Experiences with a Distributed Deduplication API

Fred Douglis, Andrew Huber, Donna Lewis, Rachel Traylor
firstname.lastname@dell.com

Dell EMC

Abstract

DD BOOST™ is an API to support distributed deduplication, with clients performing chunking and fingerprinting and a deduplicating storage server determining the uniqueness of each chunk. By suppressing the transmission of duplicates, DD BOOST reduces network load; by pushing the pre-processing to clients, the server offloads computation and reduces memory requirements. With *virtual synthetic* writes that explicitly direct a server to interleave existing data with new data, performance can be improved still further. The net result is a dramatic increase in throughput compared to writing content to the server via NFS, with a median reduction in network traffic of 90-99%. This paper reports empirical results from customer deployments of DD BOOST over several years, as well as lab benchmarks.

I. Introduction

Deduplication is a common technique in the storage industry, particularly in the context of file backup [1]. Deduplicated storage permits content that appears multiple times to refer to a single instance of the repeated content. This saves space within the storage system, especially when the same content is written many times (as backups are), but there are other ramifications from this transformation. For example, deduplication can cause poor locality when reconstructing the original data, because a file consists of references to individual unique data units that have been written over a prolonged interval [2], [3]. The degraded locality can impact deduplication write performance as well when the index of unique chunks is dispersed on disk.

In a sense, deduplicated storage was an outgrowth of earlier work on deduplicated network transfers. *Rsync* would have the sender compute weak and strong hashes of blocks in a file and the receiver compute a rolling checksum to see if the weak hash matched over any contiguous region of that size; if the strong hashes also matched, the block could be copied from the receiver rather than sent on the network [4]. Spring and Wetherall devised a mechanism to cache recently transferred data and replace repeated instances of previous network transfers with pointers into the recent history [5], and the Low-Bandwidth File System (LBFS) used a similar technique to avoid transferring file blocks already known to the recipient [6]. Content would be broken into *chunks* based on characteristics of the data, then *fingerprinted* with a collision-resistant hash to uniquely identify each chunk. LBFS did not deduplicate data at the storage layer, because its emphasis was on optimizing network transfers rather than disk space. Venti [7] was the first storage system to deduplicate at a level smaller than a file, using hashes of fixed-sized file blocks.[1]

Commercial deduplication systems focused for years on *backup* workloads, because they were intended to be drop-in replacements for tape-based backup systems. With tapes, the lack of random access meant that unless one periodically copied the entire contents of the file system onto a consolidated set of tapes, a full restore could require reading an arbitrary number of past tapes; hence it led to the model of periodic *full* backups followed by a small number of *incremental* backups until the next full [8]. By moving backups to magnetic disk but deduplicating repeated content, disk-based backup systems could provide better performance and ease of management than tape at comparable cost [9]. Variable-sized chunks were more effective than fixed-sized file blocks, because content within a backup stream was likely to shift.

However, the full content would be sent over the network to the storage server even if it would be identified as duplicates of content that is already stored. In addition, the more clients sending backups to the storage server, the higher the load on the server (processing, for chunking and fingerprinting the content streams; DRAM and NVRAM for buffering content as it's processed).

Our DD BOOST library enables applications, such as backup software, to perform much of the work of deduplicating storage on *clients* rather than Dell EMC Data Domain™ *servers* [10]. Rather than simply mounting the deduplicating storage via a standard NAS protocol such as NFS, a DD BOOST-enabled client writes data through a library that buffers content, performs content-defined chunking, fingerprints the chunks, and sends just the fingerprints to the server. The server identifies which chunks are not already stored, if any, and the client library compresses and transfers the new content. Thus network loads are reduced to the extent that the data is deduplicated and compressed, and the processing to chunk and fingerprint duplicates is left to the clients. DD BOOST, which has been deployed commercially since 2010, has been found to decrease network traffic by about one order of magnitude. Shifting the initial fingerprint computation to the clients allows the server to support additional clients with the same resources: in lab experiments, we find the peak one-minute CPU utilization on the server is also reduced by an order of magnitude in DD BOOST compared to NFS.

Virtual synthetic writes are another option for backup

[1] Typically the distinction between *blocks* and *chunks* is whether they are fixed in size or determined by the content [1].

clients. They mix writes of new data with directives to the backup server to copy existing data by reference; i.e., rather than identifying and removing duplicate data, they instantiate duplicates through low-overhead operations. For instance, an update to 1GB of a 1TB file system might result in writing 1GB but creating a new "full backup" on the server with a new copy of the other 999GB. (See §II-B for details.) This can result in orders of magnitude reduction of the data that would be necessary for a full backup to be transferred in its entirety.

In the rest of this paper, we provide additional background on DD BOOST (§II), describe the architecture of DD BOOST (§III) , evaluate the system through customer telemetry and lab tests (§IV), and then discuss lessons learned (§V) and related work (§VI) before concluding.

II. Background

A. History

DD BOOST originated from an earlier library based on the Symantec OpenStorage (OST) protocol [11]. OST was created to allow backup software systems such as Symantec NetBackup to interoperate with purpose-built backup appliances (PBBAs), such as Data Domain systems, over a standard API. By providing an implementation of OST, our PBBA could be used by any application that used the OST APIs. The original OST-based library, introduced in 2007, provided traditional server-based deduplication of the application's data. Data Domain extended it in 2009 to include client-side chunking and fingerprinting. Similar to LBFS [6], the client sends the fingerprints to the PBBA, which looks up the fingerprints to determine which chunks the client must send to the server. (Unlike LBFS, the file is stored in a deduplicated fashion on the server as well.) By building this distributed deduplication capability into the OST implementation, any application using OST benefits from distributed deduplication.

In 2010 the OST-based approach was generalized into a client library available to all applications. Defining the DD BOOST API offered a number of additional benefits: features not supported by OST could be made available to applications (such as asynchronous file replication for duplicating a backup to a second site), API call overhead could be reduced by calling DD BOOST APIs directly rather than via OST APIs; and the APIs could be more general and less backup-specific, hence better-suited to applications other than just backup software.

B. Usage Models

The way in which PBBAs are used has changed dramatically over time [12]. Here we describe some of the key usage models.

Fulls and incrementals. This is the original mode and is still used by many customers. A system writes all data (typically weekly), then periodically (daily) writes all files changed since the last full backup [8]. (There are other approaches, such as having many levels and writing new files changed since the last backup of the next higher level.) Generally, the backup software uses the modification timestamp of files to determine what to write, so any change to a file results in the entire file being backed up again.

