Nitro: A Capacity-Optimized SSD Cache for Primary Storage

Cheng Li, *Rutgers Univerisity;* Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace, *EMC Corporation*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_1

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

Nitro: A Capacity-Optimized SSD Cache for Primary Storage

Cheng Li[†], Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace
[†]*Rutgers University* *EMC Corporation – Data Protection and Availability Division*

Abstract

For many primary storage customers, storage must balance the requirements for large capacity, high performance, and low cost. A well studied technique is to place a solid state drive (SSD) cache in front of hard disk drive (HDD) storage, which can achieve much of the performance benefit of SSDs and the cost per gigabyte efficiency of HDDs. To further lower the cost of SSD caches and increase effective capacity, we propose the addition of data reduction techniques.

Our cache architecture, called `Nitro`, has three main contributions: (1) an SSD cache design with adjustable deduplication, compression, and large replacement units, (2) an evaluation of the trade-offs between data reduction, RAM requirements, SSD writes (reduced up to 53%, which improves lifespan), and storage performance, and (3) acceleration of two prototype storage systems with an increase in IOPS (up to 120%) and reduction of read response time (up to 55%) compared to an SSD cache without Nitro. Additional benefits of Nitro include improved random read performance, faster snapshot restore, and reduced writes to SSDs.

1 Introduction

IT administrators have struggled with the complexity, cost, and overheads of a primary storage architecture as performance and capacity requirements continue to grow. While high IOPS, high throughput, and low latency are necessary for primary workloads, many customers have budget limitations. Therefore, they also want to maximize capacity and management simplicity for a given investment. Balancing these requirements is an ongoing area of storage research.

Fundamentally though, the goals of high performance and cost-efficient storage are in conflict. Solid state drives (SSDs) can support high IOPS with low latency, but their cost will be higher than hard disk drives (HDDs) for the foreseeable future [24]. In contrast, HDDs have high capacity at relatively low cost, but IOPS and latency are limited by the mechanical movements of the drive. Previous work has explored SSDs as a cache in front of HDDs to address performance concerns [1, 7, 30], and the SSD interface has been modified for caching purposes [26], but the cost of SSDs continues to be a large fraction of total storage cost.

Our solution, called Nitro, applies advanced data reduction techniques to SSD caches, increasing the effective cache size and reducing SSD costs for a given system. Deduplication (replacing a repeated data block with a reference) and compression (e.g. LZ) of storage have become the primary strategies to achieve high space and energy efficiency, with most research performed on HDD systems. We refer to the combination of deduplication and compression for storage as capacity-optimized storage (COS), which we contrast with traditional primary storage (TPS) without such features.

Though deduplicating SSDs [4, 13] and compressing SSDs [10, 19, 31] has been studied independently, using both techniques in combination for caching introduces new complexities. Unlike the variable-sized output of compression, the Flash Translation Layer (FTL) supports page reads (e.g. 8KB). The multiple references introduced with deduplication conflicts with SSD erasures that take place at the block level (a group of pages, e.g. 2MB), because individual pages of data may be referenced while the rest of a block could otherwise be reclaimed. Given the high churn of a cache and the limited erase cycles of SSDs, our technique must balance performance concerns with the limited lifespan of SSDs. We believe this is the first study combining deduplication and compression to achieve capacity-optimized SSDs.

Our design is motivated by an analysis of deduplication patterns of primary storage traces and properties of local compression. Primary storage workloads vary in how frequently similar content is accessed, and we wish to minimize deduplication overheads such as in-memory indices. For example, related virtual machines (VMs) have high deduplication whereas database logs tend to have lower deduplication, so Nitro supports targeting deduplication where it can have the most benefit. Since compression creates variable-length data, Nitro packs compressed data into larger units, called Write-Evict Units (WEUs), which align with SSD internal blocks. To extend SSD lifespan, we chose a cache replacement policy that tracks the status of WEUs instead of compressed data, which reduces SSD erasures. An important finding is that replacing WEUs instead of small data blocks maintains nearly the same cache hit ratio and performance of finer-grained replacement, while extending SSD lifespan.

To evaluate Nitro, we developed and validated a simulator and two prototypes. The prototypes place Nitro in front of commercially available storage products. The first prototype uses a COS system with deduplication and compression. The system is typically targeted for storing highly redundant, sequential backups. Therefore, it has lower random I/O performance, but it becomes a plausible primary storage system with Nitro acceleration. The second prototype uses a TPS system without deduplication or compression, which Nitro also accelerates.

Because of the limited computational power and memory of SSDs [13] and to facilitate the use of off-the-shelf SSDs, our prototype implements deduplication and compression in a layer above the SSD FTL. Our evaluation demonstrates that Nitro improves I/O performance because it can service a large fraction of read requests from an SSD cache with low overheads. It also illustrates the trade-offs between performance, RAM, and SSD lifespan. Experiments with prototype systems demonstrate additional benefits including improved random read performance in aged systems, faster snapshot restore when snapshots overlap with primary versions in a cache, and reduced writes to SSDs because of duplicate content. In summary, our contributions are:

- We propose Nitro, an SSD cache that utilizes deduplication, compression, and large replacement units to accelerate primary I/O.
- We investigate the trade-offs between deduplication, compression, RAM requirements, performance, and SSD lifespan.
- We experiment with both COS and TPS prototypes to validate Nitro's performance improvements.

2 Background and Discussion

In this section, we discuss the potential benefits of adding deduplication and compression to an SSD cache and then discuss the appropriate storage layer to add a cache.

Leveraging duplicate content in a cache. I/O rates for primary storage can be accelerated if data regions with different addresses but duplicate content can be reused in a cache. While previous work focused on memory caching and replicating commonly used data to minimize disk seek times [14], we focus on SSD caching.

We analyzed storage traces (described in §5) to understand opportunities to identify repeated content. Figure 1 shows the deduplication ratio (defined in §5.1) for 4KB blocks for various cache sizes. The deduplication ratios increase slowly for small caches and then grow rapidly to $\sim 2.0X$ when the cache is sufficiently large to hold the working set of unique content. This result confirms that a cache has the potential to capture a significant fraction of potential deduplication [13].

This result motivates our efforts to build a deduplicated SSD cache to accelerate primary storage. Adding

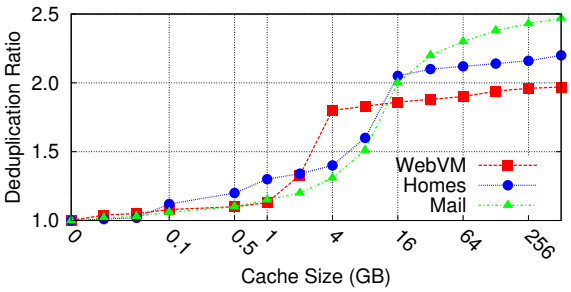


Figure 1: Caching tens of thousand of blocks will achieve most of the potential deduplication.

deduplication to a storage system increases complexity, though, since infrastructure is needed to track the liveness of blocks. In contrast, caching requires less complexity, since cache misses do not cause a data loss for write-through caching, though performance is affected. Also, the overhead of calculating and managing secure fingerprints must not degrade overall performance.

