorithms against an offline optimal "best case" baseline to assess how much room an algorithm might have for improvement [5, 9, 11, 43]. For cache replacement, Belady's MIN algorithm [2] has long been used as that baseline.

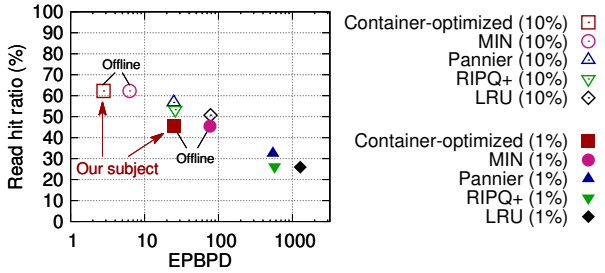


Figure 1: A RHR vs. endurance (EPBPD, on a log scale) scatter-plot of results for different caching algorithms. We report the average RHR and EPBPD across a set of traces, using cache sizes of 1% or 10% of the WSS. (Descriptions of the datasets and the full box-and-whisker plots appear below.) RIPQ+ and Pannier are online algorithms, described in §5.3. The goal of our container-optimized offline heuristic is to reduce EPBPD with the same RHR as MIN, an offline algorithm that provides the optimal RHR without considering erasures.

Figure 1 plots read hit ratio (RHR) against the EPBPD required by a given algorithm assuming SSD sizes of 1% or 10% of the working set size (WSS) of a collection of traces (described in §5.1). MIN achieves an average RHR improvement of 10%–75% compared to LRU, a widely used online algorithm. Using future knowledge makes a huge difference to RHR: Pannier covers only 33% to 52% of the gap from LRU to MIN, while RIPQ+ (described in §5.3) sees about the same RHR as LRU. However, MIN is suboptimal when considering not only RHRs but also flash endurance: it will insert data into a cache if it will be accessed before something currently in the cache, even if the new data will itself be evicted before being accessed. Such an insertion increases the number of writes, and therefore the number of erasures, without improving RHR; we refer to these as *wasted writes*. In this paper we explore the tradeoffs between RHR and erasures when future accesses are known.

A second complicating factor arises in the context of flash storage: containers. Not only should an offline algorithm not insert unless it improves the RHR, it must be aware of the container layout. For example, data that will be invalidated around the same time would benefit by being placed in the same container, so the container can be erased without the need to copy blocks.

We believe that an offline *optimal* flash replacement algorithm not only requires future knowledge but is computationally infeasible. We develop a series of heuristics that use future knowledge to make best-effort cache replacement decisions, and we compare these heuristics to each other and to online algorithms. Our *container-optimized offline heuristic* maintains the same RHR as MIN. The heuristic identifies a block that is inserted into the cache but evicted before being read, and omits that insertion in order to avoid needing to erase the region where that block is stored. At the same time, the heuristic consolidates blocks that will be evicted at around the same time into the same container when possible, to minimize GC activities. One important finding is that simple techniques (e.g., omitting insertions and GC copies that are never reread) provide most of the benefit; the more complicated ones (e.g., consolidating blocks that die together) have a marginal impact on saving erasures at relatively large cache sizes. Alternatively, we describe other approaches to maximize RHR subject to erasure constraints. Figure 1 provides examples in which the average RHR is the same maximum achievable by MIN, while lowering EPBPD by 56%–67%. Interestingly, the container-based online algorithms reduce EPBPD relative to LRU by similar factors.

Specifically, we make the following contributions:

- We thoroughly investigate the problem space of offline compound object caching in flash.
- We identify and evaluate a set of techniques to make offline flash caching container-optimized.
- We present a multi-stage heuristic that approximates the offline optimal algorithm; our heuristic can serve as a useful approximate baseline for analyzing any online flash caching algorithm.
- We experiment with our heuristic on a wide range of traces collected from production/deployed systems, to validate that it can provide a practical upper bound for both RHR and lifespan.

2 Background and Related Work

Here we provide a brief background of the offline caching algorithms, discuss the challenges of finding an offline optimal caching algorithm for container-based flash, and describe some previous analytical efforts.

2.1 Belady's MIN and its Limitations

Belady's MIN algorithm [2] replaces elements whose next reference is the furthest in the future, and it is provably optimal with respect to the read hit ratio given certain assumptions [25, 26]. In particular, it applies in a single level of a caching hierarchy in which all blocks (or pages) *must* be inserted. For instance, it applies to demand-paging in a virtual memory environment.

Our environment is slightly different. We assume a DRAM cache at the highest level of the cache hierarchy and a flash device serving as an intermediate cache between DRAM and magnetic disks. A block that is read from disk into DRAM and then evicted from DRAM can be inserted into flash to make subsequent accesses faster, but it can also be removed from DRAM without inserting into flash ("read-around"). Similarly, writes need not be inserted into flash as long as persistent writes will be stored on disk (see §3.4).

Since this is not a demand-fetch algorithm, MIN is not

necessarily the optimal strategy. Consider a simple 2-location cache with the following access sequence:

A,B,C, A,B,D, A,B,C, A,B,D ...

In a demand-fetch algorithm a missing block must be inserted into the cache, replacing another one; in this case the hit rate will be $\frac{1}{3}$, as B will always be replaced by C or D before the next access. With read-around it is not necessary for C and D to be inserted into cache, allowing hits on both A and B for a hit rate of $\frac{2}{3}$. We note, however, that such behavior may be emulated by a demand-fetch algorithm using one more cache location, which is reserved for those elements which would not be inserted into cache in the read-around algorithm. The hit rate for a read-around algorithm with N cache locations is thus bounded by the performance of MIN with N+1 locations, a negligible difference for larger values of N which we ignore in the remainder of the paper.

Even if MIN provides the optimal RHR, we argue below that it can write more blocks than another approach providing the same RHR with fewer erasures. We use MIN to refer to a variant of Belady's algorithm that does not insert a block into the cache if it will not be reread, while M^+ is a further enhancement that does not insert a block that will not be reread *prior to eviction*.

Temam [39] extends Belady's algorithm by exploiting spatial locality to take better advantage of processor caches. Gill [10] applies Belady's policy to multi-level cache hierarchies. His technique is useful for iterating across multiple runs of a cache policy. However, since Belady targets general local memory caching, it is not directly applicable to container-based flash caching due to the inherent difference between DRAM and flash.

2.2 Container-based Caching Algorithms

Previous work shows that various container-based flash cache designs lead to different performance–lifespan trade-offs [22, 23, 30, 31, 38]. SDF [31], Nitro [22], and SRC [30] use a large write unit aligned to the flash erasure unit size to improve cache performance. RIPQ [38] leverages another level of indirection to track reaccessed photos within containers. Pannier [23] explicitly exploits hot/cold block and invalidation mixtures for container-based caching to further improve performance and reduce flash erasures. However, it is not known how much headroom in both performance and lifespan might exist for any state-of-the-art flash caching algorithms. To give a clear idea of how well an online flash caching algorithm performs, we need an offline optimal algorithm that incorporates performance and lifespan of the flash cache.

2.3 Analytical Approaches

Considerable prior work has explored the offline optimality of caching problems in various contexts from a theoretical perspective. Albers et al. [1] and Brehob et al. [4] prove the NP-hardness of optimal replacement for non-standard caches. Chrobak et al. [6] prove the strong NP-completeness of offline caching supporting elements with varying sizes (i.e., costs). Neither explicitly studies the offline optimality of the flash caching problem with two goals that are essentially in conflict.

