Pannier: A Container-based Flash Cache for Compound Objects

Cheng Li
Rutgers University
chenglii@cs.rutgers.edu

Philip Shilane
EMC Corporation
philip.shilane@emc.com

Fred Douglis
EMC Corporation
fred.douglis@emc.com

Grant Wallace
EMC Corporation
grant.wallace@emc.com

ABSTRACT

Classic caching algorithms leverage recency, access count, and/or other properties of cached blocks at per-block granularity. However, for media such as flash which have performance and wear penalties for small overwrites, implementing cache policies at a larger granularity is beneficial. Recent research has focused on buffering small blocks and writing in large granularities, called containers, but it has not explored the ramifications and best strategies for caching compound blocks consisting of logically distinct, but physically co-located, blocks. Containers may have highly diverse blocks, with mixtures of frequently accessed, infrequently accessed, and invalidated blocks.

We propose and evaluate Pannier, a flash cache middleware that provides high performance while extending flash lifespan. Pannier uses three main techniques: (1) leveraging block access counts to manage cache containers, (2) incorporating block liveness as a property to improve flash cache space efficiency, and (3) designing a multi-step feedback controller to ensure a flash cache does not wear out in its lifespan while maintaining performance. Our evaluation shows that Pannier improves flash cache performance and extends lifespan beyond previous per-block and container-aware caching policies. More fundamentally, our investigation highlights the importance of creating new policies for caching compound blocks in flash.

Categories and Subject Descriptors

D.4.7 [**Software**]: Operating Systems—*Organization and Design*

General Terms

Experimentation, measurement, performance

Middleware'15 December 07–11, 2015, Vancouver, BC, Canada.
Copyright © 2015 ACM. ISBN 978-1-4503-3618-5/15/12 ... $15.00
http://dx.doi.org/10.1145/2814576.2814734.

Keywords

Flash cache, eviction algorithm, I/O throttling

1. INTRODUCTION

Flash caches using solid-state drives (SSDs) are typically layered in front of hard disk drives (HDDs) to achieve high system performance, but the hardware characteristics of flash are at odds with the assumptions of standard cache eviction algorithms. We focus on achieving high cache performance for a second level cache, which exists at a layer below a storage system's RAM cache and above the hard drive's internal caches. Increasing cache performance is of particular value to data-intensive computing. Flash is often used for caching in this environment since it provides significant performance gains over HDDs at a higher density than DRAM. A difficulty with using flash, though, is that state-of-the-art caching algorithms make caching decisions at per-block granularity [5, 10, 12, 18, 24, 27, 33, 38], which does not align to the erasure units of flash and causes internal copy-forward operations. As an example, cached blocks may be kilobytes in size, while throughput and erasure properties are optimized for flash writes that are megabytes [23, 35] that correspond to the flash erasure unit. Evicting individual blocks from flash results in internal operations to copy live blocks to new locations and flash erasures, which are the primary source of performance degradation and lifespan reduction [6, 22]. Note that we use the term *block* to refer to the unit of cache insertion, read, and management; which is not necessarily the same as the flash erasure unit, sometimes also referred to as a block in the literature.

To optimize write throughput and reduce flash erasures, small blocks are often buffered together to form a container [17, 23, 35]. Such designs overwrite entire containers so the flash translation layer (FTL) can avoid relocating individual blocks, which extends the flash's lifespan. Because the FTL does not have access to block properties such as dirty/clean status and access count, the cache's status is managed at a user level, though there has been research on moving cache functionality into the FTL [31].

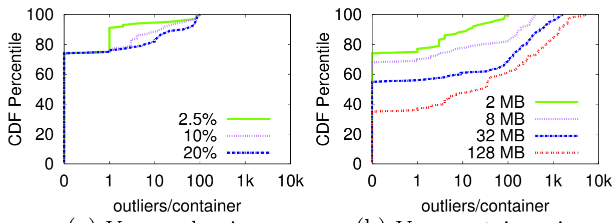
Fundamentally though, block-based caching and container-based caching are in conflict. Per-block tracking can maintain the highest accuracy for access patterns but also requires the most resources such as RAM and creates the most flash fragmentation. For example, MQ [38] maintains a per-block access counter for cached blocks and evicted blocks, which scales RAM with the number of cached blocks. On the other hand, container-based approaches track cache status at a coarse granularity to reduce resources [17], which makes it difficult to distinguish between the cache status of blocks within a container. While recent cache algorithms have combined per-block tracking information with large container writes, they have not fully considered the implications of compound block caching [23, 35].

A new issue arises for a container cache since blocks within a container may have highly varying access patterns and even vary in their liveness state. We refer to heterogeneous containers as *divergent containers*. Blocks in a container may exist in one of three states: hot, cold, or invalid. Hot and cold are intuitive terms indicating opposite regions on the access count spectrum, and a research challenge is to incorporate these concepts into a cache algorithm. In our analysis, we found blocks that have several orders of magnitude more accesses than other blocks within the same container. Besides access differences, a new issue arises as cache blocks become invalidated (*i.e.*, deleted or overwritten). Per-block caching would immediately mark the space for invalidated blocks as free. However, container caches on flash avoid small overwrites so containers become partially dead. Better understanding these new challenges allows us to fully explore the potential of container-based flash caching. We have specifically designed a flash cache middleware, called *Pannier*, to manage a container cache with divergent containers. Pannier serves as a second level cache (underneath the DRAM cache) for a storage system, which can be shared by multiple applications. Pannier uses a combination of block access history and invalidation to determine which containers to keep versus evict and whether to copy blocks from evicted containers and possibly regroup by access patterns.

The success of a flash cache policy depends on multiple, interacting goals: maximizing performance (high IOPS and low latency) while also controlling erasures. While some administrators wish to maximize performance and are willing to replace flash as necessary, most want consistent performance where the flash cache lifespan matches the overall system lifespan. To meet this goal, we present a multi-step feedback algorithm called *Throttle Insertions and Reinsertions for Erasures* (TIRE). When the flash writes are within a quota (per time period), client insertions and internal copy-forward operations (called reinsertions) are allowed. As the writes increase, we increase a threshold necessary for block insertions and reinsertions, where the value is based on its access count. To better quantify the value of data written to the cache, we present a new metric called *flash usage effectiveness* (FUE) that combines hit ratio and erasures into a single number for comparison purposes (§4.1).

In summary, our contributions are:

- We propose Pannier, a container-based flash cache that manages compound blocks, and we compare to multiple state-of-the-art caching policies.
- We manage divergent containers with diverse blocks, varying access characteristics and liveness status.
- We present TIRE, a multi-step feedback controller that preserves most of the hot data while ensuring the flash device does not wear out before its targeted lifespan.
- We experiment with a Pannier prototype to validate performance and lifespan improvements.



(a) Vary cache size (b) Vary container size

Figure 1: Impact of cache size and container size on outliers of block access count.

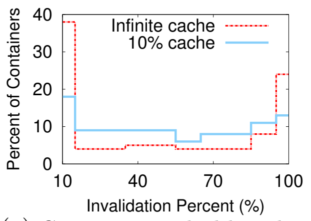
2. DIVERGENT CONTAINERS

In this section, we discuss the background and challenges of a flash cache design based on containers. We use LRU+, a simple container-based caching policy derived from Nitro [17], as an example to identify design issues. Nitro also supported deduplication and compression, which we disable to focus on the caching algorithm. LRU+ tracks the most recent access to a container (*e.g.*, the most recent read of any block in a container) and evicts the least recently used container. First, we show divergent containers exist, for example having a large variance of access counts among the contained blocks. Then we demonstrate that block invalidation is common and causes space inefficiency.

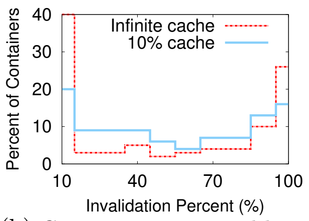
Block access pattern variability. In container caching, the granularity of a flash write is an entire container consisting of multiple blocks inserted by a client. We study the access pattern of blocks within a container and analyze the number of outlier blocks per container. For this analysis, we define outlier blocks as those that have access counts greater or equal to 5× the average (rounded up) accesses of blocks in the container. As an example, in a 2MB container with 512 4KB blocks, if 100 blocks each have 10 accesses and the remaining blocks have zero accesses, the rounded mean is 2, and the outlier threshold is 10. Then there are 100 outlier blocks in the container.