Leveraging compression in a cache. Compression, like deduplication, has the potential to increase cache capacity. Previous studies [5, 10, 29, 31] have shown that local compression saves from 10-60% of capacity, with an approximate mean of 50% using a fast compressor such as LZ. Potentially doubling our cache size is desirable, as long as compression and decompression overheads do not significantly increase latency. Using an LZ-style compressor is promising for a cache, as compared to a HDD system that might use a slower compressor that achieves higher compression. Decompression speed is also critical to achieve low latency storage, so we compress individual data blocks instead of concatenating multiple data blocks before compression. Our implementation has multiple compression/decompression threads, which can leverage future advances in multicore systems.

A complexity of using compression is that it transforms fixed-sized blocks into variable-sized blocks, which is at odds with the properties of SSDs. Similar to previous work [10, 19, 31], we pack compressed data together into larger units (WEUs). Our contribution focuses on exploring the caching impact of these large units, which achieves compression benefits while decreasing SSD erasures.

Appropriate storage layer for Nitro. Caches have been added at nearly every layer of storage systems: from client-side caches to the server-side, and from the protocol layer (e.g. NFS) down to caching within hard drives. For a deduplicated and compressed cache, we believe there are two main locations for a server-side cache. The first is at the highest layer of the storage stack, right after processing the storage protocol. This is the server's first opportunity to cache data, and it is as close to the client as possible, which minimizes latency.

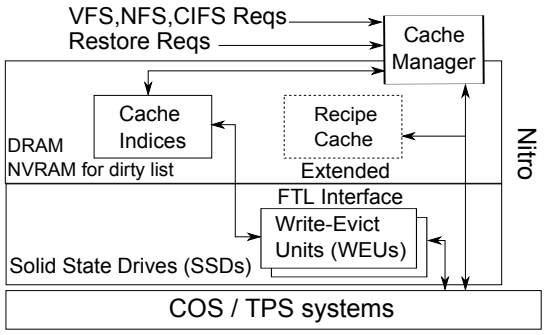


Figure 2: SSD cache and disk storage.

The second location to consider is post-deduplication (and compression) within the system. The advantage of the post-deduplication layer is that currently existing functionality can be reused. Of course, deduplication and compression have not yet achieved wide-spread implementation in storage systems. An issue with adding a cache at the post-deduplication layer is that some mechanism must provide the file recipe, a structure mapping from file and offset to fingerprint (e.g. SHA-1 hash of data), for every cache read. Loading file recipes adds additional I/O and latency to the system, depending on the implementation. While we added Nitro at the protocol layer, for COS systems, we evaluate the impact of using file recipes to accelerate duplicate reads (§3.1). We then compare to TPS systems that do not typically have file recipes, but do benefit from caching at the protocol layer.

3 Nitro Architecture

This section presents the design of our Nitro architecture. Starting at the bottom of Figure 2, we use either COS or TPS HDD systems for large capacity. The middle of the figure shows SSDs used to accelerate performance, and the upper layer shows in-memory structures for managing the SSD and memory caches.

Nitro is conceptually divided into two halves shown in Figure 2 and in more detail in Figure 3 (steps 1-6 are described in §3.2). The top half is called the *CacheManager*, which manages the cache infrastructure (indices), and a lower half that implements SSD caching. The CacheManager maintains a file index that maps the file system interface (<filehandle, offset>) to internal SSD locations; a fingerprint index that detects duplicate content before it is written to SSD; and a dirty list that tracks dirty data for write-back mode. While our description focuses on file systems, other storage abstractions such as volumes or devices are supported. The CacheManager is the same for our simulator and prototype implementations, while the layers below differ to either use simulated or physical SSDs and HDDs (§4).

We place a small amount of NVRAM in front of our cache to buffer pending writes and to support write-back caching: check-pointing and journaling of the dirty list. The CacheManager implements a dynamic prefetching scheme that detects sequential accesses when the consecutive bytes accessed metric (M11 in [17]) is higher than a threshold across multiple-streams. Our cache is scan-resistant because prefetched data that is accessed only once in memory will not be cached. We currently do not cache file system metadata because we do not expect it to deduplicate or compress well, and we leave further analysis to future work.

3.1 Nitro Components

Extent. An extent is the basic unit of data from a file that is stored in the cache, and the cache indices reference extents that are compressed and stored in the SSDs. We performed a large number of experiments to size our extents, and there are trade-offs in terms of read-hit ratio, SSD erasures, deduplication ratio, and RAM overheads. As one example, smaller extents capture finer-grained changes, which typically results in higher deduplication ratios, but smaller extents require more RAM to index. We use the median I/O size of the traces we studied (8KB) as the default extent size. For workloads that have differing deduplication and I/O patterns than what we have studied, a different extent size (or dynamic sizing) may be more appropriate.

Write-Evict Unit (WEU). The Write-Evict Unit is our unit of replacement (writing and evicting) for SSD. File extents are compressed and packed together into one WEU in RAM, which is written to an SSD when it is full. Extents never span WEUs. We set the WEU size equal to one or multiple SSD block(s) (the unit for SSD erase operation) depending on internal SSD properties, to maximize parallelism and reduce internal fragmentation. We store multiple file extents in a WEU. Each WEU has a header section describing its contents, which is used to accelerate rebuilding the RAM indices at start-up. The granularity of cache replacement is an entire WEU, thus eliminating copy forward of live-data to other physical blocks during SSD garbage collection (GC). This replacement strategy has the property of reducing erasures within an SSD, but this decision impacts performance, as we discuss extensively in §6.1. WEUs have generation numbers indicating how often they have been replaced, which are used for consistency checks as described later.

File index. The file index contains a mapping from filehandle and offset to an extent's location in a WEU. The location consists of the WEU ID number, the offset within the WEU, and the amount of compressed data to read. Multiple file index entries may reference the same extent due to deduplication. Entries may also be marked as dirty if write-back mode is supported (shown in gray in Figure 3).

Fingerprint index. To implement deduplication within the SSDs, we use a fingerprint index that maps from ex-

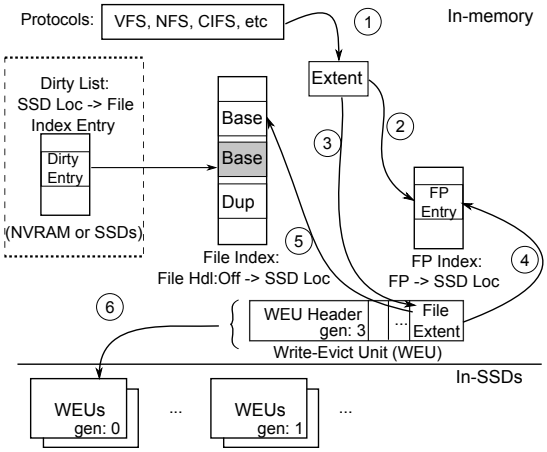


Figure 3: File index with base and duplicate entries, fingerprint index, file extents stored in WEUs, and a dirty list.

tent fingerprint to an extent's location within the SSD. The fingerprint index allows us to find duplicate entries and effectively increase the cache capacity. Since primary workloads may have a wide range of content redundancy, the fingerprint index size can be limited to any arbitrary level, which allows us to make trade-offs between RAM requirements and how much potential deduplication is discovered. We refer to this as the *fingerprint index ratio*, which creates a partial fingerprint index. For a partial fingerprint index, a policy is needed to decide which extents should be inserted into/evicted from the fingerprint index. User-specified configurations, folder/file properties, or access patterns could be used in future work. We currently use LRU eviction, which performed as well as more complicated policies.

