Generating Realistic Datasets for Deduplication Analysis

Appears in the proceedings of the 2012 USENIX Annual Technical Conference (ATC 2012)

Vasily Tarasov[1], Amar Mudrankit[1], Will Buik[2], Philip Shilane[3], Geoff Kuenning[2], Erez Zadok[1]
[1]*Stony Brook University,* [2]*Harvey Mudd College, and* [3]*EMC Corporation*

Abstract

Deduplication is a popular component of modern storage systems, with a wide variety of approaches. Unlike traditional storage systems, deduplication performance depends on data content as well as access patterns and meta-data characteristics. Most datasets that have been used to evaluate deduplication systems are either unrepresentative, or unavailable due to privacy issues, preventing easy comparison of competing algorithms. Understanding how both content and meta-data evolve is critical to the realistic evaluation of deduplication systems.

We developed a generic model of file system changes based on properties measured on terabytes of real, diverse storage systems. Our model plugs into a generic framework for emulating file system changes. Building on observations from specific environments, the model can generate an initial file system followed by ongoing modifications that emulate the distribution of duplicates and file sizes, realistic changes to existing files, and file system growth. In our experiments we were able to generate a 4TB dataset within 13 hours on a machine with a single disk drive. The relative error of emulated parameters depends on the model size but remains within 15% of real-world observations.

1 Introduction

The amount of data that enterprises need to store increases faster than prices drop, causing businesses to spend ever more on storage. One way to reduce costs is deduplication, in which repeated data is replaced by references to a unique copy; this approach is effective in cases where data is highly redundant [11,15,17]. For example, typical backups contain copies of the same files captured at different times, resulting in deduplication ratios as high as 95% [9]. Likewise, virtualized environments often store similar virtual machines [11]. Deduplication can be useful even in primary storage [15], because users often share similar data such as common project files or recordings of popular songs.

The significant space savings offered by deduplication have made it an almost mandatory part of the modern enterprise storage stack [5, 16]. But there are many variations in how deduplication is implemented and which optimizations are applied. Because of this variety and the large number of recently published papers in the area, it is important to be able to accurately compare the performance of deduplication systems.

The standard approach to deduplication is to divide the data into *chunks*, hash them, and look up the result in an index. Hashing is straightforward; chunking is well understood but sensitive to parameter settings. The indexing step is the most challenging because of the immense number of chunks found in real systems.

The chunking parameters and indexing method lead to three primary evaluation criteria for deduplication systems: (1) space savings, (2) performance (throughput and latency), and (3) resource usage (disk, CPU, and memory). All three metrics are affected by the data used for the evaluation and the specific hardware configuration. Although previous storage systems could be evaluated based only on the I/O operations issued, deduplication systems need the actual content (or a realistic re-creation) to exercise caching and index structures.

Datasets used in deduplication research can be roughly classified into two categories. (1) Real data from customers or users, which has the advantage of representing actual workloads [6, 15]. However, most such data is restricted and has not been released for comparative studies. (2) Data derived from publicly available releases of software sources or binaries [10, 24]. But such data cannot be considered as representative of the general user population. As a result, neither academia nor industry have wide access to representative datasets for unbiased comparison of deduplication systems.

We created a framework for *controllable data generation*, suitable for evaluating deduplication systems. Our dataset generator operates at the file-system level, a common denominator in most deduplication systems: even block- and network-level deduplicators often process file-system data. Our generator produces an initial file system image or uses an existing file system as a starting point. It then *mutates* the file system according to a *mutation profile*. To create profiles, we analyzed data and meta-data changes in several public and private datasets: home directories, system logs, email and Web servers, and a version control repository. The total size of our datasets approaches 10TB; the sum of observation periods exceeds one year, with the longest single dataset exceeding 6 months' worth of recordings.

Our framework is versatile, modular, and efficient. We use an in-memory file system tree that can be populated and mutated using a series of composable modules. Researchers can easily customize modules to emulate file system changes they observe. After all appropriate mutations are done, the in-memory tree can be quickly written to disk. For example, we generated a 4TB file system on a machine with a single drive in only 13 hours, 12 of which were spent writing data to the drive.

2 Previous Datasets

To quantify the lack of readily available and representative datasets, we surveyed 33 deduplication papers published in major conferences in 2000–2011: ten papers were Usenix ATC, ten in Usenix FAST, four in SYSTOR, two in IEEE MSST, and the remaining seven elsewhere. We classified 120 datasets used in these papers as: (1) Private datasets accessible only to particular authors; (2) Public datasets which are hard or impossible to reproduce (e.g., CNN web-site snapshots on certain dates); (3) Artificially synthesized datasets; and (4) Public datasets that are easily reproducible by anyone.

We found that 29 papers (89%) used *at least one* private dataset for evaluation. The remaining four papers (11%) used artificially synthesized datasets, but details of the synthesis were omitted. This makes it nearly impossible to fairly compare many papers among the 33 surveyed. Across all datasets, 64 (53%) were private, 17 (14%) were public but hard to reproduce, and 11 (9%) were synthetic datasets without generation details. In total, 76% of the datasets were unusable for cross-system evaluation. Of the 28 datasets (24%) we characterized as public, twenty were smaller than 1GB in logical size, much too small to evaluate any real deduplication system. The remaining eight datasets contained various operating system distributions in different formats: installed, ISO, or VM images.

Clearly, the few publicly available datasets do not adequately represent the entirety of real-world information. But releasing large real datasets is challenging for privacy reasons, and the sheer size of such datasets makes them unwieldy to distribute. Some researchers have suggested releasing hashes of files or file data rather than the data itself, to reduce the overall size of the released information and to avoid leaking private information. Unfortunately, hashes alone are insufficient: much effort goes into chunking algorithms, and there is no clear winning deduplication strategy because it often depends on the input data and workload being deduplicated.

3 Emulation Framework

In this section we first explain the generic approach we took for dataset generation and justify why it reflects many real-world situations. We then present the main components of our framework and their interactions. For the rest of the paper, we use the term *meta-data* to refer to the file system name-space (file names, types, sizes, directory depths, etc.), while *content* refers to the actual data within the files.

3.1 Generation Methods

Real-life file systems evolve over time as users and applications create, delete, copy, modify, and back up files. This activity produces several kinds of correlated information. Examples include 1) Identical downloaded files; 2) Users making copies by hand; 3) Source-control systems making copies; 4) Copies edited and modified by users and applications; 5) Full and partial backups repeatedly preserving the same files; and 6) Applications creating standard headers, footers, and templates.

To emulate real-world activity, one must account for all these sources of duplication. One option would be to carefully construct a statistical model that generates duplicate content. But it is difficult to build a statistics-driven system that can produce correlated output of the type needed for this project. We considered directly generating a file system containing duplicate content, but rejected the approach as impractical and non-scalable.

Instead, we emulate the evolution of real file systems. We begin with a simulated *initial snapshot* of the file system at a given time. (We use the term “snapshot” to refer to the complete state of a file system; our usage is distinct from the copy-on-write snapshotting technology available in some systems.) The initial snapshot can be based on a live file system or can be artificially generated by a system such as Impressions [1]. In either case, we *evolve* the snapshot over time by applying *mutations* that simulate the activities that generate both unique and duplicate content. Because our evolution is based on the way real users and applications change file systems, our approach is able to generate file systems and backup streams that accurately simulate real-world conditions, while offering the researcher the flexibility to tune various parameters to match a given situation.