Figure 1 plots the cumulative distribution function for the outlier blocks per container for one of our experimental traces (Trace 3 in §4.2). The x-axis indicates the number of outlier blocks per container, and the y-axis shows the fraction of containers. We fix the container size to 2MB, and Figure 1(a) shows that about 30% of the containers, across all cache sizes, have at least one outlier block. Larger cache sizes show more outlier blocks for the containers since containers stay in the cache longer. One extremely divergent container had a block with 873 accesses as well as numerous blocks with either zero or one accesses. In Figure 1(b), we fix the cache size to 10% of the working set size and vary the container size from 2 to 128MB. As the container size increases, the block access counts become more variable because physically co-located blocks may have very different access counts in a large container, thus more outlier blocks are observed per container. For example, we observed one 128MB container had a block with 2,334 accesses while the vast majority of blocks had zero to two accesses. We observed outlier blocks across all of our traces.

This characterization indicates that evicting the entire container may cause performance loss due to access count variability in blocks. Our solution in Pannier is to give containers a time period within the cache for their access counts



(a) Containers packed based on locality



(b) Containers segregated by clean/dirty state

Figure 2: Characterization of container invalidation when using different container packing strategies.

to stabilize, and then we copy blocks that are accessed with similar counts to form new containers such that blocks in containers will generally age at similar rates.

Incorporating invalidation into a container cache. When a client overwrites a block, the modified block is written into a new container to avoid excessive FTL copy-forward. The block in the old container becomes *invalidated* in the sense that it is no longer the valid version to return.

To better understand the characteristics of invalidation, Figure 2 plots the distribution of container invalidation for trace 1 (§4.2). The horizontal axis shows the invalidation ratio for containers, and the vertical axis is the percentage of containers. We studied the impact of two common packing strategies when forming containers before writing to flash. We used the LRU+ caching scheme with an infinite cache size and a cache sized to 10% of the working set. Figure 2(a) shows the results when packing by logical address, meaning that logically near blocks are inserted into the same container. Figure 2(b) shows the results when packing blocks into either clean or dirty containers, where dirty blocks must be stored to HDD before eviction. In each experiment, we used two open containers to accentuate the impact of packing. For both packing strategies, results for the infinite cache show a mostly bimodal distribution with containers either fully invalidated (rightmost bar) or fully valid (leftmost bar), while the 10% cache shows a smoother variation. A simple container-based LRU+ replacement policy that only considers recency knowledge will keep a recently accessed container, even if it is mostly invalidated, leading to poor space utilization. For one cache configuration and trace combination, we observed 69% of all cached blocks were invalidated. Clearly, invalidation is an important property for a container cache algorithm regardless of the packing scheme.

3. PANNIER ARCHITECTURE

This section presents the design of Pannier with a focus on container-based flash caching. The goal of Pannier, as a second level storage cache, is to agnostically cache high-value content for the applications to maximize performance while maintaining the lifespan of the flash device. From the discussion in §2, Pannier also needs to select divergent containers with large access count variability and/or high invalidation to consolidate valid blocks to new containers.

Figure 3 shows the overall architecture, components, and examples of the insertion and lookup paths, which are discussed in greater detail in §3.1. Pannier must handle well-studied per-block caching concerns, such as segregating hot and cold data, along with container-specific issues that have

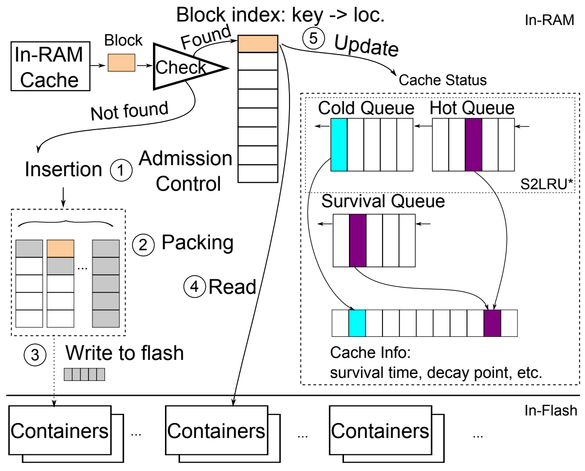


Figure 3: Container-based flash caching architecture. The dashed rectangle shows the cache status managed by a set of queues.

not been thoroughly investigated. Because of complex access and invalidation patterns, a container may have combinations of hot, cold, and invalid blocks. For these reasons, we combine a segmented LRU structure for hot/cold data segregation along with a new *survival queue* that identifies divergent containers.

Consider the following example of a client insertion and lookup. (1) When the client inserts data from a read-miss or for a new write, the admission control policy decides whether to insert the new cache block. (2) The packing policy determines how to assign the block to an in-RAM container. (3) Pannier seals the container when it is full and writes it to the flash. (4) On a lookup, Pannier checks the in-RAM indices for the flash location based on the block's key (*e.g.*, file handle and offset) and services the read from flash. (5) Due to the read hit, the cache-status module is updated, causing the corresponding container's position to change in the in-memory queues.

3.1 Pannier Components

In this section, we define the major cache components.

Block. A block is the generic term we use for the minimal unit of access to the cache. A client can read, insert, or invalidate a block.

Block index. An in-memory index maps from a block's key (*e.g.*, LBA) to a location in flash. For a client 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 a container and referenced from the index. When invalidating a block, the block's index entry is removed.

Container. We use 2MB as the default container size (§5.2 highlights the relevant factors). Each container has a header section to keep the metadata describing the blocks in that container so we can reconstruct the index efficiently at startup time. In-memory, open containers hold newly inserted blocks and can support in-place updates. Once a container is sealed and persisted in the flash cache, Pannier redirects overwrites to a new open container without updating the sealed container.

S2LRU+. Advanced caching designs [14, 18, 38] show that a segmented cache structure provides segregation of hot and cold data, thus protecting cache locality for a second level cache. We decided to use S2LRU [14] as a building basis for Pannier. S2LRU partitions the cache into two segments, one for the probationary and the other for the protected segments. In the standard S2LRU algorithm, new data are inserted to the most recently used (MRU) position in the probationary segment. On a hit, data are promoted from the probationary segment to the MRU position in the protected segment. Data hit in the protected segment are promoted to the MRU position of the same segment. When the protected segment exceeds its predefined size (*e.g.*, half of the cache size), the LRU data are migrated to the MRU position in the probationary segment. For clarity, we rename the protected segment and probationary segment the *hot queue* and *cold queue*, respectively.

An intuitive way to make S2LRU container-aware is to manage at the container granularity instead of block granularity, which we call *S2LRU+*. Specifically, S2LRU+ inserts a new container into the MRU position in the cold queue. Whenever there is a hit to a container, S2LRU+ promotes the container to the hot queue, and the migration from hot queue to cold queue is also at container granularity. Entries in the S2LRU+ queue are references to container information structures. A drawback of S2LRU+, though, is that it is not designed to handle divergent containers.

S2LRU*. We further modified S2LRU+ to create S2LRU* for Pannier, by modifying the insertion, promotion and reinsertion operations to better support divergent containers and the segregation of hot and cold blocks (§3.2).

Survival queue. In addition to the S2LRU* structure, Pannier uses a survival queue, a priority-based queue structure to identify divergent containers that may otherwise survive for a long time because of a small number of repeatedly accessed blocks. Once such containers are identified, hot blocks can be copied to form new containers to segregate hot/cold data, and the original container is freed. Pannier assigns a *decay point* and a *survival time* to each container to describe when to inspect a container to age access counts and how long it can stay in the cache. The survival queue is ranked using the survival time. The survival queue consists of pointers to container information structures shared with the S2LRU* structures. We discuss the operation of the survival queue in §3.2.

Pannier manages a wall clock counter in the cache which is incremented for each insertion operation, *i.e.*, caching a read-miss or a write. Because writes cause blocks in our immutable containers to be invalidated, the wall clock is incremented for every write. The decay point and survival time are assigned using the wall clock value.

Invalidation and access bitmaps. Within each container, Pannier tracks which blocks are valid or invalidated using an invalidation bitmap with a bit for each block. Similarly, we use an access bitmap to track if a block has been read since it was written to flash. If a block is invalidated, we clear the access bit for the corresponding block.