Recipe cache. To reduce misses on the read-path, we create a cache of file recipes (Figure 2), which represent a file as a sequence of fingerprints referencing extents. This allows us to check the fingerprint index for already cached, duplicate extents. File recipes are a standard component of COS systems and can be prefetched to our cache, though this requires support from the COS system. Since fingerprints are small (20 bytes) relative to the extent size (KBs), prefetching large lists of fingerprints in the background can be efficient compared to reading the corresponding data from HDD storage. A recipe cache can be an add-on for TPS to opportunistically improve read performance. We do not include a recipe cache in our current TPS implementation because we want to isolate the impact of Nitro without changing other properties of the underlying systems. Its impact on performance is discussed in §6.2.

Dirty list. The CacheManager supports both write-through and write-back mode. Write-through mode assumes all data in the cache are clean because writes to the system are acknowledged when they are stable both in SSD and the underlying storage system. In contrast, write-back mode treats writes as complete when data are cached either in the NVRAM or SSD. In write-back mode, a dirty list tracks dirty extents, which have not yet propagated to the underlying disk system. The dirty list can be maintained in NVRAM (or SSD) for consistent logging since it is a compact list of extent locations. Dirty extents are written to the underlying storage system either when they are evicted from the SSD or when the dirty list reaches a size watermark. When a dirty file index entry is evicted (base or duplicate), the file recipe is also updated. The CacheManager then marks the corresponding file index entries as clean and removes the dirty list entries.

3.2 Nitro Functionality

File read path. Read requests check the file index based on filehandle and offset. If there is a hit in the file index, the CacheManager will read the compressed extent from a WEU and decompress it. The LRU status for the WEU is updated accordingly. For base entries found in the file index, reading the extent's header from SSD can confirm the validity of the extent. When reading a duplicate entry, the CacheManager confirms the validity with WEU generation numbers. An auxiliary structure tracks whether each WEU is currently in memory or in SSD.

If there is a file index miss and the underlying storage system supports file recipes (i.e. COS), the CacheManager prefeteches the file recipe into the recipe cache. Subsequent read requests reference the recipe cache to access fingerprints, which are checked against the cache fingerprint index. If the fingerprint is found to be a duplicate, then cached data can be returned, thus avoiding a substantial fraction of potential disk accesses. The CacheManager updates the LRU status for the fingerprint index if there is a hit. If a read request misses in both the file and fingerprint indices, then the read is serviced from the underlying HDD system, returned to the client, and passed to the cache insertion path.

File write path. On a write, extents are buffered in NVRAM and passed to the CacheManager for asynchronous SSD caching.

Cache insertion path. To demonstrate the process of inserting an extent into the cache and deduplication, consider the following 6-step walk-through example in Figure 3: (1) Hash a new extent (either from caching a read miss or from the file write path) to create a fingerprint. (2) Check the fingerprint against the fingerprint index. If the fingerprint is in the index, update the appropriate LRU status and go to step 5. Otherwise continue with step 3. (3) Compress and append the extent to a WEU that is in-memory, and update the WEU header. (4) Update the fingerprint index to map from a fingerprint to WEU location. (5) Update the file index to map from file handle and offset to WEU. The first entry for the

cached extent is marked as a "Base" entry. Note that the WEU header only tracks the base entry. (6) When an in-memory WEU becomes full, increment the generation number and write it to the SSD. In write-back mode, dirty extents and clean extents are segregated into separate WEUs to simplify eviction, and the dirty-list is updated when a WEU is migrated to SSD.

Handling of duplicate entries is slightly more complicated. Once a WEU is stored in SSD, we do not update its header because of the erase penalty involved. When a write consists of duplicate content, as determined by the fingerprint index, a duplicate entry is created in the file index (marked as "Dup") which points to the extent's location in SSD WEU. Note that when a file extent is over-written, the file index entry is updated to refer to the newest version. Previous version(s) in the SSD may still be referenced by duplicate entries in the file index.

SSD cache replacement policy. Our cache replacement policy selects a WEU from the SSD to evict before reusing that space for a newly packed WEU. The CacheManager initiates cache replacement by migrating dirty data from the selected WEU to disk storage and removing corresponding invalid entries from the file and fingerprint indices. To understand the interaction of WEU and SSDs, we experimented with moving the cache replacement decisions to the SSD, on the premise that the SSD FTL has more internal knowledge. In our co-designed SSD version (§4), the CacheManager will query the SSD to determine which WEU should be replaced based on recency. If the WEU contains dirty data, the CacheManager will read the WEU and write dirty extents to underlying disk storage.

Cleaning the file index. When evicting a WEU from SSD, our in-memory indices must also be updated. The WEU metadata allows us to remove many file index entries. It is impractical, though, to record back pointers for all duplicate entries in the SSD, because these duplicates may be read/written hours or days after the extent is first written to a WEU. Updating a WEU header with a back pointer would increase SSD churn. Instead, we use asynchronous cleaning to remove invalid, duplicate file index entries. A background cleaning thread checks all duplicate entries and determines whether their generation number matches the WEU generation number. If a stale entry is accessed by a client before it is cleaned, then a generation number mismatch indicates that the entry can be removed. All of the WEU generation numbers can be kept in memory, so these checks are quick, and rollover cases are handled.

Faster snapshot restore/access. Nitro not only accelerates random I/Os but also enables faster restore and/or access of snapshots. The SSD can cache snapshot data as well as primary data for COS storage, distinguished by separate snapshot file handles.

We use the standard snapshot functionality of the storage system in combination with file recipes for COS. When reading a snapshot, its recipe will be prefetched from disk into a recipe cache. Using the fingerprint index, duplicate reads will access extents already in the cache, so any shared extents between the primary and snapshot versions can be reused, without additional disk I/O. To accelerate snapshot restores for TPS, integration with differential snapshot tracking is needed.

System restart. Our cache contains numerous extents used to accelerate I/O, and warming up a cache after a system outage (planned or unplanned) could take many hours. To accelerate cache warming, we implemented a system restart/crash recovery technique [26]. A journal tracks the dirty and invalid status of extents. When recovering from a crash, the CacheManager reads the journal, the WEU headers from SSD (faster than reading all extent headers), and recreates indices. Note that our restart algorithm only handles base entries and duplicate entries that reference dirty extents (in write-back mode). Duplicate entries for clean extents are not explicitly referenced from WEU headers, but they can be recovered efficiently by fingerprint lookup when accessed by a client, with only minimal disk I/O to load file recipes.