Other researchers have looked at related problems. Horwitz et al. [14] formulate the index register allocation problem to the shortest path problem with a general graph model and prove the optimality of the allocation algorithm. Ben-Aroya and Toledo [3] analyze a variety of offline/online wear-leveling algorithms for flash-based storage systems. Although not directly related to our problem, these works provide insights into the offline optimality of container-based flash caching.

3 Quest for the Offline Optimal

A flash device contains a number of *flash blocks*, the unit of erasures, referred to in our paper as *containers* to avoid confusion with file blocks. But many of the issues surrounding flash caching arise even in the absence of containers. We refer to the case where each file block can be erased individually as **unit caching**, and we describe the metrics (§3.1) and algorithms (§3.2) in that context. This separates the general problem of deciding what to write into the flash cache in the first place from the overhead of garbage collecting containers; we return to the impact of containers in §3.3. In §3.2–3.3 we also introduce a set of techniques for eliminating wasted writes. We then discuss how to handle user-level writes in §3.4. Finally, we summarize the algorithms of interest in §3.5.

3.1 Metrics

The principal metrics of concern are:

Read Hit Ratio (RHR): The ratio of read I/O requests satisfied by the cache (DRAM cache + flash cache) over total read requests.

The Number of Flash Erasures: In order to compare the impact on lifespan across different algorithms and workloads, we focus on the EPBPD required to run a given algorithm on a given workload and cache size. The total number of erasures is the product of EPBPD, capacity, and workload duration.

Flash Usage Effectiveness (FUE): The FUE metric [23] endeavors to balance RHR and erasures. It is defined as the number of bytes of flash hit reads divided by flash writes, including client writes and internal copy-forward (CF) writes. A score of 1 means that, on average, every byte written to flash is read once, so higher scores are better. It can serve as a utility function to evaluate different algorithms. We define *Weighted FUE* (**WFUE**), a variant of FUE that considers both RHR and erasures and uses a weight to specify their relative importance:

Technique	Description	C
R_N	omit insertions reread $< N$ times	✗
TRIM	notify GC to omit dead blocks	✓
CFR	avoid wasted CF blocks	✓
E	segregate blocks by evict time	✓

Table 1: Summary of offline heuristic techniques used for eliminating wasted writes to the flash cache. C: container-optimized.

$$\text{WFUE} = \alpha * (\text{RHR}_A/\text{RHR}_M) + (1-\alpha) * (E_M - E_A)/E_M$$

The utility of an algorithm is determined by comparing the RHR and erasures (E) incurred by the algorithm, denoted by A, to the values for M^+ (an improved MIN, described in §3.2, denoted here by M for simplicity).[1] If α is low and an algorithm saves many writes in exchange for a small reduction in RHR, WFUE will increase.

3.2 Objectives and Algorithms

Depending on the goals of end users, we may have different objective functions. Optimizing for RHR irrespective of erasures is trivial: the performance metric RHR serves as a naïve but straightforward goal for which MIN can easily get an optimal solution, without considering the flash endurance. Taking erasures into account, we identify three objectives of interest. We describe each briefly to set the context for comparison, then elaborate on heuristics to optimize for them. (We do not claim their optimality, leaving such analysis to future work.)

O1: Maximal RHR The purpose of objective **O1** is to *minimize erasures subject to maximal RHR*. If we consider the RHR obtained by MIN, there should be a sequence of cache operations that will preserve MIN's hit ratio while reducing the number of erasures. Belady's **MIN** caches any block that either fits in the cache, or which will be reaccessed sooner than some other block in the cache. It does not take into account whether the block it inserts will itself be evicted from the cache before it is accessed.

The first step to reducing erasures while keeping the maximal RHR is to identify *wasted cache writes* due to eviction. **Algorithm M+** is a variant of MIN that identifies which blocks are added (via reads or writes) to the cache and subsequently evicted without rereference, then no longer inserts them into the cache (R_N in Table 1, where $N = 1$).

It is unintuitive, but cache writes can be wasted even if they result in a read hit. As an example, assume block A is in cache at time t_0, and will next be accessed at time $T > t_0$. If block B is accessed at t_0, and will be accessed exactly one more time at time $T-1$, MIN dictates that A be replaced by B. However, by removing B, there is still one miss (on B rather than A), while an extra write has occurred. Leaving A in the cache would have the same RHR but one fewer write into the cache.

Ultimately, our goal is to identify a Pareto-optimal solution set where it is impossible to reduce the number of erasures without reducing RHR. This requires that no block be inserted if it does not improve RHR, but the complexity of considering every insertion decision in that light is daunting. Thus we start with eliminating cache insertions that are completely wasted and leave additional trades of one miss against another to future work.

An offline heuristic **Algorithm H** that approximates **M+** works as follows:

Step 1 Annotate each entry with its next reference.

Step 2 Run **MIN** to annotate the trace with a sequence of cache insertions and evictions, given a cache capacity. Note all insertions that result in being evicted without at least one successful reference.

Step 3 Replay the annotated trace: do not cache a block that had not been accessed before eviction.

O2: Limited Erasures In some cases a user will be willing to sacrifice RHR in order to reduce the number of erasures. In fact, given limits on the total number of erasures of a given region of flash, it may be essential to make that tradeoff. Thus, **O2** first limits erasures to a particular rate, such as 5 EPBPD. (The EPBPD rate is multiplied by the size of the flash cache and the duration of the original trace to compute a total budget for erasures.) *Given an erasure limit, the goal of **O2** is to maximize RHR.* Note that the rate of erasures is averaged across an entire workload, meaning that the real limit is the total number of erasures; EPBPD is a way to normalize that count across workloads or configurations.

We can modify **Algorithm H** for **O2** to have a threshold. $\mathbf{H}_T$ works as follows:

Step 1 Run **H** and record all insertions. Annotate each record with the number of read hits absorbed as a result of that insertion, and count the total number of insertions resulting in a given number of read hits.

Step 2 Compute the number of cache insertions I performed in the run of **H** and the number of insertions I' allowed to achieve the EPBPD threshold T. If $I > I'$ then count the cumulative insertions CI resulting in 1 read hit, 2 read hits, and so on until $I - CI = I'$. Identify the reuse count, R, at which eliminating these low-reuse insertions brings the total EPBPD to the threshold T. Call the number of cache insertions with R reuses that must also be eliminated the leftover, L.

Step 3 Rerun **H**, skipping all insertions resulting in

[1] Though these two factors have different ranges and respond differently to changes, WFUE controls the value of both via normalization so that the higher each factor yields, the better the algorithm performs with respect to that goal. A negative value due to high erasures would demonstrate the deficiency of the algorithm in saving erasures. Hence, WFUE can serve as a general metric for quantitatively comparing different heuristics.

fewer than R read hits, and skipping the first L insertions resulting in exactly R hits.

Algebraically, we can view the above description as follows: Let A_i represent the count of cache insertions absorbing i hits.

$$I = \sum_{i=1}^{n} A_i \qquad \text{and} \qquad CI = (\sum_{i=1}^{R-1} A_i) + L$$

We identify R such that this results in $I - CI = I'$.

O3: Maximize WFUE The goal of **O3** is to *maximize WFUE*, which combines RHR and erasures into a single score to simplify the comparison of techniques (§3.1). Intuitively, the user may want to get the highest read hits per erasure (i.e., best "bang-for-the-buck" considering the user pays the cost of device endurance for RHR).