Our mutation process can operate on file systems in two dimensions: space and time. The “space” dimension is equivalent to a single snapshot, and is useful to emulate deduplication in primary storage (e.g., if two users each have an identical copy of the same file). “Time” is equivalent to backup workloads, which are very common in deduplication systems, because snapshots are taken within some pre-defined interval (e.g., one day). Virtualized environments exhibit both dimensions, since users often create multiple virtual machines (VMs) with identical file systems that diverge over time because they are used for different purposes. Our system lets researchers create mutators for representative VM user classes and generate appropriately evolved file systems. Our system’s ability to support logical changes in both space and time lets it evaluate deduplication for all major use cases: primary storage, backup, and virtualized environments.

3.2 Fstree Objects

Deduplication is usually applied to large datasets with hundreds of GB per snapshot and dozens of snapshots. Generating and repeatedly mutating a large file system would be unacceptably slow, so our framework performs

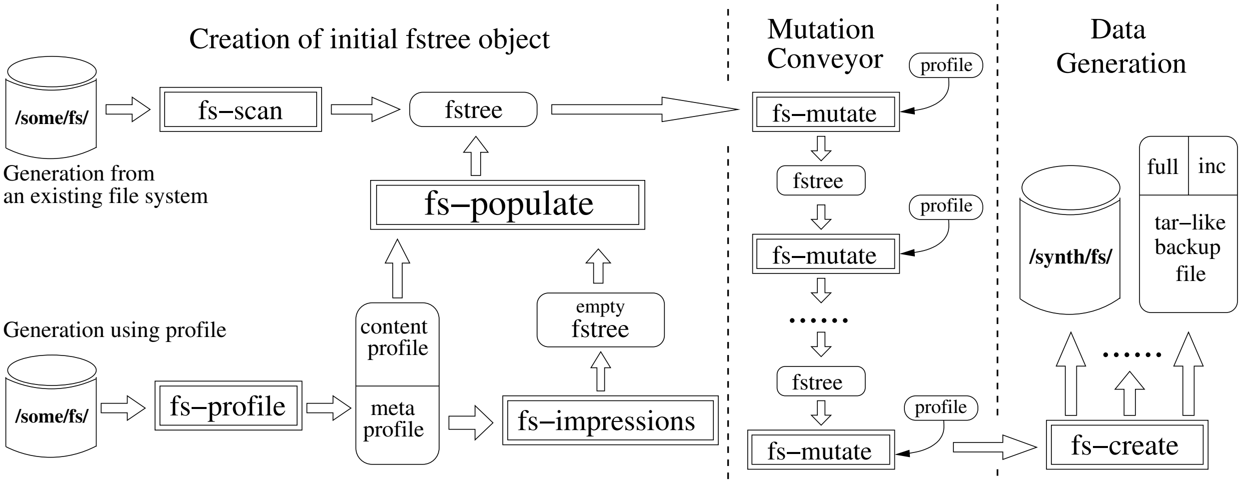


Figure 1: Action modules and their relationships. Double-boxed rectangles represent action modules and rectangles with rounded corners designate fstrees and other inputs and outputs.

most of its work without I/O. Output happens only at the end of the cycle when the actual file system is created.

To avoid excess I/O, we use a small in-memory representation—an *fstree*—that stores only the information needed for file system generation. This idea is borrowed from the design of Filebench [8]. The fstree contains pointers to *directory* and *file* objects. Each directory tracks its parent and a list of its files and subdirectories. The file object does not store the file's complete content; instead, we keep a list of its logical *chunks*, each of which has an identifier that corresponds to (but is not identical to) its deduplication hash. We later use the identifier to generate unique content for the chunk. We use only 4 bytes for a chunk identifier, allowing up to 2^{32} unique chunks. Assuming a 50% deduplication ratio and a 4KB average chunk size, this can represent 32TB of storage. Note that a single fstree normally represents a single snapshot, so 32TB is enough for most modern datasets. For larger datasets, the identifier field can easily be expanded.

To save memory, we do not track per-object user or group IDs, permissions, or other properties. If this information is needed in a certain model (e.g., if some users modify their files more often than others), all objects have a variable-sized private section that can store any information required by a particular emulation model.

The total size of the fstree depends on the number of files, directories, and logical chunks. File, directory, and chunk objects are 29, 36, and 20 bytes, respectively. Representing a 2TB file system in which the average file was 16KB and the average directory held ten files would require 9GB of RAM. A server with 64GB could thus generate realistic 14TB file systems. Note that this is the size of a *single* snapshot, and in many deduplication studies one wants to look at 2–3 months worth of daily backups. In this case, one would write a snapshot after each fstree mutation and then continue with the same in-memory fstree. In such a scenario, our system is capable of producing datasets of much larger sizes; e.g., for 90 full backups we could generate 1.2PB of test data.

Our experience has shown that it is often useful to save fstree objects (the object, not the full file system) to persistent storage. This allows us to reuse an fstree in different ways, e.g., representing the behavior of different users in a multi-tenant cloud environment. We designed the fstree so that it can be efficiently serialized to or from disk using only a single sequential I/O. Thus it takes less than two minutes to save or load a 9GB fstree on a modern 100MB/sec disk drive. Using a disk array can make this even faster.

3.3 Fstree Action Modules

An fstree represents a static image of a file system tree—a snapshot. Our framework defines several operations on fstrees, which are implemented by pluggable *action modules*; Figure 1 demonstrates their relationships. Double-boxed rectangles represent action modules; rounded ones designate inputs and outputs.

FS-SCAN. One way to obtain an initial fstree object (to be synthetically modified later) is to scan an existing file system. The FS-SCAN module does this: it scans content and meta-data, creates file, directory, and chunk objects, and populates per-file chunk lists. Different implementations of this module can collect different levels of detail about a file system, such as recognizing or ignoring symlinks, hardlinks, or sparse files, storing or skipping file permissions, using different chunking algorithms, etc.

FS-PROFILE, FS-IMPRESSIONS, and FS-POPULATE. Often, an initial file system is not available, or cannot be released even in the form of an fstree due to sensitive data. FS-PROFILE, FS-IMPRESSIONS, and FS-POPULATE address this problem. FS-PROFILE is similar to FS-SCAN, but does not collect such detailed information, instead gathering only a statistical profile. The spe-

Name	Total size (GB)	Total files (thousands)	Snapshots & period	Avg. snapshot size (GB)	Avg. number of files in a snapshot (thousands)
Kernels (Linux 2.6.0–2.6.39)	13	903	40	0.3	23
CentOS (5.0–5.7)	36	1,559	8	4.5	195
Home	3,482	15,352	15 weekly	227	1,023
MacOS	4,080	83,220	71 daily	59	1,173
System Logs	626	2,672	8 weekly	78	334
Sources	1,331	1,112	8 weekly	162	139

Table 1: Summary of analyzed datasets.

cific information collected depends on the implementation, but we assume it does not reveal sensitive data. We distinguish sub-parts: the *meta profile*, which contains statistics about the meta-data, and the *content profile*.