4 Nitro Implementation

To evaluate Nitro, we developed a simulator and two prototypes. The CacheManager is shared between implementations, while the storage components differ. Our simulator measures read-hit ratios and SSD churn, and its disk stub generates synthetic content based on fingerprint. Our prototypes measure performance and use real SSDs and HDDs.

Potential SSD customization. Most of our experiments use standard SSDs without any modifications, but it is important to validate our design choices against alternatives that modify SSD functionality. Previous projects [1, 3, 13] showed that the design space of the FTL can lead to diverse SSD characteristics, so we would like to understand how Nitro would be affected by potential SSD changes. Interestingly, we found through simulation that Nitro performs nearly as well with a commercial SSD as with a customized SSD.

We explored two FTL modifications, as well as changes to the standard GC algorithm. First, the FTL needs to support aligned allocation of contiguous physical pages for a WEU across multiple planes in aligned blocks, similar to vertical and horizontal super-page striping [3]. Second, to quantify the best-case of using SSD as a cache, we push the cache replacement functionality to the FTL, since the FTL has perfect information about page state. Thus, a new interface allows the CacheManager to update indices and implement write-back mode before eviction. We experimented with mul-

tiple variants and present WEU-LRU, an update to the greedy SSD GC algorithm that replaces WEUs.

We also added the SATA TRIM command [28] in our simulator, which invalidates a range of SSD logical addresses. When the CacheManager issues TRIM commands, the SSD performs GC without copying forward data. Our SSD simulator is based on well-studied simulators [1, 3] with a hybrid mapping scheme [15] where blocks are categorized into data and log blocks. Page-mapped log blocks will be consolidated into block-mapped data blocks through merge operations. Log blocks are further segregated into sequential regions and random areas to reduce expensive merge operations.

Prototype system. We have implemented a prototype Nitro system in user space, leveraging multi-threading and asynchronous I/O to increase parallelism and with support for replaying storage traces. We use real SSDs for our cache, and either a COS or TPS system with hard drives for storage (§ 5). We confirmed the cache hit ratios are the same between the simulator and prototypes. When evicting dirty extents from SSD, they are moved to a write queue and written to disk storage before their corresponding WEU is replaced.

5 Experimental Methodology

In this section, we first describe our analysis metrics. Second, we describe several storage traces used in our experiments. Third, we discuss the range of parameters explored in our evaluation. Fourth, we present the platform for our simulator and prototype systems.

5.1 Metrics

Our results present overall system IOPS, including both reads and writes. Because writes are handled asynchronously and are protected by NVRAM, we further focus on read-hit ratio and read response time to validate Nitro. The principal evaluation metrics are:

IOPS: Input/Output operations per second.

Read-hit ratio: The ratio of read I/O requests satisfied by Nitro over total read requests.

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

SSD erasures: The number of SSD blocks erased, which counts against SSD lifespan.

Deduplication and compression ratios: Ratio of the data size versus the size after deduplication or compression ($\geq$ 1X). Higher values indicate more space savings.

5.2 Experimental Traces

Most of our experiments are with real-world traces, but we also use synthetic traces to study specific topics.

FIU traces: Florida International University (FIU) collected storage traces across multiple weeks, including WebVM (a VM running two web-servers), Mail (an email server with small I/Os), and Homes (a file server with a large fraction of random writes). The FIU traces contain content fingerprint information with small granularity (4KB or 512B), suitable for various extent size studies. The FIU storage systems were reasonably sized, but only a small region of the file systems was accessed during the trace period. For example, WebVM, Homes and Mail have file system sizes of 70GB, 470GB and 500GB in size, respectively, but we measured that the traces only accessed 5.3%, 5.8% and 11.5% of the storage space, respectively [14]. The traces have more writes than reads, with write-to-read ratios of 3.6, 4.2, and 4.3, respectively. To our knowledge, the FIU traces are the only publicly available traces with content.

Boot-storm trace: A "boot-storm" trace refers to many VMs booting up within a short time frame from the same storage system [8]. We first collected a trace while booting up one 18MB VM kernel in Xen hypervisor. The trace consisted of 99% read requests, 14% random I/O, and 1.2X deduplication ratio. With this template, we synthetically produced multiple VM traces in a controlled manner representing a large number of cloned VMs with light changes. Content overlap was set at 90% between VMs, and the addresses of duplicates were shifted by 0-15% of the address space.

Restore trace: To study snapshot restore, we collected 100 daily snapshots of a 38GB workstation VM with a median over-write rate of 2.3%. Large read I/Os (512KB) were issued while restoring the entire VM.

Fingerprint generation. The FIU traces only contain fingerprints for one block size (e.g. 4KB), and we want to vary the extent size for experiments (4-128KB), so it is necessary to process the traces to generate extent fingerprints. We use a multi-pass algorithm, which we briefly describe. The first pass records the fingerprints for each block read in the trace, which is the initial state of the file system. The second pass replays the trace and creates extent fingerprints. An extent fingerprint is generated by calculating a SHA-1 hash of the concatenated block fingerprints within an extent, filling in unspecified block fingerprints with unique values as necessary. Write I/Os within the trace cause an update to block fingerprints and corresponding extent fingerprints. A final pass replays the modified trace for a given experiment.

Synthetic compression information. Since the FIU traces do not have compression information, we synthetically generate content with intermingled unique and repeated data based on a compression ratio parameter. Unless noted, the compression ratio is set for each extent using a normal distribution with mean of 2 and variance of 0.25, representing a typical compression ratio for primary workloads [29]. We used LZ4 [9] for compression and decompression in the prototype.

Variable	Values
Fingerprint index ratio (%)	**100**, 75, 50, 25, 0 (off)
Compression	**on**, off
Extent size (KB)	4, **8**, 16, 32, 64, 128
Write/Evict granularity	**WEU**, extent
Cache size (% of volume)	0.5, 1, **2**, 5
WEU size (MB)	0.5, 1, **2**, 4
Co-design	**standard SSD**, modified SSD
Write-mode	write-through, **write-back**
Prefetching	dynamic up to 128KB
Backend storage	COS, TPS

Table 1: Parameters for Nitro with default values in bold.

5.3 Parameter Space

Table 1 lists the configuration space for Nitro, with default values in bold. Due to space constraints, we interleave parameter discussion with experiments in the evaluation section. While we would like to compare the impact of compression using WEU-caching versus plain extent-based caching, it is unclear how to efficiently store compressed (variable-sized) extents to SSDs without using WEUs or an equivalent structure [10, 19, 31]. For that reason, we show extent caching without compression, but with or without deduplication, depending on the experiment. The cache is sized as a fraction of the storage system size. For the FIU traces, a 2% cache corresponds to 1.4GB, 9.4GB, and 10GB for WebVM, Homes and Mail traces respectively. Most evaluations are with the standard SSD interface except for a co-design evaluation. We use the notation Deduplicated (D), Non-deduplicated (ND), Compressed (C) and Non-compressed (NC). Nitro uses the WEU (D, C) configuration by default.