A common model for this, such as the one used by Dell EMC's Networker™ backup software, aggregates all files in a backup into a single larger file. The format of the aggregate is similar to the well-known Unix *tar* file. Figure 1 provides a simplified example of this usage model, for both (a) full and (b) incremental backups. For the full backup, the client ① writes a large backup file containing files *f1..fn*. ② The entire file is sent to the server. ③ The server chunks and fingerprints the file. ④ The server saves all unique chunks in compressed form and adds their fingerprints to its index. The file is represented as a *recipe* consisting of its fingerprints.

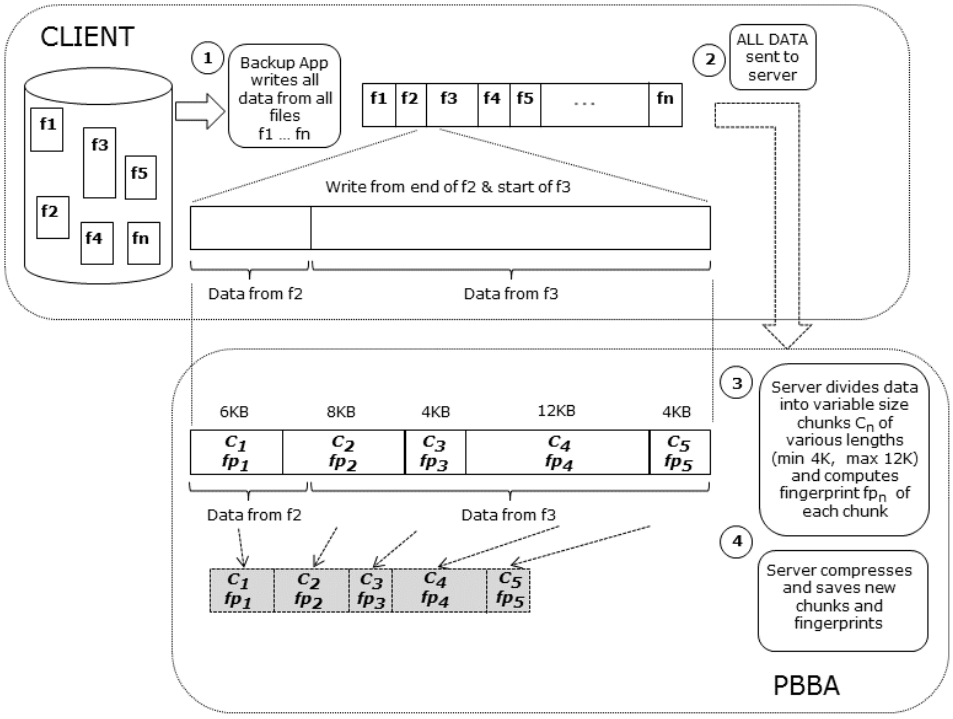
In Figure 1(b), file *f2* is extended and part of file *f3* is overwritten, resulting in ① an incremental file containing both *f2* and *f3* being written. ② The entire incremental backup file is sent to the PBBA for deduplication. The PBBA ③ performs the chunking and computes fingerprints. Finally, ④ it looks up the fingerprints, then compresses and stores those chunks that do not deduplicate.

With DD BOOST, the first full backup is much like the process in Figure 1(a), though any chunks that repeat earlier content in the same backup or other previously stored files can be transferred by fingerprint rather than the full content. Figure 2 shows an incremental backup using DD BOOST. As with the NFS example, files *f2* and *f3* are modified. ① The client backup application writes the incremental backup into DD BOOST. ② DD BOOST buffers the data and processes each buffer by chunking the data and fingerprinting each chunk. ③ DD BOOST sends only the fingerprints and lengths of each chunk in the buffer to the PBBA. ④ The PBBA looks up each fingerprint and sends the client a list of those chunks not found. ⑤ DD BOOST receives that list and ⑥ packs the requested chunks into compression regions. ⑦ The PBBA saves the new chunks and adds their fingerprints to its index.

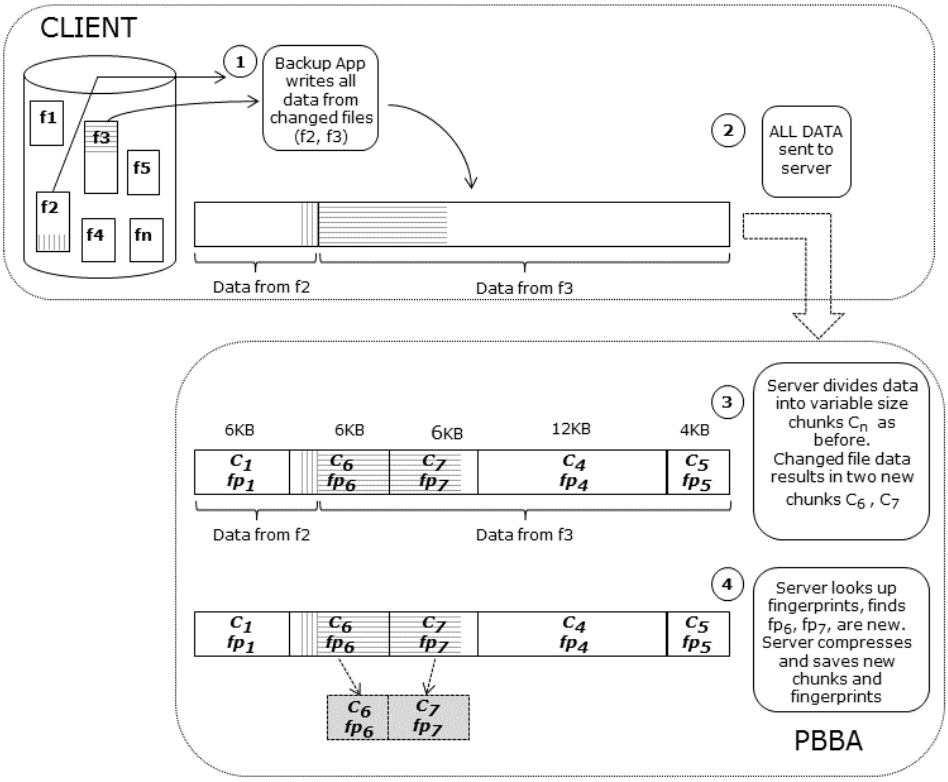
Thus, with DD BOOST, unchanged parts of a file within an incremental backup would be fingerprinted but not transferred over the network. The full backup would benefit much more, since many files would not be modified at all. In this example, just two chunks are sent in their entirety to create the incremental backup, with other chunks being sent by reference.