Ghost cache and access counter. Each block has an 8-bit access counter, and Pannier uses a ghost cache [12, 18, 38] to manage a counter for blocks in the cache and those recently evicted from the cache. Since the ghost cache only keeps the meta-data (key and access count), it can actually track more than the physical cache size, and there are techniques to reduce tracking overheads [8]. We set the ghost cache to hold 2× of the cache size. The ghost cache is used for insertion and reinsertion described in §3.2.

Conditions	Actions
1. $\Sigma E(t) \le 0$	no insertion, wait for next quantum
2. $\hat{E}(t) \le q$	cache everything
3. $\hat{E}(t) \in (q, K \cdot q]$	within quota slack: split $(q, K \cdot q]$ into three intervals, 1) cache read misses and reinsertions, 2-3) set acc. threshold to 1, 2 respec.
4. $\hat{E}(t) > K \cdot q$	beyond quota slack: no insertion, wait for next quantum

Table 1: TIRE is a multi-step feedback controller.

TIRE: Throttle Insertions and Reinsertions for Erasures. Typical MLC NAND flash devices have a usable lifespan of 2-10K erase cycles, and our admission control component ensures the flash cache does not wear out during its lifespan. Based on industrial specifications [29, 30], we set the flash erasure per block per day (EPBPD) quota to 8.3 and 5, representing 3 year and 5 year amortization periods, respectively.

The principle of the TIRE admission policy is to maintain the flash erasure quota. TIRE uses a credit-based approach to manage the quota. Specifically, we split the execution time into quanta (*e.g.*, 5-minute) as an accounting period, and we set the erasure quota q for that time period t. TIRE outputs an erasure delta ($\Delta E(t)$), which is the running erasure credit for the quanta that may be negative or positive. TIRE also calculates an erasure credit ($\Sigma E(t)$), which is the running credit of erasures allowed, which again may be negative or positive. We derive the current erasure credit from the erasure credit of the last quantum and the delta erasure ($\Sigma E(t) = \Sigma E(t-1) + \Delta E(t-1)$). If there are remaining erasures for one quantum, then the erasure credit is carried over to the next quantum. Within a quantum, we use a multi-step bang-bang controller [34] to decide with fine granularity what to put in the cache. For each quantum, TIRE computes the instantaneous erasures ($\hat{E}$) since the beginning of the quantum.

We next walk through the TIRE policy in greater detail, shown in Table 1, discussed from top to bottom. (1) When the erasure credit ($\Sigma E(t)$) is non-positive at the beginning of a quantum, TIRE rejects all insertions and reinsertions until the next quantum. When the erasure credit is positive, the remaining checks are considered. (2) If the instantaneous erasures are below the quota ($\hat{E}(t) \le q$), then all insertions (reads and writes) as well as reinsertions are allowed.

We next consider the case that the erasures increase beyond the quota. We define an access threshold as the minimum accesses needed for a block to be accepted into the cache. The design rationale of TIRE's policy is to gradually raise the access threshold of writing blocks into the cache. TIRE uses a slack factor K to control how many erasures are

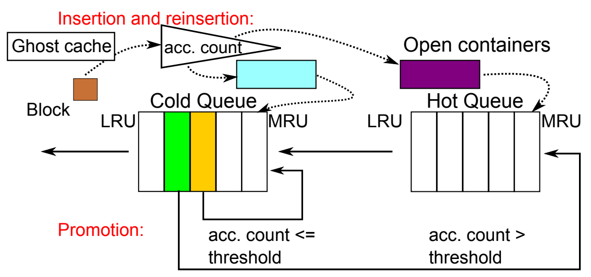


Figure 4: S2LRU*: insertion, promotion and reinsertion of containers in Pannier.

allowed beyond the quota within a time period. (3) When the instantaneous erasures are between q and $K \cdot q$, TIRE dynamically adjusts the throttling policy. Specifically, we partition the interval within $(q, K \cdot q]$ and increase the access threshold accordingly. We set $K = 4$ and split $(q, K \cdot q]$ equally into 3 intervals (discussed in §5.2). TIRE assigns the following three policies (1-3) to each interval: interval 1) accepts insertions for read misses and reinsertions but does not accept writes (new dirty data from a client); intervals 2-3) increase the access threshold to 1 and 2, respectively, for read misses and reinsertions. This means that as the erasures grow beyond q and up to $K \cdot q$, we increase the threshold for block accesses necessary before writing into the cache. We found increasing the access threshold can significantly reduce erasures with limited impact on hit ratio (see §5.2). (4) When erasures grow beyond the slack value in a time period ($\hat{E}(t) > K \cdot q$), no further insertions are allowed until the next quantum, which may start with a negative erasure credit because we allowed erasures beyond the quota in the current quantum.

Sampled distributions. Pannier leverages sampling techniques to track trends in the cache including access and reuse distributions. Pannier also samples a subset of containers to track access percentage. We found a small sampling rate (*e.g.*, 3%) provides sufficient accuracy, while minimizing memory overheads. These sampled distributions guide the eviction and copy-forward operations in §3.2.

3.2 Pannier Functions

Next, we describe Pannier's key functions and operations.

Promotion. We configured S2LRU* by setting a minimum number of accesses needed before a container is promoted to the hot queue (Figure 4), since a container may contain numerous blocks. This technique avoids cold containers polluting the cache, and we discuss its impact in §5.2.

Seal, eviction, invalidation and access. Figure 5 shows pseudocode of how Pannier handles events related to containers and blocks. We describe the events of sealing, eviction, invalidation and access to a block. Let A, I, and C represent the percentage of accessed, invalidated, and cold blocks in a container, respectively; then $A + I + C = 100\%$.

When an in-memory container is full, it is sealed and written to flash. The `OnSeal` function sets the decay point and survival time for a container. The decay point is the period until the access counters for blocks are decremented, similar to clock-based aging algorithms, so our access counters are *aged*. The intuition of survival time is to give a container sufficient time to allow servicing read hits and invalidating

```
void OnSeal(Container c):
  c.decay_point = c.seal_time + lifetime
  c.survival_time = c.seal_time + lifetime

Container OnEvict():
  Container c = Q.top() // Survival queue Q
  if c.decay_point < now():
    AgeContainer(c)
  if c.survival_time < now():
    Copy forward objects with frequency > 0
    Q.pop() and return c
  else:
    return LRU container from cold queue

void OnInvalidation(Container c):
  c.survival_time = now() + lifetime * (100% -c.I%)

void OnAccess(Container c, Object obj):
  Increment the frequency of obj
  bool refill = false
  if c.decay_point < now():
    AgeContainer(c)
    if c.A% > access_threshold:
      refill = true
  if c.A% increased or refill:
    c.survival_time = now() + lifetime * (100% -c.I%)

void AgeContainer(Container c):
  step = ceil((now() - decay_point)/lifetime))
  Decrement object frequency by step
  Recompute c.A%
  c.decay_point = now() + lifetime
```

Figure 5: Seal, aging, eviction and access handling for cached blocks and containers in Pannier.

overwritten blocks before analyzing the stable pattern for the container. The *lifetime* is determined as the 99th percentile of the reuse distance distribution for sampled blocks, representing a sufficient survival time in the cache.

When the cache is full and a container must be selected for eviction, `OnEvict` is called, which first checks if the container needs to be aged (described below). Then it checks if the survival time for the top container in the queue has expired. If so, Pannier copies blocks with non-zero accesses forward to new containers using the TIRE policy and the packing rules described in the reinsertion section. If the survival time of the container has not expired, then Pannier evicts the LRU container from the cold queue.

On block invalidation (`OnInvalidation`), the survival time of the container is reset to a fraction of the default lifetime, which is proportional to the valid blocks in the container. For example, if the container is 100% invalidated, then the survival time is set to the current time, thus it is evicted from the queue immediately. If the container state has not changed when the survival time expires, Pannier evicts this container and reclaims the space. This design leverages the bimodal pattern observed in §2 to set the appropriate survival time.

On a read access (`OnAccess`), Pannier increments the access count of the block. If the decay point of the container has expired, Pannier ages the container. If the percentage of the aged accessed blocks for a container is high then we extend the survival time of the container. Pannier uses the median value of a sampled distribution of container access percentage as the `access_threshold` value. Since the access count is decremented for aged containers, Pannier only copies hot blocks forward and consolidates them into new containers in the `OnEvict` operation. The advantage of ag-

ID	Trace	Description	Duration	WSS	Reads	R:W	Seq. (KB/IO)
1	MSR prn0	print server	7 day	61GB	14GB	1:1.2	24
2	MSR src11	source control server	7 day	157GB	780GB	3:1	51
3	MS map	hosts of a live map imagery	24 hr	703GB	1.9TB	3.2:1	40

Table 2: Properties of traces used in the performance experiments (WSS = working set size).

ing containers during `OnAccess` is that we apply the aging procedure to containers that are changing, whereas containers that are not accessed will move to the top of the survival queue and be evicted.