Several existing tools can generate a static file system image based on a meta-data profile [1, 8], and any of these can be reused by our system. A popular option is Impressions [1], which we modified to produce an fstree object instead of a file system image (FS-IMPRESSIONS). This fstree object is *empty*, meaning it contains no information about file contents. FS-POPULATE fills an empty fstree by creating chunks based on the content profile. Our current implementation takes the distribution of duplicates as a parameter; more sophisticated versions are future work.

The left part of Figure 1 depicts the two current options for creating initial fstrees. This paper focuses on the mutation module (below).

FS-MUTATE. FS-MUTATE is a key component of our approach. It mutates the fstree according to the changes observed in a real environment. Usually it iterates over all files and directories in the fstree and deletes, creates, or modifies them. A single mutation can represent weekly, daily, or hourly changes; updates produced by one or more users; etc. FS-MUTATE modules can be chained as shown in Figure 1 to represent multiple changes corresponding to different users, different times, etc. Usually, a mutation module is controlled by a parameterized profile based on real-world observations. The profile can also be chosen to allow microbenchmarking, such as varying the percentage of unique chunks to observe changes in deduplication behavior. In addition, if a profile characterizes the changes between an empty file system and a populated one, FS-MUTATE can be used to generate an initial file system snapshot.

FS-CREATE. After all mutations are performed, FS-CREATE generates a final dataset in the form needed by a particular deduplication system. In the most common case, FS-CREATE produces a file system by walking through all objects, creating the corresponding directories and files, and generating file contents based on the chunk identifiers. Content generation is implementation-specific; for example, contents might depend on the file type or on an entropy level. The important property to preserve is that the same chunk identifiers result in the same content, and different chunk identifiers produce different content. FS-CREATE could also generate tar-like files for input to a backup system, which can be significantly faster than creating a complete file system because it can use sequential writes. FS-CREATE could also generate only the files that have changed since the previous snapshot, emulating data coming from an incremental backup.

4 Datasets Analyzed

To create a specific implementation of the framework modules, we analyzed file system changes in six different datasets; in each case, we used FS-SCAN to collect hashes and file system tree characteristics. We chose two commonly used public datasets, two collected locally, and two originally presented by Dong et al. [6].

Table 1 describes important characteristics of our six datasets: total size, number of files, and per-snapshot averages. Our largest dataset, MacOS, is 4TB in size and has 83 million files spanning 71 days of snapshots.

Kernels: Unpacked Linux kernel sources from version 2.6.0 to version 2.6.39.

CentOS: Complete installations of eight different releases of the CentOS Linux distribution from version 5.0 to 5.7.

Home: Weekly snapshots of students' home directories from a shared file system. The files consisted of source code, binaries, office documents, virtual machine images, and miscellaneous files.

MacOS: A Mac OS X Enterprise Server that hosts various services for our research group: email, mailing lists, Web-servers, wiki, Bugzilla, CUPS server, and an RT trouble-ticketing server.

System Logs: Weekly unpacked backups of a server's `/var` directory, mostly consisting of emails stored by a list server.

Sources: Weekly unpacked backups of source code and change logs from a Perforce version control repository.

Of course, different environments can produce significantly different datasets. For that reason, our design is flexible, and our prototype modules are parameterized by profiles that describe the characteristics of a particular dataset's changes. If necessary, other researchers can use our profile collector to gather appropriate distributions, or implement a different FS-MUTATE model to express the changes observed in a specific environment.

For the datasets that we analyzed, we will release all profiles and module implementations publicly. We expect that future papers following this project will also publish their profiles and mutation module implementations, especially when privacy concerns prevent the release of the whole dataset. This will allow the community to reproduce results and better compare one deduplication system to another.

5 Module Implementations

There are many ways to implement our framework's modules. Each corresponds to a model that describes a dataset's behavior in a certain environment. An ideal model should capture the characteristics that most affect the behavior of a deduplication system. In this section we first explore the space of parameters that can affect the performance of a deduplication system, and then present a model for emulating our datasets' behavior. Our implementation can be downloaded from *https://avatar.fsl.cs.sunysb.edu/groups/deduplicationpublic/*.

5.1 Space Characteristics

Both content and meta-data characteristics are important for accurate evaluation of deduplication systems. Figure 2 shows a rough classification of relevant dataset characteristics. The list of properties in this section is not intended to be complete, but rather to demonstrate a variety of parameters that it might make sense to model.

Previous research has primarily focused on characterizing *static* file system snapshots [1]. Instead, we are interested in characterizing the file system's *dynamic* properties (both content and meta-data). Extending the analysis to multiple snapshots can give us information about file deletions, creations, and modifications. This in turn will reflect on the properties of static snapshots.

Any deduplication solution divides a dataset into chunks of fixed or variable size, indexes their hashes, and compares new incoming chunks against the index. If a new hash is already present, the duplicate chunk is discarded and a mapping that allows the required data to be located later is updated.

Therefore, the total number of chunks and the number of unique chunks in a dataset affects the system's performance. The performance of some data structures used in deduplication systems also depends on the distribution of duplicates, including the percentage of chunks with a certain number of duplicates and even the ordering of duplicates. E.g., it is faster to keep the index of hashes in RAM, but for large datasets a RAM index may be economically infeasible. Thus, many deduplication systems use sophisticated index caches and Bloom filters [25] to reduce RAM costs, complicating performance analysis.

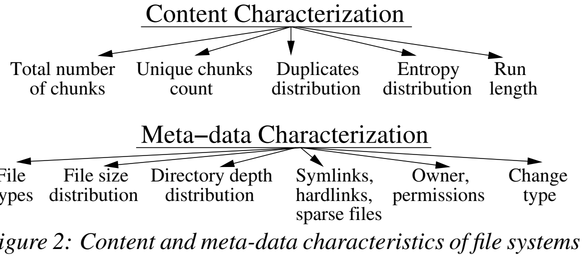


Figure 2: Content and meta-data characteristics of file systems that are relevant to deduplication system performance.

For many systems, it is also important to capture the entropy distribution inside the chunks, because most deduplication systems support local chunk compression to further reduce space. Compression can be enabled or disabled intelligently depending on the data type [12].

A deduplication system's performance depends not only on content, but also on the file system's *meta-data*. When one measures the performance of a conventional file system (without deduplication), the file size distribution and directory depth strongly impact the results [2]. Deduplication is sometimes an addition to existing conventional storage, in which case file sizes and directory depth will also affect the overall system performance.

The run lengths of unique or duplicated chunks can also be relevant. If unique chunks follow each other closely (in space and time), the storage system's I/O queues can fill up and throughput can drop. Run lengths depend on the ways files are modified: pure extension, as in log files; simple insertion, as for some text files; or complete rewrites, as in many binary files. Run lengths can also be indirectly affected by file size distributions, because it often happens that only a few files in the dataset change from one backup to another, and the distance between changed chunks within a backup stream depends on the sizes of the unchanged files.

Content-aware deduplication systems sometimes use the file header to detect file types and improve chunking; others use file owners or permissions to adjust their deduplication algorithms. Finally, symlinks, hardlinks, and sparse files are a rudimentary form of deduplication, and their presence in a dataset can affect deduplication ratios.