Virtual Synthetic backups. One of the biggest changes over time has been the use of incremental strategies by backup applications. After some number of incremental backups, the application may want to create an up-to-date full backup by combining the incremental backups made since the last complete backup. The full backup provides the application an ability to consolidate the content, so a future restore can operate on the unified backup file rather than starting from an old "full" backup followed by a large number of updates. Without this consolidation, each update must be processed sequentially and may require obsolete data to be read and then later overwritten.

In fact, some applications choose to synthesize a new full backup after *every* update. This allows the restore functionality



(a) First full backup. ① Individual files f1..fn are aggregated into a single stream of data and ② transferred to the server. The top of the figure shows a blown-up representation of part of this stream, as it is ③ chunked and fingerprinted on the PBBA. Chunks in this example are variable-sized, averaging 8K with minimum and maximum sizes of 4K and 12K, respectively. ④ The server packs together and compresses chunks, adding their fingerprints to its index.



(b) Incremental backup. ① The backup application writes an aggregated backup file consisting of updated files. Here, f2 has had data appended and the first part of f3 has been overwritten. ② The entirety of f2 and f3 are sent to the PBBA, which ③ chunks and fingerprints the data. In this example, C_1, C_4, and C_5 are unchanged, while the new C_6 and C_7 are saved. The incremental file, with recipe $<C_1, C_6, C_7, C_4, C_5>$, is added to the PBBA file system.

Fig. 1. Before DD BOOST, a client would send backups to the PBBA using NFS. Not only (a) the first full backup but (b) all subsequent incremental and full backups would be transferred in their entirety. All fingerprinting, deduplication, and compression would be done on the PBBA.

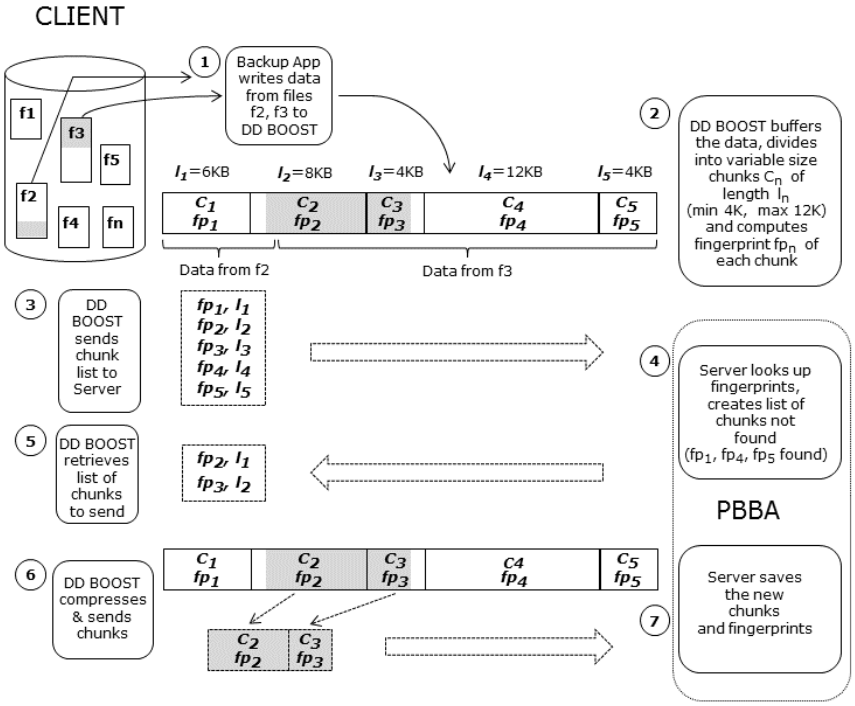


Fig. 2. With DD BOOST, after the full backup is written to the PBBA, subsequent incremental backups send a subset of the files. Some chunks in the incremental may be duplicates, which the PBBA will identify as being unnecessary to transmit. The DD BOOST client library then packs the new chunks into groups of approximately 128 KB before compressing them and sending them to the PBBA.

to ignore the existence of incremental backups, by always operating on a single file representing a point-in-time view of the system. The specifics of what gets written varies by application. A traditional Unix file system backup would generally use modification timestamps, so it might write parts of a file that have not actually changed (similar to the DD BOOST incremental backup). For this application, the real difference is that a *full* backup is created by reference to the previous full and subsequent updates, rather than by writing the data as a new file. A database or virtual machine backup may use logs or change-block tracking [13], respectively, to know precisely what has changed. Since these applications tend to write only data that they know have changed rather than all data when performing an incremental backup, the amount of data written on the client is reduced. Updating unchanged data requires just the explicit reference to large sequences of the previous version rather than per-chunk index lookups, reducing the load on both the client and the PBBA.

What *is* written will not deduplicate as well as a full backup would, because the unchanged data is not repeatedly written. When the application does want to produce a new full backup, most if not all of the data needed already exists on the server. By providing *synthetic* writes (also called *virtual* writes), DD BOOST is able to efficiently support these incremental merge strategies. We refer to backups utilizing these writes as *Virtual Synthetic* (VS) backups [10].

A synthetic write sends no data from the client to the PBBA. Instead of providing data to be written to a file on the PBBA, a synthetic write provides a list of data regions already on the PBBA to be written to the new file. These data regions are simply portions of existing files. Thus a new complete backup file is created from a previous backup file and one or more incremental backup files, without first reading the data. Synthetic writes can be intermingled with normal writes if there is additional client-side data that has not previously been written to the server.

While the incrementals do not deduplicate well (somewhat limiting the benefits of distributing deduplication to the client), synthetic writes have the opposite effect. They are efficient, so systems tend to use them liberally, generating a "full" backup after hours or days rather than weekly. This drives up deduplication on the PBBA by frequently creating new files with the same underlying content, in turn having surprising implications on other activity on the PBBA. For instance, dramatic increases in deduplication significantly impacted Data Domain's garbage collection performance, leading to a new design [14].

