Austere Flash Caching with Deduplication and Compression

Qiuping Wang and Jinhong Li, *The Chinese University of Hong Kong;* Wen Xia, *Harbin Institute of Technology, Shenzhen;* Erik Kruus and Biplob Debnath, *NEC Labs;* Patrick P. C. Lee, *The Chinese University of Hong Kong*

https://www.usenix.org/conference/atc20/presentation/wang-qiuping

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Austere Flash Caching with Deduplication and Compression

Qiuping Wang[†], Jinhong Li[†], Wen Xia[‡], Erik Kruus[*], Biplob Debnath[*], and Patrick P. C. Lee[†]
[†]*The Chinese University of Hong Kong* [‡]*Harbin Institute of Technology, Shenzhen* [*]*NEC Labs*

Abstract

Modern storage systems leverage flash caching to boost I/O performance, and enhancing the space efficiency and endurance of flash caching remains a critical yet challenging issue in the face of ever-growing data-intensive workloads. Deduplication and compression are promising data reduction techniques for storage and I/O savings via the removal of duplicate content, yet they also incur substantial memory overhead for index management. We propose AustereCache, a new flash caching design that aims for memory-efficient indexing, while preserving the data reduction benefits of deduplication and compression. AustereCache emphasizes austere cache management and proposes different core techniques for efficient data organization and cache replacement, so as to eliminate as much indexing metadata as possible and make lightweight in-memory index structures viable. Trace-driven experiments show that our AustereCache prototype saves 69.9-97.0% of memory usage compared to the state-of-the-art flash caching design that supports deduplication and compression, while maintaining comparable read hit ratios and write reduction ratios and achieving high I/O throughput.

1 Introduction

High I/O performance is a critical requirement for modern data-intensive computing. Many studies (e.g., [1, 6, 9, 11, 20, 21, 24, 26, 31, 34, 35, 37]) propose solid-state drives (SSDs) as a flash caching layer atop hard-disk drives (HDDs) to boost performance in a variety of storage architectures, such as local file systems [1], web caches [20], data centers [9], and virtualized storage [6]. SSDs offer several attractive features over HDDs, including high I/O throughput (in both sequential and random workloads), low power consumption, and high reliability. In addition, SSDs have been known to incur much less cost-per-GiB than main memory (DRAM) [27], and such a significant cost difference still holds today (see Table 1). On the other hand, SSDs pose unique challenges over HDDs, as they not only have smaller available capacity, but also have poor endurance due to wear-out issues. Thus, in order to support high-performance workloads, caching as many objects as possible, while mitigating writes to SSDs to avoid wear-outs, is a paramount concern.

We explore both deduplication and compression as data reduction techniques for removing duplicate content on the I/O path, so as to mitigate both storage and I/O costs. Deduplication and compression target different granularities of data reduction and are complementary to each other: while deduplication removes chunk-level duplicates in a coarse-grained but lightweight manner, compression removes byte-level duplicates within chunks for further storage savings. With the ever-increasing growth of data in the wild, deduplication and/or compression have been widely adopted in primary [18, 23, 36] and backup [40, 42] storage systems. In particular, recent studies [24, 26, 37] augment flash caching with deduplication and compression, with emphasis on managing variable-size cached data in large replacement units [24] or designing new cache replacement algorithms [26, 37].

Type	Brand	Cost-Per-GiB ($)
DRAM	Crucial DDR4-2400 (16 GiB)	3.75
SSD	Intel SSD 545s (512 GiB)	0.24
HDD	Seagate BarraCuda (2 TiB)	0.025

Table 1: Cost-per-GiB of DRAM, SSD, and HDD based on the price quotes in January 2020.

Despite the data reduction benefits, existing approaches [24,26,37] of applying deduplication and compression to flash caching inevitably incur substantial memory overhead due to expensive index management. Specifically, in conventional flash caching, we mainly track the logical-to-physical address mappings for the flash cache. With both deduplication and compression enabled, we need dedicated index structures to track: (i) the mappings of each logical address to the physical address of the non-duplicate chunk in the flash cache after deduplication and compression, (ii) the cryptographic hashes (a.k.a. fingerprints (§2.1)) of all stored chunks in the flash cache for duplicate checking in deduplication, and (iii) the lengths of all compressed chunks that are of variable size. It is desirable to keep all such indexing metadata in memory for high performance, yet doing so aggravates the memory overhead compared to conventional flash caching. The additional memory overhead, which we refer to as *memory amplification*, can reach at least 16× (§2.3) and unfortunately compromise the data reduction effectiveness of deduplication and compression in flash caching.

In this paper, we propose AustereCache, a memory-efficient flash caching design that employs deduplication and compression for storage and I/O savings, while substantially mitigating the memory overhead of index structures in similar designs. AustereCache advocates *austere* cache management on the data layout and cache replacement policies to limit the memory amplification due to deduplication and compression. It builds on three core techniques: (i) *bucketization*, which achieves lightweight address mappings by determinis-

tically mapping chunks into fixed-size buckets; (ii) *fixed-size compressed data management*, which avoids tracking chunk lengths in memory by organizing variable-size compressed chunks as fixed-size subchunks; and (iii) *bucket-based cache replacement*, which performs memory-efficient cache replacement on a per-bucket basis and leverages a compact sketch data structure [13] to track deduplication and recency patterns in limited memory space for cache replacement decisions.

We implement an AustereCache prototype and evaluate it through testbed experiments using both real-world and synthetic traces. Compared to CacheDedup [26], a state-of-the-art flash caching system that also supports deduplication and compression, AustereCache uses 69.9-97.0% less memory than CacheDedup, while maintaining comparable read hit ratios and write reduction ratios (i.e., it maintains the I/O performance gains through flash caching backed by deduplication and compression). In addition, AustereCache incurs limited CPU overhead on the I/O path, and can further boost I/O throughput via multi-threading.

The source code of our AustereCache prototype is available at: **http://adslab.cse.cuhk.edu.hk/software/austerecache**.

2 Background

We first provide deduplication and compression background (§2.1). We then present a general flash caching architecture that supports deduplication and compression (§2.2), and show how such an architecture incurs huge memory amplification (§2.3). We finally argue that state-of-the-art designs are limited in mitigating the memory amplification issue (§2.4).

2.1 Deduplication and Compression

Deduplication and compression are data reduction techniques that remove duplicate content at different granularities.

Deduplication. We focus on *chunk-based deduplication*, which divides data into non-overlapping data units called *chunks* (of size KiB). Each chunk is uniquely identified by a *fingerprint (FP)* computed by some cryptographic hash (e.g., SHA-1) of the chunk content. If the FPs of two chunks are identical (or distinct), we treat both chunks as duplicate (or unique) chunks, since the probability that two distinct chunks have the same FP is practically negligible. Deduplication stores only one copy of duplicate chunks (in physical space), while referring all duplicate chunks (in logical space) to the copy via small-size pointers. Also, it keeps all mappings of FPs to physical chunk locations in an index structure used for duplicate checking and chunk lookups.

Chunk sizes may be fixed or variable. While content-based variable-size chunking generally achieves high deduplication savings due to its robustness against content shifts [42], it also incurs high computational overhead. On the other hand, fixed-size chunks fit better into flash units and fixed-size chunking often achieves satisfactory deduplication savings [26]. Thus, this work focuses on fixed-size chunking.

Compression. Unlike deduplication, which provides coarse-grained data reduction at the chunk level, *compression* aims for fine-grained data reduction at the byte level by transforming data into more compact form. Compression is often applied to the unique chunks after deduplication, and the output compressed chunks are of variable-size in general. For high performance, we apply sequential compression (e.g., Ziv-Lempel algorithm [43]) that operates on the bytes of each chunk in a single pass.

2.2 Flash Caching

We focus on building an SSD-based flash cache to boost the I/O performance of HDD-based primary storage, by storing the frequently accessed data in the flash cache. Flash caching has been extensively studied and adopted in different storage architectures (§7). Existing flash caching designs, which we collectively refer to as *conventional flash caching*, mostly support both *write-through* and *write-back* policies for read-intensive and write-intensive workloads, respectively [22]; the write-back policy is viable for flash caching due to the persistent nature of SSDs. For write-through, each write is persisted to both the SSD and the HDD before completion; for write-back, each write is completed right after it is persisted to the SSD. To support either policy, conventional flash caching needs an SSD-HDD translation layer that maps each *logical block address (LBA)* in an HDD to a *chunk address (CA)* in the flash cache.

