Efficient Corruption-Free Duplicate Elimination
in Distributed Storage Systems:
Extended Version

João Barreto and Paulo Ferreira
Distributed Systems Group - INESC-ID/Technical University of Lisbon
[joao.barreto,paulo.ferreira]@inesc-id.pt
November 2008

1 Introduction

This document is an addenda to the paper entitled "dedupFS: Efficient Corruption-Free Data Deduplication in Distributed Storage Systems". The following sections describe the complete evaluation of dedupFS, which is only summarized in the main paper. For a copy of the main paper, please send an email to the authors.
2 Complete Evaluation

This section evaluates dedupFS under different aspects, from bandwidth efficiency to synchronization performance, comparing it with relevant alternatives such as rsync, xdelta, diff, LBFS and gzip-compressed file transfer. We analyze experimental results taken from representative workloads.

2.1 Experimental Setting

We have obtained the results by synchronizing different workloads between two distributed sites. A first site ran Ubuntu 6.06 (kernel version 2.6.15) on a Pentium 4 3GHz processor with 1GB of RAM. The second site ran Debian 4.0 (kernel version 2.6.18) on an Athlon 64 processor 3200+ with 2 GB of RAM. All experiments were run during periods of negligible load from other processes running at each machine (approximately < 1% CPU usage by other applications). A 100 Mbps full-duplex Ethernet LAN interconnected both sites.

2.1.1 Methodology

We consider different, real workloads when emulating collaborative update activity at each site. For different workloads and settings, we measured both the transmitted volume of data, and the access and synchronization performance of each evaluated solution. For the first measures, we use statistical data that each evaluated solution outputs (as we explain next). Hereafter, we use the symbol B to represent an 8-bit byte, and the symbols KB, MB and GB to, respectively denote 1024 B, 1024 KB and 1024 MB.

With respect to performance measures, we used the time command of Linux, with millisecond precision. Unless when stated, the performance results are an average of 3 executions of the same experiment.
Each workload represents the evolution of a shared directory, containing a given set of files and sub-directories, along a given time period. The workload directory consists of a single r-unit. We study two types of workloads: *sequential* and *concurrent* workloads.

**Sequential workloads.** Sequential workloads capture scenarios where a single writer site is producing a chronologically ordered sequence of versions of the workload directory, $d_1, d_2, ..., d_n$. At each version, the writer site may modify files and sub-directories from the previous version, as well as create new files and sub-directories. Reader sites, in turn, keep replicas of the workload directory and occasionally synchronize with the writer’s replicas, in order to obtain new versions as they become available.

Except where noted, we restrict our evaluation to the transition from $d_1$ to $d_n$, for two reasons. The first reason is simplicity, since analyzing each intermediate version transition is not relevant for most of following evaluation.\(^1\) Secondly, for most workloads we consider (e.g. open-source code projects), the intermediate versions are not publicly available.

The experimental procedure emulates the synchronization of the shared workload directory from one writer site to one reader site.\(^2\) Starting with both sites empty, the procedure consists of the following steps, which Figure 1 depicts:

1. We recursively copy the contents of $d_1$ into the writer site. In the case of dedupFS, such files and directories are copied into directory "1" of the local dedupFS mount directory.
2. We synchronize the (previously empty) reader site with the writer site. In this moment, both sites hold $d_1$.
3. We recursively copy the contents of $d_n$ into the writer site. In the case of dedupFS, into directory "2" of the local dedupFS mount directory.
4. We again synchronize the reader site with the writer site. In this moment, both sites hold $d_1$ and $d_n$.

**Concurrent workloads.** Concurrent workloads illustrate multiple-writer scenarios. In this case, two writer sites, $a$ and $b$, start with a common, initial version of the workload directory, $d_1$. Each writer site then independently modifies its local replica of the workload directory, respectively producing versions $d_a$ and $d_b$. After such a concurrent update activity, both sites synchronize their replicas.

More precisely, the procedure is the one that Figure 1 presents:

1. We recursively copy the contents of $d_1$ into directory "1" of $A$’s mount directory.

---

\(^1\)The option of a two-version analysis was also taken for the evaluation of most relevant related work, e.g. [14, 10, 6, 7].

\(^2\)Again, this is the same experimental procedure that relevant related work, such as [14, 10, 6, 7], follows.
2. We synchronize $B$ with $A$. In this moment, both sites hold $d_1$.

3. We recursively copy the contents of $d_a$ into directory "2a" of $A$’s mount directory. In parallel, we recursively copy the contents of $d_b$ into directory "2b" of $B$’s mount directory.