To compare the tradeoffs between RHR and erasures, we consider a variant of **H**, **Algorithm $\mathbf{H}_N$**, which omits cache insertions that are reread $< N$ times (R_N in Table 1). This is similar to the threshold-based **Algorithm $\mathbf{H}_T$**, but the decision about the number of reaccesses necessary to justify a cache insertion is *static*. An increase in the minimum number of reads per cache insertion should translate directly to a higher FUE, though the writes due to GC are also a factor. For WFUE, the value of α determines whether such a threshold is beneficial.

3.3 Impact from Containers

The metrics and objectives described in §3.1–3.2 apply to the unit caching scenario, in which each block may be erased separately, but they also apply to the container environment. The aim is still to minimize erasures subject to a maximal RHR, to maximize RHR subject to a limit on erasures, or to maximize a utility function of the two.

However, the approach to *solving* the optimization problem varies when containers are considered. This complexity arises because there is an extra degree of freedom: not only does an algorithm need to decide *whether* to cache a block, it must decide *where* it goes and whether to reinsert it during GC. Regarding placement, one option is to cache data in a container-oblivious manner. For instance, a host could write each block to a distinct location in flash and rely on an FTL to reorganize data to consolidate live data and reuse areas of dead data. This might result in a significant overhead from the FTL unwittingly copying forward blocks that MIN knows are no longer needed, so adding the SSL *TRIM* [40] command to inform the FTL that a block is dead can reduce erasures significantly (see §6.1). As shown in Table 1, we categorize TRIM as *container-optimized*, because an FTL itself manages data at the granularity of containers.

For CF, the first step is to supplement the annotations from §3.2 with information about blocks that are CF and not reaccessed. Copy-Forward Reduction (*CFR* in Table 1) effectively extends TRIM with the logic of R_1, by identifying "wasted" CFs; however, eliminating *all* needless CFs is difficult. With the smallest cache, on average this reduces the erasures due to wasted CFs from 4% to 1%; repeating this step a few more times brings it down another order of magnitude but does not completely eliminate wasted CFs. This is because (for a small cache) there is always a block to CF that has not yet been verified as a useful copy nor marked as wasted. Note that while it seems appealing to simply not copy something forward that was not copied forward in a previous iteration, the act of excluding a wasted copy makes new locations for data available, perturbing the sequence of operations that follow. This makes the record from the previous run only partly useful for avoiding mistakes in the subsequent run, an example of the "butterfly effect."

Still, writing in a container-optimized manner can improve on the naive data placement of M^+, which uses the FTL to fill a container at a time. By segregating blocks by their expiration time as they are written to flash (E in Table 1), we may be able to erase one container, without the need to CF, as soon as all the blocks within it are no longer needed. We refer to E as *container-optimized* since it explicitly consolidates data that die together at the host or application level.

Since the purpose of our study is to provide a best-case comparison point for real-world cache replacement algorithms, we focus henceforth on the container-based cache replacement policies. Note that if containers consist of only one block, any approaches that are specifically geared to containers should work for the unit caching replacement policy. In the next section, we describe the algorithms in greater detail, using **C** to represent the offline heuristic **H** in the context of containers.

3.4 Impact from Dirty Data

The results from the various algorithms depend significantly on how the cache treats writes into the file system. For example, Pannier [23] requires that all file writes be inserted into the cache, with the expectation that the cache be an internally consistent representation of the state of the file system at any given time. All writes are immediately reflected in stable storage, which is appropriate for an online algorithm; Pannier's comparison to Belady used the same approach even with future knowledge, writing all user-level writes into SSD.

With future knowledge, however, one can argue that a "dead write" that will be overwritten before being read need not be inserted into the cache. The same is true of a write that is never again referenced, though in that case it should be written through to disk. Since we model only the flash cache RHR and endurance, we place these dead writes into DRAM but not into flash.

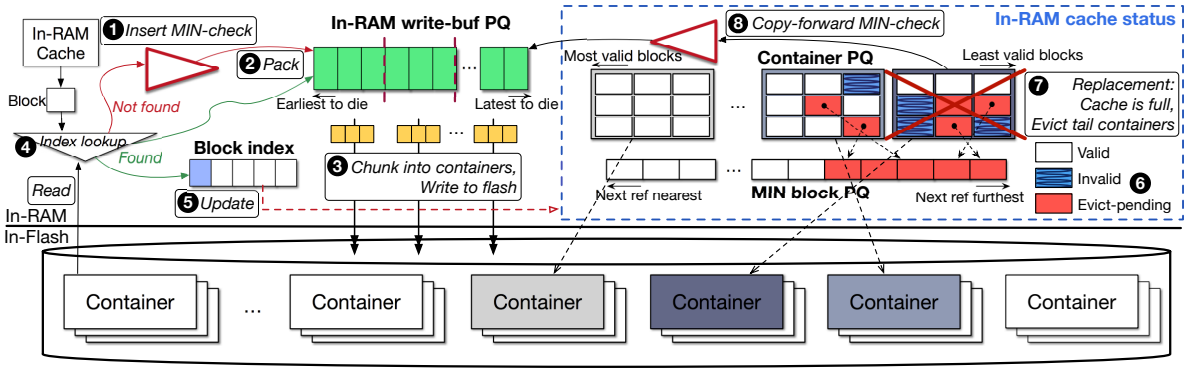


Figure 2: Container-optimized offline flash caching framework. PQ: priority queue.

3.5 Algorithm Granularity

For the remainder of this paper, we compare the following algorithms. **M** refers to variants of MIN while **C** refers to container-optimized algorithms. A table summarizing these (and other) algorithms appears in §5.3.

M Belady's MIN, which does not insert a block that will be overwritten or never reread by the client.

M+ A variant of MIN, which identifies a block that is inserted into cache but evicted before being read, and omits that insertion. It also uses TRIM to avoid CF once the last access to a block occurs.

$\mathbf{M}_N$ A variant of M^+, which does not insert blocks with accesses $< N$. M_1 is equivalent to M^+, while M_N generalizes it to $N > 1$.

$\mathbf{M}_T$ A variant of M^+, which eliminates enough low-reuse cache insertions to get the best RHR under a specific erasure limit (see §3.2).

C Each block is inserted if and only if M^+ would insert it. However, a write buffer of W containers is set aside in memory, and the system GCs whenever it is necessary to make that number of containers free to overwrite. Containers are filled using the *eviction timestamp* indicating when, using M^+, a block would be removed from the cache. The contents of the containers are ordered by eviction time, so the first container has the m blocks that will be evicted first, the next container has the next m blocks, and so on.

$\mathbf{C}_N$, $\mathbf{C}_T$ Analogous to M_N, M_T.

4 Offline Approximation

Here we describe how to evolve the container-oblivious MIN algorithm to a container-optimized heuristic C, which provides the same RHR but significantly fewer erasures. We also explain creating C_N and C_T.

4.1 Key Components

Figure 2 depicts the framework architecture and examples of the insertion and lookup paths. A detailed discussion appears in the next subsection, but the following components are the major building blocks.

Block Index An in-memory index maps from a block's key (e.g., LBA) to a location in flash. Upon a read, the in-memory index is checked for the block's key, and if found, the flash location is returned. Newly inserted blocks are added to the in-memory write buffer queue first. Once the content of the writer buffer is persisted in the flash cache, all blocks are assigned an address (an index entry) used for reference from the index. When invalidating a block, the block's index entry is removed.