In this work, we explore how to augment conventional flash caching with deduplication and compression to achieve storage and I/O savings, so as to address the limited capacity and wear-out issues in SSDs. Figure 1 shows the architecture of a general flash caching system that deploys deduplication and compression. We introduce two index structures: (i) *LBA-index*, which tracks how each LBA is mapped to the FP of a chunk (the mappings are many-to-one as multiple LBAs may refer to the same FP), and (ii) *FP-index*, which tracks how each FP is mapped to the CA and the length of a compressed chunk (the mappings are one-to-one). Thus, each cache lookup triggers two index lookups: it finds the FP of an LBA via the LBA-index, and then uses the FP to find the CA and the length of a compressed chunk via the FP-index. We also maintain a *dirty list* to track the list of LBAs of recent writes in write-back mode.

We now elaborate the I/O workflows of the flash caching system in Figure 1. For each write, the system partitions the written data into fixed-size chunks, followed by deduplication and compression: it first checks if each chunk is a duplicate; if not, it further compresses the chunk and writes the compressed chunk to the SSD (the compressed chunks can be packed into large-size units for better flash performance and endurance [24]). It updates the entries in both the LBA-index and the FP-index accordingly based on the FP of the chunk; in write-through mode, it also stores the fixed-size chunk in the HDD in uncompressed form. For each read, the sys-

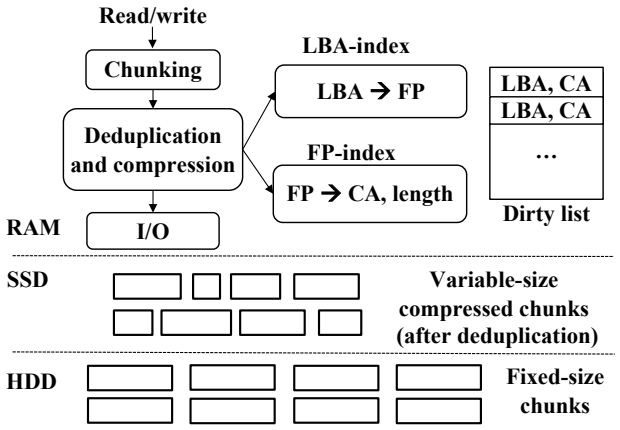


Figure 1: Architecture of a general flash caching system with deduplication and compression.

tem checks if the LBA is mapped to any existing CA via the lookups to both the LBA-index and the FP-index. If so (i.e., cache hit), the system decompresses and returns the chunk data; otherwise (i.e., cache miss), it fetches the chunk data from the HDD into the SSD, while it applies deduplication and compression to the chunk data as in a write.

2.3 Memory Amplification

While deduplication and compression intuitively reduce storage and I/O costs in flash caching by eliminating redundant content on the I/O path, both techniques inevitably incur significant memory costs for their index management. Specifically, if both index structures are entirely stored in memory for high performance, the memory usage is significant and much higher than that in conventional flash caching; we refer to such an issue as *memory amplification* (over conventional flash caching), which can negate the data reduction benefits of deduplication and compression.

We argue this issue through a simple analysis on the following configuration. Suppose that we deploy a 512 GiB SSD as a flash cache atop an HDD that has a working set of 4 TiB. Both the SSD and the HDD have 64-bit address space. For deduplication, we fix the chunk size as 32 KiB and use SHA-1 (20 bytes) for FPs. We also use 4 bytes to record the compressed chunk length. In the worst case, the LBA-index keeps 4 TiB / 32 KiB $= 128 \times 2^{20}$ (LBA, FP) pairs, accounting for a total of 3.5 GiB (each pair comprises an 8-byte LBA and a 20-byte FP). The FP-index keeps 512 GiB / 32 KiB $= 16 \times 2^{20}$ (FP, CA) pairs, accounting for a total of 512 MiB (each pair comprises a 20-byte FP, an 8-byte CA, and a 4-byte length). The total memory usage of both the LBA-index and the FP-index is **4 GiB**. In contrast, conventional flash caching only needs to index 16×2^{20} (LBA, CA) pairs and the memory usage is **256 MiB**. This implies that flash caching with deduplication and compression amplifies the memory usage by **16**×. If we use a more collision-resistant hash function, the memory amplification is even higher; for example, it becomes **22.75**× if each FP is formed by SHA-256 (32 bytes). Note that our analysis does not consider other metadata for deduplication and compression (e.g., reference counts for deduplication), which further aggravates memory amplification over conventional flash caching.

In addition to memory amplification, deduplication and compression also add *CPU overhead* to the I/O path. Such overhead comes from: (i) the FP computation of each chunk, (ii) the compression of each chunk, and (iii) the lookups to both the LBA-index and the FP-index.

2.4 State-of-the-Art Flash Caches

We review two state-of-the-art flash caching designs, Nitro [24] and CacheDedup [26], both of which support deduplication and compression. We argue that both designs are still susceptible to memory amplification.

Nitro [24]. Nitro is the first flash cache that deploys deduplication and compression. To manage variable-size compressed chunks (a.k.a. extents [24]), Nitro packs them in large data units called *Write-Evict Units (WEUs)*, which serve as the basic units for cache replacement. The WEU size is set to align with the flash erasure block size for efficient garbage collection. When the cache is full, Nitro evicts a WEU based on the least-recently-used (LRU) policy. It manages index structures in DRAM (or NVRAM for persistence) to track all chunks in WEUs. If the memory capacity is limited, Nitro stores a *partial* FP-index in memory, at the expense that deduplication may miss detecting and removing some duplicates.

In addition to the memory amplification issue, organizing the chunks by WEUs may cause a WEU to include *stale chunks*, which are not referenced by any LBA in the LBA-index as their original LBAs may have been updated. Such stale chunks cannot be recycled immediately if their hosted WEUs also contain other valid chunks that are recently accessed due to the LRU policy, but instead occupy the cache space and degrade the cache hit ratio.

CacheDedup [26]. CacheDedup focuses on cache replacement algorithms that reduce the number of *orphaned entries*, which refer to either the LBAs that are in the LBA-index but have no corresponding FPs in the FP-index, or the FPs that are in the FP-index but are not referenced by any LBA. It proposes two deduplication-aware cache replacement policies, namely D-LRU and D-ARC, which augment the LRU and adaptive cache replacement (ARC) [29] policies, respectively. It also proposes a compression-enabled variant of D-ARC, called CD-ARC, which manages variable-size compressed chunks in WEUs as in Nitro [24]; note that CD-ARC suffers from the same stale-chunk issue as described above. CacheDedup maintains the same index structures as shown in Figure 1 (§2.2), in which the LBA-index stores LBAs to FPs, and the FP-index stores FPs to CAs and compressed chunk lengths. If it keeps both the LBA-index and the FP-index in memory for performance concerns, it still suffers from the same memory amplification issue. A follow-up work CDAC [37] improves the cache replacement of CacheDedup by incorporating ref-

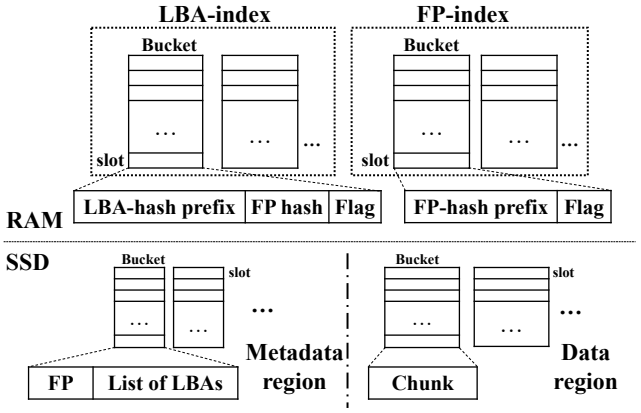


Figure 2: Bucketized data layouts of AustereCache in the LBA-index, the FP-index, as well as the metadata and data regions in flash.

erence counts and access patterns, but incurs even higher memory overhead for maintaining additional information.

3 AustereCache Design

AustereCache is a new flash caching design that leverages deduplication and compression to achieve storage and I/O savings as in prior work [24,26,37], but puts specific emphasis on reducing the memory usage for indexing. It aims for austere cache management via three key techniques.

- **Bucketization (§3.1).** To eliminate the overhead of maintaining address mappings in both the LBA-index and the FP-index, we leverage deterministic hashing to associate chunks with storage locations. Specifically, we hash index entries into equal-size partitions (called *buckets*), each of which keeps the *partial* LBAs and FPs for memory savings. Based on the bucket locations, we further map chunks into the cache space.
- **Fixed-size compressed data management (§3.2).** To avoid tracking chunk lengths in the FP-index, we treat variable-size compressed chunks as fixed-size units. Specifically, we divide variable-size compressed chunks into smaller fixed-size *subchunks* and manage the subchunks without recording the compressed chunk lengths.
- **Bucket-based cache replacement (§3.3).** To increase the likelihood of cache hits, we propose cache replacement on a per-bucket basis. In particular, we incorporate recency and deduplication awareness based on *reference counts* (i.e., the counts of duplicate copies referencing each unique chunk) for effective cache replacement. However, tracking reference counts incurs non-negligible memory overhead. Thus, we leverage a fixed-size compact sketch data structure [13] for reference count estimation in limited memory space with bounded errors.

