Can't We All Get Along? Redesigning Protection Storage for Modern Workloads

Yamini Allu, Fred Douglis, Mahesh Kamat, Ramya Prabhakar, Philip Shilane, and Rahul Ugale, *Dell EMC*

https://www.usenix.org/conference/atc18/presentation/allu

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Can't We All Get Along?
Redesigning Protection Storage for Modern Workloads

Yamini Allu Fred Douglis[*] Mahesh Kamat Ramya Prabhakar Philip Shilane Rahul Ugale

Dell EMC

Abstract

Deduplication systems for traditional backups have optimized for large sequential writes and reads. Over time, new applications have resulted in nonsequential accesses, patterns reminiscent of primary storage systems. The Data Domain File System (DDFS) needs to evolve to support these modern workloads by providing high performance for nonsequential accesses without degrading performance for traditional backup workloads.

Based on our experience with thousands of deployed systems, we have updated our storage software to distinguish user workloads and apply optimizations including leveraging solid-state disk (SSD) caches. Since SSDs are still significantly more expensive than magnetic disks, we make our system cost-effective by caching metadata and file data rather than moving everything to SSD. We dynamically detect access patterns to decide when to cache, prefetch, and perform numerous other optimizations. We find that on a workload with nonsequential accesses, with SSDs for caching metadata alone, we measured a 5.7× improvement on input/output operations per second (IOPS) when compared to a baseline without SSDs. Combining metadata and data caching in SSDs, we measured a further 1.7× IOPS increase. Adding software optimizations throughout our system added an additional 2.7× IOPS improvement for nonsequential workloads. Overall, we find that both hardware and software changes are necessary to support the new mix of sequential and nonsequential workloads at acceptable cost. Our updated system is sold to customers worldwide.

1 Introduction

With traditional backups, an application periodically copies the entire contents of a file system into the backup environment, with changes since the last copy added at shorter intervals. These are called *full* and *incremental* backups, respectively [8]. Deduplicating file systems leverage the redundancy across full backups by storing a single copy of data, with the granularity of duplicate detection varying from whole files [2] to individual file blocks [31] or variable-sized "chunks" that are determined on the fly via content characteristics [28, 36]. Leveraging a log-structured file system [32] to store nonduplicate data results in append-only write operations. Nonsequential reads are needed for index lookups and when deduplication results in the physical fragmentation of unique data [20]. However, index lookups can be limited to trade deduplication efficiency to improve performance both for writes and reads by writing some duplicates to improve data locality [13].

There have been recent reports about the impact of evolving workloads on system performance. An article [3] provided an overview of the impact of increasing numbers of small files and higher deduplication ratios (and other changing properties) on the Data Domain File System (DDFS) as a whole, but with relatively few details or quantitative analysis. As an example, garbage collection (GC) was slowed by these changing workloads and a new algorithm was needed [11].

Indeed, GC is not the only aspect of the system that must be rethought to handle modern workloads. While Data Domain was one of the original "purpose-built backup appliances," modern data protection workloads impose very different requirements. These workloads include frequent updates to arbitrary locations in backup files, direct access to individual files rather than the aggregates created by traditional backup applications, and direct read/write access to files in the appliance by applications on other hosts. This last class of usage is particularly demanding, as it generally involves large amounts of **nonsequential I/O (NSIO)**.

Thus, we are at an inflection point where we need to rethink and redesign backup systems to enable *optimized*

[*]Current affiliation: Perspecta Labs

performance for non-traditional data protection workloads with nonsequential accesses for our customers. This goes beyond previous work on improving deduplicating storage by reducing fragmentation of sequentially read data [20], as it strives to provide improved NSIO performance without degrading the performance of traditional, sequential, workloads. These two types of application must coexist regardless of the distribution of workloads between the two categories.

In this paper, we describe the evolution of DDFS to support both traditional and nontraditional workloads based on our experience with deployed systems. Traditional workloads mostly have large sequential writes and reads with low metadata operations. Nontraditional workloads have many small files, more metadata operations, and frequent nonsequential accesses. Our new DDFS design supports higher IOPS for both metadata operations and nonsequential reads and writes and is already used by our customers[1]

We expanded our storage system, which had used only hard disk drives (HDDs), to also use solid-state disks (SSDs). These cache index data, directory structures, deduplicating file recipes, and ultimately file data. Because the capacity of the SSDs is a small fraction of the overall system (e.g., 1%), we must make a number of tradeoffs. For instance, while the index that maps fingerprints to disk location stores information in the SSDs for all chunks, we use a shorter form of the fingerprint that can have occasional hash collisions. We rely on metadata accessed when reading a chunk to provide the full fingerprint for confirmation, and when a mismatch is detected, the full on-disk index is consulted.

In addition, we have made several changes to our software stack. These include dynamic assessment of prefetching and caching behavior based on access patterns; data alignment, using application-specific chunk size; scheduler changes for quality of service; increasing the parallelism of I/O requests to a single file; minimizing writes by queuing metadata updates in memory; and support for smaller access sizes.

Through lab experiments, we demonstrate the impact of these changes on the performance of certain applications. With a NSIO workload, with SSDs for caching metadata, we measured a 5.7× IOPS improvement relative to a system without SSDs. Adding data caching in SSDs, we measured a further 1.7× IOPS increase. Combining SSD caching with software optimizations throughout our system added an additional 2.7× IOPS increase for NSIO workloads. We measured similar factors of reductions in terms of average latency of accesses for both reads and writes. Importantly, performance for traditional workloads running concurrently remained high and even increased (when run separately) due to software improvements. We provide detailed experimental results in §6.

In summary, the main contributions of this paper are:

1. We extend support to modern backup applications, with NSIO access patterns. We ensure our new design for the DDFS software stack benefits both traditional and nontraditional backup and restore tasks.

2. We optimize the file system to better utilize the benefits of flash in our software stack, while minimizing the cost of adding flash by selectively storing metadata on SSDs.

3. Experimental results using our techniques show orders of magnitude improvement in IOPS and reduced latencies in nonsequential workloads. Even traditional backup workloads show 25%-30% improvement in restore throughput because the SSD cache reduces disk accesses. In experiments where both traditional and NSIO workloads execute concurrently, our system maintains high performance for both workloads.

The rest of the paper is organized as follows. §2 provides a brief overview of deduplication in file systems and recent changes in backup applications that motivated us to re-architect our file system design. §3 describes our high-level architecture and design and §4 presents the detailed file system modifications in the DDFS stack. §5 states our experimental platform and workloads used in our study. Detailed experimental results are provided in §6. We discuss related work in §7. §8 concludes our study and discusses future extensions.