Figure 3 provides an example of the VS approach, analogous to the updates using NFS or DD BOOST. ① The first full backup is sent the same way as with DD BOOST. (Some duplicates may be identified in this process, but usually not many.) Then, parts of *f2* and *f3* are updated. The next backup combines ② references to unchanged files with ③ modified files (sent using the same DD BOOST protocol). Thus, instead of having both a full and incremental backup, a new file is synthesized to represent the current state of the system. In this

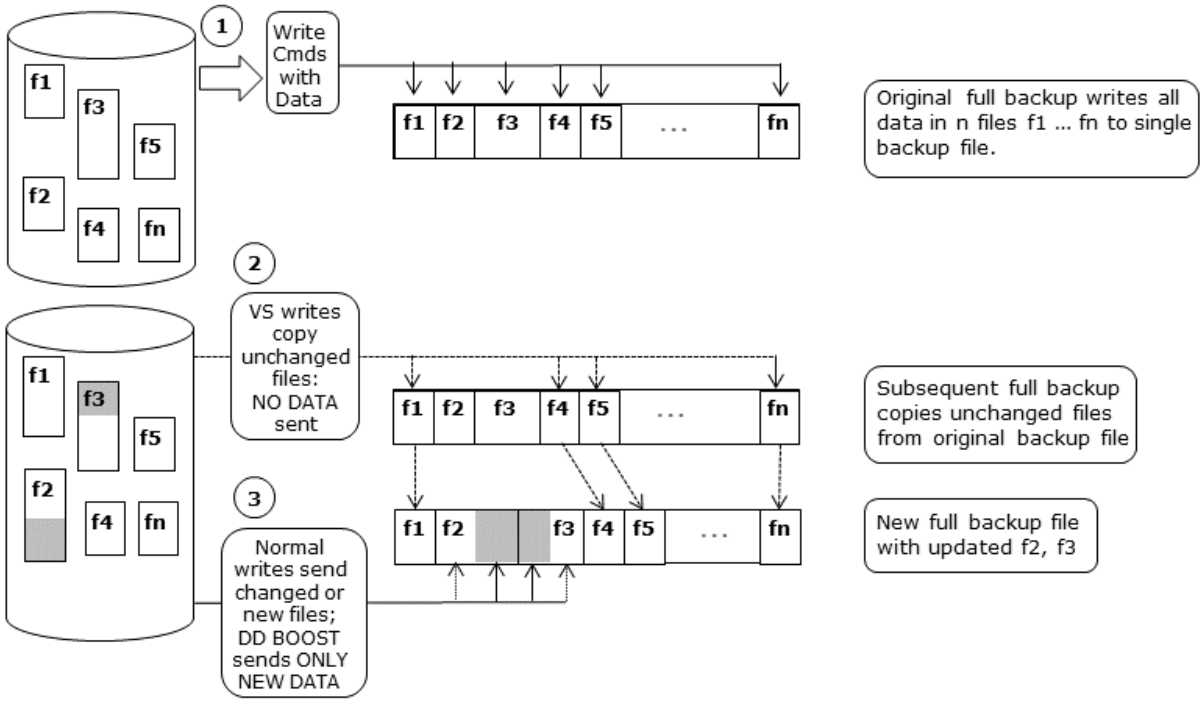


Fig. 3. VS backups start with a full backup, similar to NFS or DD BOOST. After this, they can send incremental updates indefinitely, combining regions from the full backup to create a new full backup on the PBBA. Here, the client uses special DD BOOST VS commands to direct the PBBA to copy files *f1* and *f4..fn* by reference. It writes files *f2* and *f3* using standard DD BOOST operations that perform client-side deduplication. The gray areas represent chunks that are sent from the client, while the dashed arrows point at unfilled areas that are deduplicated via DD BOOST.

figure, the shaded areas in *f2* and *f3* are new chunks that are transferred to the PBBA, while the white parts of these files are chunks that match the previous backup and are deduplicated.

Finally, it is important to note that these models are not exclusive to the DD BOOST API. For instance, OST has "optimized synthetic backups," which direct the storage server to create a new backup from the specified data [11]. This is compared to regular "synthetic backups" that would be constructed on a separate *media server*, requiring data transfer from the storage server to the media server and back.

III. Architecture

This section presents the architecture and APIs for DD BOOST.

A. DD BOOST *Functions*

DD BOOST currently supports about 100 functions:

Library initialization / utility routines (31 calls) These initialize DD BOOST and allow setting/getting various library and server parameters, managing errors, connecting to the server, checksums, etc.

Standard file system calls (34 calls) These support basic file and directory operations such as open, close, read, write, truncate, sync, get attributes, etc.

Additional file capabilities (17 calls) These include APIs to create snapshots, clone or copy a file or directory, perform synthetic / virtual writes, report file statistics, etc.

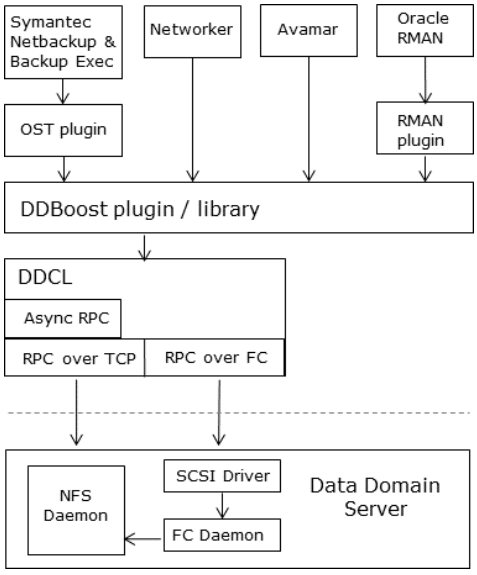


Fig. 4. DD BOOST architectural overview.

Special features (12 calls) These APIs provide custom features such as comparing all or parts of two files, creating and managing access control tokens, and performing file replication.

Figure 4 depicts how DD BOOST interacts with applications

and Data Domain PBBAs.[2] The DD BOOST library sits in the middle, with some applications using it directly and others (NetBackup, BackupExec, and Oracle RMAN) going through specialized APIs that have been modified to use DD BOOST. It manages interactions with the PBBA via RPCs sent over either TCP or FibreChannel (FC). (RPCs sent over FC are received as SCSI commands by the SCSI driver on the server and passed to a FC daemon, which extracts the embedded RPC and forwards it on to the NFS daemon.) Some simple DD BOOST calls (e.g, library initialization) are handled entirely in the DD BOOST library. Other operations result in one or more RPCs to the server. For example, a write call may only buffer the user data until there is sufficient data for chunking and fingerprinting. If sufficient data is provided or has accumulated, the write call will result in one or more "send references" RPCs sending the list of chunks to the server, one or more "receive references" RPCs retrieving the list of chunks that need to be sent to the server, and then a number of "send data" RPCs to send those chunks to the server. Additionally, the first write of a file will send an *open* RPC, as the file is not opened on the server until it is actually written in order to improve performance and conserve server resources. If a client uses excessive resources, for instance by opening too many parallel connections, it will be throttled by the PBBA.

To guard against accidental (or intentional) misrepresentation of the fingerprint of a chunk [15], the PBBA recomputes fingerprints before storing them locally. While this adds some computational load, the effort is only expended on new data.

B. Nonsequential I/O

The DD BOOST API was originally intended for backing up data to a PBBA using an interface derived from the days of tape backup. Whether a file is an aggregate of many individual files (similar to a "tar" file), the concatenation of raw disk blocks, or a single file copied directly from the client to the PBBA, it would be written sequentially from start to finish.