3.1 Bucketization

Figure 2 shows the bucketized data layouts of AustereCache in both index structures and the flash cache space. We now do not consider compression, which we address in §3.2.

AustereCache partitions both the LBA-index and the FP-index into equal-size *buckets* composed of a fixed number of equal-size *slots*. Each slot corresponds to an LBA and an FP in the LBA-index and the FP-index, respectively. In addition, AustereCache divides the flash cache space into a *metadata region* and a *data region* that store metadata information and cached chunks, respectively; each region is again partitioned into buckets with multiple slots. Note that both regions are allocated the same numbers of buckets and slots as in the FP-index, such that each slot in the FP-index is a one-to-one mapping to the same slots in the metadata and data regions.

To reduce memory usage, each slot stores only the *prefix* of a key, rather than the full key. AustereCache first computes the hashes of both the LBA and the FP, namely *LBA-hash* and *FP-hash*, respectively. It stores the prefix bits of the LBA-hash and the FP-hash as the primary keys in one of the slots of a bucket in the LBA-index and the FP-index, respectively. Keeping only partial keys leads to hash collisions for different LBAs and FPs. To resolve hash collisions, AustereCache maintains the full LBA and FP information in the metadata region in flash, and any hash collision only leads to a cache miss without data loss. Also, by choosing proper prefix sizes, the collision rate should be low. AustereCache currently fixes 128 slots per bucket, mainly for efficient cache replacement (§3.3). For 16-bit prefixes as primary keys, the hash collision rate is only $1-(1-\frac{1}{2^{16}})^{128} \approx 0.2\%$, which is sufficiently low.

Write path. To write a unique chunk identified by an (LBA, FP) pair to the flash cache, AustereCache updates both the LBA-index and the FP-index as follows. For the LBA-index, it uses the suffix bits of the LBA-hash to identify the bucket (e.g., for 2^k buckets, we check the k-bit suffix). It scans all slots in the corresponding bucket to see if the LBA-hash prefix has already been stored; otherwise, it stores the entry in an empty slot or evicts the least-recently-accessed slot if the bucket is full (see cache replacement in §3.3). It writes the following to the slot: the LBA-hash prefix (primary key), the FP-hash, and a valid flag that indicates if the slot stores valid data. Similarly, for the FP-index, it identifies the bucket and the slot using the FP-hash, and writes the FP-hash prefix (primary key) and the valid flag to the corresponding slot.

Based on the bucket and slot locations in the FP-index, AustereCache identifies the corresponding buckets and slots in the metadata and data regions of the flash cache. For the metadata region, it stores the complete FP and the list of LBAs; note that the same FP may be shared by multiple LBAs due to deduplication. We now fix the slot size as 512 bytes. If the slot is full and cannot store more LBAs, we evict the oldest LBA using FIFO to accommodate the new one. For the data region, AustereCache stores the chunk in the corresponding slot, which is also the CA.

Deduplication path. To perform deduplication on a written chunk identified by an (LBA, FP) pair, AustereCache first identifies the bucket of the FP-index using the suffix bits of

the FP-hash, and then searches for any slot that matches the same FP-hash prefix. If a slot is found, AustereCache checks the corresponding slot in the metadata region in flash and verifies if the input FP matches the one in the slot. If so, it means that a duplicate chunk is found, so AustereCache appends the LBA to the LBA list if the LBA does not exist before; otherwise, it implies an FP-hash prefix collision. When such a collision occurs, AustereCache invalidates the collided FP in the metadata region in flash and writes the chunk as described above (recall that the collision is unlikely from our calculation).

Read path. To read a chunk identified by an LBA, AustereCache first queries the LBA-index for the FP-hash using the LBA-hash prefix, followed by querying the FP-index for the slot that contains the FP-hash prefix. It then checks the corresponding slot of the metadata region in flash if an LBA is found in the LBA list. If so, the read is a cache hit and AustereCache returns the chunk from the data region; otherwise, the read is a cache miss and AustereCache accesses the chunk in the HDD via the LBA.

Analysis. We show via a simple analysis that the bucketization design of AustereCache has low memory usage. Suppose that we use a 512 GiB SSD as the flash cache with a 4 TiB working set of an HDD. We fix the chunk size as 32 KiB. Since each bucket has 128 slots, the LBA-index needs at most 2^{20} buckets to reference all chunks in the HDD, while the FP-index needs at most 2^{17} buckets to reference all chunks in the SSD. In addition, we store the first 16 prefix bits of both the LBA-hash and the FP-hash as the partial keys in the LBA-index and the FP-index, respectively. Since we use suffix bits to identify a bucket, we need 20 and 17 suffix bits to identify a bucket in the LBA-index and the FP-index, respectively. Thus, we configure an LBA-hash with $16+20=36$ bits and an FP-hash with $16+17=33$ bits.

We now compute the memory usage of each index structure, to which we apply bit packing for memory efficiency. For the LBA-index, each slot consumes 50 bits (i.e., a 16-bit LBA-hash prefix, a 33-bit FP-hash, and a 1-bit valid flag), so the memory usage of the LBA-index is $2^{20} \times 128 \times 50$ (bits) $=$ 800 MiB. For the FP-index, each slot consumes 17 bits (i.e., a 16-bit FP-hash prefix and a 1-bit valid flag), so the memory usage of the FP-index is $2^{17} \times 128 \times 17$ (bits) $=$ 34 MiB. The total memory usage of both index structures is 834 MiB, which is only around 20% of the 4 GiB memory space in the baseline (§2.3). While we do not consider compression, we emphasize that even with compression enabled, the index structures incur no extra overhead (§3.2).

Comparisons with other data structures. We may construct the LBA-index and the FP-index using other data structures for further memory savings. As an example, we consider the B+-tree [12], which is a balanced tree structure that organizes all leaf nodes at the same level. Suppose that we store index mappings in the leaf nodes that reside in flash, while the non-leaf nodes are kept in memory for referencing the leaf nodes. We evaluate the memory usage of the LBA-index and the FP-index as follows.

Suppose that each leaf node is mapped to a 4 KiB SSD page. For the LBA-index, each leaf node stores at most $\lfloor \frac{4096}{8+20} \rfloor = 146$ (LBA, FP) pairs (for an 8-byte LBA and a 20-byte FP). Referencing each leaf node takes 16 bytes (including an 8-byte LBA key and an 8-byte pointer). As there are 128×2^{20} (LBA, FP) pairs, the memory usage of the LBA-index is $\frac{128 \times 2^{20}}{146} \times 16 \approx 14.0$ MiB (note that we exclude the memory usage for referencing non-leaf nodes). For the FP-index, each leaf node stores at most $\frac{4096}{20+8+4} = 128$ (FP, CA) pairs (for a 20-byte FP, an 8-byte CA, and a 4-byte length). Referencing each leaf node takes 28 bytes (including a 20-byte FP key and an 8-byte pointer). As there are 16×2^{20} (FP, CA) pairs, the memory usage of the FP-index is 3.5 MiB. Both the LBA-index and the FP-index incur much less memory usage than our current bucketization design (see above).

We can further use an in-memory Bloom Filter [8] to query for the existence of index mappings. For an error rate of 0.1%, each mapping uses 14.4 bits in a Bloom Filter. To track both 128×2^{20} (LBA, FP) pairs in the LBA-index and 16×2^{20} (FP, CA) pairs in the FP-index, we need an additional memory usage of 259.2 MiB.

We can conduct similar analyses for other data structures. For example, for the LSM-tree [32], we can maintain an in-memory structure to reference the on-disk LSM-tree nodes (a.k.a. *SSTables* [33]) that store the index mappings for the LBA-index and the FP-index. Then we can accordingly compute the memory usage for the LBA-index and the FP-index.

Even though these data structures support memory-efficient indexing, they incur additional flash access overhead. First, using B+-trees or LSM-trees for both the LBA-index and the FP-index incurs two flash accesses (one for each index structure) for indexing each chunk, while AustereCache issues only one flash access in the metadata region. Also, both the B+-tree and the LSM-tree have high write amplification [33] that degrades I/O performance. For these reasons, and perhaps more importantly, the synergies with compressed data management and cache replacement (see the following subsections), we settle on our proposed bucketized index design.

3.2 Fixed-Size Compressed Data Management

AustereCache can compress each unique chunk after deduplication for further space savings. To avoid tracking the length of the compressed chunk (which is of variable-size) in the index structures, AustereCache slices a compressed chunk into fixed-size *subchunks*, while the last subchunk is padded to fill a subchunk size. For example, for a subchunk size of 8 KiB, we store a compressed chunk of size 15 KiB as two subchunks, with the last subchunk being padded.