5.4 Experimental Platform

Our prototype with a COS system is a server equipped with 2.33GHz Xeon CPUs (two sockets, each with two cores supporting two hardware threads). The system has 36GB of DRAM, 960MB of NVRAM, and two shelves of hard drives. One shelf has 12 1TB 7200RPM SATA hard drives, and the other shelf has 15 7200RPM 2TB drives. Each shelf has a RAID-6 configuration including two spare disks. For comparison, the TPS system is a 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. Before each run, we reset the initial state of the HDD storage based on our traces.

Both prototypes use a Samsung 256GB SSD, though our experiments use a small fraction of the available SSD, as controlled by the cache capacity parameter. According to specifications, the SSD supports >100K random read IOPS and >90K random write IOPS. Using a SATA-2 controller (3.0 Gbps), we measured 8KB SSD random reads and writes at 18.7K and 4.2K IOPS, respectively. We cleared the SSD between experiments.

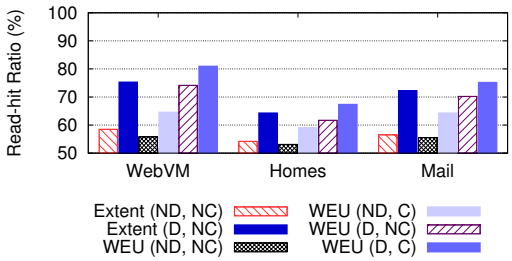


Figure 4: Read-hit ratio of WEU-based vs. Extent-based for all workloads. Y-axis starts at 50%.

We set the SSD simulation parameters based on the Micron MLC SSD specification [23]. We vary the size of each block or flash chip to control the SSD capacity. Note that a larger SSD block size has longer erase time (e.g., 2ms for 128KB and 3.8ms for 2MB). For the unmodified SSD simulation, we over-provision the SSD capacity by 7% for garbage collection, and we reserve 10% for log blocks for the hybrid mapping scheme. No space reservation is used for the modified SSD WEU variants.

6 Evaluation

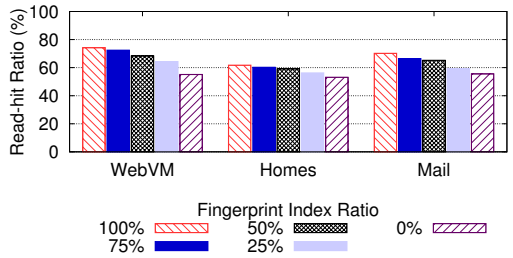
This section presents our experimental results. We first measure the impact of deduplication and compression on caching as well as techniques to reduce in-memory indices and to extend SSD lifespan. Second, we evaluate Nitro performance on both COS and TPS prototype systems and perform sensitivity and overhead analysis. Finally, we study Nitro's additional advantages.

6.1 Simulation Results

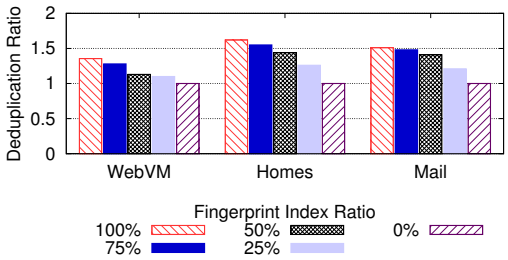
We start with simulation results, which demonstrate caching improvements with deduplication and compression and compare a standard SSD against a co-design that modifies an SSD to specifically support caching.

Read-hit ratio. We begin by showing Nitro's effectiveness at improving the read-hit ratio, which is shown in Figure 4 for all three FIU traces. The trend for all traces is that adding deduplication and compression increases the read-hit ratio.

WEU (D, C) with deduplication (fingerprint index ratio set to 100% of available SSD extent entries) and compression represents the best scenario with improvements of 25%, 14% and 20% across all FIU traces as compared to a version without deduplication or compression (WEU (ND, NC)). Adding compression increases the read-hit ratio for WEU by 5-9%, and adding deduplication increases the read-hit ratio for WEU by 8-19% and extents by 6-17%. Adding deduplication consistently offers a greater improvement than adding compression, suggesting deduplication is capable of increasing the read-hit ratio for primary workloads that contain many duplicates



(a) Impact of fingerprint index ratio on read-hit ratio



(b) Impact of fingerprint index ratio on deduplication ratio

Figure 5: Fingerprint index ratio impact on read-hit ratio and deduplication for WEU (D, NC).

like the FIU traces. Comparing WEU and extent-based caching with deduplication, but without compression (D, NC), extent-based caching has a slightly higher hit-ratio by 1-4% due to finer-grained evictions. However, the advantages of extent-based caching are offset by increased SSD erasures, which are presented later. In an experiment that increased the cache size up to 5% of the file system size, the combination of deduplication and compression (D, C) showed the largest improvement. These results suggest Nitro can extend the caching benefits of SSDs to much larger disk storage systems.

Impact of fingerprint index ratio. To study the impact of deduplication, we adjust the fingerprint index ratio for WEU (D, NC). 100% means that all potential duplicates are represented in the index, while 0% means deduplication is turned off. Decreasing the fingerprint index ratio directly reduces the RAM footprint (29 bytes per entry) but also likely decreases the read-hit ratio as the deduplication ratio drops.

Figure 5(a) shows the read-hit ratio drops gradually as the fingerprint index ratio decreases. Figure 5(b) shows that the deduplication ratio also slowly decreases with the fingerprint index ratio. Homes and Mail have higher deduplication ratios ($\geq$1.5X) than WebVM, as shown in Figure 1. Interestingly, higher deduplication ratios in the Homes and Mail traces do not directly translate to higher read-hit ratios because there are more writes than reads ($\sim$4 W/R ratio), but do increase IOPS (§6.2). Nitro users could limit their RAM footprint by setting the fingerprint index ratio to 75% or 50%, which results in a 16-22% RAM savings respectively and a decrease in read-hit ratio of 5-11%. For example, when reducing the fingerprint index from 100% to 50% for the Mail trace (10GB

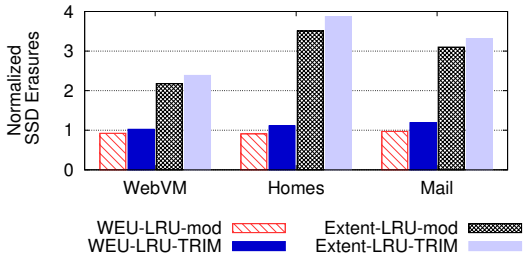


Figure 6: Number of SSD erasures for modified and unmodified SSD variants.

cache size), $\geq$131,000 duplicate extents are not cached by Nitro, on average.