When aging a container (`AgeContainer`), we calculate how many lifetimes have passed since the last aging step, which is the amount to decrement from every block's access count. We then adjust the access bitmap, since block access counts may have reached zero. Finally, we adjust the decay point to allow another round of aging.

Reinsertion. When the cache is full and a container has been selected for eviction, Pannier copies accessed blocks forward to new containers according to the TIRE policy that monitors erasures. However, Figure 4 shows our technique for grouping blocks based on their access count. We use the sampled per-block access count distribution to set the threshold for selecting between open containers for the cold and hot queues. When sealed, a container is placed into the MRU position of either the cold or hot queues. We use four open containers for insertion and reinsertion by default. We categorize open containers into two for clean and two for dirty blocks and then pack blocks based on access count.

Restart and crash recovery. Pannier currently checkpoints cache status information and dirty blocks to flash at the container granularity based on either a client `SYNC` operation or every 30 seconds. Since our contribution focuses on container caching, we refer the reader to previous work [25, 31] for specific implementation details.

For restart and crash recovery, a journal tracks the dirty and invalid status of blocks. When recovering from a crash, Pannier reads the journal and the container headers from flash and recreates the indices and cache status information.

4. EXPERIMENTAL METHODOLOGY

This section describes the metrics, traces, configuration for Pannier, and experimental systems.

4.1 Metrics

Our results present overall system IOPS, including both reads and writes. Because writes are handled asynchronously and are protected by battery-backed DRAM in our prototype, we focus on the read-hit ratio and the read response time to validate Pannier. The principal metrics are:

Input/Output operations per second (IOPS): Client read and write operations per second.

Read-hit ratio (RHR): The ratio of read I/O requests satisfied by Pannier over total read requests. Pannier splits large requests into 4KB blocks. If all 4KB blocks are in the cache, the request is a hit. Otherwise it is a miss since the HDD I/O for missing blocks dominates the latency.

Read response time (RRT): The average elapsed time from the dispatch of one read request to when it finishes, characterizing the user-perceivable latency including HDD latency for cache misses.

Erasure per block per day (EPBPD): The flash erasures normalized against the number of flash blocks given a cache size and a period of time.

Flash usage effectiveness (FUE): We introduce the FUE metric as the number of bytes of flash reads divided by flash writes, including client writes and internal copy-forward writes. A score of 1 means that, on average, every byte written to flash is read once, so higher scores are better. FUE combines RHR and erasures into a single score to simplify the comparison of techniques, though a client may be willing to trade-off flash lifespan for greater performance, which is not captured by FUE.

4.2 Trace Description

Storage traces. We used a set of 69 traces from 3 repositories. The repositories are:

EMC-VMAX traces. 48 traces of EMC VMAX primary storage servers that span at least 24 hours and have at least 1GB of both reads and writes [32].

MS Production Server traces. All nine storage traces from a diverse set of Microsoft Corporation production servers captured using event tracing for windows instrumentation, with at least 700k accesses [15].

MSR-Cambridge traces. 12 block level traces lasting for 168 hours on 13 servers, representing a typical enterprise data center [21]. We narrowed the available traces to 12 to include appropriate traces for cache studies. The properties include a working set size greater than 25GB, $\geq$5% of capacity accessed, and read/write balance ($\leq$45% writes)[1].

For performance experiments, we selected three traces from the public dataset so that interested readers can repeat our experiments. Note that the cache size for each workload is 10% of the working set size for each dataset unless otherwise stated. Table 2 lists the trace characteristics to study the behavior of Pannier with different workloads. MSR `prn0` is collected from one print server, which is characterized by small I/Os and a relatively small working set size. MSR `src11` is collected from a source control server with a high overwrite ratio. MS `map` hosts map images and has a large working set size.

Synthetic traces. We created a set of synthetic traces to study the impact of access count variability and invalidation on divergent containers. We used ProWGen [4] to generate a set of read-only web traces with a range of object sizes. The working set size of these objects is 85GB. We simulate a busy web server with 67% utilization of the available throughput [11, 20]. We controlled the object access count by adjusting a Zipf distribution, with values between 0.5 and 1 as in previous work [3, 4, 11]. We set the median object size to be 60KB with a standard deviation of 8MB based on previous studies of static web content [11, 20]. We further vary the invalidation in a controlled manner to study the interplay of invalidation and access patterns.

[1] The trace names are: `hm0`, `prn0`, `prn1`, `proj0`, `proj1`, `proj4`, `src10`, `src11`, `src12`, `usr1`, `usr2`, `web2`.

Variable	Values
Container size (MB)	**2**, 4, 8, 16, 32, 64, 128
DRAM cache size (%)	1, 2.5, **5**, 10, 20
Flash cache size (%)	1, 2.5, 5, **10**, 20
Ghost cache on (re)insertion	**on**, off
Open (re)insertion containers	1, 2, **4**, 8
Packing policy	LBA, dirty/clean, **acc.**
Promotion threshold (%)	0 (off), 1, 2, **5**, 12
Write-mode	through, **back**
Admission control policy	**off**, static, TIRE
Intervals in TIRE	1, 2, **3**, 4, 5
Slack factor	1.1, 1.5, 2, **4**, 8, 16
TIRE quantum (min)	1, 2, **5**, 10, 15

Table 3: Pannier parameters with default values in bold.

Policy	Abbr.	Description	I	C	F
LRU	L	least recently used			
S2LRU	L_2	two segments LRU			
MQ	M	based on frequency hint			
Belady	B	know future access			x
LRU+	L^+	container-based LRU		x	
S2LRU+	L_2^+	container-based S2LRU		x	
RIPQ+	R^+	RIPQ with overwrites		x	
Pannier	P	our design	x	x	

Table 4: Cache policies, invalidation-aware (I), container-aware (C), or future knowledge (F). We segregate per-block policies from container-aware policies. The abbreviations are used in figures in §5.1.

4.3 Parameter Space

Table 3 lists the configurations for Pannier, with default values in bold. Due to space limitations, we interleave parameter discussion with experiments in §5.

Table 4 shows the caching schemes selected to represent coverage of past and present work including block-based caching such as LRU and container-based caching such as LRU+. We use *I* and *C* to indicate whether the scheme is invalidation-aware or container-aware. The configurations for previous work are the default in their papers (*e.g.*, number of queues). The Belady optimal replacement algorithm assumes that future accesses are known in advance [2], and it replaces the blocks in the cache with the most distant accesses. If the new caching block's next access is the furthest, then it bypasses inserting the block into the cache. For all container-based caching policies, we set the container size to be the same as Pannier.

RIPQ+. RIPQ [35] is a container-based flash caching framework to approximate several caching algorithms that use queue structures. One of the key techniques in RIPQ is that as a block in a container is accessed, its ID is copied into a virtual container in RAM. When RIPQ selects a container for eviction, any blocks that are referenced by virtual containers are copied forward. We added an overwrite operation to RIPQ (not previously supported), and we refer to the resulting system as RIPQ+. For an overwrite, RIPQ+ inserts the modified block into the corresponding open container and updates the index structures to reference the new version. When evicting the previous container, RIPQ+ determines the invalidated block is no longer referenced and can be removed. RIPQ+ supports invalidation operations but does not explicitly incorporate invalidation in the eviction algorithm, so we did not mark RIPQ+ as invalidation-aware.

4.4 Systems

For our experiments, we built a full-system simulator that allows us to measure metrics such as hit-ratio and flash erasures based on the Micron MLC flash specification [19]. We vary the size of each plane or flash chip to control the SSD capacity. We over-provision the SSD capacity by 7% for FTL garbage collection in the per-block flash cache experiments. No FTL space reservation is used for the container-based experiments [17]. We set the flash block size to 2MB and discuss its impact along with sizing containers in §5.2.

We place a small DRAM cache (5% of the flash cache size) with an LRU policy in front of the flash cache to represent a use case where the flash cache is used as a second-level cache. Four in-RAM containers are used for newly inserted blocks, and four in-RAM containers are used for re-insertions.