AustereCache allocates the same number of consecutive slots as that of subchunks in the FP-index (and hence the metadata and data regions in flash) to organize all subchunks

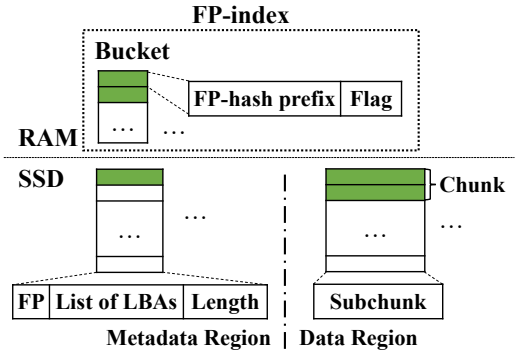


Figure 3: Fixed-size compressed data management, in which multiple consecutive slots are used for handling multiple fixed-size subchunks of a compressed chunk.

of a compressed chunk; note that the LBA-index remains unchanged, and each of its slots still references a chunk. Figure 3 shows an example in which a chunk is stored as two subchunks. For the FP-index, each of the two slots stores the corresponding FP-hash prefix, with an additional 1-bit valid flag indicating that the slot stores valid data. For the metadata region, it also allocates two slots, in which the first slot stores not only the full FP and the list of LBAs (§3.1), but also the length of the compressed chunk, while the second slot can be left empty to avoid redundant flash writes. For the data region, it allocates two slots for storing the two subchunks. Note that our design incurs no memory overhead for tracking the length of the compressed chunk in any index structure.

The read/write workflows with compression are similar to those without compression (§3.1), except that AustereCache now finds consecutive slots in the FP-index for the multiple subchunks of a compressed chunk. Note that we still keep 128 slots per bucket. However, since each slot now corresponds to a smaller-size subchunk, we need to allocate more buckets in the FP-index as well as the metadata and data regions in flash (the number of buckets in the LBA-index remains unchanged since each slot in the LBA-index still references a chunk). As we allocate more buckets for the FP-index, the memory usage also increases. Nevertheless, AustereCache still achieves memory savings for varying subchunk sizes (§5.4).

3.3 Bucket-Based Cache Replacement

Implementing cache replacement often requires priority-based data structures that decide which cached items should be kept or evicted, yet such data structures incur additional memory overhead. AustereCache opts to implement per-bucket cache replacement, i.e., the cache replacement decisions are based on only the entries within each bucket. It then implements specific cache replacement policies that incur no or limited additional memory overhead. Since each bucket is now configured with 128 slots, making the cache replacement decisions also incurs limited performance overhead.

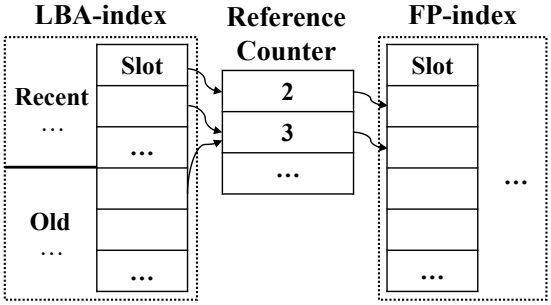


Figure 4: Cache replacement in the FP-index. When a bucket in the FP-index is full, the slot with the least reference counts (e.g. the slot with reference count 2) will be evicted.

For the LBA-index, AustereCache implements a bucket-based least-recently-used (LRU) policy. Specifically, each bucket sorts all slots by the recency of their LBAs, such that the slots at the lower offsets correspond to the more recently accessed LBAs (and vice versa). When the slot of an existing LBA is accessed, AustereCache shifts all slots at lower offsets than the accessed slot by one, and moves the accessed slot to the lowest offset. When a new LBA is inserted, AustereCache stores the new LBA in the slot at the lowest offset and shifts all other slots by one; if the bucket is full, the slot at the highest offset (i.e., the least-recently-accessed slot) is evicted. Such a design does not incur any extra memory overhead for maintaining the recency information of all slots.

For the FP-index, as well as the metadata and data regions in flash, we incorporate both deduplication and recency awareness into cache replacement. First, to incorporate deduplication awareness, AustereCache tracks the reference count for each FP-hash (i.e., the number of LBAs that share the same FP-hash). For each LBA being added to (resp. deleted from) the LBA-index, AustereCache increments (resp. decrements) the reference count of the corresponding FP-hash. When inserting a new FP to a full bucket, it evicts the slot that has the lowest reference count among all the slots in the same bucket. It also invalidates the corresponding slots in both the metadata and data regions in flash.

Simple reference counting does not address recency. To also incorporate recency awareness, AustereCache divides each LBA bucket into *recent slots* at lower offsets and *old slots* at higher offsets (now being divided evenly by half), as shown in Figure 4. Each LBA in the recent (resp. old) slots contributes to a count of two (resp. one) to the reference counting. Specifically, each newly inserted LBA is stored in the recent slot at the lowest offset in the LBA-index (see above), so AustereCache increments the reference count of the corresponding FP-hash by two. If an LBA is demoted from a recent slot to an old slot or is evicted from the LBA-index, AustereCache decrements the reference count of the corresponding FP-hash by one; similarly, if an LBA is promoted from an old slot to a recent slot, AustereCache increments the reference count of the corresponding FP-hash by one.

Maintaining reference counts for all FP-hashes, however, incurs non-negligible memory overhead. AustereCache addresses this issue by maintaining a Count-Min Sketch [13] to track the reference counts in a fixed-size compact data structure with bounded errors. A Count-Min Sketch is a two-dimensional counter array with r rows of w counters each (where r and w are configurable parameters). It maps each FP-hash (via an independent hash function) to one of the w counters in each of the r rows, and increments or decrements the mapped counters based on our reference counting mechanism. AustereCache can estimate the reference count of an FP-hash using the minimum value of all mapped counters of the FP-hash. Depending on the values of r and w, the error bounds can be theoretically proven [13].

Currently, our implementation fixes $r = 4$ and w equal to the total number of slots in the LBA-index. We justify via a simple analysis that sketch-based reference counting achieves significant memory savings. Referring to the analysis in §3.1, each FP-hash has 33 bits. If we track the reference counts of all FP-hashes, we need 2^{33} counters. On the other hand, if we use a Count-Min sketch, we set $r = 4$ and $w = 2^{27}$ (the total number of slots in the LBA-index), so there are $r \times w = 2^{29}$ counters, which consume only 1/16 of the memory usage of tracking all FP-hashes.

Our bucket-based cache replacement design works at the slot level. By using reference counting to make cache replacement decisions, AustereCache can promptly evict any stale chunk that is not referenced by an LBA, as opposed to the WEU design in Nitro and CD-ARC of CacheDedup (§2.4).

4 Implementation

We implement an AustereCache prototype as a user-space block device in C++ on Linux; the user-space implementation (as in Nitro [24]) allows us to readily deploy fast algorithms and multi-threading for performance speedups. Specifically, our AustereCache prototype issues reads and writes to the underlying storage devices via `pread` and `pwrite` system calls, respectively. It uses SHA-1 from the Intel ISA-L Crypto library [3] for chunk fingerprinting, LZ4 [4] for lossless stream-based compression, and XXHash [5] for fast hash computations in the index structures. We also integrate the cache replacement algorithms in CacheDedup [26] into our prototype for fair comparisons (§5). Our prototype now contains around 4.5 K LoC.

We leverage multi-threading to issue multiple read/write requests in parallel for high performance. Specifically, we implement bucket-level concurrency, such that each read/write request needs to acquire an exclusive lock to access a bucket in both the LBA-index and the FP-index, while multiple requests can access different buckets simultaneously.

5 Evaluation

We experiment AustereCache using both real-world and synthetic traces. We consider two variants of AustereCache: (i) AC-D, which performs deduplication only without compression, and (ii) AC-DC, which performs both deduplication and compression. We compare AustereCache with the three cache replacement algorithms of CacheDedup [26]: D-LRU, D-ARC, and CD-ARC (§2.4) (recall that CD-ARC combines D-ARC with the WEU-based compressed chunk management in Nitro [24]). For consistent naming, we refer to them as *CD-LRU-D*, *CD-ARC-D*, and *CD-ARC-DC*, respectively (i.e., the abbreviation of CacheDedup, the cache replacement algorithm, and the deduplication/compression feature). We summarize our evaluation findings as follows.