WEU vs. SSD co-design. So far, we considered scenarios where the SSD is unmodified. Next we compare our current design to an alternative that modifies an SSD to directly support WEU caching. In this experiment, we study the impact of silent eviction and the WEU-LRU eviction policy (discussed in §4) on SSD erasures. Our co-design specifically aligns WEUs to SSD blocks (WEU-LRU-mod). We also compare our co-design to variants using the TRIM command (WEU/extent-LRU-TRIM), which alerts the FTL that a range of logical addresses can be released. Figure 6 plots SSD erasures normalized relative to WEU-LRU without SSD modifications (1.0 on the vertical axis) and compares WEU versus extent caching.

SSD erasures are 2-4X higher for the extent-LRU-mod approach (i.e. FlashTier [26] extended to use an LRU policy) and extent-LRU-TRIM approach as compared to both WEU versions. This is because the CacheManager lacks SSD layout information so that extent-based eviction cannot completely avoid copying forward live SSD data. Interestingly, utilizing TRIM with the WEU-LRU-TRIM approach has similar results to WEU-LRU-mod, which indicates the CacheManager could issue TRIM commands before overwriting WEUs instead of modifying the SSD interface. We also analyzed the impact of eviction policy on read-hit ratio. WEU-LRU-mod achieves a 5-8% improvement in read-hit ratio compared to an unmodified version across the FIU traces.

Depending on the data set, the number of SSD erasures varied for the FTL and TRIM alternatives, with results between 9% fewer and 20% more erasures than using WEUs. So, using WEUs for caching is a strong alternative when it is impractical to modify the SSD or when the TRIM command is unsupported. Though not shown, caching extents without SSD modifications or TRIM (naive caching) resulted in several orders of magnitude more erasures than using WEUs.

6.2 Prototype System Results

Next, we report the performance of Nitro for primary workloads on both COS and TPS systems. We then

Metric (%)	*Trace*	*Extent*	*Nitro WEU Variants*			
		ND, NC	ND, NC	ND, C	D, NC	D, C
COS system						
IOPS	WebVM	251	307	393	532	661
	Homes	259	341	432	556	673
	Mail	213	254	292	320	450
RRT	WebVM	52	54	63	72	78
	Homes	46	49	55	57	62
	Mail	50	53	61	67	72
TPS system						
IOPS	WebVM	93	113	148	198	264
	Homes	90	130	175	233	287
	Mail	56	75	115	122	165
RRT	WebVM	39	41	49	58	64
	Homes	39	42	47	49	54
	Mail	41	44	51	57	61

Table 2: Performance evaluation of Nitro and its variants. We report IOPS improvement and read response time (RRT) reduction percentage relative to COS and TPS systems without an SSD cache. The standard deviation is $\leq$7.5%.

present sensitivity and overheads analysis of Nitro. Note that the cache size for each workload is 2% of the file system size for each dataset unless otherwise stated.

Performance in COS system. We first show how a high read-hit ratio in our Nitro prototype translates to an overall performance boost. We replayed the FIU traces at an accelerated speed to use $\sim$95% of the system resources, (reserving 5% for background tasks), representing a sustainable high load that Nitro can handle. We setup a warm cache scenario where we use the first 16 days to warm the cache and then measure the performance for the following 5 days.

Table 2 lists the improvement of total IOPS (reads and writes), and read response time reduction relative to a system without an SSD cache for all FIU traces. For example, a decrease in read response time from 4ms to 1ms implies a 75% reduction. For all traces, IOPS improvement is $\geq$254%, and the read response time reduction is $\geq$49% for Nitro WEU variants. In contrast, the Extent (ND, NC) column shows a baseline SSD caching system without the benefit of deduplication, compression, or WEU. The read-hit ratio is consistent with Figure 4.

We observe that with deduplication enabled (D, NC), our system achieves consistently higher IOPS compared to the compression-only version (ND, C). This is because finding duplicates in the SSD prevents expensive disk storage accesses, which have a larger impact than caching more data due to compression. Nitro (D, C) achieves the highest IOPS improvement (673%) in Homes using COS. As explained before, a high deduplication ratio indicates that duplicate writes are canceled, which contributes to the improved IOPS. For Mail, the increase of deduplication relative to compression-only version is smaller because small I/Os (29% of I/Os are $\leq$ the 8KB extent size) can cause more reads from disk on the write path, thus negating some of the benefits of duplicate hits in the SSDs.

Compared to extent-based caching, WEU (D, C) improves non-normalized IOPS up to 120% and reduces read response time up to 55%. Compared to WEU (ND, NC), extent-based caching decreases IOPS 13-22% and increases read response time 4-7%. This is partially because extent-based caching increases the SSD write penalty due to small SSD overwrites. From SSD random write benchmarks, we found that 2MB writes (WEU size) have $\sim$60% higher throughput than 8KB writes (extent size), demonstrating the value of large writes to SSD.

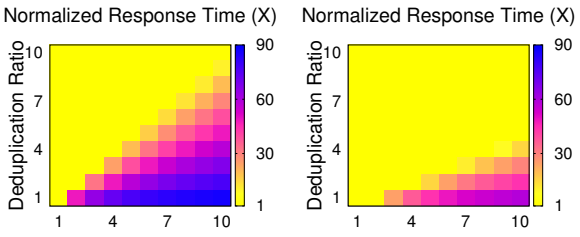
We also performed cold cache experiments that replay the trace from the last 5 days without warming up Nitro. Nitro still improves IOPS up to 520% because of sequential WEU writes to the SSD. Read response time reductions are 2-29% for Nitro variants across all traces because fewer duplicated extents are cached in the SSD.

Performance in TPS system. Nitro also can benefit a TPS system (Table 2). Note that Nitro needs to compute extent fingerprints before performing deduplication, which is computation that can be reused in COS but not TPS. In addition, Nitro cannot leverage a recipe cache for TPS to accelerate read requests, which causes 5-14% loss in read hit-ratio for our WEU variants.

For all traces, the improvement of total IOPS (reads and writes) is $\geq$75%, and the read response time reduction is $\geq$41% for Nitro WEU variants. While deduplication and compression improve performance, the improvement across Nitro variants is lower relatively than for our COS system because storage systems without capacity-optimized techniques (e.g. deduplication and compression) have shorter processing paths, thus better baseline performance. For example, overwrites in existing deduplication systems can cause performance degradation because metadata updates need to propagate changes to an entire file recipe structure. For these reasons, the absolute IOPS is higher than COS with faster read response times. Cold cache results are consistent with warm cache results.

Sensitivity analysis. To further understand the impact of deduplication and compression on caching, we use synthetic traces to investigate the impact on random read performance, which represents the worst-case scenario for Nitro. Note that adding non-duplicate writes to the traces would equivalently decrease the cache size (e.g. multi-stream random reads and non-duplicate writes). Two parameters control the synthetic traces: (1) The ratio of working set size versus the cache size and (2) the deduplication ratio. We vary both parameters from 1 to 10, representing a large range of possible scenarios.