In-RAM Write Buffer An in-memory write buffer is used to hold newly inserted blocks and supports in-place updates. The write buffer is implemented as a priority queue where the blocks are ranked based on their eviction timestamps (described in greater detail in §4.2). The write buffer queue is filled cumulatively and updated in an incremental fashion. Once the write buffer is full, its blocks are copied into containers, *sealed* and *persisted* in the flash cache. The advantage of cumulative packing and batch flushing is that the blocks with close eviction timestamps get allocated to the same container so that erasures are minimized. Overwrites to existing blocks stored in flash are redirected to the write buffer without updating the sealed container.

In-RAM Cache Status A few major in-memory data structures construct and reflect the runtime cache status. Once a container is sealed, its information structure is inserted into a **container priority queue** (PQ), a structure to support container-level insertion and eviction. Whenever a container is updated (e.g., a block is invalidated or evict-pending, etc.), its relevant position in the queue is updated. In addition to the container PQ, a **block-level MIN PQ** is designed to support the extended Belady logic and track the fine-grained block-level information. We discuss the operations of the MIN PQ in §4.2.

4.2 Container-optimized Offline Heuristic

The multi-stage heuristic C offers the optimal RHR while attempting to approach the practically lowest number of erasures on the flash cache. In the following, we describe the container-optimized heuristic pipeline (Figure 2) in

```
1 void Lookup(Object obj):
2   // WB: Write buffer priority queue
3   if WB.exist(obj.key) or INDEX.exist(obj.key):
4     Object existing = Read(obj.key)
5     OnAccess(existing, obj)
6   else: OnInsert(obj) // Upon a miss
7
8 void OnInsert(Object obj):
9   if not MINFull():
10    if obj.next_ref == INF: return
11    if Rec[obj.key].read_freq <= read_freq_thresh:
12      return
13    MIN_Q.insert(obj) // Insert into MIN queue
14  else:
15    Object victim = MIN_Q.top()
16    if obj.next_ref > victim.next_ref:  return
17    if Rec[obj.key].read_freq <= read_freq_thresh:
18      return
19    // Trigger evict on MIN queue
20    EvictMIN(MIN_Q, victim)
21    MIN_Q.insert(obj)
22  if WB.full(): OnSeal()
23  WB.insert(obj) // Pack into WB
24  EvictFlash()
25
26 void OnSeal():
27  Object obj = WB.begin()
28  // Iterate through all sorted objs
29  while obj != WB.end():
30    FreeCList[curr_ptr].insert(obj)
31    if FreeCList[curr_ptr].full():
32      // C_Q: Container queue
33      C_Q.insert(FreeCList[curr_ptr++])
34    obj = WB.next()
35  WB.clear()
36
37 void EvictMIN(Object victim):
38  MIN_Q.pop();
39  if WB.exist(victim.key): // Remove if in WB
40    WB.erase(victim)
41    return
42  Container c = GetContainer(victim)
43  victim.evict_pending = true
44  c.num_evict_pending++
45  C_Q.update(c) // Update c's position in C_Q
46
47 void EvictFlash():
48  while FlashFull():
49    Container c = C_Q.pop()
50    GC(c) // Garbage collect the evicted c
51
52 void OnCopyForward(Object obj):
53  if obj.next_ref == INF: return
54  MIN_Q.insert(obj)
55
56 void OnAccess(Object old_obj, Object new_obj):
57  if old_obj.evict_pending:
58    Count access as a miss
59    return
60  Update hit stats
61  old_obj.next_ref = new_obj.next_ref
62  // Update obj's position in MIN queue
63  MIN_Q.update(old_obj)
64  old_obj.evict_time = new_obj.evict_time
```

Figure 3: Functions handling events for flash-cached blocks and containers in the container-optimized offline heuristic.

detail. Figure 3 shows the pseudocode of how the offline heuristic handles different events.

Insert, Access and Seal We describe inserting a block, accessing a block, and sealing/persisting a container. ❶ When a client inserts a block upon a miss and the cache is not full, the `OnInsert` function first checks if the block's next reference timestamp is `INFINITE` (i.e., the block is never read again in the future or the next reference is a write). If so, `OnInsert` simply bypasses it and returns. Otherwise, an insertion record is checked to see if the block exceeds `read_freq_thresh`, a configurable read hit threshold. For instance, setting the `read_freq_thresh` to 1 filters out those that are inserted but evicted before being read, avoiding a number of

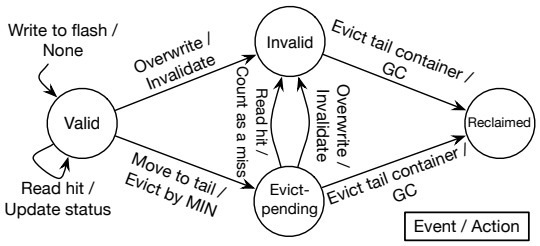


Figure 4: State transitions of blocks in our heuristic.

wasted writes. C_N and C_T also take advantage of this scheme for trading-off RHR with endurance: setting the threshold higher additionally filters out the less *useful* writes, reducing RHR but decreasing the number of erasures. The threshold can be fixed (C_N) or computed based on the total erasure budget (C_T, as described for H_T in §3.2). More useful writes, which result in a greater number of read hits, still take place. We study the trade-offs in §6.3.

Once the threshold test succeeds, if the cache is full and the block will be referenced furthest in the future (i.e., the block has a greater next reference timestamp than the most distant block (`victim`) currently stored in the cache), `OnInsert` returns without inserting it. When both checks are passed, `EvictMIN` function is triggered to evict the `victim` and the new block is inserted into the MIN queue (`MIN_Q`). At the same time, ❷ the block is added to the in-memory write buffer queue (`WB`).

❸ When `WB` is full, all the blocks held in it, sorted based on their eviction timestamp, are copied into multiple containers from the free container list `FreeCList`. `curr_ptr` maintains a pointer to the first available free container in `FreeCList`. (We compare the sorted approach to FIFO insertion of the blocks in §6.2.) The `OnSeal` function then persists the open containers in the flash cache.

Lookup ❹ On a `Lookup`, both the `WB` and in-memory index (`INDEX`) are referenced to locate the block, and the read is serviced. On a read access (`OnAccess`), ❺ the read hit updates the existing block's block-level metadata (`old_obj`) and `old_obj`'s position is updated in `MIN_Q` on the next access time. Upon a miss `OnInsert` is triggered as described.

Invalidation and Eviction ❻ The container-optimized offline caching introduces another new block state – *evict-pending*. Evict-pending describes the state when a block is evicted from `MIN_Q` (transitioning from the *valid* state to evict-pending) but temporarily resides in the GC area of the flash, pending being reclaimed. Figure 4 shows the state transitions of a block in the heuristic. A block is inserted/reinserted into the flash cache with a valid state. Once it is overwritten, the old block in the flash is marked as invalid and the updated data is inserted into `WB`. Overwriting an evict-pending block makes it transition to the invalid state. If the `victim` to be evicted

from MIN_Q happens to reside in WB, EvictMIN directly removes it from the memory. The on-flash container maintains a num_evict_pending counter. On evict-pending, the corresponding container increments its counter and updates its position in the container PQ C_Q.