Our prototype, used for performance experiments, is a primary storage server equipped with four 1.6GHz Xeon CPUs and 8GB DRAM with battery protection. There are 11 1TB 7200RPM disk drives in a RAID-5 configuration.

We use a Samsung 256GB SSD, though we constrain our experiments to use a fraction of the available SSD, as controlled by the cache capacity experimental parameter. Before each experiment, we overwrite the flash device. According to specifications, the SSD supports >100K random read IOPS and >90K random write IOPS. Using a SATA-2 controller, we measured 4KB SSD random reads at 37.8K/s.

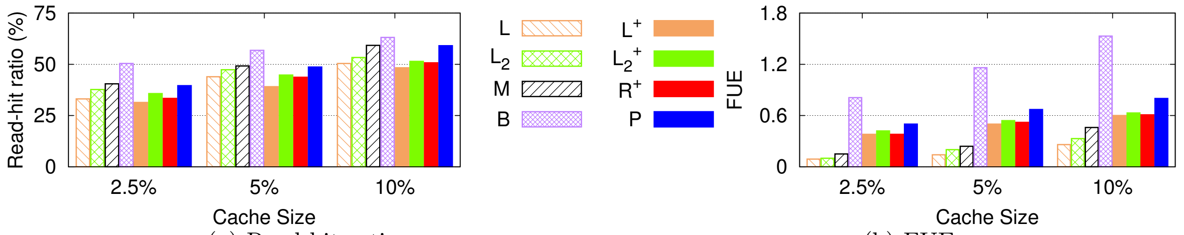
5. EVALUATION

We now evaluate Pannier's efficacy at improving flash cache performance and lifespan. We first compare hit-ratios and flash erasures across cache algorithms. Second, we study the impact and tradeoffs of each major component of Pannier. Third, we evaluate the performance of our prototype primary storage system. Fourth, we study an erasure efficient version of Pannier with the TIRE algorithm. Finally, we present sensitivity and overhead analysis.

5.1 Comparing Caching Algorithms

We first compare block-based and container-based caching algorithms to Pannier using a simulator with a focus on read-hit ratio and FUE (bytes read / bytes written).

Hit-ratio improvement. Figure 6(a) shows the read-hit ratio results across different caching schemes. We report the average hit-ratio across all 69 traces, and we vary the cache size between 2.5%, 5%, and 10% of the working set size for each trace. The standard deviation of read-hit ratios is $\leq$1%. We classify the results into a per-block group in the first 4 bars and container-based group in the last 4 bars. For space reasons, we abbreviate each technique as shown in Table 4. The per-block group includes LRU, S2LRU, MQ and Belady. The results show that S2LRU and MQ improve hit-ratio over LRU because they leverage additional hot/cold and access count information at a per-block level. Compared to LRU, MQ improves the read-hit ratio up to



(a) Read-hit ratio (b) FUE

Figure 6: Read-hit ratio and FUE for various cache policies and sizes.

7.3% on average. Belady shows the highest read-hit ratio by leveraging future knowledge.

The container-based group has LRU+, S2LRU+, RIPQ+ and Pannier, which generally have lower hit-ratios than per-block schemes due to coarser tracking. LRU+ does not have a separate segment to protect frequently accessed containers and is not invalidation-aware, which leads to the lowest read-hit ratio in this group. S2LRU+ and RIPQ+ are also not invalidation-aware, but they do segregate frequently accessed containers, so they have similar read-hit ratios. Pannier directly manages invalidation and variability in containers, which achieves up to 16% read-hit ratio improvement (with an average of 9.1%) when compared with LRU+, and it is competitive with per-block schemes such as MQ. There is still large gap from MQ to Belady. As will be shown later, Pannier also significantly improves flash lifespan compared to per-block schemes.

FUE results. Next, we analyze results from the perspective of flash usage effectiveness. Similar to the previous figure, Figure 6(b) plots the average FUE for all caching schemes across 69 traces while varying the cache size on the x-axis.

Per-block caching schemes such as LRU, S2LRU and MQ achieve low FUE scores (≤ 0.46). This means, on average, less than half of the bytes written (including internal flash writes) are read. Per-block caching schemes are unaware of flash properties and evict individual blocks, which internally causes FTL copy-forward and excessive flash erasures. We observed 2.8× more erasures on average for per-block schemes compared to container-based schemes. Although S2LRU and MQ achieve higher hit-ratios compared to container-based schemes, their FUE scores are lower because high erasure numbers negate the benefit of hit-ratio improvement.

Container-based schemes show consistently higher FUE scores than the per-block schemes. For S2LRU+ versus S2LRU, FUE improves $\geq$1.9× on average because eviction at container granularity reduces fragmentation in flash garbage collection. This effect is more pronounced with smaller cache sizes because smaller caches experience higher write amplification. RIPQ+ ignores invalidation, which leads to under-utilized flash space, which impacts FUE. Pannier has a higher FUE than all other block-based caching schemes and all per-block schemes except Belady's. Compared to LRU+, Pannier handles both invalidation and divergent containers, thus achieves a $\sim$34% improvement in the FUE metric.

Belady's algorithm shows the highest read-hit ratio and

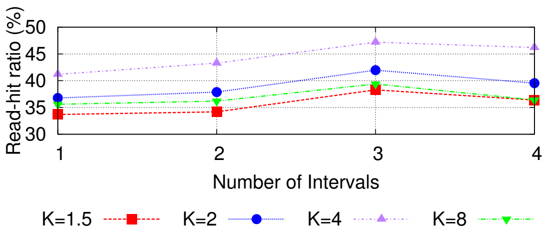


Figure 7: Tradeoff in TIRE controller. Y-axis start at 30%. The standard deviation is $\leq$1.5%.

FUE across all traces. Using future knowledge, Belady can decide not to insert blocks that would be evicted before being accessed. This perfect admission control results in fewer flash erasures and higher FUE than other algorithms.

5.2 Pannier Features and Tradeoffs

To isolate the contributions of the different features of Pannier, we first discuss the tradeoffs in the TIRE controller. Then, we study the impact of Pannier's invalidation-aware eviction scheme and container aging. Next, we study the impact of a ghost cache on container packing, insertion, reinsertion and promotion threshold. Finally, we discuss the impact of container size.

Tradeoffs in TIRE. We study four parameters in Pannier's TIRE controller: the slack factor K (short-term allowance beyond the quota), the number of intervals, access count thresholds for insertion/reinsertion and the feedback quantum. Figure 7 plots the read-hit ratio for the TIRE controller with S2LRU*. We vary the number of intervals between 1 and 4, and we vary the slack factor between 1.1 to 16, though we only plot 1.5 to 8 for space reasons. The cache is set to 10% of the working set size, and the EPBPD goal is set to 5. Other cache sizes show consistent results.

The read-hit ratio is maximized for TIRE with three intervals and a slack value of 4. This result shows that a cache may temporarily need to use more erasures than a stricter quota would allow ($K = 4$ versus $K = 1.5$). Increasing the threshold for insertion/reinsertion (3 intervals versus 1) also helps prevent writes for less useful blocks. By comparison, we turned off TIRE and found that 22 out of the 69 traces had more erasures than the limit of 5, though the read-hit ratio was higher than for TIRE (51.4% versus 47.9%). Next, we investigated the access count threshold and found that TIRE's dynamic approach has a higher hit ratio than a static value of 2 (47.9% versus 45.4%). Though not plotted,

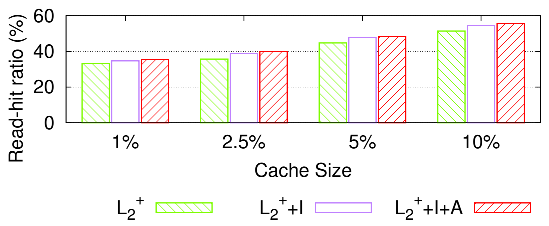


Figure 8: Impact of including invalidation (I) and aging (A) for S2LRU*. The standard deviation is ≤0.5%.

we also investigated the quantum granularity from 1 to 15 minutes, and found the highest read-hit ratio at 5 minutes, though the differences were small at other quantum values.