- Overall, AustereCache reduces memory usage by 69.9-97.0% compared to CacheDedup (Exp#1). It achieves the memory savings via different design techniques (Exp#2).
- AC-D achieves higher read hit ratios than CD-LRU-D and comparable read hit ratios as CD-ARC-D, while AC-DC achieves higher read hit ratios than CD-ARC-DC (Exp#3).
- AC-DC writes much less data to flash than CD-LRU-D and CD-ARC-D, while writing slightly more data than CD-ARC-DC due to padding (§3.2) (Exp#4).
- AustereCache maintains its substantial memory savings for different chunk sizes and subchunk sizes (Exp#5). We also study how it is affected by the sizes of both the LBA-index and the FP-index (Exp#6).
- AustereCache achieves high I/O throughput for different access patterns (Exp#7), while incurring small CPU overhead (Exp#8). Its throughput further improves via multi-threading (Exp#9).

Traces	Working Set (GiB)	Unique Data (GiB)	Write-to-Read Ratio
WebVM	2.71	69.37	3.24
Homes	19.19	240.00	10.81
Mail	59.01	983.78	5.09

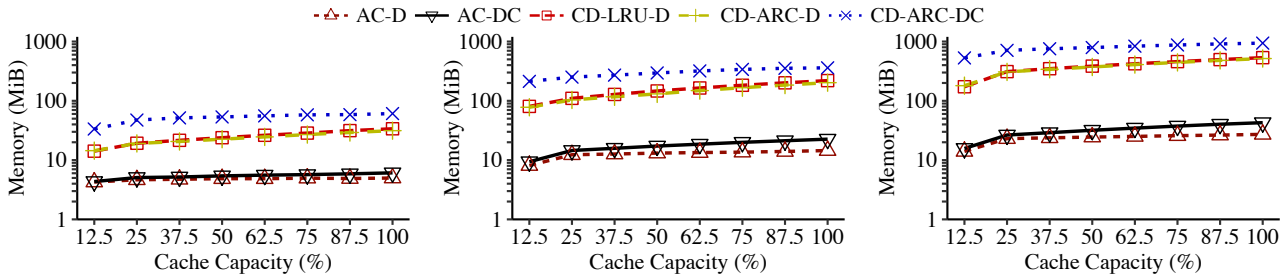
Table 2: Basic statistics of FIU traces in 32 KiB chunks.

5.1 Traces

Our evaluation is driven by two traces.

FIU [23]. The FIU traces are collected from three different services with diverse properties, namely *WebVM*, *Homes*, and *Mail*, for the web, NFS, and mail services, respectively. Each trace describes the read/write requests on different chunks (of size 4 KiB or 512 bytes each), each of which is represented as an MD5 fingerprint of the chunk content.

To accommodate different chunk sizes, we take each trace of 4 KiB chunks and perform two-phase trace conversion as in [24]. In the first phase, we identify the initial state of the disk by traversing the whole trace and recording the LBAs of all chunk reads; any LBA that does not appear is assumed to have a dummy chunk fingerprint (e.g., all zeroes). In the second phase, we regenerate the trace of the corresponding chunk size based on the LBAs and compute the new chunk fingerprints. For example, we form a 32 KiB chunk by concatenating eight contiguous 4 KiB chunks and calculating a



(a) WebVM (b) Homes (c) Mail

Figure 5: Exp#1 (Overall memory usage). Note that the y-axes are in log scale.

new SHA-1 fingerprint for the 32 KiB chunk. Table 2 shows the basic statistics of each regenerated FIU trace on 32 KiB chunks.

The original FIU traces have no compression details. Thus, for each chunk fingerprint, we set its *compressibility ratio* (i.e., the ratio of raw bytes to the compressed bytes) following a normal distribution with mean 2 and variance 0.25 as in [24].

Synthetic. For throughput measurement (§5.5), we build a synthetic trace generator to account for different access patterns. Each synthetic trace is configured by two parameters: (i) *I/O deduplication ratio*, which specifies the fraction of writes that can be removed on the write path due to deduplication; and (ii) *write-to-read ratio*, which specifies the ratios of writes to reads.

We generate a synthetic trace as follows. First, we randomly generate a working set by choosing arbitrary LBAs within the primary storage. Then we generate an access pattern based on the given write-to-read ratio, such that the write and read requests each follow a Zipf distribution. We derive the chunk content of each write request based on the given I/O deduplication ratio as well as the compressibility ratio as in the FIU trace generation (see above). Currently, our evaluation fixes the working set size as 128 MiB, the primary storage size as 5 GiB, and the Zipf constant as 1.0; such parameters are all configurable.

5.2 Setup

Testbed. We conduct our experiments on a machine running Ubuntu 18.04 LTS with Linux kernel 4.15. The machine is equipped with a 10-core 2.2 GHz Intel Xeon E5-2630v4 CPU, 32 GiB DDR4 RAM, a 1 TiB Seagate ST1000DM010-2EP1 SATA HDD as the primary storage, and a 128 GiB Intel SSDSC2BW12 SATA SSD as the flash cache.

Default setup. For both AustereCache and CacheDedup, we configure the size of the FP-index based on a fraction of the working set size (WSS) of each trace, and fix the size of the LBA-index four times that of the FP-index. We store both the LBA-index and the FP-index in memory for high performance. For AustereCache, we set the default chunk size and subchunk size as 32 KiB and 8 KiB, respectively. For CD-ARC-DC in CacheDedup, we set the WEU size as 2 MiB (the default in [26]).

5.3 Comparative Analysis

We compare AustereCache and CacheDedup in terms of memory usage, read hit ratios, and write reduction ratios using the FIU traces.

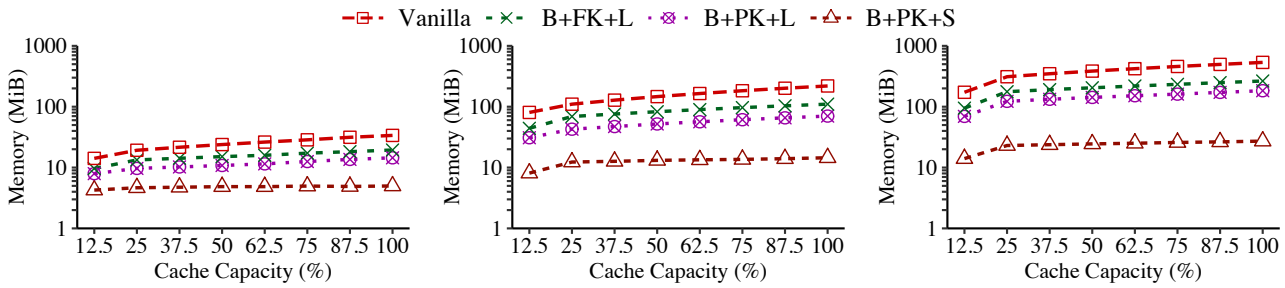
Exp#1 (Overall memory usage). We compare the memory usage of different schemes. We vary the flash cache size from 12.5% to 100% of WSS of each FIU trace, and configure the LBA-index and the FP-index based on our default setup (§5.2). To obtain the actual memory usage (rather than the allocated memory space for the index structures), we call `malloc_trim` at the end of each trace replay to return all unallocated memory from the process heap to the operating system, and check the residual set size (RSS) from `/proc/self/stat` as the memory usage.

Figure 5 shows that AustereCache significantly saves the memory usage compared to CacheDedup. For the non-compression schemes (i.e., AC-D, CD-LRU-D, and CD-ARC-D), AC-D incurs 69.9-94.9% and 70.4-94.7% less memory across all traces than CD-LRU-D and CD-ARC-D, respectively. For the compression schemes (i.e., AC-DC and CD-ARC-DC), AC-DC incurs 87.0-97.0% less memory than CD-ARC-DC.

AustereCache achieves higher memory savings than CacheDedup in compression mode, since CD-ARC-DC needs to additionally maintain the lengths of all compressed chunks, while AC-DC eliminates such information. If we compare the memory overhead with and without compression, CD-ARC-DC incurs 78-194% more memory usage than CD-ARC-D across all traces, implying that compression comes with high memory usage penalty in CacheDedup. On the other hand, AC-DC only incurs 2-58% more memory than AC-D.

Exp#2 (Impact of design techniques on memory savings). We study how different design techniques of AustereCache help memory savings. We mainly focus on bucketization (§3.1) and bucket-based cache replacement (§3.3); for fixed-size compressed data management (§3.2), we refer readers to Exp#1 for our analysis.

We choose CD-LRU-D of CacheDedup as our baseline and compare it with AC-D (both are non-compressed versions), and add individual techniques to see how they contribute to the memory savings of AC-D. We consider four variants:



(a) WebVM (b) Homes (c) Mail

Figure 6: Exp#2 (Impact of design techniques on memory savings).