Let V, I, and E represent the percentage of valid, invalidated and evict-pending blocks in a container, respectively; then $V+I+E=100\%$. The priority of a container is calculated using V. When the cache is full, EvictFlash selects the container with the lowest V (i.e., the fewest valid blocks) for eviction.

Copy-forwarding and GC ❼ When the cache is full and a container has been selected for eviction, the heuristic copies valid blocks forward to the in-memory write buffer. Function OnCopyForward is called to check if the reinserted block is useful. All the invalidated and evict-pending blocks get erased in the flash. The selected container is then reclaimed and inserted back to FreeCList.

❽ The check for a "useful" reinserted block looks for future references and (optionally) confirms the block will not be evicted before it is read.

5 Experimental Methodology

Throughout our analyses, we set the flash erasure unit size to 2MB. We place a small DRAM cache (5% of the flash cache size) in front of the flash cache to represent the use case where the flash cache is used as a second-level cache. (The DRAM cache uses the MIN eviction policy for offline flash cache algorithms and LRU for online ones.) Because some of the datasets in the repositories we accessed have too small a working set for 5% of the smallest flash cache to hold the maximum number of in-memory containers, we restrict our analyses to those datasets with a minimum 32GB working set.

This section describes the traces; implementation and configuration for the experimental system; and the set of caching algorithms evaluated.

5.1 Trace Description

We use a set of 34 traces from 3 repositories:

EMC-VMAX Traces: This includes 25 traces of EMC VMAX primary storage servers [37] that span at least 24 hours, have at least 1GB of both reads and writes, and meet the minimum working set threshold (slightly over half the available traces).

MS Production Server Traces: This includes 3 storage traces from a diverse set of Microsoft Corporation production servers captured using event tracing for windows instrumentation [18], meeting the 32GB minimum.[2]

MSR-Cambridge Traces: This includes 6 block-level traces lasting for 168 hours on 13 servers, representing a typical enterprise datacenter [28]. We narrowed available traces to 6 to include appropriate traces for cache studies. The properties include a working set size greater than 32GB, $\geq 5\%$ of capacity accessed, and read/write balance ($\leq 45\%$ writes).[3]

Given a raw trace, we annotate it by making a full pass over the trace and marking each 4KB I/O block with its next reference timestamp. Large requests in traces are split into 4KB blocks, each of which is annotated with the timestamp of its next occurrence individually. The annotated trace is then fed to the cache simulator. Round 1 simulation, e.g., M, may generate the insert log that can be used by round 2 simulation (e.g., M^+, M_N) to filter out blocks that will be evicted before being read.

5.2 Implementation and Configuration

For the experimental evaluation, we built our container-optimized offline caching heuristic by adding about 3,900 lines of C++ code to a full-system cache simulator. It reports metrics such as hit ratio and flash erasures based on the Micron MLC flash specification [27].

The size of the flash cache for each trace is determined by a fixed fraction of the WSS of the trace (from 1–10%). For C, a write buffer queue (with default size equal to 4 containers) is used for newly inserted blocks and is a subset of this DRAM cache. We over-provision the flash capacity by 7% by default; this extra capacity is used for FTL GC or to ensure the ability to manually clean containers in the container-optimized case. We discuss varying the over-provisioning space in §6.2.

We conduct the simulation study on 4 VMs each equipped with 4 cores and 128 GB DRAM. All tests are run in parallel (using GNU parallel [12]). We measured the CPU time of heuristic M^+ and C looping over all traces, not including the runtime of trace annotating and insertion log generating pre-runs. We ran the experiments 5 times and variance was low. C takes 21.7% longer (2.47 hr) than M^+ (2.03 hr) for the smallest cache size due to the overhead of PQ used by C under intensive GCs. The heuristic keeps track of more metadata in 10% cache size. Thus, with the least amount of GCs, it takes M^+ almost as long (2.19 hrs) as C does (2.21 hrs), to replay all traces. The results show that our heuristic simulation can process a large set of real-world traces within a reasonable time. This strengthens our confidence that our offline heuristics can serve as a practically useful tool.

5.3 Caching Algorithms

Table 2 shows the caching algorithms selected to represent past and present work. We select the classic LRU algorithm, two state-of-the-art container-based online algorithms (described next), and a variety of offline algo-

[2] The traces are: BuildServer, DisplayAdsPayload, DevelopmentToolsRelease.

[3] The traces are: prn0, prn1, proj0, proj4, src12, usr2.

Policy	Abbrev.	Description	O	C
LRU	L	least recently used	✗	✗
RIPQ+	R^+	RIPQ with overwrites, segmented-LRU	✗	✗
Pannier	P	container-based, S2LRU*, survival queue, insertion throttling	✗	✓
MIN	M	FK, do not insert data whose next ref. is the most distant	✓	✗
MIN+	M^+	FK, do not insert data evicted without read (R_1+TRIM, **O1**)	✓	✓
MIN+write-threshold	M_T	FK, limit number of insertions (R_T+TRIM, **O2**)	✓	✓
MIN+insertion-removal	M_N	FK, do not insert data with accesses $< N$ (R_N+TRIM, **O3**)	✓	✓
Container-optimized	C	FK, container-optimized (R_1+TRIM+CFR+E, **O1**)	✓	✓
C+write-threshold	C_T	FK, container-optimized (R_T+TRIM+CFR+E, **O2**)	✓	✓
C+insertion-removal	C_N	FK, container-optimized, do not insert data w/ acc. $< N$ (R_N+TRIM+CFR+E, **O3**)	✓	✓

Table 2: Caching algorithms. FK: future knowledge, O: offline, C: container-optimized.

rithms (as described in §4). The configurations for previous work are the default in their papers (e.g., number of queues) unless otherwise stated.

RIPQ+ is based on RIPQ [38], a container-based flash caching framework to approximate several caching algorithms that use queue structures. As a block in a container is accessed, its ID is copied into a virtual container in RAM, and when a container is selected for eviction, any blocks referenced by virtual containers are copied forward. We adopt a modified version of RIPQ that handles overwrite operations, referred to as RIPQ+ [23]. We use the segmented-LRU algorithm [17] and a container size of 2MB with 8 insertion points.

Pannier [23] is a container-based flash caching mechanism that identifies divergent (heterogeneous) containers where blocks held therein have highly varying access patterns. Pannier uses a priority-based survival queue to rank containers based on their survival time, and it selects a container for eviction that has either reached the end of its survival time or is the least-recently used in a segmented-LRU structure. During eviction, frequently accessed blocks are copied forward into new containers. Pannier also uses a multi-step credit-based throttling scheme to ensure flash lifespan.

6 Evaluation

In §6.1 we evaluate a number of caching algorithms with respect to RHR and EPBPD. We also evaluate the contribution of various techniques on the improvement in endurance. This is followed in §6.2 by a sensitivity analysis of some of the parameters, the use of the write buffer and overprovisioning. We then consider tradeoffs that improve endurance at a cost in RHR (§6.3).

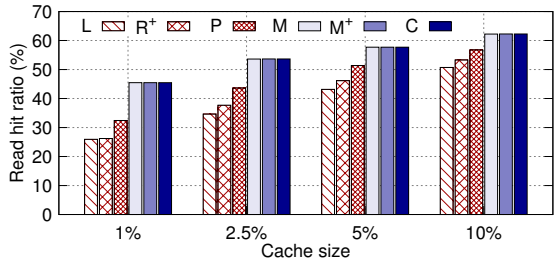
6.1 Comparing Caching Algorithms

We first compare the three online algorithms from Table 2 with the RHR-maximizing offline algorithms: MIN, M^+ and C. We focus initially on **O1**, minimizing erasures subject to a maximal read hit ratio. Figure 5 shows the RHR and EPBPD results across all 34 traces, while varying the cache size for each trace.