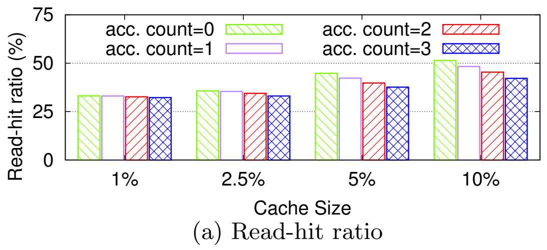
Impact of eviction and container aging. Figure 8 shows the impact of eviction and aging in Pannier. We start with plain S2LRU+ and add invalidation-awareness and aging, which constitute S2LRU* ($L_2^+ + I + A$). Across cache sizes, adding invalidation-awareness and aging increases the read-hit ratio. With handling of invalidation, heavily invalidated containers are quickly evicted without traversing the hot and cold queues, which leads to a ∼3.2% read-hit ratio improvement. However, the invalidation-aware scheme does not have an impact on FUE. Adding aging into Pannier increases the read-hit ratio (1.2%) and FUE (0.15) slightly. Each technique adds a small, but cumulative, improvement to read-hit ratio.

Impact of ghost cache and promotion threshold. Due to space constraints, we summarize the impact of a ghost cache for insertion, reinsertion and packing blocks into containers. We also show the impact of a promotion threshold.

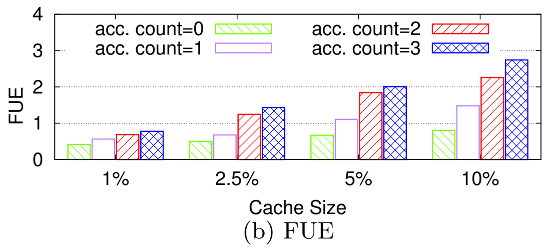
First, we found that leveraging a ghost cache with historical access counts of recent blocks is an effective technique to improve FUE through admission control. The policy is to only insert blocks into the cache that have more than a threshold of accesses. Figure 9 plots the read-hit ratio and FUE when varying the access count threshold between [0, 3]. An access count threshold of 0 indicates that admission control is disabled. The results show that for small cache sizes (*e.g.*, 1% and 2.5%), the read-hit ratio loss is low (≤2%) when increasing the insertion threshold. The impact is more pronounced for large cache sizes. We observed 2-3× improvement in FUE when increasing the threshold, demonstrating that this admission control policy significantly reduces flash erasures and increases FUE with a slight decrease in read-hit ratio. We incorporate this property in our insertion/reinsertion policy to limit erasures when a quota is enforced (§5.4).

Second, we use the ghost cache to segregate frequently accessed blocks into containers for either the hot or cold queue during the original insertion into the cache as well as reinsertion during copy-forward. We refer to this as a packing decision. Using four open containers for insertion and reinsertion, we found that either insertion or reinsertion leveraging the ghost cache separately improves the hit-ratio 1-1.5%, and the combination increases the hit-ratio ∼3%. FUE scores also increased marginally with the ghost cache. Overall, the ghost cache has a modest impact for packing containers, but a much larger impact on admission control.

Finally, we study the impact of including a promotion threshold. We use S2LRU* as a baseline system and vary the promotion threshold as the percentage of blocks in a container. For example, in a container with 512 blocks, the promotion threshold of 1% indicates 5 accesses are needed to promote the container. We found that setting the container promotion threshold to 5% achieves a consistent read-hit ratio increase of ∼2%, with the FUE score increasing slightly (≤ 0.1). A higher threshold causes the read-hit ratio and FUE score to drop. For Pannier, we set the promotion threshold to 5% as the default value.



(a) Read-hit ratio



(b) FUE

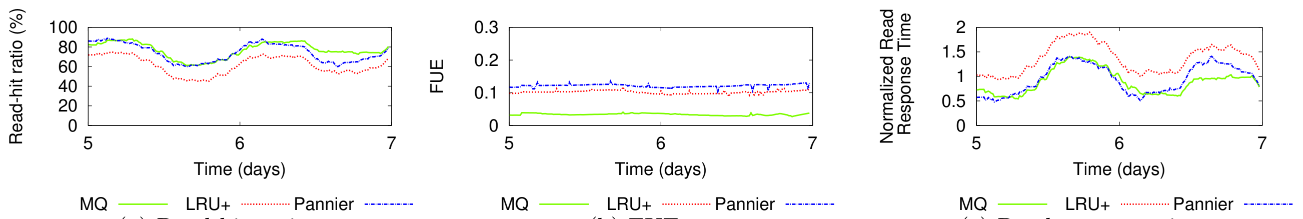
Figure 9: Tradeoffs in access-count based admission control for S2LRU+. The standard deviation is ≤0.5%.

Impact of container size. We analyze the impact of container size in terms of performance, resource overhead and flash erasures. We use a multi-threaded benchmark with a mix of random 4KB reads and random container writes of varying sizes as the typical I/O pattern for Pannier. As we vary the write size, we found that when the size increases from 2MB to 32MB, the write throughput increases from 28 to 32 MBps. Further increasing the write I/O size slowly increases the throughput but also significantly increases the read tail latency. For example, the 99^{th} percentile read response time is ∼1.6s when using a 1GB I/O size for writes. Escalation of tail latencies in the lower-level can have unexpected impacts on upper-level applications [7].

A large container size decreases DRAM requirements for maintaining cache status in the queue, but there are drawbacks. In contrast, smaller containers use less memory for open in-memory containers that are being packed. Importantly, flash erasures can be minimized when the cache eviction size is aligned with flash block size since this avoids incorrect copy-forward decisions by the FTL. For these reasons we set 2MB as the default container size for Pannier to align with the flash block size. For flash devices that have differing characteristics than what we have studied, a different container size may be more appropriate.

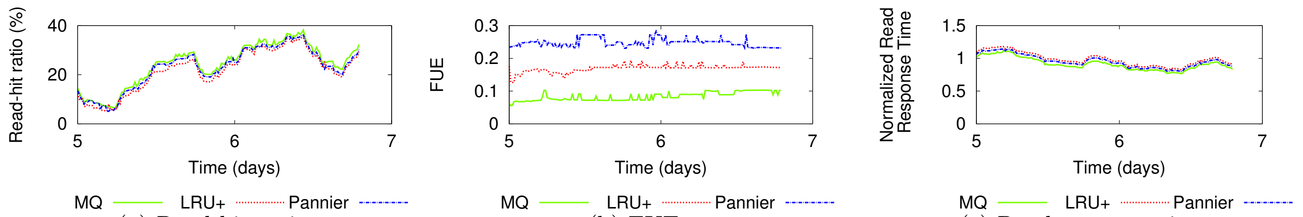
5.3 Prototype System Results

Next, we report the performance of Pannier without the TIRE controller (*i.e.*, without throttling SSD writes). We replayed the three selected traces at an accelerated speed to achieve ∼80% of the system saturation throughput, rep-



(a) Read-hit ratio (b) FUE (c) Read response time

Figure 10: Performance results for MSR `prn0`. The standard deviations are 2%, 0.04, and 0.1 for read-hit ratio, FUE and normalized response time, respectively.



(a) Read-hit ratio (b) FUE (c) Read response time

Figure 11: Performance results for MSR `src11`. The standard deviations are 1.5%, 0.03, and 0.07 for read-hit ratio, FUE and normalized response time, respectively.

resenting a sustainable high load that Pannier can handle. We created a warm cache scenario where we use the first two thirds of the trace to warm the cache and then measure the performance for the remaining trace. We primarily compare with MQ and LRU+ because they represent the best per-block scheme (MQ) and the container-aware baseline scheme (LRU+). We normalize the performance against LRU because it represents a typical per-block baseline scheme.

Improved read response time. We first show how a high read-hit ratio in Pannier prototype translates to an overall performance boost with the MSR `prn0` trace. Figure 10 presents the read-hit ratio, FUE and read response time results of MQ, LRU+ and Pannier. We size the cache to be 10% of the working set size, and present results for the last two days of the trace. Figure 10(a) shows that Pannier achieves a 15% read-hit ratio increase compared to LRU+. MQ shows the highest read-hit ratio with an increase of 17%. Figure 10(b) shows that Pannier and LRU+ are consistently 2× better than MQ in FUE, which shows the value of container-granularity writes. Figure 10(c) shows the read response time of MQ, LRU+ and Pannier normalized against LRU. Pannier shows a 16% read response time reduction and 29% IOPS increase compared to LRU, and MQ has an even lower response time in the final day.