2 Background and Motivation

Here we provide an overview of our protection storage and a more detailed discussion of the changes that motivated our architecture modifications.

2.1 Deduplicating Protection Storage

Deduplication is common in commercial products and has been widely discussed including survey articles [30, 35]. Here we provide a brief overview of the specifics of our system; see Zhu, et al. [36] for additional details.

Each file is represented by a Merkle tree [26], which is a hierarchical set of hashes. The lowest level of the tree is file data, and hashes to many chunks[2] are aggregated into a new chunk one level higher in the tree. The fingerprint of that chunk is stored a level higher, and so on. The root of the Merkle tree represents a recipe for a single

[1]The data cache is not yet commercially available.

[2]In the interest of brevity, we shall refer to the unit of deduplication as a chunk even if the system uses fixed-sized blocks.

file, and the filename-to-recipe mapping is managed by a *directory manager*. Chunks are aggregated into write units, called *containers*, which are megabytes in size, and may be compressed in smaller units of tens of chunks.

Thus if a file is read sequentially, the system uses the directory manager to find the root of the Merkle tree, makes its way to the lowest level of metadata, and identifies the fingerprints of potentially thousands of data chunks to access. The chunks themselves may be scattered throughout the system, though there are techniques to alleviate read fragmentation [13,20]. There is an index that maps each fingerprint to a container, and the container has metadata identifying chunk offsets.

The storage system is log-structured [32], so whenever new data or metadata chunks get written, they are added to new containers. Garbage collection is necessary to reclaim free space, which can arise from deleted files removing the last reference to a chunk, as well as extra duplicates that are written due to imperfect deduplication [11].

2.2 Changing Environments

The improvements to our deduplicating storage system were motivated by changes in hardware and in applications [3]. Dramatic increases in the capacity of individual disks meant that a purely disk-based system would not have sufficient input/output operations per second (IOPS) to perform the necessary index lookups to deduplicate effectively. Moving the index to SSDs was a necessary step to improving performance, but it was not sufficient to handle the other changes.

The most extreme requirements on performance derive from two changes in workload. Initially, our systems had to deal with the change from periodic full backups to generating full backups by transferring changes since the last backup with *virtual synthetic* full backups [3,12] and change-block tracking [33] backups. With virtual synthetic and change-block tracking, changes to a backup would be written into protection storage, then a new “full backup” would be created by making a copy of the file metadata with the changes included. This would often be done at intervals of hours, rather than the weekly periodic backups from traditional workloads; thus the amount of metadata in the file system would grow by orders of magnitude, could not be cached in DRAM, and was slow to access on disk. The access patterns also changed from fully sequential writes (full backups, which would write a backup from start to finish on a regular basis) to “incremental” writes that would have monotonically increasing offsets but might skip large regions of a file.

Even greater stress to system performance arose with scenarios where data in protection storage are accessed *in place* after a failure. As an example, a virtual machine (VM) backed by a `vmdk` file might be booted and run from protection storage even while its `vmdk` image is migrated to a primary storage server. Accesses to any data not yet received by primary storage would be served through explicit I/Os from protection storage. It is characterized by nonsequential accesses, with additional sequential accesses introduced by storage migration in the background. Read operations during this period often include access of backup data for browsing and recovery of small files from large backups. Read-write operations from a running VM further stress deduplicating storage due to nonsequential reads and overwrites. This is somewhat analogous to storage vMotion [24], when a virtual disk can be accessed while it migrates. This workload in turn has implications on data formats, deduplication units, and physical devices.

Data formats A number of data protection applications perform transformations on data during the backup process. For example, some legacy backup applications have been described as creating a *tar*-like file concatenating data and metadata from many individual files in primary storage, to create a single backup file in protection storage [22]. It is not feasible to restore an individual file from this large aggregate, so there has been a shift towards backing up individual files in their “native” format. This in turn can lead to millions or billions of individual files, making the performance of namespace operations very important.

Unit of deduplication While content-defined chunking is a well studied topic, there is usually an assumption of little knowledge about the data type. As an example, without application-specific knowledge, variable-sized chunks are generally able to localize the impact of small edits when forming chunks. When application knowledge is available, it can increase efficiency such as deduplicating virtual machine disk images [14] in fixed-size units corresponding to disk blocks. (That is, updates to one part of the file do not shift content in other parts of the file.) More generally, application-specific deduplication must align the unit of deduplication appropriately, whether it is a database record or a block storage system directly performing backups.

Devices In addition to using SSDs to store metadata such as the deduplication index, we need to cache file metadata (the recipes that uniquely identify the individual chunks within a file) and file data blocks themselves. SSD caching, including the implications of retrofitting this to an existing disk-based data protection system, is a focus of this paper.

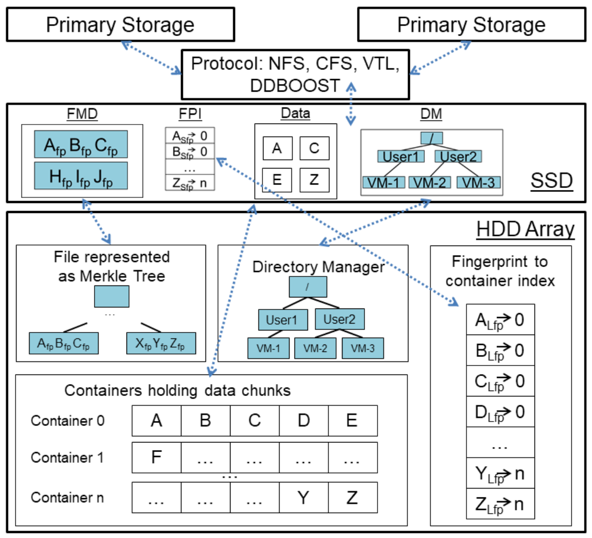


Figure 1: SSD caches: file metadata (FMD), fingerprint index (FPI), data, and directory manager (DM)

Mixed workloads While DDFS was originally designed to support large sequential access patterns, the shift to new workloads does not mean systems no longer have sequential accesses. Instead, there can be mixes of both across and within files. As an example, a system may restore a VM image by sequentially accessing it to create a copy on another system, but during the restore operation the VM image is accessed with reads and writes at more arbitrary locations. There may also be accesses related to ongoing backups relative to this file. DDFS needs to treat the types of accesses differently (*e.g.*, prefetching file data and metadata for the sequential restore) and provide different qualities of service based on resource requirements.