Dependencies. An additional issue is that many of the parameters mentioned above depend on each other, so considering their statistical distributions independently is not possible. For example, imagine that emulating the changes to a specific snapshot requires removing N files. We also want the total number of chunks to be realistic, so we need to remove files of an appropriate size. Moreover, the distribution of duplicates needs to be preserved, so the files that are removed should contain the appropriate number of unique and duplicated chunks. Preserving such dependencies is important, and our FS-MUTATE implementation (presented next) allows that.

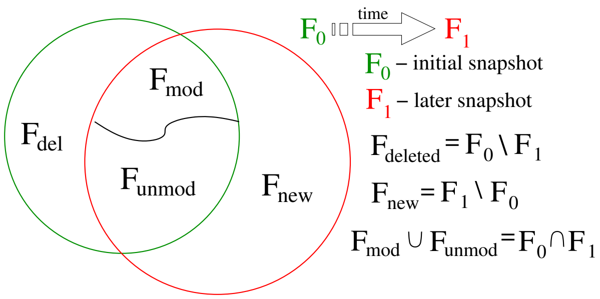

time
F_0
F_1
F_0 – initial snapshot
F_1 – later snapshot
F_{mod}
F_{del}
F_{unmod}
F_{new}
$F_{deleted} = F_0 \setminus F_1$
$F_{new} = F_1 \setminus F_0$
$F_{mod} \cup F_{unmod} = F_0 \cap F_1$

Figure 3: Classification of files. F_0 and F_1 are files from two subsequent snapshots.

5.2 Markov & Distribution (M&D) Model

We call our model *M&D* because it is based on two abstractions: a Markov model for classifying file changes, and a multi-dimensional distribution for representing statistical dependencies between file characteristics.

Markov model. Suppose we have two snapshots of a file system taken at two points in time: F_0 and F_1. We classify files in F_0 and F_1 into four sets: 1) F_{del}: files that exist in F_0, but are missing in F_1. 2) F_{new}: files that exist in F_1, but are missing in F_0. 3) F_{mod}: files that exist in both F_0 and F_1, but were modified. 4) F_{unmod}: files in F_0 and F_1 that were not modified. The relationship between these sets is depicted in Figure 3. In our study, we identify files by their full pathname, i.e., a file in the second snapshot with the same pathname as one in the first is assumed to be a later version of the same file.

Analysis of our datasets showed that the file sets defined above remain relatively stable. Files that were unmodified between snapshots $F_0 \rightarrow F_1$ tended to remain unmodified between snapshots $F_1 \rightarrow F_2$. However, files still migrate between sets, with different rates for different datasets. To capture such behavior we use the Markov model depicted in Figure 4. Each file in the fstree has a state assigned to it in accordance with the classification defined earlier. In the fstree representing the first snapshot, all files have the New state. Then, during mutation, the file states change with precalculated probabilities that have been extracted by looking at a window of three real snapshots, covering two file transitions: between the first and second snapshots and between the second and third ones. This is the minimum required to allow us to calculate conditional probabilities for the Markov model. For example, if some file is modified between snapshots $F_0 \rightarrow F_1$ and is also modified in $F_1 \rightarrow F_2$, then this is a Modified→Modified (MM) transition. Counting the number of MM transitions among the total number of state transitions allows us to compute the corresponding probability; we did this for each possible transition.

Some transitions, such as Deleted→New (DN), may seem counterintuitive. However, some files are recreated after being deleted, producing nonzero probabilities for this transition. Similarly, if a file is renamed or moved, it will be counted as two transitions: a removal and a creation. In this case, we allocate duplicated chunks to the new file in a later stage.

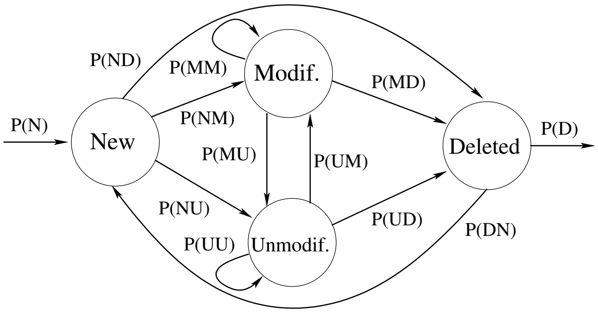


Figure 4: Markov model for handling file states. State transitions are denoted by the first letters of the source and destination states. For example, NM denotes a New→Modified transition and P(NM) is the transition's probability.

The Markov model allows us to accurately capture the rates of file appearance, deletion, and modification in the trace. Table 2 presents the average transition probabilities observed for our datasets. As mentioned earlier, in all datasets files often remain Unchanged, and thus the probabilities of UU transitions are high. The chances for a changed file to be re-modified are around 50% for many of our datasets. The probabilities for many other transitions vary significantly across different datasets.

Multi-dimensional distribution. When we analyzed real snapshots, we collected three multi-dimensional file distributions: $M_{del}(p_1, ..., p_{n_{del}})$, $M_{new}(p_1, ..., p_{n_{new}})$, and $M_{mod}(p_1, ..., p_{n_{mod}})$ for deleted, new, and modified files, respectively. The parameters of these distributions (p_1, ..., p_n) represent the characteristics of the files that were deleted, created, or modified. As described in Section 5.1, many factors affect deduplication. In this work, we selected several that we deemed most relevant for a generic deduplication system. However, the organization of our FS-MUTATE module allows the list of emulated characteristics to be easily extended.

All three distributions include these parameters:

depth: directory depth of a file;
ext: file extension;
size: file size (in chunks):
uniq: the number of chunks in a file that are not present in the previous snapshot (i.e., unique chunks);
dup1: the number of chunks in a file that have only one duplicate in the entire previous snapshot; and
dup2: the number of chunks in a file that occur exactly twice in the entire previous snapshot.

We consider only the chunks that occur up to 3 times in a snapshot because in all our snapshots these chunks constituted more than 96% of all chunks.

During mutation, we use the distribution of new files:

$$M_{new}(depth, ext, size, uniq, dup1, dup2)$$

Dataset	N	NM	NU	ND	MU	MD	MM	UM	UD	UU	DN	D
Kernels	5	32	65	3	49	3	48	17	3	80	1	3
CentOS	13	4	22	74	43	2	55	4	1	95	1	10
Home	4	2	78	20	54	10	36	0.14	0.35	99.51	6	0.50
MacOS	0.1	11	78	11	37.46	0.03	62.51	0.05	0.03	99.92	1	0.03
System Logs	2	9	90	1	44.40	0.18	55.42	0.03	0.01	99.06	4	0.02
Sources	0.2	7	88	5	58.76	0.04	41.20	0.07	0	99.93	0	0.01

Table 2: Probabilities (in percents) of file state transitions for different datasets. N: new file appearance. D: file deletion. NM: New→Modified transition. NU: New→Unmodified transition. ND: New→Deleted transition, etc.

to create the required number of files with the appropriate properties. E.g., if $M_{new}(2, \text{“.c”}, 7, 3, 1, 1)$ equals four, then FS-MUTATE creates four files with a "`.c`" extension at directory depth two. The size of the created files is seven chunks, of which three are unique, one has a single duplicate, and one has two duplicates across the entire snapshot. The hashes for the remaining two chunks are selected using a per-snapshot (not per-file) distribution of duplicates, which is collected during analysis along with M_{new}. Recall that FS-MUTATE does not generate the content of the chunks, but only their hashes. Later, during on-disk snapshot creation, FS-CREATE will generate the content based on the hashes.