Thus DD BOOST was optimized for writing files in their entirety, starting at offset zero, with no change in offset or overwriting of existing content. Random writes were supported, but due to the difficulty of chunking data when using variable size chunks, random writes would cause deduplication to revert to the server using OST-level calls such as NFS writes. Read performance was less important than write performance, since backups (writing data) were done frequently but restores (reading data) were rare.

Over time users increasingly wanted to use the PBBA as a more general file server or in other modes such as change-block tracking [16]; thus, supporting nonsequential I/O became more important. Nonsequential reads are easily supported, since the metadata for a deduplicated file can be used to map an offset to a specific chunk, but the system must be careful not to allow readahead to adversely impact performance. Write performance is still the greatest priority, however: to map efficiently from an offset to a chunk rather than the other way around, *fixed-sized* chunking has advantages. For some workloads, fixed chunks moderately degrade deduplication due to shifting content [17], [18], but fixing the unit of deduplication is useful for maintaining acceptable write performance for arbitrary offsets. This is especially true for workloads such as VM images that always write full blocks that are aligned to some blocksize. The choice of variable-sized or fixed-sized chunking can be specified on a per-file basis.

[2]Currently, DD BOOST works only with Data Domain systems. It could be standardized similar to OST to work with other PBBAs.

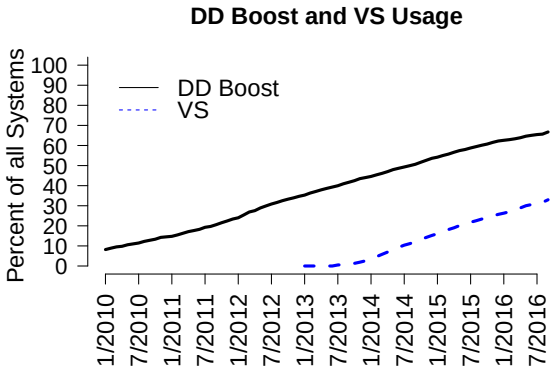


Fig. 5. DD BOOST and VS were introduced in 2010 and 2013, respectively; both took time to attain a sizable fraction of deployed Data Domain systems.

IV. Evaluation

We divide our evaluation into an overview of customer use and lab benchmarks to compare protocols head-to-head. We answer the following questions:

- How many systems use DD BOOST and VS backups?
- How much bandwidth is saved via DD BOOST?
- How do DD BOOST and NFS scale under load when writing backups? How much does the cumulative change in the backups affect performance? How much does VS help?
- How do processor loads for DD BOOST compare to NFS?

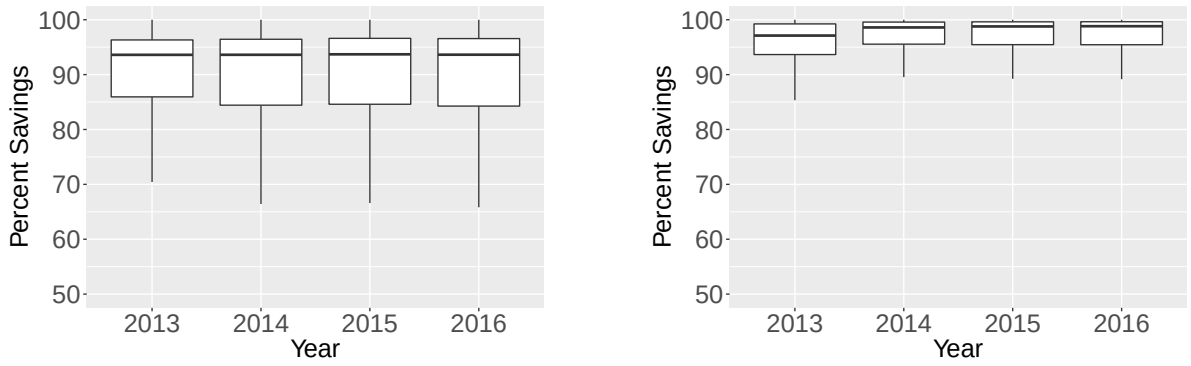
A. Telemetry

As DD BOOST was first released in 2010, we analyzed daily reports from customer systems to ascertain the rate of adoption and the bandwidth reductions achieved. Figure 5 shows that the fraction of systems using DD BOOST has increased steadily,[3] as has those using VS backups.

Systems using VS tend to have much higher deduplication rates than those without, and the writes performed on the PBBA via a synthetic write directive are counted as pre-compression writes. This results in many systems having bandwidth improvement rates that round to 100%.

Therefore, we evaluate the bandwidth savings by dividing PBBAs depending on their use of VS. Figure 6(a) shows box-and-whisker plots, by year, for systems with no VS. (We start in 2013 because before then there are few systems using

[3]It starts with about 10% of systems having DD BOOST enabled as of the start of 2010, as this includes those using the earlier OST version.



(a) No VS

(b) Use VS

Fig. 6. Bandwidth savings from DD BOOST, grouped by year, and segregated by use of VS backups.

DD BOOST and no systems using VS.) We see that even the systems that do not use VS see median savings over 90%, compared to a median of 99% savings for systems with VS. Figure 6(b) shows the corresponding plots when VS is used.

The box plots for the non-VS systems are remarkably similar across years, and each shows a large variance from the median. We have not analyzed these metrics enough to know the cause behind systems saving as little as 60–70% of bandwidth, but an educated guess might be that they write changed blocks but do not write periodic full backups that would deduplicate well on the clients. As they do not use VS, these systems aren't getting the "credit" for performing server-side operations without requiring network I/O.

The VS systems show the median and bottom quartile rising steadily: this may indicate that the increase in adoption rates shown in Figure 5 also results in more widespread use of VS on each system, or even that systems that started using VS sometime during the year had less overall benefits. In the steady state (after the first year), even the outliers at the low range are saving 90% of network traffic.

The box plots for Figure 5 are Tukey box plots, in that they do not show data that were outliers via the interquartile (IQR) method. For systems with no VS, the percentage of data as outliers ranged from 15-16%, and the median percent savings of the outliers increased from 26-32%. For systems with VS, the percentage of outliers decreased from 10.2% to 7.5% from 2013–2016, while the median percent savings of the outliers increased from 72% to 82%.

B. Lab Experiments

We used an internal load generator [14], [19] to evaluate DD BOOST in comparison to a direct NFS mount to the PBBA. The generator creates N parallel streams, each with its own "lineage." Within a lineage, one version of a file has a predetermined amount of overlap with the previous file, which is transformed through a series of moves, deletions, and insertions: approximately 1% is added and deleted with each new generation, with 3% of data shuffled internally. Compression within each file is fixed at 2:1 for these experiments, through a combination of uncompressible and extremely compressible data. The changes are deterministic, so adjacent generations within a lineage will contain largely the same data; across lineages, no duplication exists.