4. We synchronize $B$ with $A$. In this moment, $B$ holds $d_1$, $d_a$ and $d_b$.

The contents of $d_a$ and $d_b$ can reflect out-of-bound interactions that have passed unnoticed by dedupFS while each site concurrently produced each such divergent version, as Figure 1 illustrates. Notice, however, that such interactions are already implicit in the contents of $d_a$ and $d_b$ and, therefore, do not constitute any explicit step of the experiment.

### 2.1.2 Workloads

Workload selection was initially driven by the standard workloads that related data deduplication literature uses to evaluate proposed solutions. In particular, we have closely followed the set of workloads adopted by Jain, Dahlin and Tewari when evaluating the TAPER protocol [10]. Their workloads reflect a wide range of realistic usage scenarios. Furthermore, with rare exceptions\(^3\), their set of workloads is representative of the workloads found in other related work.

The standard workloads consist of sequential workloads. In order to attain a more complete evaluation, we introduce two concurrent workloads, obtained from real data.

Table 1 presents the considered workloads along with their overall characteristics, including number of versions, number of files and plain sizes. Every workload comprises considerably large overall contents (ranging from more than 100 MB up to more than 2 GB) and file sets (ranging from more than 850 files up to more than 42,000 files). The large dimensions of the considered workloads prove the robustness of the current dedupFS prototype implementation, which was able to correctly store and synchronize such large file sets. At the same time, such large samples lend strong statistical value to the results that we present next.

\(^3\)For instance, [6] uses animation rendering data snapshots. We could not use such a workload, since it was not publicly available.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Type</th>
<th>#Files</th>
<th>$d_1$ size (B)</th>
<th>$d_a$ size (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>course-docs</td>
<td>Concurrent</td>
<td>9,485</td>
<td>29,567,088</td>
<td>82,480,704 ($d_a$); 43,418,272 ($d_b$)</td>
</tr>
<tr>
<td>student-projects</td>
<td>Concurrent</td>
<td>6,487</td>
<td>22,643,346</td>
<td>30,160,055 ($d_a$); 9,083,818 ($d_b$)</td>
</tr>
<tr>
<td>gcc</td>
<td>Sequential</td>
<td>42,036</td>
<td>142,105,997</td>
<td>173,078,627</td>
</tr>
<tr>
<td>emacs</td>
<td>Sequential</td>
<td>4,210</td>
<td>45,988,215</td>
<td>54,730,114</td>
</tr>
<tr>
<td>linux</td>
<td>Sequential</td>
<td>28,573</td>
<td>156,568,220</td>
<td>162,381,404</td>
</tr>
<tr>
<td>usrbin</td>
<td>Sequential</td>
<td>8,101</td>
<td>201,681,977</td>
<td>333,910,547</td>
</tr>
</tbody>
</table>

Table 1: Overall characteristics of evaluated workloads.
We may divide the workloads into the following three categories:

1. Collaborative document editing.

This category is the closest to the collaborative scenarios that the thesis mainly targets. Furthermore, it is the only category to include concurrent workloads.

Two workloads exist in this category. We obtained both from real data from the Distributed Systems (DS) and Software Engineering (SE) undergraduate courses of the Instituto Superior Técnico faculty of Lisbon. Data spans across a complete 6-month semester.

A first workload, called course-docs, consists of snapshots of the CVS repositories used by the lecturers of both courses to keep pedagogical material (e.g. tutorials, lecture slides, code examples and project specifications) and private documents (e.g. individual student evaluation notes and management documents). The type of files varies significantly, ranging from text files such as Java code files or html files, to binary files such as pdf documents, Java libraries, Microsoft Word documents and Microsoft Excel worksheets.

Both courses were tightly coordinated, since their students are partially evaluated based on a large code project that is common to both courses. Consequently, lecturers of both courses carried an eminently collaboratively activity, involving regular meetings.

The lecturers maintained two CVS repositories: a common DS&SE repository, which lecturers of both courses accessed to share documents related to both courses; and a DS-only repository, which only DS lecturers used.

The course-docs workload starts ($d_1$) with a snapshot of both repositories at the beginning of the semester. In this moment, both repositories have old documents from previous years. Such an initial version then diverges to two concurrent versions. A first one, $d_a$, contains a snapshot of the DS-only repository at the end of the semester; a second one, $d_b$ includes the new contents of the common DS&SE repository at the end of the semester.

We believe that this workload is an very good case to evaluate the amount of out-of-band redundancy. The concurrent evolutions from $d_1$ to $d_a$ and $d_b$ depend on data that was copied from $d_1$, which dedupFS is able to detect and exploit. However, the users of both repositories had regular out-of-band interactions, such as frequent email exchanges, and weekly meetings. Such out-of-band interactions allow new data (i.e. not originally present in $d_1$) to be exchanged between $d_a$ and $d_b$ outside the control of dedupFS.