When selecting files for deletion, FS-MUTATE uses the deleted-files distribution:

$$M_{del}(depth, ext, size, uniq, dup1, dup2, state)$$

This contains an additional parameter—*state*—that allows us to elegantly incorporate a Markov model in the distribution. The value of this parameter can be one of the Markov states New, Modified, Unmodified, or Deleted; we maintain the state of each file within the fstree. A file is created in the New state; later, if FS-MUTATE modifies it, its state is changed to Modified; otherwise it becomes Unmodified. When FS-MUTATE selects files for deletion, it limits its search to files in the state given by the corresponding M_{del} entry. For example, if $M_{del}(2, \text{“.c”}, 7, 3, 1, 1, \text{“Modified”})$ equals one, then FS-MUTATE tries to delete a single file in the Modified state (all other parameters should match as well).

To select files for modification, FS-MUTATE uses the M_{mod} distribution, which has the same parameters as M_{del}. But unlike deleted files, FS-MUTATE needs to decide *how* to change the files. For every entry in M_{mod}, we keep a list of *change descriptors*, each of which contains the file's characteristics *after* modification:

1. File size (in chunks);
2. The number of unique chunks (here and in the two items below, duplicates are counted against the entire snapshot);
3. The number of chunks with one duplicate;
4. The number of chunks with two duplicates; and
5. Change pattern.

All parameters except the last are self-explanatory. The change pattern encodes the way a file was modified. We currently support the following three options: B—the file was modified in the beginning (this usually corresponds to prepend); E—the file was modified at the end (corresponds to file extension or truncation); and M—the file was modified somewhere in the middle, which corresponds to the case when neither the first nor the last chunk were modified, but others have changed. We also support combinations of these patterns: *BE*, *BM*, *EM*, and *BEM*. To recognize the change pattern during analysis, we sample the corresponding chunks in the old and new files. Table 3 presents the average change patterns for different datasets. For all datasets the number of files modified in the beginning is high. This is a consequence of chunk-based analysis: files that are smaller than the chunk size contain a single chunk. Therefore, wherever small files are modified, the first (and only) chunk differs in two subsequent versions, which our analysis identifies as a change in the file's beginning. For the System Logs dataset, the number of files modified at the end is high because logs are usually appended. In the Sources dataset many files are modified in the middle, which corresponds to small patches in the code.

Dataset	B	E	M	BE	BM	ME	BEM
Kernels	52	8	7	14	5	3	11
CentOS	69	3	2	8	2	1	15
Home	38	3	8	10	11	1	29
MacOS	53	21	1	12	1	1	11
Sys. Logs	42	34	5	6	0	1	10
Sources	20	6	41	7	7	1	18

Table 3: Probabilities of the change patterns for different datasets (in percents).

We collect change descriptors and the M_{mod} distribution during the analysis phase. During mutation, when a file is selected for modification using M_{mod}, one of the aforementioned change descriptors is selected randomly and the appropriate changes are applied.

It is possible that the number of files that satisfy the distribution parameters is larger than the number that need to be deleted or modified. In this case, FS-MUTATE randomly selects files to operate on. If not enough files with the required properties are in the fstree, then FS-MUTATE tries to find the best match based on a simple

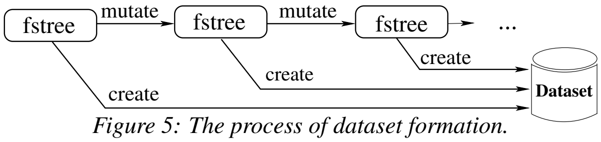
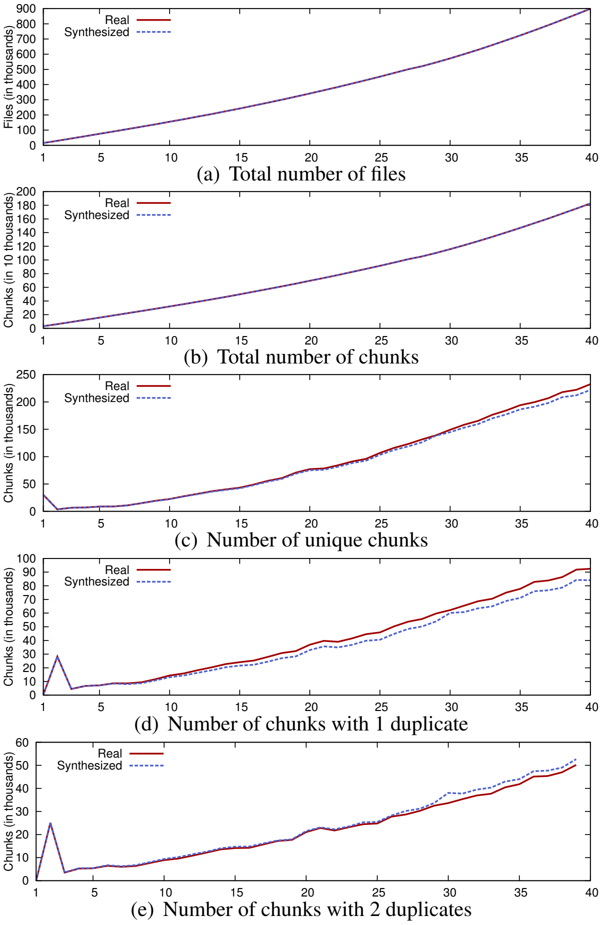


Figure 5: The process of dataset formation.



(a) Total number of files

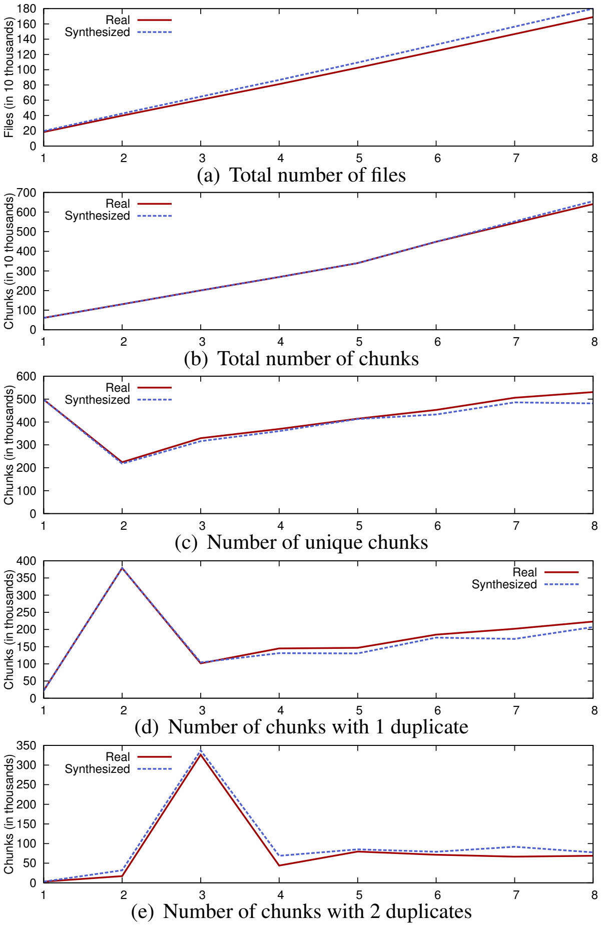
(b) Total number of chunks

(c) Number of unique chunks

(d) Number of chunks with 1 duplicate

(e) Number of chunks with 2 duplicates

Figure 6: Emulated parameters for Kernels real and synthesized datasets as the number of snapshots in them increases.