3 Modernizing Protection Storage

To motivate our caching decisions, consider the steps necessary to access data within a file at an arbitrary offset. Figure 1 shows four caches: file metadata (FMD), fingerprint to container index (FPI), data, and directory manager (DM). They are shown as SSD caches, though initially they existed only in DRAM. First, we find the entries in the file's Merkle tree corresponding to the desired data offset (FMD cache). Traversing the tree itself involves a level of indirection as every chunk within the tree is referenced by hash which is translated to a container using a fingerprint index (FPI cache). We then read in the portion of the file tree, which leads us back through the fingerprint index to access data chunks (data cache) that are returned to the client. Please note that the fingerprint index is shown in a simplified form relative to updates discussed in §4.7. Finding the file's top-level information involves a directory structure (DM cache) that is also used for namespace changes.

For largely sequential accesses, the overhead of retrieving various types of metadata will be amortized across many data accesses. For instance, if an application reads 1 MB of fixed-sized 4 KB chunks (256 in total), and the fingerprints of those chunks are all contained in a single chunk in the Merkle tree, then the cost of the directory lookup and the first few levels of the Merkle tree are amortized across 256 chunk reads. If the locality of those chunks is high, the first lookup in the FPI will lead to a container that populates the DRAM FPI for the rest of the 1-MB read.

For random accesses, especially to individual files, each read can result in a DM lookup, on-disk Merkle tree traversals, on-disk FPI lookups, and finally a data access. While caching will not significantly help completely random accesses, any amount of locality can result in substantial improvement.

We therefore use SSD to cache several metadata structures as well as file data. We controlled the costs of our design by using low-cost SSD that totaled 1% of the total hard drive capacity. Our selected SSDs only support three full erasures per day, so our design attempts to minimize writes. At the time of writing this article, SSD costs approximately 8× more per GB than HDD, so adding a 1% SSD cache increases the hardware capacity costs by 8% [1]. While some backup customers appreciate all-flash options, many remain sensitive to costs.

3.1 Caching the File Metadata

While our system attempts to group metadata chunks together, locality can become fragmented for multiple reasons, such as GC repositioning chunks and related files sharing previously written chunks. To decrease the latency for accessing file metadata (FMD) (i.e. the Merkle trees), we cache FMD in flash. Also for NSIO, accessing each data chunk requires accessing a metadata chunk that is unlikely to be reaccessed in the near future. This doubles the number of I/Os needed to serve a client request, so prefetching metadata chunks and caching in flash will decrease overall latency. There are multiple challenges we considered while designing the FMD cache.

We noted that metadata chunks can be of variable size, and not align with a flash erasure unit. We therefore packed metadata chunks into a multi-MB cache blocks and created a caching policy similar to Nitro [18]. Briefly, we maintain a single time stamp per cache block and perform LRU eviction using that time stamp to evict an entire cache block at a time. While LRU is a simple policy, and more advanced techniques [19] could be used, we have found that chunks within cache blocks tend to age at similar rates. We track the amount of data written to the FMD and will throttle new insertions to maintain our long term average of three writes per day times the capacity of the cache.

Importantly, we must determine which metadata chunks to add to the cache, as our capacity is insufficient to cache metadata chunks for all files in the system. 10% of the SSD cache is allocated to the FMD. Rather than simply inserting all FMD, we use admission control to determine what is appropriate to cache (§4).

3.2 Caching the Fingerprint Index

For any given chunk, on the read path the system must map from its unique fingerprint to its location in storage. The fingerprint to container index (FPI) performs that function. Historically, in DDFS the index would be on disk, with a small subset cached in DRAM. Our index design actually requires two I/Os for each access because there are typically two layers to the index. On a read operation, where the FPI mapping was not in the cache, there would be I/O to disk. However, the container that would then be loaded would include metadata for other chunks in the container, and their fingerprints would be cached. Since reads were typically large restore operations, accesses would be sequential and many other fingerprints would be found in the cache. As lower cost, denser hard drives have become available, they have been added to our systems. Unfortunately, IOPS per capacity have decreased for denser drives, and this further motivates the need to use SSD to accelerate NSIO such as fingerprint index accesses.

To support NSIO, the system keeps the entire FPI in SSD, but because space is limited, DDFS makes a concession. Each record stores a short version of each fingerprint in SSD along with the corresponding container and other information. Rather than storing all 20 bytes of a fingerprint, it stores four bytes. In the case of duplicate short fingerprints, only one copy is recorded in the FPI. More details are in §4. Figure 1 labels the full fingerprints as Lfp on disk and the short fingerprints as Sfp in SSD. It is possible the FPI will incorrectly match a query fingerprint based on the first bytes of a short fingerprint in the cache, but this false positive case will be detected. If the needed chunk is not found in the container referenced by the short fingerprint, then the full on-disk index is consulted. Latency is higher in this case, but as it is infrequent, overall performance improves dramatically while controlling SSD costs. FPI occupies 50% of the SSD cache.

For the data locality of traditional backups, for every 1MB external read, we issue an average of eight I/Os to disk where two are for the FPI. When the FPI is moved

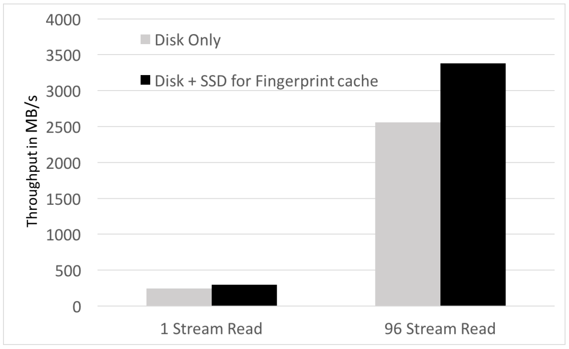


Figure 2: Performance evaluation of a NSIO workload with and without the fingerprint index cache in SSD.

to SSD, we should see a benefit of at least 25% on disk bound systems. For data with bad locality, we will issue multiple FPI lookups per client read, so FPI in SSD would offer even more benefit. In Figure 2, we compare overall throughput of our system when the FPI is in SSD versus only on HDD. We show that having a fingerprint cache in SSD improves restore performance at higher stream counts, when disk is a bottleneck, by up to 32%. More experimental details are provided in §6.

3.3 Caching the Chunks

Once the system knows where to find a chunk, it loads the storage container holding it. With traditional workloads and significant spatial locality, two properties hold that are not true for NSIO workloads:

1. Once accessed, a particular chunk is unlikely to be accessed again unless the same content appears multiple times in the restore stream.
2. Other chunks in the same storage container are reasonably likely to be accessed as well, so the system benefits from caching that container's data and metadata. The container metadata can be used to avoid FPI lookups when locality is high [36].