Figure 7 shows projected 2D contour graphs from a 3D plot for (D, NC) and (D, C). The metric is read re-



(a) No compression (b) 2X compression

Figure 7: Sensitivity analysis of (D, NC) and (D, C).

sponse time in COS with Nitro normalized against that of fitting the entire data set in SSD (lower values are better). The horizontal axis is the ratio of working set size versus cache size, and the vertical axis is the deduplication ratio. The bottom left corner (1, 1) is a working set that is the same size as the cache with no deduplication. We can derive the effective cache size from the compression and deduplication ratio. For example, the effective cache size for a 16GB cache in this experiment expands to 32GB with a 2X deduplication ratio configuration, and further to 64GB when adding 2X compression.

First, both deduplication and compression are effective techniques to improve read response time. For example, when the deduplication ratio is high (e.g. $\geq$5X such as for multiple, similar VMs), Nitro can achieve response times close to SSD even when the working set size is 5X larger than the cache size. The combination of deduplication and compression can support an even larger working set size. Second, when the deduplication ratio is low (e.g.$\leq$2X), performance degrades when the working set size is greater than twice the cache size. Compression has limited ability to improve response time, and only a highly deduplicated scenario (e.g. VM boot-storm) can counter a large working set situation. Third, there is a sharp transition from high response time to low response time for both (D, NC) and (D, C) configurations (values jump from 1 to $>$ 8), which indicates that (slower) disk storage has a greater impact on response time than (faster) SSDs. As discussed before, the performance for Nitro in the TPS system is always better than the COS system.

Nitro overheads. Figure 8 illustrates the performance overheads of Nitro with low and high hit-ratios. We performed a boot-storm experiment using a real VM boot trace (§5) synthetically modified to create 60 VM traces. For the 59 synthetic versions, we set the content overlap ratio to 90%. We set the cache size to achieve 0% (0GB) and 100% (1.2GB) hit-ratios in the SSD cache. With these settings, we expect Nitro's performance to approach the performance of disk storage and SSD storage.

In both COS and TPS 0% hit-ratio configurations, we normalized against corresponding systems without SSDs. All WEU variants impose $\leq$7% overhead in response time because extent compression and fingerprint calculation are performed off the critical path. In the 100% hit-ratio scenario, we normalize against a system with all data fitting in SSD without WEUs. WEU (ND, NC) imposes a 2% increase in response time. Compression-only (ND, C) and deduplication-only (D, NC) impose 11% and 6.2% overhead on response time respectively. WEU (D, C) overhead ($\leq$18%) mainly comes from decompression, which requires additional time when reading compressed extents from SSD. Although we are not focused on comparing compression algorithms, we did quantify that gzip achieves 23-47% more compression than LZ4 (our default), which improves the read-hit ratio, though decompression is 380% slower for gzip.

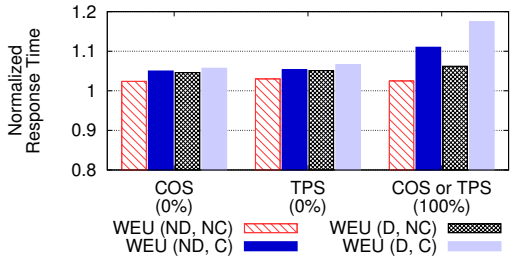


Figure 8: Overheads of Nitro prototypes with the cache sized to have 0% and 100% hit-ratios. Y-axis starts at 0.8. The standard dev. is $\leq$3.8% in all cases.

6.3 Nitro Advantages

There are additional benefits because Nitro effectively expands a cache: improved random read performance in aged COS, faster snapshot restore performance, and write reductions to SSD.

Random read performance in aged COS system. For HDD storage systems, unfortunately, deduplication can lead to storage fragmentation because a file's content may be scattered across the storage system. A previous study considered sequential reads from large backup files [18], while we study the primary storage case with random reads across a range of I/O sizes.

Specifically, we wrote 100 daily snapshots of a 38GB desktop VM to a standard COS system, a system augmented with the addition of a Nitro cache, and a TPS system. To age the system, we implemented a retention policy of 12 weeks to create a pattern of file writes and deletions. After writing each VM, we measured the time to perform 100 random reads for I/O sizes of 4KB to 1MB. The Nitro cache was sized to achieve a 50% hit ratio (19GB). Figure 9 shows timing results for the 1st generation (low-gen) and 100th generation (high-gen) normalized to the response time for COS low-gen at 4KB. For the TPS system, we only plot the high-gen numbers, which were similar to the low-gen results, since there was no deduplication-related fragmentation.

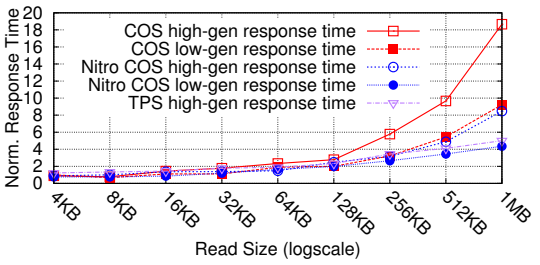


Figure 9: Response time diverges for random I/Os.

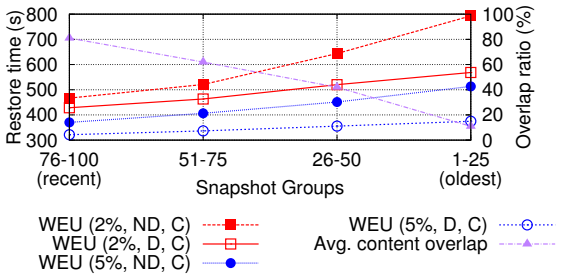


Figure 10: Nitro improves snapshot restore performance.

As the read size grows from 4KB to around 128KB, the response times are stable and the low-gen and high-gen results are close to each other for all systems. However, for larger read sizes in the COS high-gen system, the response time grows rapidly. The COS system's logs indicate that the number of internal I/Os for the COS system is consistent with the high response times. In comparison to the COS system, the performance gap between low-gen and high-gen is smaller for Nitro. For 1MB random reads, Nitro COS high-gen response times (76ms) are slightly faster than COS low-gen, and Nitro COS low-gen response times (39ms) are slightly faster than a TPS high-gen system. By expanding an SSD cache, Nitro can reduce performance differences across random read sizes, though the impact of generational differences is not entirely removed.

Snapshot restore. Nitro can also improve the performance of restoring and accessing standard snapshots and clones, because of shared content with a cached primary version. Figure 10 plots the restore time for 100 daily snapshots of a 38GB VM (same sequence of snapshots as the previous test). The restore trace used 512KB read I/Os, which generate random HDD I/Os in an aged, COS system described above.

We reset the cache before each snapshot restore experiment to the state when the 100th snapshot is created. We evaluate the time for restoring each snapshot version and report the average for groups of 25 snapshots with the cache sized at either 2% or 5% of the 38GB volume. The standard deviation for each group was $\leq$7s. Group 1-25 has the oldest snapshots, and group 76-100 has the most recent. For all cache sizes, WEU (D, C) has consistently faster restore performance than a compression-only version (ND, C). For the oldest snapshot group (1-25) with a 5% cache size, WEU (D, C) achieves a shorter restore time (374s) when deduplication and compression are enabled as compared to the system with compression only (513s). The recent snapshot group averages 80% content overlap with the primary version, while the oldest group averages 20% content overlap, as plotted against the right axis. Clearly, deduplication assists Nitro in snapshot restore performance.