(a) Total number of files

(b) Total number of chunks

(c) Number of unique chunks

(d) Number of chunks with 1 duplicate

(e) Number of chunks with 2 duplicates

Figure 7: Emulated parameters for CentOS real and synthesized datasets as the number of snapshots in them increases.

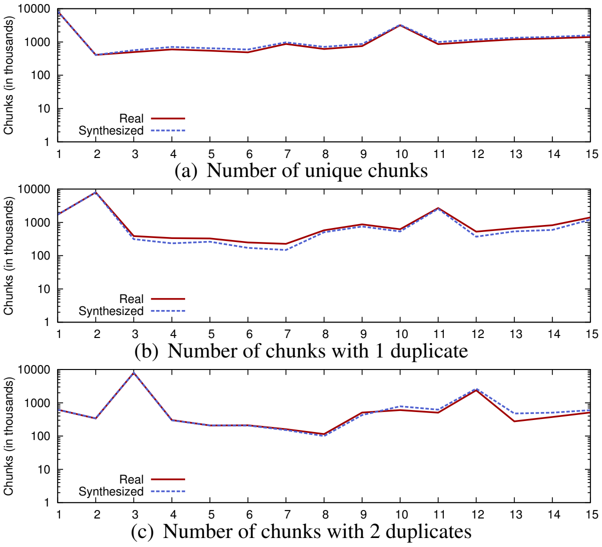
heuristic: the file that matches most of the properties. Other definitions of best match are possible, and we plan to experiment with this parameter in the future.

Multi-dimensional distributions capture not only the statistical frequency of various parameters, but also their interdependencies. By adding more distribution dimensions, one can easily emulate other parameters.

Analysis. To create profiles for our datasets, we first scanned them using the FS-SCAN module mentioned previously. We use variable chunking with an 8KB average size; variable chunking is needed to properly detect the type of file change, since prepended data causes fixed-chunking systems to see a change in every chunk. We chose 8KB as a compromise between accuracy (smaller sizes are more accurate) and the speed of the analysis, mutation, and file system creation steps.

The information collected by FS-SCAN was loaded into a database; we then used SQL queries to extract distributions. The analysis of our smallest dataset (Kernels) took less than 2 hours, whereas the largest dataset (MacOS) took about 45 hours of wall-clock time on a single workstation. This analysis can be sped up by parallelizing it. However, since it needs to be done only once to extract a profile, a moderately lengthy computation is acceptable. Mutation and generation of a file system run much faster and are evaluated in Section 6. The size of the resulting profiles varied from 8KB to 300KB depending on the number of changes in the dataset.

Chunk generation. Our FS-CREATE implementation generates chunk content by maintaining a randomly generated buffer. Before writing a chunk to the disk, this buffer is XORed with the chunk ID to ensure that each ID produces a unique chunk and that duplicates have the same content. We currently do not preserve the chunk's entropy because our scan tool does not yet collect this data. FS-SCAN collects the size of every chunk, which is kept in the in-memory fstree object for use by FS-CREATE. New chunks in mutated snapshots have their size set by FS-MUTATE according to a per-snapshot chunk-size distribution. However, deduplication systems can use *any* chunk size that is larger than or equal to the one that FS-SCAN uses. In fact, sequences of identical chunks may appear in several subsequent snapshots. As these sequences of chunks are relatively long, any



(a) Number of unique chunks

(b) Number of chunks with 1 duplicate

(c) Number of chunks with 2 duplicates

Figure 8: Emulated parameters for Homes real and synthesized datasets as the number of snapshots in them increases.

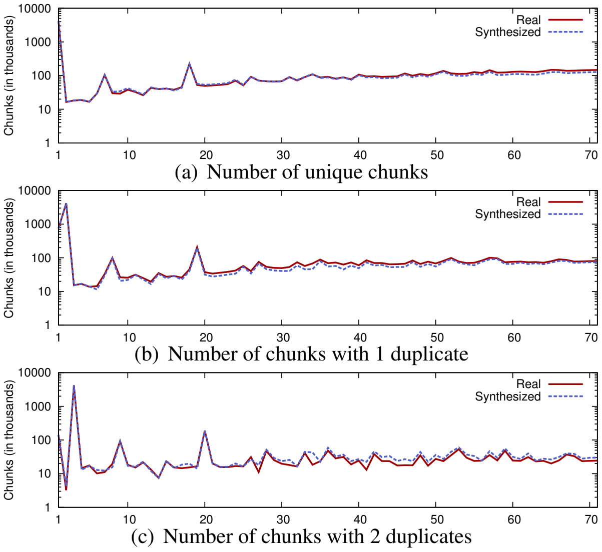
chunking algorithm can detect an appropriate number of identical chunks across several snapshots.

Security guarantees. The FS-SCAN tool uses 48-bit fingerprints, which are prefixes of 16 byte MD5 hashes; this provides a good level of security, although we may be open to dictionary attacks. Stronger anonymization forms can be easily added in the future work.

6 Evaluation

We collected profiles for the datasets described in Section 4 and generated the same number of synthetic snapshots as the real datasets had, chaining different invocations of FS-MUTATE so that the output of one mutation served as input to the next. All synthesized snapshots together form a synthetic dataset that corresponds to the whole real dataset (Figure 5). We generated the initial fstree object by running FS-SCAN on the real file system. Each time a new snapshot was added, we measured the total files, total chunks, numbers of unique chunks and those that had one and two duplicates, directory depth, file size and file type distributions.

First, we evaluated the parameters that FS-MUTATE emulates. Figures 6–11 contain the graphs for the real and synthesized Kernels, CentOS, Homes, MacOS, System Logs, and Sources datasets, in order. The Y axis scale is linear for the Kernels and Sources datasets (Figures 6–7) and logarithmic for the others (Figures 8–11). We present file and chunk count graphs only for the Kernels and CentOS datasets. The relative error of these two parameters is less than 1% for all datasets, and the graphs look very similar: monotonic close-to-linear growth. The file count is insensitive to modification operations because files are not created or removed, which explains its high accuracy. The total chunk count is maintained because we carefully preserve file size during creation, modification, and deletion.