For NSIO, in contrast, the locality of access within a container may be highly variable, and the reuse of specific data may be more commonplace. For instance, a data chunk might be written and then read, with a gap between the accesses that would be too large for the data to reside in a client or server DRAM cache. The SSD data cache is intended to provide a large caching level to optimize those access patterns, but it needs to dynamically identify what patterns it encounters.

On a data miss for sequential reads, we load the desired chunk as well as the following chunks that may be accessed. This helps to warm our cache and improves access times. We avoid loading chunks during writes except when read-modify-write operations are necessary. Techniques to identify such cases and modifications to DDFS are described in §4. 35% of the SSD cache is reserved for data chunks.

3.4 Caching Directories

The directory manager (DM) manages the mapping from file paths to Merkle trees. Thus, the first time a file is opened, the system must access this mapping to find the root of the tree. For large files such as VM images that are opened once and then accessed over time, the cost of the DM lookup is insignificant (once for a large file), but if there are numerous files to open (such as the result of backing up a file system as individual files); this cost can be a significant performance penalty.

In DDFS, data for DM resides on HDD, but a full copy is now cached in SSD for performance. Since such a cache is straightforward, our experiments focus on applications that are not namespace-intensive, so we do not consider the DM cache further. DM is allocated 5% of the SSD cache.

4 File System Modifications to Support Nonsequential Workloads

Our goal is to enable faster accesses for new workloads while continuing to support traditional sequential backup/restore workloads without performance degradation. Besides the flash caches described previously, numerous changes were needed in the file system to support NSIO. We begin by presenting our technique for identifying the type of client access, which determines if optimizations are applied. We then describe the most important file system changes.

4.1 Detecting Workload Types

To decide whether NSIO processing is needed, the incoming I/O requests must be analyzed to determine the type of access. Defining “sequential” is itself a challenge, as access patterns may not be strictly *sequential* even if they are *predictable* [17].

The access pattern detection algorithm partitions large files into regions and keeps a history of recent I/Os (specifically, 16 I/Os) per region as shown in Figure 3. There are two kinds of detection to check for data sequentiality and access patterns.

By default, all incoming I/Os are assumed to be sequential until there is enough history of previous I/Os to check for other types. Once the history buffer is full, if a new I/O is not within a threshold distance of one of the previous 16 I/Os, it is considered nonsequential.

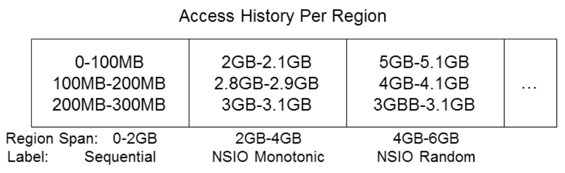


Figure 3: Access history for three regions of a file, labeled sequential, NSIO monotonic, and NSIO random.

The reason for comparing with several past accesses is to avoid detecting re-ordered asynchronous I/O operations from a client as NSIO. By keeping multiple regions of access patterns within a file, we allow combinations of sequential and NSIO accesses to the same file to coexist without NSIO patterns hiding the existence of simultaneous sequential access. One example of this is NSIO from accessing a live VM image while simultaneously performing vMotion; another is the result of reordering of asynchronous I/O operations on a client. Our region size is a minimum of 2GB and grows to maintain at most 16 regions per file. The memory required for tracking a file is $\leq$3KB.

Referring to Figure 3, besides sequential I/O, we also label two variants of NSIO: NSIO monotonic and NSIO random. Monotonic refers to accesses that are to the same or non-consecutive increasing offsets. Random refers to accesses that do not have a discernible pattern. The monotonic pattern is particularly common when a backup client generates a synthetic full backup by first copying the previous full backup (an efficient metadata operation in deduplicated storage) and then overwrites regions at increasing offsets in the file. Distinguishing NSIO monotonic from random patterns allows us to implement different caching and eviction methodologies.

4.2 Prefetching Content

One of the uses of identifying accesses based on history per region is to prefetch and cache content. Importantly, we also avoid caching content that will not be reused. Our options are to load data into DRAM for immediate use or load into SSD if reuse is expected.

Specifically, when access patterns are labeled as sequential or NSIO monotonic, we can prefetch and load into DRAM, because we know the data or metadata will be used soon and SSD caching is unlikely to provide further benefit. For NSIO random I/O, we prefetch into SSD because we need to cache most of the active data set, which is larger than DRAM, to get the benefit of caching. In order to warm-up the cache sooner for every file with NSIO random I/O, we first load 128KB around the current I/O (e.g. 8KB) for caching in SSD since it may be reused, and there is little additional latency for

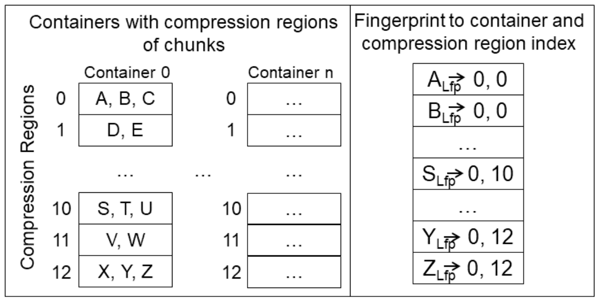


Figure 4: The fingerprint index is updated to map from fingerprint to container and compression region. The structured is presented in a simplified form.

loads of that size in our system.

4.3 Direct Read of Compression Regions

As described in §3, accessing a region of a file involves identifying the fingerprint representing that chunk from the file recipe, checking the fingerprint index for the corresponding location on disk, and then reading the chunk. For sequential accesses, we implemented an optimization in the early versions of the file system. The fingerprint index maps to container, so we read in the container's metadata region into RAM, which consists of a list of fingerprints for chunks within each compression region. We then determine which compression region to read and decompress to find the needed chunk. For sequential (or nearly sequential) accesses, we typically find most needed fingerprints in the RAM cache without the need to query the fingerprint index [36].

While this previous optimization dramatically reduces fingerprint index accesses for sequential I/O, it is inefficient for NSIO. A client's nonsequential read requires a fingerprint index read, container metadata read, and compression region read, *i.e.* three reads in total. Because future accesses are unlikely to remain within the same container, there is no amortization of reading a container's metadata. To remove the container metadata read for NSIO cases, we adjusted our fingerprint index to map from fingerprint to a compression region offset and size *within* a container (Figure 4). This allows us to perform direct compression region reads without first reading in container metadata, reducing the number of accesses from three to two. We dynamically decide based on access patterns whether to read compression region metadata or not.