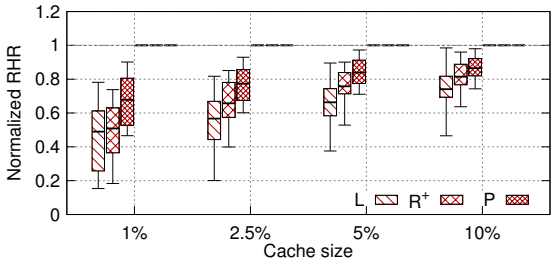
In Figure 5(a), we see that LRU (the left bar in each set) obtains the lowest hit rate because it has neither future knowledge nor a sophisticated cache replacement algorithm. RIPQ+ has about the same RHR as LRU; its benefits arise in reducing erasures rather than increasing hit rate. Pannier achieves up to 26% improvement in RHR over LRU and RIPQ+. By leveraging future knowledge, MIN, M^+ and C achieve the (identical) highest hit ratio, which improves upon Pannier by 9.7%–40%. The gap is widest for the smallest cache sizes. Figure 5(b) shows the range of the normalized RHR (normalized against the best-case C), with a similar trend as shown in Figure 5(a).

Figure 5(c) and 5(d) show that online algorithms incur the most erasures. Pannier performs slightly better than RIPQ+ due to the divergent container management and more efficient flash space utilization. Though it is not explicitly erasure-aware, MIN saves up to 86% of erasures compared to Pannier. This is because with perfect future knowledge, MIN can decide not to insert blocks that would never be referenced or whose next reference is a write. This implicit admission control mechanism results in significantly fewer flash erasures and higher RHR. M^+ further reduces erasures by 31% compared to MIN, because M^+ avoids inserting blocks that would be evicted before being accessed and uses TRIM to avoid copying blocks during GC if they will not be rereferenced. Variation does exist, as shown in Figure 5(d): the 10% and 25%-ile (the lower bound of box and whiskers) breakdown of M^+ are closer to 1 than those of MIN; the lower bounds never reach below 1 while the upper bounds (75% and 90%-ile) are far above, especially for small cache sizes. At small cache sizes, the container-optimized offline heuristic, C, further improves on M^+ by 40% by lowering GC costs.

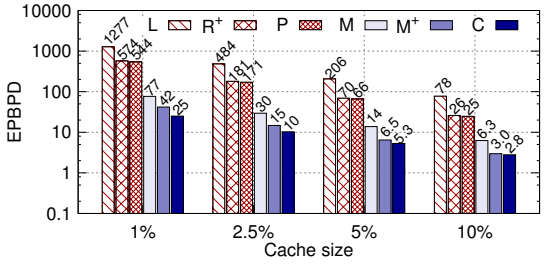
One way to understand the differences among MIN, M^+, and C is to view the contributions of the individual improvements. The cumulative fraction of erasures saved by the techniques used by M^+ or C (summarized in Table 1) is depicted in Figure 6, normalized against the erasures used by MIN. Surprisingly, we find that simple techniques such as R_1 and TRIM have a greater impact on reducing erasures compared to more advanced techniques such as the container-optimized E.



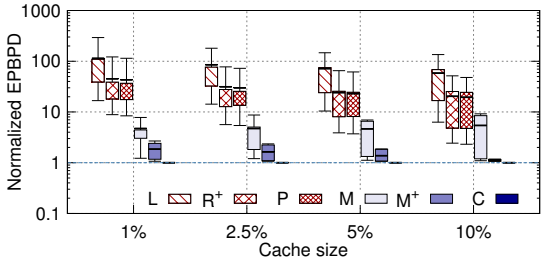
(a) Read hit ratio.



(b) Normalized read hit ratio. $M/M^+/C$ are all 1 due to normalization.



(c) EPBPD.



(d) Normalized EPBPD.

Figure 5: RHR and EPBPD for various online/offline caching algorithms and sizes. EPBPD is shown on a log scale, with values above the bars to ease comparisons. The box-and-whisker plots in (b) and (d) show the {10, 25, average, 75, 90}%-ile breakdown of the normalized RHR and EPBPD, respectively; each is normalized to that of the best-case C.

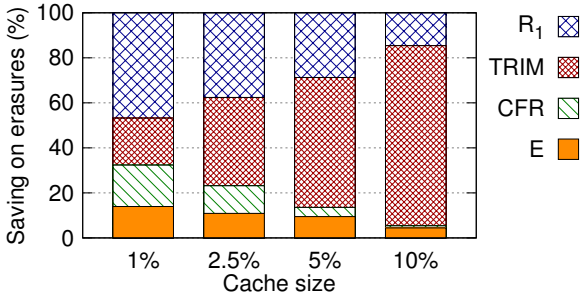


Figure 6: Fraction of erasures saved in different stages. R_1: never inserting blocks that will be evicted before being read. TRIM: removing blocks that would not be reread at FTL layer, CFR: avoiding wasted CFs, E: packing blocks in write buffer using eviction timestamp.

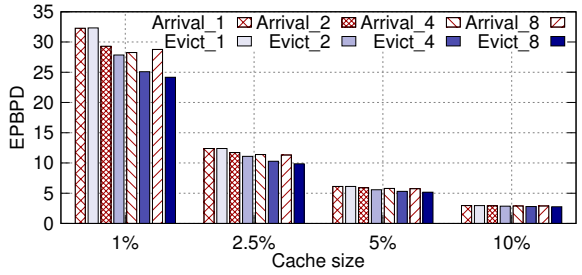
- The top component in each stacked bar (**R_1**) shows the relative improvement from preventing the insertion of blocks that will be evicted without being referenced; for the smallest cache, this accounts for about half the overall improvement from all techniques, but it is a relatively small improvement for the largest cache.
- For MIN, using **TRIM** to avoid the FTL copying of blocks that will not be reaccessed has an enormous contribution for the largest cache (˜80% of all erasures eliminated), but it is a much smaller component of the savings from the smallest caches. Note that we convert MIN to M^+ by avoiding (1) unread blocks due to eviction and (2) FTL GC for unneeded blocks.
- Adding a check for blocks that are copied forward (**CFR**) but then evicted without being rereferenced has a moderate (10%) impact on the smallest cache, but little impact on the largest. This is done only for C, as the copy-forwarding within the FTL for M^+ occurs in a separate component.
- Using 4 write buffers and grouping by eviction timestamp (**E**) has a similar effect to **CFR** on smaller caches and a nontrivial improvement for larger ones.

6.2 Sensitivity Analysis

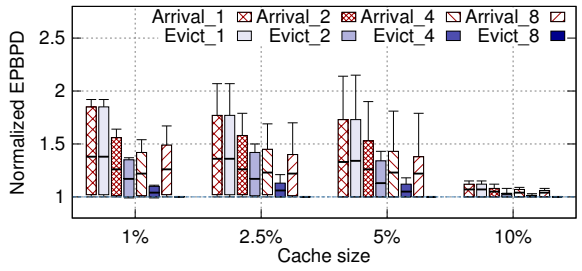
Thus far we have focused on default configurations. Here we compare the impact of different in-memory write buffer designs and sizes on erasures. Then we examine the impact of over-provisioned capacity.