While previous work using this load generator has applied a *uniform* distribution to the choice of locations for update operations [14], [19], experience with VS backups led to a more detailed examination of change rates in user backups. For our experiments, we use a *normal* distribution, which concentrates updates more frequently in a smaller range of data. This results in larger sequences that are more conducive to stitching together via the VS protocol, and the parameters have been validated against sample customer datasets.

Briefly, the load generator works as follows. In these tests, we want to compare the performance of writing and reading files with good locality and those with poorer locality. The initial file, which we call *generation 0* ("gen-0"), is written completely sequentially. Because it does not deduplicate with any existing content, it should be packed into the minimum number of storage containers needed to hold the compressed data in the file. The first change to the file is called generation 1; we also refer to this as a *low-generation* file. It has some fragmentation as many chunks refer to those from generation 0 while some are newly added or modified. *High-generation* files have been evolved through many iterations of changes to content. Thus a high-gen write has gone through many more rounds of transformation and will deduplicate less. In addition, each generation written to a PBBA results in more fragmentation, impacting read and write performance [2], [3].

Once the first generation is written, each generation is transformed using a set of rules and distributions derived from customer datasets. The file is modified by a sequence of *additions*, *deletions*, and *modifications* over clusters of data. Based on analysis of backup datasets, we default to an average of one cluster of added data, one cluster of deleted data, and two clusters of modified data per gigabyte. The location of the center of the first update is picked randomly, and the location of each update after the first, relative to the previous update, is based on a distribution: half the time the update is within 1%

of the entire file size, relative to the previous cluster, with a mean distance of 8% of the file size and a standard deviation of 18%. The σ value for the normal distribution, i.e., the range of most updates from the center of the cluster, is 0.25% of the file size for modifications, 0.45% for deletions, and 0.9% for additions.

We iterate through all generations of a lineage, but we only write the following versions to the PBBA: 0 (the initial file), 1, then every fifth version $5, 10, ..., 40$ before timing the writes to high-gen data, generation 41. We compare the write performance and read performance, respectively, for NFS and DD BOOST on the *gen-0*, *low-gen*, and *high-gen* versions.

To compare the different protocols directly, we used an EMC Data Domain DD2500 server (8 cores@2.20GHz, 64GB DRAM, 3 shelves of disks, 2 10GbE ports). We have four identical clients running CentOS 6.5, with 12 cores@2.67GHz and 32 GB DRAM. Two clients share a given 10GbE port, and except for the single-stream case, operate in parallel on $\frac{N}{4}$ 20 GB files apiece. All measurements are repeated three times and error bars are shown.

Figure 7(a) shows effective network bandwidth for gen-0, low-gen, and high-gen writes. (This is the data per unit time written by a client; since the data for each generation is fixed, it is inversely proportional to how quickly the operation completes.) The x-axis shows the number of parallel streams, ranging from 1–64. When writing gen-0, DD BOOST and NFS are nearly identical, gaining a slight improvement when moving from 1 stream to 8 streams but not improving much beyond that. The lines for gen-0 DD BOOST and NFS nearly overlap at this scale, though DD BOOST transfers only half the data due to compression; DD BOOST's relative effective bandwidth is only 40% of NFS with one stream but 10–25% better with parallel streams. Once the data is highly duplicated, NFS shows little benefit from increased streams past the first version, while DD BOOST improves with offered load.

DD BOOST gets similar bandwidth for low-gen and high-gen writes, though the error bars highlight greater inconsistency when reaching 64 streams. High-gen scales better with more parallel streams, perhaps due to dataset size. In the cases of low-gen and high-gen, DD BOOST transfers only about 5% of the data NFS writes to the PBBA.

We also measured the CPU utilization of the clients and PBBA during the experiments. For gen-0, where the clients must write all data to the PBBA, the peak one-minute utilization on each client is under 0.5% for both DD BOOST and NFS, but NFS is roughly $\frac{1}{2}$ that of DD BOOST. Since both are so low, this distinction is unimportant, but more importantly the utilization for DD BOOST falls below that of NFS for the low-gen and high-gen runs. This indicates the benefit of not transmitting duplicates compensates for the overhead of chunking and fingerprinting.

Figure 7(b) shows the bandwidth for read performance—something for which PBBAs are not optimized. With low parallelism, both protocols perform similarly, though NFS has a slight advantage. There is greater differentiation at higher stream counts, but they remain close as long as there has been some fragmentation. NFS does consistently outperform DD BOOST for reads of the gen-0 file.

Finally, we show the benefit of VS backups in Figure 7(c). It shows the bandwidth, in terms of user writes, for standard DD BOOST and DD BOOST with VS, on the low-gen and high-gen datasets. (No VS backup is possible for gen-0.) The standard DD BOOST curves are the same as in Figure 7(a); by comparison, VS consistently outperforms regular DD BOOST, though DD BOOST closes the gap by scaling better with higher numbers of parallel streams.

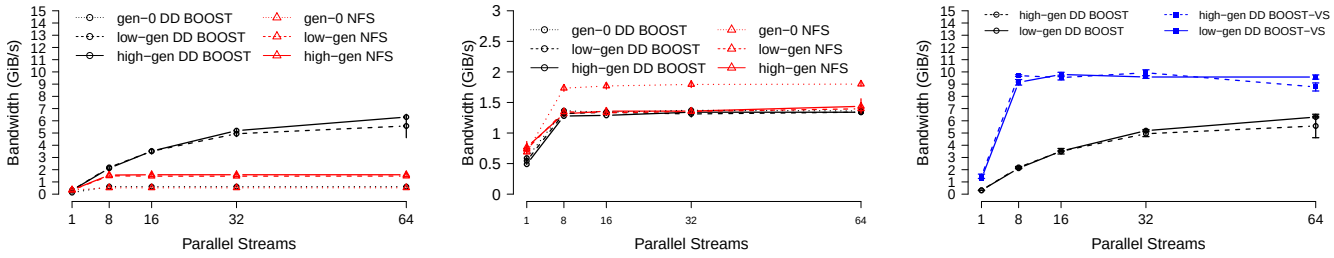
V. Lessons Learned

DD BOOST has been a victim of its own success. As customers used DD BOOST for traditional backups and found it helpful, they wanted to do more with it. This led to uses that the original design did not consider and did not support well if at all. Some of the lessons and challenges are:

In-line transformations. When doing a complete backup of a database or filesystem on a periodic basis, a high degree of deduplication would be expected if the data has not changed significantly. Only the fraction of the data that is new or modified should actually need to be stored. However we found in some database backups very little deduplication occurred despite only a small fraction of the data being different. The reason is that some backup applications don't just write the data but include metadata in-line with the data, such as a date or timestamp [20]. This metadata changes with each backup, so it will appear to the deduplication algorithm that the data has changed. We refer to this kind of metadata as *marker* data, as the applications typically use it to mark the backup data as belonging to a certain backup instance, a certain database instance, or other tags. The marked data is typically small and of a known format, so if we know marker data is present the deduplication algorithm can look for and detect the marker data. For the purposes of deduplication the marker data can be ignored (though it must still be stored along with the other data), and doing so can greatly improve deduplication. Format and content of markers are application-specific, so markers need to be handled on a per-application basis. Marker detection and processing for various important applications have been added to both the PBBA and the DD BOOST Client Library over time to maintain high deduplication rates.