To reduce SSD space for the FPI entries, we limit the entry size to twelve bytes. Four bytes come from the shortened fingerprint. Four bytes are used for the container ID, which is sufficient since it is relative to the lowest container ID within the system. The remaining four bytes are used to describe the compression region within the container with bits allocated to the compression region offset and size within the container as well as internal uses. To reduce the number of bits required, compression regions are written at sector boundaries. When indexing a fingerprint, we use a hash of the first eight bytes to select a FPI bucket. In combination with the four bytes short fingerprint, the collision rate is below 0.01%.

4.4 Higher Concurrency with Queue Changes

Applications directly accessing files from backup storage have high performance requirements, and latency is an important aspect, so I/Os must be processed as soon as they enter the file system. Unlike traditional workloads that tend to be highly sequential, with one client I/O effectively dependent on earlier I/Os to complete, NSIO has a greater need and opportunity for parallelism. For NSIO, FMD required to process the I/O may not be in memory and will require disk I/Os. Requests that are dependent on the same FMD will be processed serially in the order received; however, requests that do not require the same FMD are processed in any order and in parallel. Once the required FMD is loaded for any I/O, that request is given priority for further processing to avoid starvation. Apart from issuing parallel I/Os for FMD on disk, fingerprint lookups for multiple reads within the same file take place in parallel for NSIO.

4.5 Adjusting the Chunk Size to Improve Nonsequential Writes

For traditional large backup files, variable chunking achieves better deduplication than fixed-size chunks because it better identifies consistent chunks in the presence of insertions and deletions [34, 36], which we refer to as *shifts*. For new use cases that have block-aligned writes, such as change block tracking for VMs, shifts do not occur, and fixed-sized deduplication is effective [14]. Although variable-sized chunking has better deduplication, the performance gains achieved with fixed-size chunks outweighs the deduplication loss [27].

Based on customer configuration or backup software integration, we label workloads that will benefit from fixed-sized chunks. This simplifies the write path, as we do not need to find chunk boundaries or perform a read-modify-write for a partial overwrite of a chunk. For certain applications, such as VMs and databases, the block size is predetermined, and we set our chunk size accordingly for further efficiency.

4.6 Delayed Metadata Update

Switching to fixed-sized chunks has the added benefit that it allows for more efficient updates of file recipes during nonsequential writes. Traditionally, we would need to read in portions of the file recipe to provide fingerprints for chunks that must be read before being modified. With fixed-size chunks, we never need to modify existing chunks as they are simply replaced. We do need to update the file recipe to reference new chunks, but this can be delayed until sufficient updates have been accumulated. Since our chunk references are 28 bytes, 1MB of non-volatile memory can buffer references for nearly 300MB worth of logical writes.

4.7 Selective Fingerprint Index Queries

While our file system is designed to perform deduplication by identifying redundant chunks, we may choose to skip redundancy checks to improve performance [3]. We have found that nonsequential writes tend to consist of unique content. So to avoid fingerprint index queries that are unlikely to find a match, we disable querying the fingerprint index for small nonsequential writes ($<$128KB). Any duplicate chunks written to storage will be removed during periodic garbage collection [11].

4.8 Quality of Service and Throttling

DDFS has a quality of service (QoS) mechanism that assigns shares for external and internal workloads such as backup, restore, replication and garbage collection. These shares are used in the CPU and disk scheduler to provide QoS for the workloads. NSIO can happen as part of backup or restore, so we made changes to further split the backup and restore workload shares into sequential and nonsequential shares. The number of shares assigned to these workloads is tunable based on a customer's desired system behavior. By default, the shares for NSIO workloads are kept at 20% so as to not impact other critical workloads, but as reads and writes on backups during a restore becomes commonplace, shares may need to be increased for NSIO.

On non-uniform memory access architectures, jobs pertaining to a task are assigned a particular CPU for cache locality. Our earlier implementation used round robin assignment of jobs to CPUs. However, the resource requirements between NSIO workloads vary greatly and hence a simple round robin is insufficient. In the latest version of DDFS we have changed this assignment to least-loaded CPU instead. NSIO performance greatly depends on read performance, so we have modified our I/O scheduler to avoid read starvation and provide higher priority for read requests.

With all of the changes to increase NSIO performance, accepting more I/Os in parallel at the protocol layer usually improves overall performance. However, beyond a limit, further client requests will cause RPC timeouts, and hence I/O throttling per workload type becomes important. Based on the type of workload and the average latency, we have implemented an edge throttling mechanism where the protocol layer can query the subsystem health and insert queue delays to dynamically change the number of client accesses supported.

5 Experimental Methodology

This section describes our experimental methodology and the test environment including the system configuration and workloads used. All our results are measured on a Data Domain DD9800 [10] configured with maximum capacity. It has 60 Intel(R) Xeon(R) CPU E7-4880 v2 processors @2.50GHz, with 775GB DRAM, 8 10Gb network ports, 10.9TB SSD, and 1008 TB disk storage across 6 shelves with 4TB HDDs. Each shelf has between 1 and 4 packs, with 15 HDDs per pack. There are 20 spare HDDs. We produce accesses to the DD9800 using up to 8 clients running Linux version 2.6.32 with Intel(R) Xeon(R) CPU E5-2620 with 2.00GHz cores, 64 GB of memory, and a 10Gb Ethernet card.

We primarily use traditional and NSIO workloads for our measurements. Performance numbers for traditional backup and restore are reported using an in-house synthetic generator that randomly creates first generation backups for each stream and then modifies following generations with deletions (1%), shuffles (1%), and additions (1%) [7]. Across clients, the total size of first generation backups is 3TB, and metadata is approximately 1% of the data size. We wrote every 5th generation, though we allowed changes to accumulate in memory even for unwritten generations. This workload has 100% sequential read/write accesses to data for all generations. However, metadata accesses are NSIO. We report throughput numbers as the average of generations 41 and 42.

For a NSIO workload, we use the industry standard FIO benchmark [6] to simulate large sequential and NSIO reads as well as small NSIO reads and writes. We also present results when accessing 32 100GB VM images with mixtures of sequential I/O and NSIO, as described in each experiment. While customer VMs often share content, in order to reduce factors affecting our experiments, we have confirmed that there was no potential deduplication within or across the images. Here we report performance numbers in terms of IOPS and average latency. Unless otherwise noted, experiments were performed on an isolated system configured with fixed-sized chunks and without other read/write operations or background tasks such as garbage collection or replication.