Impact of Consolidating Blocks

We study the impact of consolidating blocks with similar eviction timestamps into the same containers and the impact of sizing the write buffer. Figure 7 plots the average EPBPD and variation across all traces, as a function of policy and write buffer size, grouped by different cache sizes. All experiments are performed using C and give the same optimal RHR. By default C uses the priority-based queue structure as the in-memory write buffer, which is filled using the eviction timestamp indicating when, using M^+, the block would be evicted from the cache. The write buffer, once full, is dispersed into containers that are written into flash. For comparison purposes we implemented a FIFO-queue-based write buffer, where the blocks are simply sorted based on their

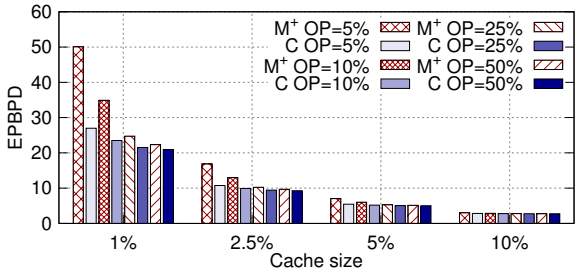


(a) EPBPD.

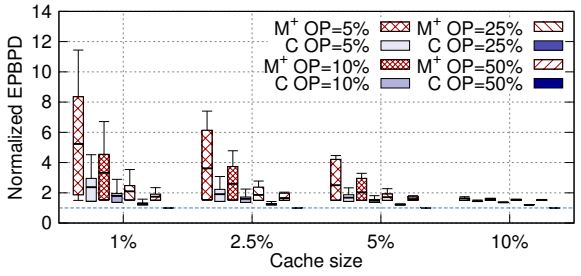


(b) Normalized EPBPD.

Figure 7: Impact of consolidating blocks based on eviction timestamp, and trade-offs in varying the size of the in-memory write buffer (a multiple of containers); `Arrival_2` means packing the blocks into the write buffer (2-container worth of capacity) based on their arrival time, `Evict_4` means packing the blocks into the 4-container-sized write buffer based on the eviction timestamp. The box-and-whisker plot in (b) shows the {10, 25, average, 75, 90}%-ile breakdown of the EPBPD, normalized to that of `Evict_8`.



(a) EPBPD.



(b) Normalized EPBPD.

Figure 8: Trade-offs in over-provisioned (OP) space. The box-and-whisker plot shows normalized EPBPD against `C OP=50%`.

arrival timestamp.[4] There is no difference in EPBPD with a queue size of 1 container, because blocks cannot be sorted by eviction timestamp with a single open container in DRAM. Increasing the write buffer size, we observe a reduction in erasures with `Evict`. This effect is more pronounced with a bigger write buffer queue. For example, for the 2.5% cache size, `Evict_2` reduces the EPBPD by 5% compared to `Arrival_2`; but this EPBPD differential increases to 13% for an 8-container write buffer. This is because a bigger `Evict` write buffer results in less fragmentation due to evict-pending blocks in containers stored in flash. The trend shown in average consistently matches that of the individual variation in Figure 7(b). Interestingly, `Arrival_8` with a 1% cache size yields a slightly higher EPBPD than that of `Arrival_4`. This is because the fraction of data copied forward internally (due to GC) is higher when using a relatively small cache and a large write buffer. We observed that while the 4 least populated containers generally were sufficiently "dead" to benefit from GC, the next 4 (5^{th}–8^{th} least populated) containers hold significantly more live data than the first four when collected in a batch; this increases CF significantly.

[4]Packing blocks based on LBA or whether clean (newly inserted) or dirty (invalidated due to overwrite), yields no impact on erasures or WFUE scores for the offline heuristics.

Impact of Over-provisioned Capacity

Next, we analyze the impact of over-provisioned (OP) capacity on erasures. Figure 8 shows the average EPBPD when varying the over-provisioned space between 5% and 50% for different cache sizes. As in the previous experiment, we omit results of RHR, which are unaffected by overprovisioning. We classify the results into block-level (M^+) and container-optimized (C) approaches, which are interspersed and grouped by the amount of over-provisioned capacity. (Thus, for a given capacity, it is easy to see the impact of the container-optimized approach.)

For both groups, a larger OP space results in lower EPBPD, because the need for GC to reclaim new space becomes less urgent than a flash equipped with relatively smaller OP space. This effect is more significant for M^+, as M^+ manages data placement in a container-oblivious manner; this results in more GCs, which in turn cause larger write amplification (more internal writes at the FTL level). We observe that for a 1% cache size, C with 10% OP incurs fewer erasures than M^+ with 25% OP. This is because C consolidates blocks that would be removed at roughly the same time, resulting in significantly fewer internal (CF) flash writes. C also avoids CF of most blocks that get evicted before reaccess. With the largest cache, however, the relative benefit from additional overprovisioning is nominal. Again, Figure 8(b) demonstrates that the variation across traces exists; and

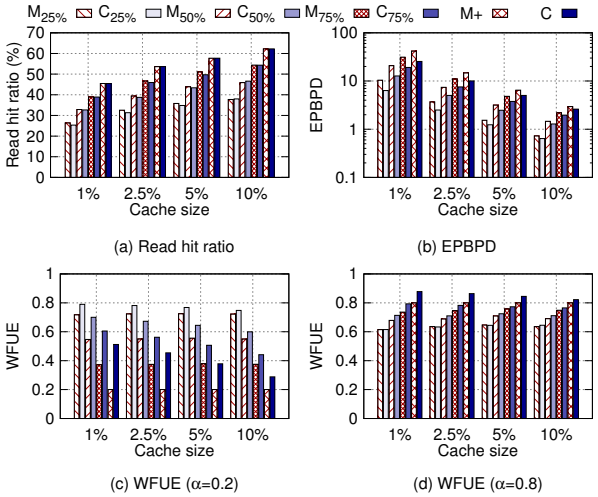


Figure 9: Trade-offs when limiting erasures: WFUE as a function of algorithm, EPBPD quota, cache size, and α weights.

it is just a matter of how many more erasures each trace incurs, compared to the best case.

6.3 Throttling Flash Writes

Thus far the evaluation has focused on **O1**, optimizing first for RHR and second for EPBPD. If erasures are more important, we can limit flash writes to reduce EPBPD at some cost to hit rate. Recall that **O2** tries to maximize RHR subject to a specific limit, whereas **O3** tries to optimize WFUE given a particular weight of the importance of RHR relative to EPBPD.

Figure 9 demonstrates the effect of C_T, which uses the admission control logic described in §3.5 to meet a specific EPBPD limit. It removes insertions with the least impact (i.e., blocks with least number of read hits) on RHR to meet the endurance goal. Figure 9 shows the results averaged across all traces when varying the EPBPD quota for a trace from 25%–75% of the EPBPD necessary for M or C respectively; this represents a reasonable range of user requirements on flash lifespan in real-world cases. For each cache size there are eight bars, with pairs of M_T and C_T algorithms as the threshold varies from 25% to 100% of total erasures. (The limits for M_T and C_T are set differently, since the maximum values vary.)