- *Vanilla*. It refers to CD-LRU-D. It maintains the LRU lists that track the LBAs and FPs being accessed in the LBA index and the FP index, respectively.
- *B+FK+L*. It deploys bucketization (B), but keeps the full keys (FK) (i.e., LBAs and FPs) in each slot. Each bucket implements the LRU policy (L) independently and keeps an LRU list of the slot IDs being accessed.
- *B+PK+L*. It deploys bucketization (B) and now keeps the prefix keys (PK) in both the LBA-index and the FP-index. It still implements the LRU policy as in B+FK+L.
- *B+PK+S*. It deploys bucketization (B) and keeps the prefix keys (PK). It maintains reference counts in a sketch (S). Note that it is equivalent to AC-D.

Figure 6 presents the memory usage versus the cache capacity, where the memory usage is measured as in Exp#1. Compared to Vanilla, B+FK+L saves the memory usage by 30.6-50.6%, while B+PK+L further increases the savings to 43.9-68.0% due to keeping prefix keys in the index structures. B+FK+S (i.e., AC-D) increases the overall memory savings to 69.9-94.9% by keeping reference counts in a sketch as opposed to maintaining LRU lists with full LBAs and FPs.

Exp#3 (Read hit ratio). We evaluate different schemes with the *read hit ratio*, defined as the fraction of read requests that receive cache hits over the total number of read requests.

Figure 7 shows the results. AustereCache generally achieves higher read hit ratios than different CacheDedup algorithms. For the non-compression schemes, AC-D increases the read hit ratio of CD-LRU-D by up to 39.2%. The reason is that CD-LRU-D is only aware of the request recency and fails to clean stale chunks in time (§2.4), while AustereCache favors to evict chunks with small reference counts. On the other hand, AC-D achieves similar read hit ratios to CD-ARC-D, and in particular has a higher read hit ratio (up to 13.4%) when the cache size is small in WebVM (12.5% WSS) by keeping highly referenced chunks in cache. For the compression schemes, AC-DC has higher read hit ratios than CD-ARC-DC, by 0.5-30.7% in WebVM, 0.7-9.9% in Homes, and 0.3-6.2% in Mail. Note that CD-ARC-DC shows a lower read hit ratio than CD-ARC-D although it intuitively stores more chunks with compression, mainly because it cannot quickly evict stale chunks due to the WEU-based organization (§2.4).

Exp#4 (Write reduction ratio). We further evaluate different schemes in terms of the *write reduction ratio*, defined as the fraction of reduction of bytes written to the cache due to both deduplication and compression. A high write reduction ratio implies less written data to the flash cache and hence improved performance and endurance.

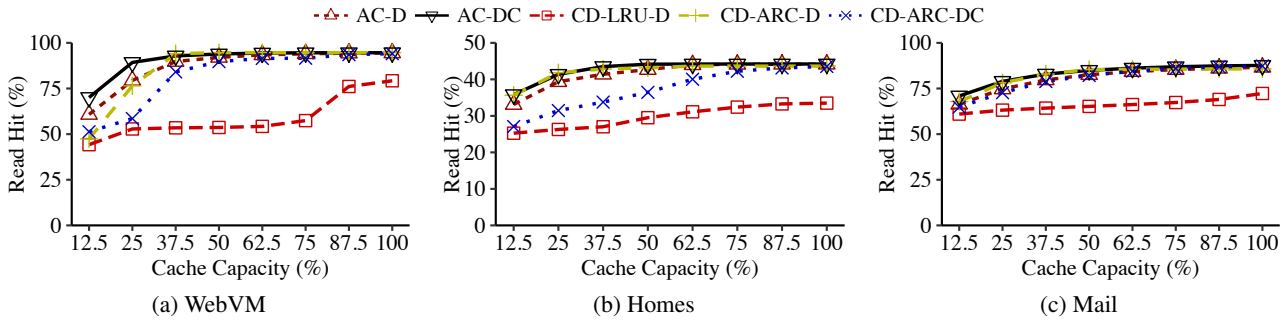
Figure 8 shows the results. For the non-compression schemes, AC-D, CD-LRU-D, and CD-ARC-D show marginal differences in WebVM and Homes, while in Mail, AC-D has lower write reduction ratios than CD-LRU-D by up to 17.5%. We find that CD-LRU-D tends to keep more stale chunks in cache, thereby saving the writes that hit the stale chunks. For example, when the cache size is 12.5% of WSS in Mail, 17.1% of the write reduction in CD-LRU-D comes from the writes to the stale chunks, while in WebVM and Homes, the corresponding numbers are only 3.6% and 1.1%, respectively. AC-D achieves lower write reduction ratios than CD-LRU-D, but achieves much higher read hit ratios by up to 39.2% by favoring to evict the chunks with small reference counts (Exp#3).

For the compression schemes, both CD-ARC-DC and AC-DC have much higher write reduction ratios than the non-compression schemes due to compression. However, AC-DC shows a slightly lower write reduction ratio than CD-ARC-DC by 7.7-14.5%. The reason is that AC-DC pads the last subchunk of each variable-size compressed chunk, thereby incurring extra writes. As we show later in Exp#5 (§5.4), a smaller subchunk size can reduce the padding overhead, although the memory usage also increases.

5.4 Sensitivity to Parameters

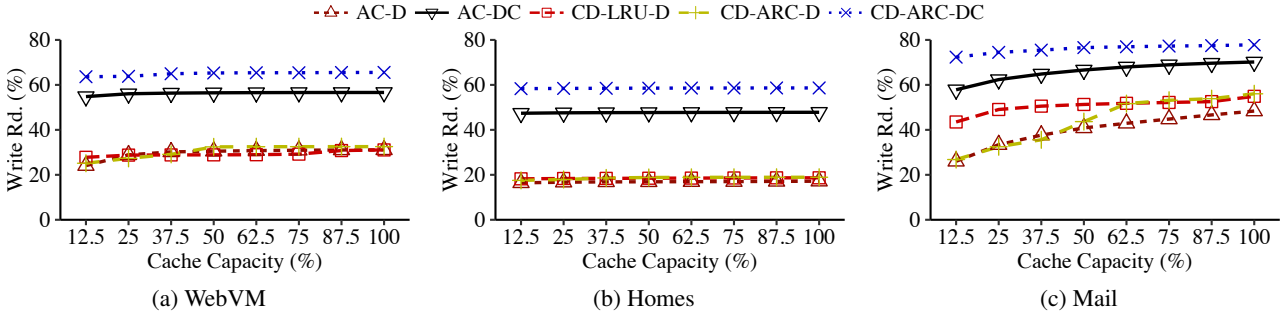
We evaluate AustereCache for different parameter settings using the FIU traces.

Exp#5 (Impact of chunk sizes and subchunk sizes). We evaluate AustereCache on different chunk sizes and subchunk sizes. We focus on the Homes trace and vary the chunk sizes and subchunk sizes as described in §5.1. For varying chunk sizes, we fix the subchunk size as one-fourth of the chunk size; for varying subchunk sizes, we fix the chunk size as 32 KiB. We focus on comparing AC-DC and CD-ARC-DC by fixing the cache size as 25% of WSS. Note that CD-ARC-DC is unaffected by the subchunk size.



(a) WebVM (b) Homes (c) Mail

Figure 7: Exp#3 (Read hit ratio).



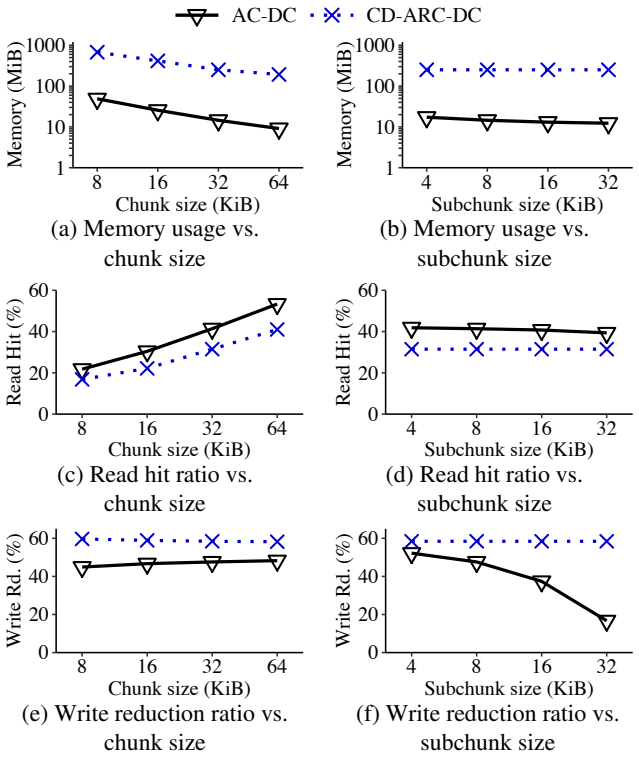
(a) WebVM (b) Homes (c) Mail

Figure 8: Exp#4 (Write reduction ratio).