All metadata fit within SSD without the need for admission control.

We show the benefits of our hardware and software optimizations in the following experiments. Each data point in our experiments with traditional backup workload was collected in runs that lasted multiple days and on data sets that were aged up to 42 generations of backups. Multiple clients were used to generate the backup workload, and results are averaged across clients. Each data point in our experiments with NSIO workload was collected by measuring the average performance (IOPS and latency) with at least three runs, and the standard deviation of results is $<1.5\%$ in all cases.

6 Evaluation

We begin by exploring the impact of caching metadata and data as well as software optimizations within DDFS for NSIO workloads. Then we investigate the impact on traditional, sequential workloads using different protocols. Finally, we study the sensitivity to different read/write ratios in NSIO workloads and the impact when storage vMotion occurs in parallel.

6.1 Caching and Software Optimizations

We investigate the impact of progressively adding metadata and data to a SSD cache as well the value of software optimizations in terms of average IOPS (Figure 5(a)) and latency (Figure 5(b)). Metadata include the FPI, FMD, and DM caches, though our tests do not perform directory operations. To avoid direct comparisons, the experiments with optimizations disabled are separated by a dashed line in each set of bars

We vary the number of VM images accessed from 1 to 32 and plot the average IOPS and latency for a NSIO workload. In these experiments, we study a read-only workload, and each VM is issued a maximum of 8 concurrent I/Os. When the flash cache is disabled, each external I/O will translate to six internal I/Os to disk. This includes two I/Os for FPI lookup to then perform one I/O for file metadata. From the file metadata, we have the chunk fingerprint and then perform two I/Os for FPI lookup and one I/O to load the data. The total number of HDD IOPS available on the test system is 24K. So, the theoretical achievable client IOPS when the cache is not available would be 4,000. Software optimizations are enabled except in one set of runs.

We see in the experiment with 32 VMs and the cache disabled, we achieve 3,200 IOPS with a latency of 35ms. When we enable the caching of metadata, every external NSIO will result in one I/O to HDD for data. We show that we can achieve 28K IOPS in a 32 VM experiment, with an average latency of 10ms. When both data and metadata are in the flash cache, IOPS are only limited by the data set size we can cache. On the test system, we can cache 100% of the data for up to 24 VMs and 75% of the data with 32 VMs. We achieve peak performance of 57K IOPS for 24 VMs with a cache hit ratio of 95%. The overall latency stays under 5ms even at peak IOPS.

We next consider the benefit of software optimizations (§4) to improve NSIO performance. Results show that using a SSD cache for NSIO without software changes would limit us to a peak NSIO performance of 20K IOPS, compared to 56K IOPS when software enhancements are enabled. Similarly, even when data and metadata are cached, latency decreases from 13ms to 5ms with the addition of software optimizations.

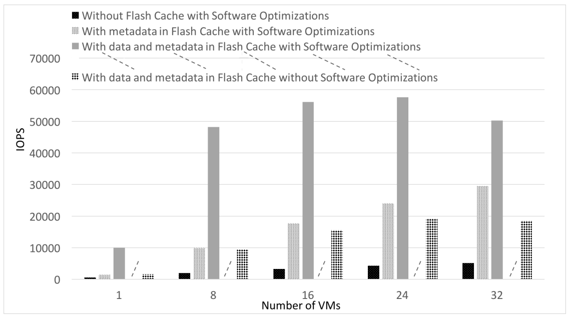
With a small cache and high churn in application workload, some portion of the I/Os will be serviced from disk. Our software optimizations remove unnecessary I/Os to disk (§4.2, §4.3, and §4.5), increase parallelism (§4.3), and improve the I/O scheduler (§4.8). With these changes, we are able to to achieve high NSIO IOPS and maintain a low latency with a SSD cache sized at 1% of the total system capacity.

6.2 Traditional and NSIO Workloads

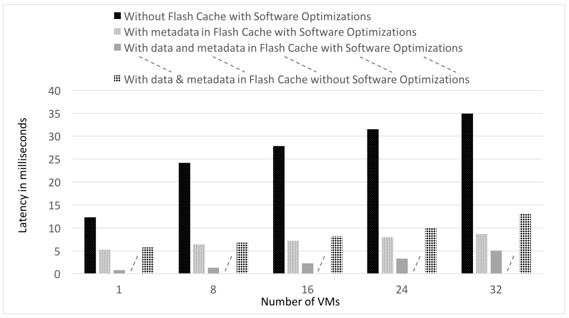
In this experiment, we evaluate both traditional and NSIO workloads running concurrently to measure the impact on traditional workloads. We run both workloads through NFS and DDBOOST protocols. DDBOOST is our proprietary protocol where segmenting and fingerprinting of data is offloaded to backup clients and only changed data is sent across the network [12]. DDBOOST performance is typically higher than NFS because deduplication reduces the amount of data transferred, and the backup server has fewer computational demands.

In this experiment, we throttle NSIO workloads on 32 VMs to a total of 10K IOPS. 2.4TB of the 3.2TB data set fit in the data cache. In Figure 6, we measure the performance of 96 streams of backup and restore workloads while varying the protocol and the fraction of reads versus writes of the NSIO workload. A 100% read NSIO workload is possible when the client writes are redirected to primary storage during a recovery operation. 70% reads are common in other recovery use cases where both writes and reads are directed to backup storage. Read/write numbers represent restore and backup performance for high-generation backups with an equal split of 48 backups and 48 restores.

Considering the difference between NFS and DDBOOST, we find the expected result that DDBOOST has higher overall throughput because of offloading tasks to clients. Across protocols, backup and restore performance is not degraded more than than 10% when NSIO runs in parallel, though there is greater impact



(a) IOPS



(b) Latency

Figure 5: Average IOPS and latency as caching and software optimizations are varied. Up to 24 VMs, 100% of the data can be cached. Caching decreases to 75% for 32 VMs. Experiments without software optimizations are separated from the rest by a dashed line.

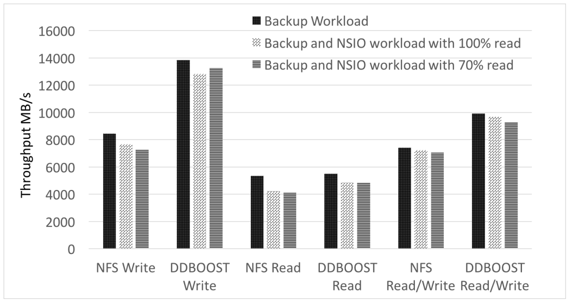


Figure 6: Traditional backup workloads have a 10% degradation when NSIO workloads are added with the DDBOOST protocol outperforming NFS.

when NSIO includes writes.