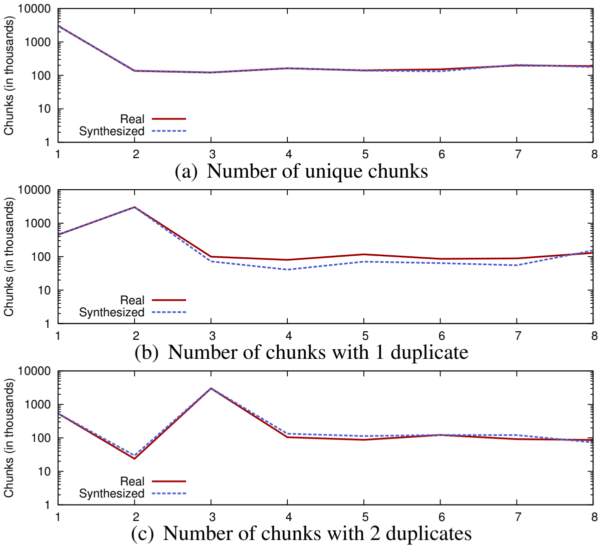


(a) Number of unique chunks

(b) Number of chunks with 1 duplicate

(c) Number of chunks with 2 duplicates

Figure 9: Emulated parameters for MacOS real and synthesized datasets as the number of snapshots in them increases.



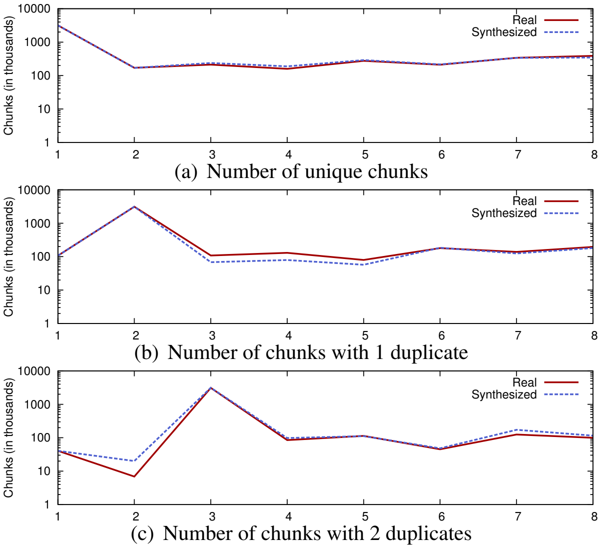
(a) Number of unique chunks

(b) Number of chunks with 1 duplicate

(c) Number of chunks with 2 duplicates

Figure 10: Emulated parameters for System Logs real and synthesized datasets as the number of snapshots in them increases.

For all datasets the trends observed in the real data are closely followed by the synthesized data. However, certain discrepancies exist. Some of the steps in our FS-MUTATE module are random; e.g., the files deleted or modified are not precisely the same ones as in the real snapshot, but instead ones with similar properties. This means that our synthetic snapshots might not have the same files that would exist in the real snapshot. As a result, FS-MUTATE cannot find some files during the following mutations and so the best-match strategy is used, contributing to the *instantaneous* error of our method.



(a) Number of unique chunks

(b) Number of chunks with 1 duplicate

(c) Number of chunks with 2 duplicates

Figure 11: Emulated parameters for Sources real and synthesized datasets as the number of snapshots in them increases.

However, because our random actions are controlled by the real statistics, the deviation is limited in the long run.

The graphs for unique chunks have an initial peak because there is only one snapshot at first, and there are not many duplicates in a single snapshot. As expected, this peak moves to the right in the graphs for chunks with one and two duplicates.

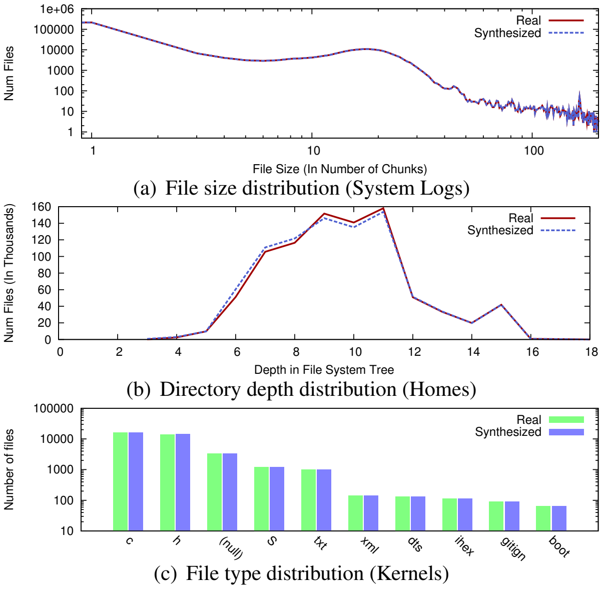
The Homes dataset has a second peak in all graphs around 10–12 snapshots (Figure 8). This point corresponds to two missing weekly snapshots. The first was missed due to a power outage; the second was missed because our scan did not recover properly from the power outage. As a result, the 10th snapshot contributes many more unique chunks in the dataset than the others.

The MacOS dataset contains daily, not weekly snapshots. Daily changes in the system are more sporadic than weekly ones: one day users and applications add a lot of new data, the next many files are copied, etc. Figure 9 therefore contains many small variations.

Table 4 shows the relative error for emulated parameters at the end of each run. Maximum deviation did not exceed 15% and averaged 6% for all parameters and datasets. We also analyzed the file size, type, and directory depth distributions in the final dataset. Figure 12 demonstrates these for several representative datasets. In all cases the accuracy was fairly high, within 2%.

The snapshots in our datasets change a lot. For example, the deduplication ratio is less than 5 in our Kernels dataset, even though the number of snapshots is 40. We expect the accuracy of our system to be higher for the datasets that change slower; for instance, datasets with identical snapshots are emulated without any error.

Performance. We measured the time of every mutation and creation operation in the experiments above.



(a) File size distribution (System Logs)

(b) Directory depth distribution (Homes)

(c) File type distribution (Kernels)

Figure 12: File size, type, and directory depth distributions for different real and synthesized dataset.

Dataset	**Files**	**Chunks**	**Unique chunks**	**1 Dup. chunks**	**2 Dup. chunks**
Kernels	< 1	< 1	4	9	5
CentOS	6	2	9	7	11
Home	< 1	< 1	12	13	14
MacOS	< 1	< 1	4	9	4
Sys. Logs	< 1	< 1	6	15	15
Sources	< 1	< 1	10	8	13

Table 4: Relative error of emulated parameters after the final run for different datasets (in percents).

Dataset	**Total size (GB)**	**Snap-shots**	**Mutat. time**	**Creat. time**	**Total time**
Kernels	13	40	30 sec	6 sec	5 min
CentOS	36	8	3 min	95 sec	13 min
Home	3,482	15	44 min	38 min	10 hr
MacOS	4,080	71	49 min	10 min	13 hr
Sys. Logs	626	8	14 min	4 hr	32 hr
Sources	1,331	8	21 min	4 hr	32 hr

Table 5: Times to mutate and generate data sets.

The Kernels, CentOS, Home, and MacOS experiments were conducted on a machine with an Intel Xeon X5680 3.3GHz CPU and 64GB of RAM. The snapshots were written to a single Seagate Savvio 15K RPM disk drive. For some datasets the disk drive could not hold all the snapshots, so we removed them after running FS-SCAN for accuracy analysis. Due to privacy constraints the System Logs and Sources experiments were run on a different machine with an AMD Opteron 2216 2.4GHz CPU, 32GB of RAM, and a Seagate Barracuda 7,200 RPM disk drive. Unfortunately, we had to share the second machine with a long-running job that periodically performed random disk reads.

Table 5 shows the *total* mutation time for all snapshots, the time to write a *single* snapshot to the disk, and the total time to perform all mutations plus write the whole dataset to the disk. The creation time includes the time to write to disk. For convenience the table also contains dataset sizes and snapshot counts.