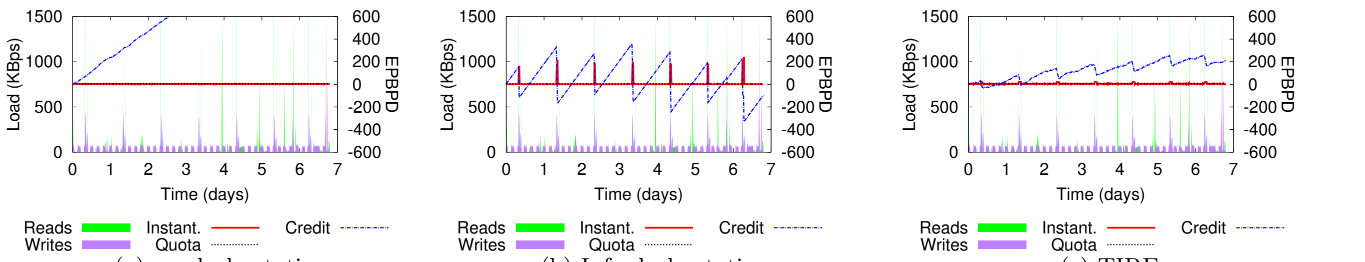
Improved IOPS. We examine the performance improvement for the MSR `src11` trace to see how Pannier behaves when the read-hit ratio improvement is small. Figure 11 presents the read-hit ratio, FUE and read response time results of MQ, LRU+ and Pannier for the `src11` trace. Figure 11(a) shows that the hit-ratio of all three techniques are similar. However, the FUE of Pannier improves ∼70% because of the high overwrite percentage in `src11`. Given similar read-hit ratios, higher FUE indicates fewer flash writes and flash queueing contention, thus, better IOPS. The read response time shows a 4% improvement over LRU, and the normalized IOPS shows a 12% improvement.

Trace	*Policy*	*RHR*	*FUE*	*EPBPD*	*Perf.(%)*	
					IOPS	RRT
`prn0`	L	67.6	0.01	13.7	100	100
	M	80.2	0.02	12.0	118	83
	L^+	63.5	0.18	4.14	102	107
	P	78.8	0.22	4.30	129	84
`src11`	L	23.0	0.07	33.7	100	100
	M	24.4	0.08	32.4	106	92
	L^+	22.8	0.13	12.38	104	106
	P	23.6	0.22	13.4	112	96
`map`	L	25.6	0.16	40.2	100	100
	M	34.6	0.17	39.0	110	85
	L^+	20.5	0.4	14.3	108	103
	P	29.3	0.45	18.9	119	89

Table 5: Performance evaluation across caching policies. The IOPS and RRT reduction percentage are relative to the average performance with LRU policy. The standard deviation for performance is ≤5.4%.

Performance result summary. Table 5 shows the results for the traces studied in our performance experiments. We observed Pannier achieves consistent performance improvement across all traces and found that the read response time directly relates to read-hit ratio. For the same read-hit ratio (the `src11` trace), a higher FUE score means fewer flash writes, thus less contention and improved IOPS. As the load intensity changes, the tradeoffs of Pannier can increase or decrease. For example, as the load intensity decreases, the IOPS improvement of Pannier is less pronounced, though the EPBPD is still ∼3× lower than the per-block scheme. A higher load intensity causes more contention in Pannier, where a modest FUE improvement alleviates some of the queueing effect, thus better IOPS. In addition, a larger flash cache will improve read-hit ratio and read response time, but the gap between each of the caching policies is smaller too.

Since the typical use case of a flash cache is behind an



(a) no slack, static (b) Inf. slack, static (c) TIRE

Figure 12: Impact of credit-based static throttling, infinite slack and TIRE on EPBPD for MSR src11 trace. We use the same scale in all subfigures for better visualization.

upper-level DRAM cache, we also experimented with different DRAM sizes. A bigger DRAM cache hides reference streams with low reuse distance from a flash cache, thus slightly increasing FUE and IOPS. The read-hit ratio and read response time achieves ≤3% improvement because the underlying (larger) flash cache is the determining factor for hits. Pannier users can tune the DRAM cache size based on the availability of resources and workload characteristics.

5.4 Pannier with TIRE

So far, we have studied the performance of Pannier for a performance-oriented customer. Next we study the performance of Pannier with erasure constraints and quantify the impact of the TIRE controller on flash performance and erasures. Figure 12 shows the advantage of Pannier's controller with an EPBPD quota of 5 for the MSR `src11` trace. We plot the trace load and the instantaneous, quota and credit EPBPD. For comparison, we use two policies: credit-based zero slack and infinite slack. For the zero slack policy, if the EPBPD credit is zero at any point in the quantum, then there is no admission for the next quantum until the EPBPD quota is positive. Another variant of this policy is infinite slack that accepts any new caching blocks into the cache and only computes the EPBPD credit at the end of the quantum.

The no slack, static policy (Figure 12(a)) strictly keeps the EPBPD target and generates a large EPBPD credit of 1612, but the read-hit ratio is only 13.8% with a FUE of 1.9. The infinite slack, static policy (Figure 12(b)) leads to a read-hit ratio of 18.3% and FUE of 0.81 with a negative EPBPD credit of -96, meaning the erasure goal was not achieved. Figure 12(c) shows that the TIRE successfully satisfies the EPBPD quota with the highest read-hit ratio of 20.4% and FUE of 0.6 with a larger EPBPD balance of 208 compared to infinite slack. Clearly, the dynamic throttling policy based on access count demonstrates interesting trade-offs between EPBPD slack and performance. As discussed earlier, turning off admission control leads to a higher read-hit ratio (23.6%) but a negative EPBPD balance of -1836, which will wear out the flash before its intended lifespan.

Table 6 summarizes the results for our traces when the TIRE controller is on. We do not include `prn0` results because its EPBPD is below 5. As the EPBPD quota increases from 5 to 8.3, read-hit ratio and FUE values increase, though the flash lifetime decreases. In general, removing the EPBPD quota decreases the normalized read response time. As an example, the MS `map` trace has 107% normalized read response time with a quota of 5 compared to 89% without any quota (Table 5). Pannier allows users to explore trade-offs between performance and flash lifespan to achieve the desired performance objectives.

	quota=5				*quota=8.3*			
Trace	*RHR*	*FUE*	*Perf.(%)* IOPS	*Perf.(%)* RRT	*RHR*	*FUE*	*Perf.(%)* IOPS	*Perf.(%)* RRT
`src11`	20.4	0.6	90	111	22.9	0.9	96	108
`map`	22.0	1.2	92	107	24.3	1.6	98	104

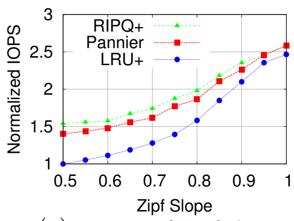
Table 6: Performance evaluation of Pannier with EPBPD quota of 5 and 8.3. The IOPS and RRT reduction percentage are relative to LRU policy in Table 5. The standard deviation for performance is ≤4.8%.

5.5 Sensitivity and Overhead Analysis

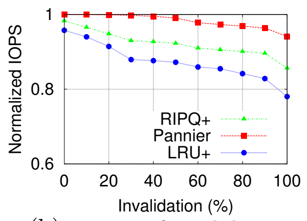
To further understand the interplay of access pattern variability and invalidation on caching, we consider a static web content flash cache use case, which is typically deployed in a content delivery network (CDN). Then, we study the resource overhead of Pannier.

Impact of access pattern variability. We first study a static web content trace generated using ProWGen with a Zipf distribution of access counts. The read-only web server trace is characterized with variable size objects and no invalidations. We compare LRU+, Pannier and RIPQ+ by varying the Zipf slope from 0.5 to 1 [3, 4, 11]. We choose RIPQ+ because it is specifically designed as a static content cache, and we use the default value of 256MB containers and 8 insertion points for their segmented queue [35]. Pannier and LRU+ also used 256MB containers for consistency. We set the cache size as 10% of the working set size.

Figure 13(a) shows the normalized IOPS as a function of Zipf slope, which controls accesses to data. We normalized against the IOPS of LRU+ when the Zipf slope is set to 0.5. When the Zipf slope is set to 0.5 (highly varying access patterns), the read-hit ratios for RIPQ+, Pannier and LRU+ are 53%, 50%, 27%, respectively. When the Zipf slope is set to 1 (low variability of access patterns), the read-hit ratios are close to 81% for all techniques. When the Zipf slope is low, access patterns are spread across containers, which lowers the read-hit ratio. This effect is more pronounced in LRU+ because one accessed block may keep an entire container alive. RIPQ+ and Pannier successfully consolidate



(a) Impact of Zipf slope



(b) Impact of invalidation

Figure 13: Impact of Zipf slope and invalidation on a static web content workload. The standard deviation for performance is ≤3.8%.

hot blocks to new containers, which translates to up to 54% IOPS improvement compared to LRU+.