When more NSIO performance is required than the sustained IOPS specified for the product, a system level QoS parameter (§4.8) allows users to choose the amount of impact on traditional workloads they find acceptable to further increase NSIO performance. Though not shown due to space limitations, we experimented with varying the QoS share allotted to NSIO versus sequential workloads. As the share for NSIO increased from 25% to 50%, IOPS increased by 32%. Increasing the share from 50% to 75% increased IOPS 18% more. When NSIO was allocated 100% of the resources, IOPS increased an additional 125% due to the complete removal of sequential I/O interference.

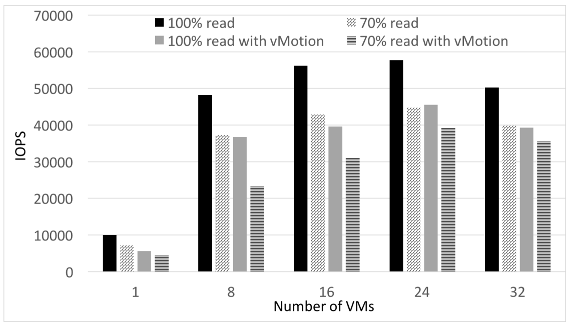
6.3 Performance during Restores

Some backup applications provide a Instant Access/Instant Restore feature where an application may be able to perform read/writes from the backup copy while a restore takes place. This feature may expose a read-only copy of a backup image for the applications to access while redirecting any writes to a write log typically located on primary storage. We simulate this workload using 100% NSIO reads. Other backup applications expose a read/write copy and send both reads and writes from the application to the exposed copy. This is simulated using a 70/30% reads/write NSIO workload. While a VM image is being accessed, backup applications also offer an option to perform storage vMotion of the VM back to primary storage. This workload is simulated by issuing sequential reads on the same VM image on which NSIO is taking place.

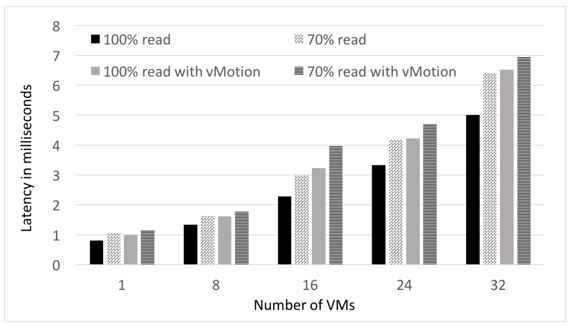
Figures 7(a) and 7(b) show an experiment where NSIO activity takes place with either 100% reads or 70/30% read/writes. With 24 VMs we see a peak of 56K IOPS and under 4 ms of latency with 100% reads. For the 70/30% read/write mix, we see a peak of 44k IOPS at 24 VMs where the cache gets nearly 95% hits. We also show that when vMotion on the same VM takes place, IOPS for NSIO drop by at most 20% and achieves a peak of 45K IOPs for 100% reads. At 70/30% read/write with vMotion, we achieve 40K IOPS. The overall result is acceptably high performance NSIO performance while vMotion takes place.

6.4 Fingerprint Cache Impact on Backup and Restore Workloads

For traditional backup workloads, even with software optimizations, disk I/O becomes a bottleneck. We previously presented experimental results for placing a FPI cache in SSD in Figure 2. For this test, we limited the total IOPS available on the test system to 4,200 by using only four disk groups, the smallest configuration possible. Other metadata and data accesses may still go to HDD, so the overall throughput improvement has many components besides fingerprint access speed. With a high stream count of 96, overall throughput increases with the SSD cache by up to 32%, which corresponds to the fraction of I/O that can be satisfied by the FPI cache in SSD.



(a) IOPS



(b) Latency

Figure 7: Average IOPS and latency for NSIO read/write access to VMs, with and without vMotion in parallel.

7 Related Work

A recent article [3] described the changes to data protection workloads and some of the changes DDFS made to address them. However, the changes were described very generally, focusing on qualitative issues but not quantitative ones. As an example, the article mentioned the addition of SSDs, short fingerprints in SSD, and selectively writing duplicates, but there were few implementation details and no experiments. While we have focused on improvements to backup storage for NSIO, backup software drives most of the client-initiated workloads [4, 8]. The work on improving garbage collection enumeration performance [11] to handle high deduplication rates and numerous individual files provided detailed performance measurements, but that effort is largely orthogonal to the improvements for NSIO described here.

SSD-assisted deduplication has taken many forms. DedupeV1 [25] and ChunkStash [9] were two early systems that moved the fingerprint index into SSD to improve performance. ChunkStash used Cuckoo Hashing [29] to reduce the impact of hash collisions, something we have not found to be a significant performance issue. PLC-cache [23] categorized deduplicated chunks by popularity to determine what to cache. Nitro [18] presented a technique for caching and evicting data chunks in large units to SSD to improve performance while reducing SSD writes, which influenced our metadata and data cache design. Kim et al. [16] modeled deduplication overheads and benefits within SSD and then accelerated performance with selective deduplication against recently written fingerprints. We view the contribution of our work as lessons learned from a deployed storage system pertaining to caching, prefetching, and scheduling, and not simply the addition of SSDs.

While there have been multiple papers regarding sequential write and read performance for deployed deduplicated storage products [5, 15, 20, 21], there has been little discussion of nonsequential workloads. Discussing the architectural changes needed to support both sequential and NSIO workloads in deduplicated storage will hopefully drive further research.

8 Conclusion and Future Work

New workloads for backup appliances and denser HDDs have placed demands on backup storage systems. DDFS has had to evolve to support not only traditional workloads (full and incremental backups with occasional restores) but also newer nonsequential workloads for thousands of customer deployments. Such workloads include direct access for reads and writes in place, as well as other workload changes such as storing individual files and eschewing periodic full backups. Additionally, traditional and newer workloads must *peacefully coexist* within the same product.

Because of the cost difference between SSDs and disk, we have chosen to cache a limited amount of metadata and file data in SSD rather than moving the entire system to SSD. We demonstrate that these caches not only improve NSIO by up to two orders of magnitude, but our system can also simultaneously support traditional workloads with consistent performance. In summary, improvements to our software and the addition of SSD caches allow DDFS to support both new and traditional workloads.