Even for the largest dataset, we completed all mutations within one hour; dataset size is the major factor in mutation time. Creation time is mostly limited by the underlying system's performance: the creation throughput of the Home and MacOS datasets is almost twice that of Kernels and CentOS, because the average file size is 2–10× larger for the former datasets, exploiting the high sequential drive throughput. The creation time was significantly increased on the second system because of a slower disk drive (7,200RPM vs. 15KRPM) and the interfering job, contributing to the 32-hour run time.

For the datasets that can fit in RAM—CentOS and Kernels—we performed an additional FS-CREATE run so that it creates data on tmpfs. The throughput in both cases approached 1GB/sec, indicating that our chunk generation algorithm does not incur much overhead.

7 Related Work

A number of studies have characterized file system workloads using I/O traces [13, 19, for example] that contain information about all I/O requests observed during a certain period. The duration of a full trace is usually limited to several days, which makes it hard to analyze long-term file system changes. Trace-based studies typically focus on the dynamic properties of the workload, such as I/O size, read-to-write ratio, etc., rather than file content as is needed for deduplication studies.

Many papers have used snapshots to characterize various file system properties [2, 3, 20, 22]. With the exception of Agrawal et al.'s study [2], discussed below, the papers examine only a single snapshot, so only static properties can be extracted and analyzed. Because conventional file systems are sensitive to meta-data characteristics, snapshot-based studies focus on size distributions, directory depths or widths, and file types (derived from extensions). File and block lifetimes are analyzed based on timestamps [2, 3, 22]. Authors often discuss the correlation between file properties, e.g., size and type [3,20]. Several studies have proposed high-level explanations for file size distributions and designed models for synthesizing specific distributions [7, 20].

Less attention has been given to the analysis of long-term file system changes. Agrawal et al. examined the trends in file system characteristics from 2000–2004 [2]. The authors presented only meta-data evolution: file count, size, type, age, and directory width and depth.

Some researchers have worked on artificial file system aging [1, 21] to emulate the fragmentation encountered in real long-lived file systems. Our mutation module modifies the file system in RAM and thus does not emulate file system fragmentation. Modeling fragmentation can be added in the future if it proves to impact deduplication systems' performance significantly.

A number of newer studies characterized deduplication ratios for various datasets. Meyer and Bolosky studied content and meta-data in primary storage [15]. The authors collected file system content from over 800 computers and analyzed the deduplication ratios of different algorithms: whole-file, fixed chunking, and variable chunking. Several researchers characterized deduplication in backup storage [17, 23] and for virtual machine disk images [11, 14]. Chamness presented a model for storage-capacity planning that accounts for the number of duplicates in backups [4]. None of these projects attempted to synthesize datasets with realistic properties.

File system benchmarks usually create a test file system from scratch. For example, in Filebench [8] one can specify file size and directory depth distributions for the creation phase, but the data written is either all zeros or random. Agrawal et al. presented a more detailed attempt to approximate the distributions encountered in real-world file systems [1]. Again, no attention was given in their study to generating duplicated content.

8 Conclusions and Future Work

Researchers and companies evaluate deduplication with a variety of datasets that in most cases are private, unrepresentative, or small in size. As a result, the community lacks the resources needed for fair and versatile comparison. Our work has two key contributions.

First, we designed and implemented a generic framework that can emulate the formation of datasets in different scenarios. By implementing new mutation modules, organizations can expose the behavior of their internal datasets without releasing the actual data. Other groups can then regenerate comparable data and evaluate different deduplication solutions. Our framework is also suitable for controllable micro-benchmarking of deduplication solutions. It can generate arbitrarily large datasets while still preserving the original's relevant properties.

Second, we presented a specific implementation of the mutation module that emulates the behavior of several real-world datasets. To capture the meta-data and content characteristics of the datasets, we used a hybrid Markov and Distribution model that has a low error rate—less than 15% during 8 to 71 mutations for all datasets. We plan to release the tools and profiles described in this paper so that organizations can perform comparable studies of deduplication systems. These powerful tools will help both industry and research to make intelligent decisions when selecting the right deduplication solutions for their specific environments.

Future Work. Our specific implementation of the framework modules might not model all parameters that potentially impact the behavior of existing deduplication systems. We plan to conduct a study similar to Park et al. [18] to create a complete list of the dataset properties that impact deduplication systems. Although we can generate an initial file system snapshot using a specially collected profile for FS-MUTATE, such approach can be limiting. We plan to perform an extensive study on how to create initial fstree objects. Many deduplication systems perform local chunk compression to achieve even higher aggregate compression. We plan to incorporate into our framework a method for generating chunks with a realistic compression ratio distribution. Finally, we want to apply clustering techniques to detect trend lines so that more generic profiles can be designed.

Acknowledgments. We thank the anonymous Usenix reviewers and our shepherd for their help. This work was made possible in part thanks to NSF awards CCF-0937833 and CCF-0937854, a NetApp Faculty award, and an IBM Faculty award.

References

[1] N. Agrawal, A. C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, 2009.

[2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, 2007.

[3] J. M. Bennett, M. A. Bauer, and D. Kinchlea. Characteristics of files in NFS environments. *ACM SIGSMALL/PC Notes*, 18(3-4):18–25, 1992.

[4] M. Chamness. Capacity forecasting in a backup storage environment. In *Proceedings of USENIX Large Installation System Administration Conference (LISA)*, 2011.

[5] EMC Corporation. EMC Centra: content addressed storage systems. Product description guide, 2004.

[6] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the Nineth USENIX Conference on File and Storage Technologies (FAST '11)*, 2011.

[7] A. B. Downey. The structural cause of file size distributions. In *Proceedings of IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, 2001.

[8] Filebench. *http://filebench.sourceforge.net*.

[9] Advanced Storage Products Group. Identifying the hidden risk of data deduplication: how the HYDRAstor solution proactively solves the problem. Technical Report WP103-3_0709, NEC Corporation of America, 2009.

[10] N. Jain, M. Dahlin, and R. Tewari. TAPER: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST '05)*, 2005.

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

[12] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and performance evaluation of lossless file data compression on server systems. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, 2009.

[13] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC '08)*, 2008.

[14] A. Liguori and E. Hensbergen. Experiences with content addressable storage and virtual disks. In *Proceedings of the Workshop on I/O Virtualization (WIOV '08)*, 2008.

[15] D. Meyer and W. Bolosky. A study of practical deduplication. In *Proceedings of the Nineth USENIX Conference on File and Storage Technologies (FAST '11)*, 2011.

[16] NetApp. NetApp deduplication for FAS. Deployment and implementation, 4th revision. Technical Report TR-3505, NetApp, 2008.

[17] N. Park and D. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2010.

[18] N. Park, W. Xiao, K. Choi, and D. J. Lilja. A statistical evaluation of the impact of parameter selection on storage system benchmark. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2011.

[19] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the Annual USENIX Technical Conference*, 2000.

[20] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP '81)*, 1981.

[21] K. A. Smith and M. I. Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the 1997 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1997)*, 1997.

[22] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 1999.

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

[24] W. Xia, H. Jiang, D. Feng, and Y. Hua. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

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