Factoring in invalidation. Next, we use a controlled approach to study the interplay of invalidation and access pattern variability. We gradually increase the percentage of overwrites from 0 to 100% to the read-only web trace. We fix the cache size and accesses as in the previous experiment, and we set the Zipf slope to 0.8 (a typically measured value [3, 4, 11]). We normalized the IOPS to the Pannier result with zero invalidations. Pannier assigns shorter survival time for heavily invalidated containers, and Figure 13(b) shows that the read-hit ratio of Pannier changes from 71% to 67%, which translates to a slight IOPS decrease of 7%. Invalidated containers in RIPQ+ and LRU+ have to go through the entire stack to be evicted, therefore, the IOPS decreases are 13% and 18%, respectively, compared to 0% invalidation. We found that a higher Zipf slope exacerbates the problem because both divergent containers and invalidation prevents containers from being evicted by LRU+, causing low space efficiency and low system IOPS.

Resource overhead. The RAM requirements for cache status include block and container records, queue structures, and sampling for threshold values. If we target a 1TB flash cache with a container size of 2MB and 4KB blocks, the RAM overhead is ∼400MB, which is small relative to the 4GB needed by a block index for any caching algorithm. Adding an optional ghost cache that tracks two times as many blocks as exist in flash adds an additional ∼4.2GB of DRAM. Note that a storage server with 1TB of flash may devote 50GB of DRAM to data caching (*e.g.*, 5% of flash).

6. RELATED WORK

Caching algorithms and NAND-flash caches, in particular, have been studied extensively in the past decades, so we briefly summarize the literature directly related to our work.

Buffer caches and hardware caches. Caches have been widely deployed as a performance accelerator over the years [5, 10, 12, 14, 18, 24, 26, 27, 31, 35, 38]. For example, MQ [38] leverages a skewed frequency distribution in the second level buffer cache to promote blocks with high frequency to increase the hit-ratio. LIRS [12] leverages a per-block LRU stack to maintain inter-reference recency. Their scheme demotes blocks with long inter-reference recency quickly without traversing the entire LRU stack. However, our container organization makes the per-block inter-reference prediction challenging because logical distinct blocks are physically co-located. One cannot infer the reuse distance from the relative position across containers. ARC [18] divides the cache into two partitions (accessed-once and accessed-many) and promotes a block from one partition to the other on a hit and adjusts the partitions' sizes to adapt to workloads. However, ARC cannot be directly transformed to support containers because block access patterns vary within a container. Qureshi *et al.* [26] found that the LRU Insertion Policy (LIP), which places the incoming line in the LRU position instead of the MRU position, reduces cache misses for memory-intensive workloads. However, a container may have a mixture of hot, cold, and invalidated blocks.

Flash caches. System researchers have studied block-based flash caches extensively [1, 25, 31]. Two recent works, SDF [23] and Nitro [17], use a large write unit that is aligned to the flash erasure unit size to improve flash cache performance. RIPQ [35] leverages another level of indirection to track reaccessed photos within containers. RIPQ only handles static, read-only content and does not make any caching decisions based on container utilization, which we updated to create RIPQ+. In summary, none of the previous work explicitly studied hot/cold block and invalidation mixtures for container-based caching, which is the focus of our work.

A large body of work on flash write buffer management has attempted to leverage flash internal structures to achieve better performance [9, 13, 16, 22], but it remains challenging to incorporate additional information such as clean/dirty status and recency at the firmware layer without modifying the FTL interface.

Container techniques in other domains. Buffering small writes into large containers has been studied extensively, and one of the best known studies is the log-structured file system (LFS) [28], which leverages large containers to achieve high write throughput for HDD systems. WOLF [36] leverages segregating active and inactive data in memory containers to reduce garbage collection overhead. However, some techniques from LFS-type systems cannot be directly applied to flash. For example, a hole plugging [37] technique is effective for disk systems, but flash does not support in-place writes. Also, as a cache, Pannier has the option to silently evict [31] cached data unlike a file system. Pannier is unique in that our use of containers balances high performance and high flash usage effectiveness, though techniques from earlier works influenced our design.

7. CONCLUSION

As flash caches have become widespread for second level caching, new caching algorithms are needed that are aware of the erasure properties of flash. Using containers to buffer small writes is the first step, but it raises the new problem of managing compound objects. This paper studies several important aspects of a container cache including: leveraging access counts to group blocks into containers; managing divergent containers with varying access counts and invalidated blocks; and limiting flash erasures with the TIRE algorithm. Our caching middleware, Pannier, has higher performance than other container-based caching algorithms. In comparison to block-based caching algorithms, Pannier's performance is competitive with significantly fewer erasures.

Acknowledgments

We thank Yue Cheng, Darren Sawyer, and the anonymous reviewers for their contributions and feedback.

8. REFERENCES

[1] A. Badam and V. S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. NSDI, 2011.

[2] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Syst. J.*, 1966.

[3] L. Breslau et al. Web Caching and Zipf-like Distributions: Evidence and Implications. INFOCOM, 1999.

[4] M. Busari and C. Williamson. ProWGen: A Synthetic Workload Generation Tool for Simulation Evaluation of Web Proxy Caches. *Comput. Netw.*, 2002.

[5] P. Cao and S. Irani. Cost-aware WWW Proxy Caching Algorithms. USITS, 1997.

[6] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. SIGMETRICS, 2009.

[7] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 2013.

[8] G. Einziger and R. Friedman. TinyLFU: A Highly Efficient Cache Admission Policy. IEEE PDP, 2014.

[9] Y. Hu et al. Performance Impact and Interplay of SSD Parallelism Through Advanced Commands, Allocation Strategy and Data Granularity. ICS, 2011.

[10] A. Jaleel et al. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). ISCA, 2010.

[11] M. Jeon et al. Workload Characterization and Performance Implications of Large-Scale Blog Servers. *ACM TOW*, 2012.

[12] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. SIGMETRICS, 2002.

[13] H. Jo et al. FAB: Flash-aware Buffer Management Policy for Portable Media Players. *IEEE TOCE*, 2006.

[14] R. Karedla, J. S. Love, and B. G. Wherry. Caching Strategies to Improve Disk System Performance. *Computer*, 1994.

[15] S. Kavalanekar et al. Characterization of Storage Workload Traces from Production Windows Servers. IISWC, 2008.

[16] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. FAST, 2008.

[17] C. Li et al. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. USENIX ATC, 2014.

[18] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. FAST, 2003.

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

[20] S. Muralidhar et al. f4: Facebook's Warm BLOB Storage System. OSDI, 2014.

[21] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-loading: Practical Power Management for Enterprise Storage. FAST, 2008.

[22] Y. Oh et al. Caching Less for Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage Systems. FAST, 2012.

[23] J. Ouyang et al. SDF: Software-defined Flash for Web-scale Internet Storage Systems. ASPLOS, 2014.

[24] V. Phalke and B. Gopinath. An Inter-reference Gap Model for Temporal Locality in Program Behavior. SIGMETRICS, 1995.

[25] D. Qin, A. D. Brown, and A. Goel. Reliable Writeback for Client-side Flash Caches. USENIX ATC, 2014.

[26] M. K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. ISCA, 2007.

[27] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-based Replacement. SIGMETRICS, 1990.

[28] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of A Log-structured File System. *ACM TOCS*, 1992.

[29] Samsung Server SSD Specification. `www.samsung.com/serverssd/`, 2015.

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

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

[32] H. Shim, P. Shilane, and W. Hsu. Characterization of Incremental Data Changes for Efficient Data Protection. USENIX ATC, 2013.

[33] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. SIGMETRICS, 1999.

[34] L. Sonneborn and F. Van Vleck. The Bang-Bang Principle for Linear Control Systems. *SIAM J. Control*, 1965.

[35] L. Tang et al. RIPQ: Effective Photo Caching Algorithm for Facebook. FAST, 2015.

[36] J. Wang and Y. Hu. WOLF–A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File System. FAST, 2002.

[37] J. Wilkes et al. The HP AutoRAID Hierarchical Storage System. SOSP, 1995.

[38] Y. Zhou, Z. Chen, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. USENIX ATC, 2001.