In the future, we expect NSIO workloads to become more common as customers increase the frequency of backups. In combination with decreasing SSD prices (though likely still more expensive than HDD), it may become worthwhile to increase our SSD cache to include most metadata and a larger fraction of active data. We will need to revisit our software design as bottlenecks shift between I/O and CPU.

Acknowledgments

We would like to acknowledge the contributions of the DDFS team including: Uday Kiran Jonnala, Sirisha Kaipa, Vrushali Kulkarni, Shuang Liang, Valiveti Narasimha, Shantanu Patwardhan, Balaji Subramanian, Pradeep Thomas, Satish Vishwanathan, Grant Wallace, Sailu Yallapragada, and Sean Ye. We appreciate the feedback of our shepherd Xiaosong Ma and the anonymous reviewers.

References

[1] Pc part picker price trends. `https://pcpartpicker.com/trends/price/internal-hard-drive/#storage.7200.6000000`. Accessed: 5-7-2018.

[2] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 1–14.

[3] Allu, Y., Douglis, F., Kamat, M., Shilane, P., Patterson, H., and Zhu, B. Backup to the future: How workload and hardware changes continually redefine Data Domain file systems. *Computer 50*, 7 (2017), 64–72.

[4] Amvrosiadis, G., and Bhadkamkar, M. Identifying trends in enterprise data protection systems. In *USENIX Annual Technical Conference (ATC'15)* (July 2015).

[5] Aronovich, L., Asher, R., Bachmat, E., Bitner, H., Hirsch, M., and Klein, S. T. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009).

[6] Axboe, J. FIO. `http://git.kernel.dk/fio.git`, 2018. Retrieved Feb 1, 2018.

[7] Botelho, F. C., Shilane, P., Garg, N., and Hsu, W. Memory efficient sanitization of a deduplicated storage system. In *USENIX Conference on File and Storage Technologies (FAST'13)* (Feb 2013).

[8] Chervenak, A., Vellanki, V., and Kurmas, Z. Protecting file systems: A survey of backup techniques. In *Joint NASA and IEEE Mass Storage Conference* (1998), vol. 99.

[9] Debnath, B. K., Sengupta, S., and Li, J. Chunkstash: Speeding up inline storage deduplication using flash memory. In *USENIX Annual Technical Conference (ATC'10)* (2010).

[10] Dell EMC. Data domain dd9800 specification sheet. `http://www.emc.com/collateral/specification-sheet/h11340-datadomain-ss.pdf`, 2017. Retrieved Feb 1, 2018.

[11] Douglis, F., Duggal, A., Shilane, P., Wong, T., Yan, S., and Botelho, F. C. The logic of physical garbage collection in deduplicating storage. In *15th USENIX Conference on File and Storage Technologies (FAST'17)* (2017).

[12] Douglis, F., Huber, A., Lewis, D., and Traylor, R. Experiences with a distributed deduplication API. In *Massive Storage Systems and Technologies (MSST'17)* (2017), IEEE.

[13] Fu, M., Feng, D., Hua, Y., He, X., Chen, Z., Xia, W., Huang, F., and Liu, Q. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *USENIX Annual Technical Conference (ATC'14)* (2014).

[14] Jin, K., and Miller, E. L. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), SYSTOR'09, ACM.

[15] Kaczmarski, M., Jiang, T., and Pease, D. A. Beyond backup toward storage management. *IBM Systems Journal 42*, 2 (2003), 322–337.

[16] Kim, J., Lee, C., Lee, S., Son, I., Choi, J., Yoon, S., Lee, H.-u., Kang, S., Won, Y., and Cha, J. Deduplication in ssds: Model and quantitative analysis. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on* (2012), IEEE.

[17] Li, C., Shilane, P., Douglis, F., Sawyer, D., and Shim, H. Assert(!Defined(Sequential I/O)). In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2014).

[18] Li, C., Shilane, P., Douglis, F., Shim, H., Smaldone, S., and Wallace, G. Nitro: a capacity-optimized ssd cache for primary storage. In *USENIX Annual Technical Conference* (June 2014), pp. 501–512.

[19] Li, C., Shilane, P., Douglis, F., and Wallace, G. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th International Middleware Conference (Middleware'15)* (Dec. 2015).

[20] Lillibridge, M., Eshghi, K., and Bhagwat, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *USENIX Conference on File and Storage Technologies (FAST'13)* (Feb 2013).

[21] Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., and Camble, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *USENIX Conference on File and Storage Technologies (FAST'09)* (2009).

[22] Lin, X., Douglis, F., Li, J., Li, X., Ricci, R., Smaldone, S., and Wallace, G. Metadata considered harmful... to deduplication. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2015).

[23] Liu, J., Chai, Y., Qin, X., and Xiao, Y. Plc-cache: Endurable ssd cache for deduplication-based primary storage. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on* (2014), IEEE.

[24] Mashtizadeh, A., Celebi, E., Garfinkel, T., and Cai, M. The design and evolution of live storage migration in VMware ESX. In *Usenix Annual Technical Conference (ATC)* (2011), vol. 11, pp. 1–14.

[25] Meister, D., and Brinkmann, A. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (2010), IEEE, pp. 1–6.

[26] Merkle, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology CRYPTO'87* (1988), Springer.

[27] Meyer, D., and Bolosky, W. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2011), pp. 229–241.

[28] Muthitacharoen, A., Chen, B., and Mazières, D. A low-bandwidth network file system. In *18th ACM Symposium on Operating Systems Principles* (2001), SOSP'01.

[29] Pagh, R., and Rodler, F. F. Cuckoo hashing. In *European Symposium on Algorithms* (2001), Springer, pp. 121–133.

[30] Paulo, J., and Pereira, J. A survey and classification of storage deduplication systems. *ACM Computing Surveys 47*, 1 (2014).

[31] Quinlan, S., and Dorward, S. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02.

[32] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992).

[33] VMware Inc. VMware vSphere Storage APIs – Data Protection (formerly known as VMware vStorage APIs for Data Protection or VADP) FAQ (1021175), Oct. 2016. `https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1021175`.

[34] Wallace, G., Douglis, F., Qian, H., Shilane, P., Smaldone, S., Chamness, M., and Hsu, W. Characteristics of backup workloads in production systems. In *USENIX Conference on File and Storage Technologies (FAST'12)* (2012).

[35] Xia, W., Jiang, H., Feng, D., Douglis, F., Shilane, P., Hua, Y., Fu, M., Zhang, Y., and Zhou, Y. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE 104*, 9 (Sept. 2016).

[36] Zhu, B., Li, K., and Patterson, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *USENIX Conference on File and Storage Technologies (FAST'08)* (Feb 2008).