Similarly, DD BOOST was modified to transform certain file paths to organize files into smaller directories. This is useful when applications require searching all files in a directory, something the PBBA was not designed to support efficiently, without directly modifying the core application.

High availability. Originally our PBBA hardware was not highly available: a hardware failure or software crash on the server would result in the storage and data being unavailable until a restart could be done, often taking many minutes or longer. This would cause any application using DD BOOST to fail since accesses to the server would timeout, unless the application itself could detect and work around the failure. (An application could possibly recover via periodic checkpoints saving work done so far, then resuming from the most recent



(a) Writes (b) Reads (c) VS (no gen-0 applicable)

Fig. 7. Effective write and read bandwidth as a function of parallel streams, for gen-0, low-gen and high-gen files. (Higher is better.)

checkpoint after detecting a failure and determining that the system was responding again.) With customers increasingly needing to backup or restore at all hours of the day, providing backup applications that operated through server failures became a high priority. This required both highly available hardware and changes to DD BOOST. We developed server systems with redundant hardware that could recover from software or hardware failure in a few minutes at most. DD BOOST then had to be updated to detect and recover properly from any server failure.

To recover from a failure, DD BOOST follows a similar checkpointing strategy to applications. After writing a fixed amount of data to a file (by default 128 MB, but configurable) DD BOOST performs a file synchronization operation that guarantees all data written to the server is in persistent storage. Between these periodic synchronization operations the data written since the previous synchronization operation is retained in buffers in the client. If a failure is detected, the file is recovered by truncating it to its size at the last successful synchronization operation, then re-writing newer data.

To detect the failure, DD BOOST has to rely on the server. Until the server recovers, DD BOOST simply sees RPCs that time out since the server is not responding. These RPCs are resent until the server recovers (if the server does not recover quickly enough, DD BOOST will consider the server dead and fail). During recovery, the server marks all files that were opened by DD BOOST as needing recovery. After recovery any RPCs that reference files that DD BOOST had open will return a special error indicating a failover has occurred and file recovery is necessary. This error causes DD BOOST to do the previously described recovery for the file before resuming normal file operations. (This is similar to distributed recovery in other file systems, e.g. Sprite [21].)

VS writes must be handled differently, since a synthetic write sends no data from the client. Instead of retaining data, DD BOOST retains the list of data regions provided with each synthetic write. During recovery, these saved lists are used to repeat the VS operations.

File usage. Backup applications using tape tended to write a backup as a single large file or several large files [22]. This meant both the DD BOOST client software and the PBBA software did not need to support many thousands of files or small files. This has changed over time, as some backups have replicated complete file systems one-for-one. It is now common for millions of files and directories to exist on the backup server [12]. While this shift did not require a change in APIs, it did require a change in internal design and implementation to efficiently support large numbers of files and directories. For example, lookup of files in a directory had to scale much better when the directory could contain thousands of files.

Virtualization. Another recent transformation of Data Domain is its availability as a virtual image, called the Data Domain Virtual Edition (DDVE) [12]. DDVEs can be used in small environments that do not require a standalone PBBA, and they can also be deployed in the cloud. A DDVE is more resource-constrained than a PBBA, so when DD BOOST allows the clients to buffer streams and pack them into compression regions, the DDVE needs less memory. Thus a DDVE can ingest data via DD BOOST when its resource requirements to ingest over NFS would be excessive.

Compression and encryption. More applications are compressing data stored in memory and in files. For security purposes data is similarly being encrypted when stored persistently. Thus backup applications are often writing compressed and/or encrypted data. This is a challenge for deduplication, as both of these tend to produce data that is essentially arbitrary and thus eliminate the opportunities for deduplication.

We deal with compression primarily by trying to work with applications to disable compression and allowing the PBBA to perform compression instead. Encryption is more complicated. Our PBBA supports encryption, so applications can leave it to the appliance. If they do not trust the PBBA to manage the data securely, the application can write fixed-sized blocks that are individually encrypted, so the same data will encrypt in the same way each time. If an entire large file is encrypted monolithically, a small change may trickle through the entire file, changing all data.

VI. Related Work

LBFS [6] has similarities to DD BOOST, in that it provided its own API for files to be deduplicated prior to network transfer.

Files would be written and cached as whole files, using a new API to write temporary files and then commit them to be written in their entirety. At that point they could be chunked, fingerprinted, and transferred to the server. In contrast, DD BOOST supports sequential writes that are chunked on the fly, as well as a reversion to NFS to support nonsequential I/O. This means that a single small random write requires only the new data to be transferred.

Embedding deduplication with network transport has been done in numerous contexts, such as rsync [4], DOT [23], czip [24], and Jumbo Store [25]. The principal distinction of DD BOOST is that it integrates network transport with deduplicating storage. Avamar™ [26], Taper [27], and Jumbo Store use hierarchical views of content (directed graphs similar to Merkle trees [28]) to identify changing content more efficiently than comparing every individual chunk. VS backups in Data Domain are similar in that they aggregate large regions of unchanged data.

More generally, there is a large body of work in deduplication optimization and the tradeoffs of different approaches. For example, there are numerous studies of the impact of deduplicating at the granularity of whole files, fixed-sized blocks, or variable-sized chunks [17], [18], [29].

VII. Conclusion and Future Work

We have described over five years of experience with a distributed deduplication API, DD BOOST. Over time, the use of *Virtual Synthetic* backups, which have the effect of efficiently creating full backup files while only transferring new data, has extended distributed deduplication further. Instead of writing a full backup and analyzing it on the PBBA or client to extract new content, the backup application provides only the new content, yet the full backup is made available efficiently on the PBBA. We have seen extensive adoption of DD BOOST, which has become a fundamental underpinning of our PBBA.