We observe in Figure 9(a) that for big cache sizes (5% and 10%) the RHR loss is about 39% for the 25% EPBPD quota. The gap reduces to 13% for the 75% EPBPD quota. As expected and shown in Figure 9(b), overall EPBPD decreases as the threshold is lowered, while C_T moderately improves upon M_T.

For WFUE, one question is what an appropriate weight α would be. Figures 9(c) and (d) plot the same algorithms but report WFUE using $\alpha = 0.2$ and $\alpha = 0.8$ (this prioritizes erasures and hit rates respectively). With $\alpha = 0.2$, the erasure savings dominate. Hence, $M_{25\%}$ and $C_{25\%}$ achieve the highest WFUE scores while M^+ and C see lower ones. Prioritizing RHR gives M^+ and C the highest WFUE across all variants. C_T consistently outperforms the corresponding M_T because it can avoid most wasted CFs and because it groups by eviction timestamp (see §6.1).

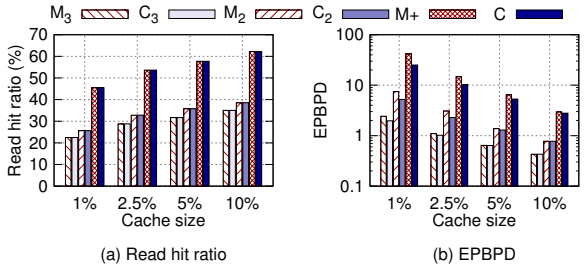


Figure 10: Trade-offs in read hit based insertion removals.

Figure 10 shows the effect of limiting flash insertions to blocks that are reread $\geq N$ times. It plots M^+ (which is equivalent to M_1), M_2, and M_3, as well as the corresponding container-optimized C algorithms. Results for RHR and EPBPD are averaged across all 34 traces. For small cache sizes (1% and 2.5%), C_2 loses an average of 41% of the RHR (Figure 10(a)), but it gets about a 79% savings in EPBPD (Figure 10(b)). Prioritizing erasures and RHR shows similar trends as the WFUE results in read hit based insertion removal tests (Figures 9(c) and (d)), hence are omitted due to space constraints.

7 Conclusion and Future Work

While it is challenging to optimize for both RHR and endurance (represented by EPBPD) simultaneously, we have presented a set of techniques to improve endurance while keeping the best possible RHR, or conversely, trade off RHR to limit the impact on endurance. In particular, our container-optimized heuristic can maintain the maximal RHR while reducing flash writes caused by garbage collection; we see improvements of 55%–67% over MIN and 6%–40% over the improved M^+, which avoids many wasted writes and uses TRIM to reduce GC overheads. Another important finding in our study indicates that simple techniques such as R_1 and TRIM provide most of the benefit in minimizing erasures. Alternatively, the flash writes can be limited to those that are rereferenced a minimum number of times. We define a new metric, Weighted Flash Usage Effectiveness, which uses the offline best case as a baseline to evaluate tradeoffs between RHR and EPBPD quantitatively.

In the future, we would like to investigate the complexity of the various algorithms (we believe them to be NP-hard). Exploring approaches to improving online flash caching algorithms is also part of our future work. We are particularly interested in heuristics to trade off one cache hit against another to further reduce cache writes without impacting RHR.

Acknowledgments

We are grateful to our shepherd, Dan Tsafrir, as well as the anonymous reviewers, for their valuable comments and suggestions that helped improve the paper. We would also like to thank Dan Arnon, Cheng Li, Stephen Manley, Darren Sawyer, and Kevin Xu for their general feedback, and Prof. Sanjeev Arora from Princeton University, and his students, Holden Lee, Fermi Ma, Karen Singh, and Cyril Zhang, for discussions on algorithm complexities. This work was supported in part by NSF award CNS-1149232.

References

[1] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. Page replacement for general caching problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, 1999.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[3] Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash memory algorithms. *ACM Trans. Algorithms*, 7(2):1–37, March 2011.

[4] M. Brehob, S. Wagner, E. Torng, and R. Enbody. Optimal replacement is np-hard for nonstandard caches. *Computers, IEEE Transactions on*, 53(1):73–76, Jan 2004.

[5] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*, 2005.

[6] Marek Chrobak, Gerhard J. Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard—even in the fault model. *Algorithmica*, 63(4):781–794, 2012.

[7] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, September 2010.

[8] Facebook Flashcache. https://github.com/facebook/flashcache.

[9] Ziqi Fan, David HC Du, and Doug Voigt. H-arc: A non-volatile memory based cache policy for solid state drives. In *Proceedings of the 30th Mass Storage Systems and Technologies Symposium (MSST'14)*. IEEE, 2014.

[10] Binny S. Gill. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, February 2008.

[11] Chris Gniady, Ali R. Butt, and Y. Charlie Hu. Program-counter-based pattern classification in buffer caching. In *Proceedings of the 6th USENIX Conference on Symposium on Operating Systems Design & Implementation (OSDI'04)*, 2004.

[12] GNU Parallel. http://www.gnu.org/software/parallel/.

[13] David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer. Flash caching on the storage client. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*, 2013.

[14] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *J. ACM*, 13(1):43–61, January 1966.

[15] Sai Huang, Qingsong Wei, Jianxi Chen, Cheng Chen, and Dan Feng. Improving flash-based disk cache with lazy adaptive replacement. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*, May 2013.

[16] Jffs2: The journalling flash file system, version 2. https://sourceware.org/jffs2/.

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

[18] S. Kavalanekar, B. Worthington, Qi Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'08)*, Sept 2008.

[19] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, February 2013.

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

[21] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *Proceedings of the 14th USENIX Confer-*

ence on File and Storage Technologies (FAST'16), February 2016.

[22] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*, June 2014.

[23] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th International Middleware Conference (Middleware'15)*, December 2015.

[24] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. Nvmkv: A scalable and lightweight flash aware key-value store. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*, June 2014.

[25] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[26] Lyle A. McGeoch and Daniel D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1-6):816–825, 1991.

[27] Micron MLC SSD Specification. http://www.micron.com/products/nand-flash, 2013.

[28] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST'08)*, February 2008.

[29] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, February 2012.

[30] Yongseok Oh, Eunjae Lee, Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Enabling cost-effective flash based caching with an array of commodity ssds. In *Proceedings of the 16th Annual Middleware Conference (Middleware'15)*, December 2015.

[31] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, 2014.

[32] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable writeback for client-side flash caches. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*, June 2014.

[33] Samsung Server SSD Specification. http://www.samsung.com/serverssd/, 2015.

[34] SanDisk SATA Solid State Drives. http://www.sandisk.com/enterprise/sata-ssd/, 2015.

[35] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu. To ARC or not to ARC. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*, July 2015.

[36] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, 2012.

[37] Hyong Shim, Philip Shilane, and Windsor Hsu. Characterization of incremental data changes for efficient data protection. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*, 2013.

[38] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. Ripq: Advanced photo caching on flash for facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, February 2015.

[39] Olivier Temam. Investigating optimal local memory performance. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. ACM, 1998.

[40] TRIM Specification. ATA/ATAPI Command Set- 2 (ACS-2). http://www.t13.org/.

[41] Yaffs (Yet Another Flash File System). http://www.yaffs.net/.

[42] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. Hec: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*. ACM, 2013.

[43] Yifeng Zhu and Hong Jiang. Race: A robust adaptive caching strategy for buffer cache. *IEEE Trans. Comput.*, 57(1):25–40, January 2008.