AC-DC maintains the significant memory savings compared to CD-ARC-DC, by 92.8-95.3% for varying chunk sizes (Figure 9(a)) and 93.1-95.1% for varying subchunk sizes (Figure 9(b)). It also maintains higher read hit ratios than CD-ARC-DC, by 5.0-12.3% for varying chunk sizes (Figure 9(c)) and 7.9-10.4% for varying subchunk sizes (Figure 9(d)). AC-DC incurs a (slightly) less write reduction ratio than CD-ARC-DC due to padding, by 10.0-14.8% for varying chunk sizes (Figure 9(e)); the results are consistent with those in Exp#4. Nevertheless, using a smaller subchunk size can mitigate the padding overhead. As shown in Figure 9(f), the write reduction ratio of AC-DC approaches that of CD-ARC-DC when the subchunk size decreases. When the subchunk size is 4 KiB, AC-DC only has a 6.2% less write reduction ratio than CD-ARC-DC. Note that if we change the subchunk size from 8 KiB to 4 KiB, the memory usage increases from 14.5 MiB to 17.3 MiB (by 18.8%), since the number of buckets is doubled in the FP-index (while the LBA-index remains the same).

Exp#6 (Impact of LBA-index sizes). We study the impact of LBA-index sizes. We vary the LBA-index size from 1× to 8× of the FP-index size (recall that the default is 4×), and fix the cache size as 12.5% of WSS.

Figure 10 depicts the memory usage and read hit ratios; we omit the write reduction ratio as there is nearly no change for varying LBA-index sizes. When the LBA-index size increases, the memory usage increases by 17.6%, 111.5%, and 160.9% in WebVM, Homes and Mail, respectively (Figure 10(a)), as we allocate more buckets in the LBA-index. Note that the increase in memory usage in WebVM is less



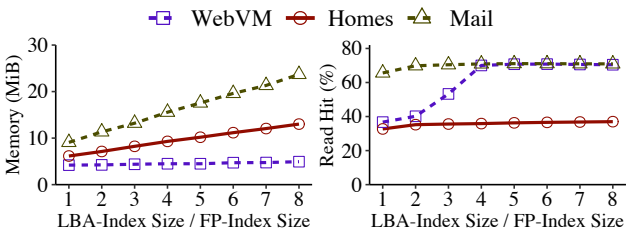
(a) Memory usage vs. chunk size (b) Memory usage vs. subchunk size

(c) Read hit ratio vs. chunk size (d) Read hit ratio vs. subchunk size

(e) Write reduction ratio vs. chunk size (f) Write reduction ratio vs. subchunk size

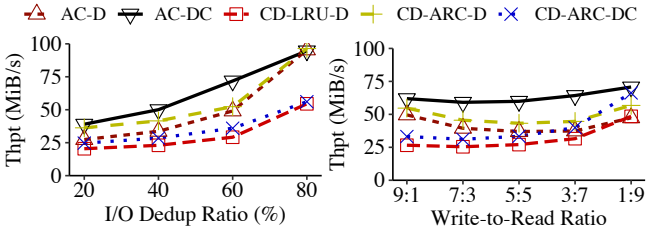
Figure 9: Exp#5 (Impact of chunk sizes and subchunk sizes). We focus on the Homes trace and fix the cache size as 25% of WSS in Homes.

than those in Homes and Mail, mainly because the WSS of WebVM is small and incurs a small actual increase of the total memory usage. Also, the read hit ratio increases with



(a) Memory usage (b) Read hit ratio

Figure 10: Exp#6 (Impact of LBA-index sizes).



(a) Throughput vs. I/O dedup ratio (write-to-read ratio 7:3) (b) Throughput vs. write-to-read ratio (I/O dedup ratio 50%)

Figure 11: Exp#7 (Throughput).

the LBA-index size, until the LBA-index reaches 4× of the FP-index size (Figure 10(b)). In particular, for WebVM, the read hit ratio grows from 36.7% (1×) to 70.4% (8×), while for Homes and Mail, the read hit ratios increase by only 4.3% and 5.3%, respectively. The reason is that when the LBA-index size increases, WebVM shows a higher increase in the total reference counts of the cached chunks than Homes and Mail, implying that more reads can be served by the cached chunks (i.e., higher read hit ratios).

5.5 Throughput and CPU Overhead

We measure the throughput and CPU overhead of AustereCache. We conduct the evaluation on synthetic traces for varying I/O deduplication ratios and write-to-read ratios. We focus on the write-back policy (§2.2), in which AustereCache first persists the written chunks to the flash cache and flushes the chunks to the HDD when they are evicted from the cache. We use direct I/O to remove the impact of page cache. We report the averaged results over five runs, while the standard deviations are small (less than 2.7%) and hence omitted.

Exp#7 (Throughput). We compare AustereCache and CacheDedup in throughput using synthetic traces. We fix the cache size as 50% of the 128 MiB WSS. Both systems work in single-threaded mode.

Figures 11(a) and 11(b) show the results for varying I/O deduplication ratios (with a fixed write-to-read ratio 7:3, which represents a write-intensive workload as in FIU traces) and varying write-to-read ratios (with a fixed I/O deduplication ratio 50%), respectively. For the non-compression schemes, AC-D achieves 18.5-86.6% higher throughput than CD-LRU-D for all cases except when the write-to-read ratio is 1:9 (slightly slower by 2.3%). Compared to CD-ARC-D, AC-D is slower by 1.1-24.5%, since both AC-D and CD-ARC-D have similar read hit ratios and write reduction ratios (§5.3), while AC-D issues additional reads and writes to the metadata region (CD-ARC-D keeps all indexing information in memory). AC-D achieves similar throughput to CD-ARC-D when there are more duplicate chunks (i.e., under high I/O deduplication ratios). For compression schemes, AC-DC achieves 6.8-99.6% higher throughput than CD-ARC-DC.

Overall, AC-DC achieves the highest throughput among all schemes for two reasons. First, AustereCache generally achieves higher or similar read hit ratios compared to CacheDedup algorithms (§5.3). Second, AustereCache incorporates deduplication awareness into cache replacement by caching chunks with high reference counts, thereby absorbing more writes in the SSD and reducing writes to the slow HDD.

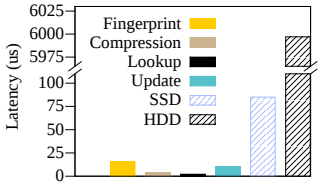


Figure 12: Exp#8 (CPU overhead).

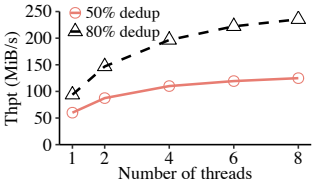


Figure 13: Exp#9 (Throughput of multi-threading).

Exp#8 (CPU overhead). We study the CPU overhead of deduplication and compression of AustereCache along the I/O path. We measure the latencies of four computation steps, including fingerprint computation, compression, index lookup, and index update. Specifically, we run the WebVM trace with a cache size of 12.5% of WSS, and collect the statistics of 100 non-duplicate write requests. We also compare their latencies with those of 32 KiB chunk write requests to the SSD and the HDD using the `fio` benchmark tool [2].

Figure 12 depicts the results. Fingerprint computation has the highest latency (15.5 μs) among all four steps. In total, AustereCache adds around 31.2 μs of CPU overhead. On the other hand, the latencies of 32 KiB writes to the SSD and the HDD are 85 μs and 5,997 μs, respectively. Note that the CPU overhead can be suppressed via multi-threaded processing, as shown in Exp#9.

Exp#9 (Throughput of multi-threading). We evaluate the throughput gain of AustereCache when it enables multi-threading and issues concurrent requests to multiple buckets (§4). We use synthetic traces with a write-to-read ratio of 7:3, and consider the I/O deduplication ratio of 50% and 80%.

Figure 13 shows the throughput versus the number of threads being configured in AustereCache. When the number of threads increases, AustereCache shows a higher throughput gain under 80% I/O deduplication ratio (from 93.8 MiB/s to 235.5 MiB/s, or 2.51×) than under 50% I/O deduplication ratio (from 60.0 MiB/s to 124.9 MiB/s, or 2.08×). A higher I/O deduplication ratio implies less I/O to flash, and AustereCache benefits more from multi-threading on parallelizing

the computation steps in the I/O path and hence sees a higher throughput gain.

6 Discussion

We discuss the following open issues of AustereCache.