Reducing writes to SSD. Another important issue is how effective our techniques are at reducing SSD writes compared to an SSD cache without Nitro. SSDs do not support in-place update, so deduplication can prevent churn for repeated content to the same, or a different, address. For WebVM and Mail, deduplication-only and compression-only reduces writes to SSD ($\geq$22%), in which compression produces more savings compared to deduplication. In the Homes, deduplication reduces writes to SSD by 39% because of shorter fingerprint reuse distance. Deduplication and compression (D, C) reduces writes by 53%. Reducing SSD writes directly translates to extending the lifespan.

7 Related Work

SSD as storage or cache. Many studies have focused on incorporating SSDs into the existing hierarchy of a storage system [2, 7, 12, 16, 30]. In particular, several works propose using flash as a cache to improve performance. For example, Intel Turbo Memory [21] adds a nonvolatile disk cache to the hierarchy of a storage system to enable fast start-up. Kgil et al. [11] splits a flash cache into separate read and write regions and uses a programmable flash memory controller to improve both performance and reliability. However, none of these systems combine deduplication and compression techniques to increase the effective capacity of an SSD cache.

Several recent papers aim to maximize the performance potential of flash devices by incorporating new strategies into the established storage I/O stack. For example, SDF [25] provides a hardware/software co-designed storage to exploit flash performance potentials. FlashTier [26] specifically redesigned SSD functionality to support caching instead of storage and introduced the idea of silent eviction. As part of Nitro, we explored possible interface changes to the SSD including aligned WEU writes and TRIM support, and we measured the impact on SSD lifespan.

Deduplication and compression in SSD. Deduplication has been applied to several primary storage systems. iDedup [27] selectively deduplicates primary workloads inline to strike a balance between performance and capacity savings. ChunkStash [6] designed a fingerprint

index in flash, though the actual data resides on disk. Dedupv1 [22] improves inline deduplication by leveraging the high random read performance of SSDs. Unlike these systems, Nitro performs deduplication and compression within an SSD cache, which can enhance the performance of many primary storage systems.

Deduplication has also been studied for SSD storage. For example, CAFTL [4] is designed to achieve best-effort deduplication using an SSD FTL. Kim et al. [13] examined using the on-chip memory of an SSD to increase the deduplication ratio. Unlike these systems, Nitro performs deduplication at the logical level of file caching with off-the-shelf SSDs. Feng and Schindler [8] found that VDI and long-term CIFS workloads can be deduplicated with a small SSD cache. Nitro leverages this insight by allowing our partial fingerprint index to point to a subset of cached entries. Another distinction is that since previous deduplicated SSD projects worked with fixed-size (non-compressed) blocks, they did not have to maintain multiple references to variable-sized data. Nitro packs compressed extents into WEUs to accelerate writes and reduce fragmentation. SAR [20] studied selective caching schemes for restoring from deduplicated storage. Our technique uses recency instead of frequency for caching.

8 Conclusion

Nitro focuses on improving storage performance with a capacity-optimized SSD cache with deduplication and compression. To deduplicate our SSD cache, we present a fingerprint index that can be tuned to maintain deduplication while reducing RAM requirements. To support the variable-sized extents that result from compression, our architecture relies upon a Write-Evict Unit, which packs extents together and maximizes the cache hit-ratio while extending SSD lifespan. We analyze the impact of various design trade-offs involving cache size, fingerprint index size, RAM usage, and SSD erasures on overall performance. Extensive evaluation shows that Nitro can improve performance in both COS and TPS systems.

Acknowledgments

We thank Windsor Hsu, Stephen Manley and Darren Sawyer for their guidance on Nitro. We also thank our shepherd Vivek Pai and the reviewers for their feedback.

References

[1] N. Agrawal et al. Design Tradeoffs for SSD Performance. USENIX ATC, 2008.

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

[3] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. ASPLOS, 2009.

[4] F. Chen, T. Luo, and X. Zhang. CAFTL: A Content-aware Flash Translation Layer Enhancing the Lifespan of Flash Memory Based Solid State Drives. FAST, 2011.

[5] C. Constantinescu, J. Glider, and D. Chambliss. Mixing Deduplication and Compression on Active Data Sets. DCC, 2011.

[6] B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. USENIX ATC, 2010.

[7] Facebook Inc. Facebook FlashCache, 2013. `https://github.com/facebook/flashcache`.

[8] J. Feng and J. Schindler. A Deduplication Study for Host-side Caches in Virtualized Data Center Environments. MSST, 2013.

[9] Google LZ4: Extremely Fast Compression Algorithm. Google, 2013. `http://code.google.com/p/lz4`.

[10] W. Huang et al. A Compression Layer for NAND Type Flash Memory Systems. ICITA, 2005.

[11] Kgil, Taeho and Roberts, David and Mudge, Trevor. Improving NAND Flash Based Disk Caches. ISCA, 2008.

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

[13] J. Kim et al. Deduplication in SSDs: Model and Quantitative Analysis. MSST, 2012.

[14] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. *ACM TOS*, 2010.

[15] S. Lee et al. A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation. *ACM TECS*, 2007.

[16] C. Li et al. Quantifying and Improving I/O Predictability in Virtualized Systems. IWQoS, 2013.

[17] C. Li et al. Assert(!Defined(Sequential I/O)). HotStorage, 2014.

[18] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. FAST, 2013.

[19] T. Makatos et al. Using Transparent Compression to Improve SSD-based I/O Caches. EuroSys, 2010.

[20] B. Mao et al. Read Performance Optimization for Deduplication Based Storage Systems in the Cloud. *ACM TOS*, 2014.

[21] J. Matthews et al. Intel Turbo Memory: Nonvolatile Disk Caches in the Storage Hierarchy of Mainstream Computer Systems. *ACM TOS*, 2008.

[22] D. Meister and A. Brinkmann. Dedupv1: Improving Deduplication Throughput Using Solid State Drives (SSD). MSST, 2010.

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

[24] D. Narayanan et al. Migrating Server Storage to SSDs: Analysis of Tradeoffs. EuroSys, 2009.

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

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

[27] K. Srinivasan et al. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. FAST, 2012.

[28] TRIM Specification. ATA/ATAPI Command Set- 2 (ACS-2). `http://www.t13.org`, 2007.

[29] G. Wallace et al. Characteristics of Backup Workloads in Production Systems. FAST, 2012.

[30] Q. Yang and J. Ren. I-CASH: Intelligently Coupled Array of SSD and HDD. HPCA, 2011.

[31] K. S. Yim, H. Bahn, and K. Koh. A Flash Compression Layer for Smartmedia Card Systems. *IEEE TOCE*, 2004.