More recently, we have released the first version of a FUSE-based file system (BOOSTFS) that extends distributed deduplication to unmodified applications [30]. Once we have experience with BOOSTFS in the field, we expect to follow up with additional analyses.

Acknowledgments

We thank George Amvrosiadis (our shepherd) and the anonymous reviewers for their comments on our paper. Jeff Ford, Philip Fote, Stephen Manley, Darren Sawyer, Philip Shilane, and Stephen Smaldone also provided comments on earlier versions. We thank Larry Brisson, Tuan Nguyen, and Yijian Wang for assistance with the performance tool, and Medha Bhadkamkar and Bruce Montague for information about OST.

References

[1] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, 2014.

[2] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki, "Reducing impact of data fragmentation caused by in-line deduplication," in *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM, 2012.

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

[4] A. Tridgell and P. Mackerras, "The rsync algorithm," Australian National University, Tech. Rep. TR-CS-96-05, 1996.

[5] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *Proceedings of the ACM SIGCOMM 2000 Conference*, 2000. [Online]. Available: http://doi.acm.org/10.1145/347059.347408

[6] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.

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

[8] A. Chervenak, V. Vellanki, and Z. Kurmas, "Protecting file systems: A survey of backup techniques," in *15th IEEE Symposium on Mass Storage Systems*, vol. 99, 1998.

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

[10] EMC Corp., *EMC Data Domain Boost for Partner Integration, Version 3.2.1, Administration Guide*, Mar. 2016, http://www.emc.com/collateral/TechnicalDocument/docu61798.pdf.

[11] *Symantec NetBackup OpenStorage SolutionsGuide for Disk; UNIX, Windows, Linux; Release 7.7*, Sep. 2015, https://www.veritas.com/support/en_US/article.DOC8574.

[12] Y. Allu, F. Douglis, M. Kamat, P. Shilane, H. Patterson, and B. Zhu, "Evolution of the Data Domain file system," *IEEE Computer*, to appear.

[13] V. 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.

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

[15] V. Henson, "An analysis of compare-by-hash." in *HotOS*, 2003.

[16] A. Natanzon, P. Shilane, M. Abashkin, L. Baruch, and E. Bachmat, "Hybrid replication: Optimizing network bandwidth and primary storage performance for remote replication," in *Networking, Architecture and Storage (NAS), 2016 IEEE International Conference on*. IEEE, 2016.

[17] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *Trans. Storage*, vol. 7, no. 4, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2078861.2078864

[18] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in *Proceedings of the USENIX Annual Technical Conference*, 2004.

[19] F. C. Botelho, P. Shilane, N. Garg, and W. Hsu, "Memory efficient sanitization of a deduplicated storage system," in *USENIX Conference on File and Storage Technologies (FAST'13)*, Feb. 2013. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/botelho

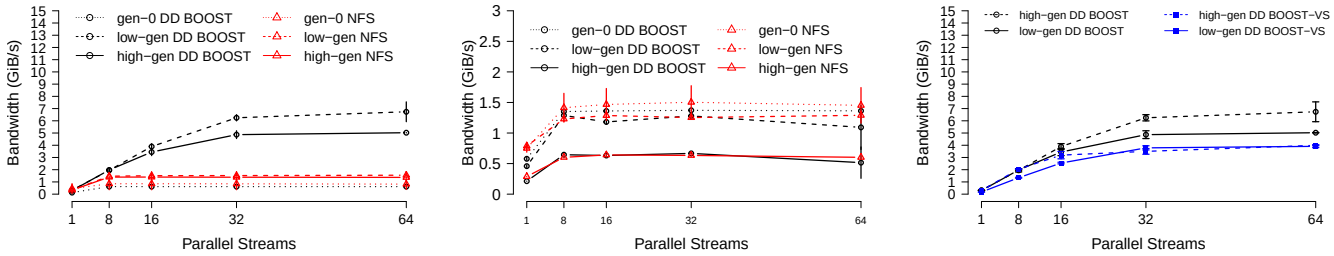
[20] X. Lin, F. Douglis, J. Li, X. Li, R. Ricci, S. Smaldone, and G. Wallace, "Metadata considered harmful…to deduplication," in *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, Santa Clara, CA, Jul. 2015. [Online]. Available: http://blogs.usenix.org/conference/hotstorage15/workshop-program/presentation/lin

[21] M. Baker and J. Ousterhout, "Availability in the sprite distributed file system," *ACM SIGOPS Operating Systems Review*, vol. 25, no. 2, 1991.

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

[23] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil, "An architecture for internet data transfer," in *NSDI*, 2006.

[24] K. Park, S. Ihm, M. Bowman, and V. S. Pai, "Supporting practical content-addressable caching with CZIP compression," in *USENIX ATC*, 2007. [Online]. Available: http://www.usenix.org/events/usenix07/tech/park.html



(a) Writes (b) Reads (c) VS (no gen-0 applicable)

Fig. 8. Using a *uniform* distribution, this shows the effective write and read bandwidth as a function of parallel streams, for gen-0, low-gen and high-gen files. (Higher is better.)

[25] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, and R. Hawkes, "Jumbo store: Providing efficient incremental upload and versioning for a utility rendering service." in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[26] J. Hamilton and E. W. Olsen, "Design and implementation of a storage repository using commonality factoring," in *Mass Storage Systems and Technologies (MSST'03)*. IEEE, 2003.

[27] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered approach for eliminating redundancy in replica synchronization," in *4th USENIX Conference on File and Storage Technologies*, 2005.

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

[29] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani, "Demystifying data deduplication," in *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*. ACM, 2008.

[30] EMC Corp., *EMC Data Domain BoostFS, Version 1.0, Configuration Guide*, Sep. 2016, https://support.emc.com/docu78742_Data-Domain-BoostFS-1.0-Configuration-Guide.pdf?language=en_US.

APPENDIX
UNIFORM DISTRIBUTION

As noted in §IV-B, previous evaluations of Data Domain in the literature used a workload generator following a uniform distribution [14], [19]. While the choice of distribution is not the focus of this paper, we include a comparison between the normal distribution presented in the evaluation (§IV) and the same experiments using a uniform distribution. Figure 8 presents the three subfigures corresponding to Figure 7, for writes and reads of NFS and DD BOOST, and writes of VS and DD BOOST. We see that the results for NFS and DD BOOST are substantially similar using the two distributions, while VS is dramatically worse on a uniform distribution than a normal one. This is because rather than clustering updates and allowing large regions of unchanged data to be referenced in a single operation, as with the normal distribution, Figure 8(c) shows that VS performs much worse with frequent switches between changed and unchanged data.