Choices of chunk/subchunk sizes. AustereCache by default uses 32 KiB chunks and 8 KiB subchunks to align with common flash page sizes (e.g., 4 KiB or 8 KiB) in commodity SSDs, while preserving memory savings even for various chunk/subchunk sizes (Exp#5 in §5.4). Larger chunk/subchunk sizes reduce the chunk management overhead, at the expense of issuing more read-modify-write operations for small requests from upper-layer applications. Efficiently managing small chunks/subchunks in large-size I/O units in flash caching [24, 25], while maintaining memory efficiency in indexing, is future work.

Impact of indexing on flash endurance. AustereCache currently reduces its memory usage by keeping only limited indexing information in memory and full indexing details in flash (i.e., the metadata region). Since the indexing information generally has a smaller size than the cached chunks, we expect that the updates of the metadata region bring limited degradations to flash endurance, compared to the writes of chunks to the data region. An in-depth analysis of how AustereCache affects flash endurance is future work.

AustereCache assumes that the flash translation layer supports efficient flash erasure management (e.g., applying write combining before writing chunks to flash). To further mitigate the flash erasure overhead, one possible design extension is to adopt a log-structured data organization in flash in order to limit random writes, which are known to degrade flash endurance [30].

7 Related Work

Flash caching. Flash caching has been extensively studied to improve I/O performance. For example, Bcache [1] is a block-level cache for Linux file systems; FlashCache [20] is a file cache for web servers; Mercury [9] is a hypervisor cache for shared storage in data centers; CloudCache [6] estimates the demands of virtual machines (VMs) and manages cache space for VMs in virtualized storage.

Several studies focus on better flash caching management. For example, FlashTier [34] exploits caching workloads in cache block management; Kim *et al.* [21] exploit application hints to cache write requests; DIDACache [35] takes a software-hardware co-design approach to eliminate duplicate garbage collection. To improve the endurance of flash caching, Cheng *et al.* [11] propose erasure-aware heuristics to admit cache insertions; S-RAC [31] selectively evicts cache items based on temporal locality; Pannier [25] manages the flash cache in large-size units (called containers) with erasure awareness; Wang *et al.* [38] use machine learning to remove unnecessary writes to flash.

Deduplication and compression. AustereCache exploits deduplication and compression in flash caching. Extensive work has shown the effectiveness of deduplication and/or compression in storage and I/O savings in primary [18, 23, 36], backup [16, 40, 42], and memory storage [19, 39]. For flash storage, CAFTL [10] implements deduplication in the flash translation layer to reduce flash writes; SmartDedup [41] organizes in-memory and on-disk fingerprints for resource-constrained devices; FlaZ [28] applies transparent and online I/O compression for efficient flash caching. Prior studies [24, 26, 37] also exploit deduplication and compression in flash caching, but incur high memory overhead in metadata management (§2.4). On the other hand, AustereCache aims for memory efficiency without compromising the storage and I/O savings achieved by deduplication and compression.

Memory-efficient designs. Prior studies propose memory-efficient data structures for flash storage. ChunkStash [15] uses fingerprint prefixes to index fingerprints on SSDs in backup deduplication. SkimpyStash [14] designs a hash-table-based index that stores chained linked lists on SSDs for deduplication systems. SILT [27] uses partial-key hashing for efficient indexing in key-value stores. TinyLFU [17] uses Counting Bloom Filters to estimate item frequencies in cache admission. Our bucketization design (§3.1) is similar to the Quotient Filter (also used in flash caching [7]) in prefix-key matching. AustereCache specifically targets flash caching with deduplication and compression, and incorporates several techniques for high memory efficiency.

8 Conclusion

AustereCache makes a case of integrating deduplication and compression into flash caching while significantly mitigating the memory overhead due to indexing. It builds on three techniques to aim for austere cache management: (i) bucketization removes address mappings from indexing; (ii) fixed-size compressed data management removes compressed chunk lengths from indexing; and (iii) bucket-based cache replacement tracks reference counts in a compact sketch structure to achieve high read hit ratios. Evaluation on both real-world and synthetic traces shows that AustereCache achieves significant memory savings, with high read hit ratios, high write reduction ratios, and high throughput.

Acknowledgments: We thank our shepherd, William Jannen, and the anonymous reviewers for their comments. This work was supported in part by RGC of Hong Kong (AoE/P-404/18), NSFC (61972441), and the Shenzhen Science and Technology Program (JCYJ20190806143405318). The corresponding author is Wen Xia.

References

[1] Bcache: A linux kernel block layer cache. `http://bcache.evilpiepirate.org/`.

[2] Fio - Flexible I/O Tester Synthetic Benchmark. http://git.kernel.dk/?p=fio.git.

[3] ISA-L_crypto. https://github.com/intel/isa-l_crypto.

[4] LZ4. https://en.wikipedia.org/wiki/LZ4_(compression_algorithm).

[5] XXHash. https://github.com/Cyan4973/xxHash.

[6] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao. CloudCache: On-demand flash cache management for cloud computing. In *Proc. of USENIX FAST*, 2016.

[7] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *Proc. of VLDB Endowment*, 5(11):1627–1637, 2012.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 12(7):422–426, 1970.

[9] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *Proc. of IEEE MSST*, 2012.

[10] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proc. of USENIX FAST*, 2011.

[11] Y. Cheng, F. Douglis, P. Shilane, G. Wallace, P. Desnoyers, and K. Li. Erasing belady's limitations: In search of flash cache offline optimality. In *Proc. of USENIX ATC*, 2016.

[12] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[13] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[14] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proc. of ACM SIGMOD*, 2011.

[15] B. K. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proc. of USENIX ATC*, 2010.

[16] A. Duggal, F. Jenkins, P. Shilane, R. Chinthekindi, R. Shah, and M. Kamat. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere! In *Proc. of USENIX ATC*, 2019.

[17] G. Einziger, R. Friedman, and B. Manes. TinyLFU: A highly efficient cache admission policy. *ACM Trans. on Storage*, 13(4):1–31, 2017.

[18] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta. Primary data deduplicationlarge scale study and system design. In *Proc. of USENIX ATC*, 2012.

[19] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. S. Lui. SmartMD: A high performance deduplication engine with mixed pages. In *Proc. of USENIX ATC*, 2017.

[20] T. Kgil and T. Mudge. FlashCache: a NAND flash memory file cache for low power web servers. In *Proc. of ACM CASES*, 2006.

[21] S. Kim, H. Kim, S.-H. Kim, J. Lee, and J. Jeong. Request-oriented durable write caching for application performance. In *Proc. of USENIX ATC*, 2015.

[22] R. Koller, , L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Proc. of USENIX FAST*, 2013.

[23] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Trans. on Storage*, 6(3):13, 2010.

[24] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proc. of USENIX ATC*, 2014.

[25] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Trans. on Storage*, 13(3):1–34, 2017.

[26] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao. CacheDedup: In-line deduplication for flash caching. In *Proc. of USENIX FAST*, 2016.

[27] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. of ACM SOSP*, 2011.

[28] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas. Using transparent compression to improve SSD-based I/O caches. In *Proc. of ACM EuroSys*, 2010.

[29] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of USENIX FAST*, 2003.

[30] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: random write considered harmful in solid state drives. In *Proc. of USENIX FAST*, 2012.

[31] Y. Ni, J. Jiang, D. Jiang, X. Ma, J. Xiong, and Y. Wang. S-RAC: SSD friendly caching for data center workloads. In *Proc. of ACM Systor*, 2016.

[32] P. O'Neil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[33] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proc. of ACM SOSP*, 2017.

[34] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: a lightweight, consistent and durable storage cache. In *Proc. of ACM EuroSys*, 2012.

[35] Z. Shen, F. Chen, Y. Jia, and Z. Shao. DIDACache: A deep integration of device and application for flash based key-value caching. In *Proc. of USENIX FAST*, 2017.

[36] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. iDedup: latency-aware, inline data deduplication for primary storage. In *Proc. of USENIX FAST*, 2012.

[37] Y. Tan, J. Xie, C. Xu, Z. Yan, H. Jiang, Y. Zhao, M. Fu, X. Chen, D. Liu, and W. Xia. CDAC: Content-driven deduplication-aware storage cache. In *Proc. of MSST*, 2019.

[38] H. Wang, X. Yi, P. Huang, B. Cheng, and K. Zhou. Efficient SSD caching by avoiding unnecessary writes using machine learning. In *Proc. of ACM ICPP*, 2018.

[39] N. Xia, C. Tian, Y. Luo, H. Liu, and X. Wang. UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling. In *Proc. of USENIX FAST*, 2018.

[40] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proc. of USENIX ATC*, 2011.

[41] Q. Yang, R. Jin, and M. Zhao. SmartDedup: Optimizing deduplication for resource-constrained devices. In *Proc. of USENIX ATC*, 2019.

[42] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of USENIX FAST*, 2008.

[43] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337 – 343, May 1977.