By studying this workload, we are able to quantify the amount of such out-of-band redundancy.

---

4The difference being that each course evaluates different aspects of the project that students deliver.
A second workload, student-projects, captures the collaborative work among students of both courses. Teams of 9 students were given the assignment of developing a code project in Java, which they were demanded to develop on a CVS repository. To ease their coding effort, they were provided with a bundle of auxiliary Java libraries and illustrative code examples. Most teams relied on this bundle as a basis from which they started developing their final project.

The project lasted for 3 months. The workload starts \( (d_1) \) with a snapshot of the initial library/code bundle that was made available to every student. We then randomly selected two final projects from the CVS repositories, respectively calling them \( d_a \) and \( d_b \). The snapshots consist mainly of binary Java libraries, Java code files and text-based configuration files, as well as Microsoft Word and pdf documents.

Similarly to the course-docs workload, the collaborative scenario captured by student-projects had sources of out-of-band redundancy. During the 3 months, the students were provided with additional code examples via the courses’ web sites. The students could freely download and use such examples in their projects; such a web source is not traceable by de-dupFS, hence contributing to out-of-band redundancy. Furthermore, the two teams had weekly laboratory courses together, where they discussed their current solution and borrowed possibly similar ideas and code samples from a common teaching assistant.

We have also evaluated the previous concurrent workloads as sequential workloads. In this case, we considered the sequential transition from \( d_1 \) to \( d_a \).

2. Software development sources.

A second set of workloads focuses on collaborative code development scenarios. These workloads include, for different real-world open-source projects, two consecutive versions of their source code releases. Namely, we have selected the source trees of recent versions of the gcc compiler, the emacs editor, and the Linux kernel. The choice of projects and versions is the same as adopted\(^5\) for the evaluation of the TAPER data deduplication scheme [10]. Using similar workloads as those used with TAPER allows us to compare our results with theirs.

The gcc workload comprises the source tree for GNU gcc versions 3.3.1 and 3.4.1. The emacs workload includes the source code for GNU Emacs versions 20.1 and 20.7. The linux workload contains the source tree of the Linux kernel versions 2.4.22 and 2.4.26. With the exception of some files in gcc, the source trees consist exclusively of ASCII text files.

3. Operating system executable binaries.

\(^5\)With the exception of the rsync workload, which we omit because we were not able to find identical versions as those that [10] consider.
The previous workloads contain a large amount of text-based files. This last workload, `usrbin`, considers binary files, which have very different characteristics when compared with text-based files; e.g. the achievable data compression rates and cross-file and cross-version data similarity tend to be substantially lower for binary than text-based files.

The `usrbin` workload includes the full contents of the `/usr/bin` directory trees of typical installations of the Ubuntu 6.06 and 7.10 64-bit Linux distributions. The `/usr/bin` directory contains most of the executable code binaries that are bundled with a Linux installation. Many of the executable files in the `/usr/bin` directories of versions 6.06 and 7.10 have common names; however, most of them differ in the source code version from which they originate, or in the compilation options that produced the code binaries.

2.1.3 Evaluated Solutions

We evaluate dedupFS with and without zlib compression. We considered different expected chunk sizes, namely 128 B, 2 KB and 8 KB. The maximum and minimum chunk size parameters were set in function of the expected chunk size; the maximum size was twice the expected chunk size, while the minimum size was a quarter of the expected chunk size.

In dedupFS, we used a directory cache of 150 directories, where each cached directory includes a maximum of 10,000 entries. Further, the size of the index array of the in-memory chunk hash table is 1024 elements. Variations to directory cache size and hash table index array size affect the performance of dedupFS. However, in order to limit our analysis to an acceptable number of variables, we assume the previous parameters as constant in the remainder of the section. We believe that the chosen values are reasonable for most devices that dedupFS targets.

We compare dedupFS with a relevant set of state-of-the-art alternative solutions to data deduplication. All state-of-the-art approaches to data deduplication are represented by at least one solution. Moreover, we also consider the basic approaches of plain transference and compression-only transference.
Table 2 sums up the characteristics of each solution evaluated, which we detail next.

- **rsync**, a Linux tool that employs compare-by-hash with fixed-size chunks [15]. We used version 2.6.3 of rsync. For a fair comparison with dedupFS, we disabled data encryption. We evaluated transference both with and without zlib compression [5] (-r and -rz options, respectively). When compression was on, we used the default compression level of zlib (level 6), the same zlib level used with dedupFS. We used the default fixed chunk size, 700 B. For data volume measures, we resorted to information output by the --stats option.

The sender site ran an rsync daemon, with access to the version directories of some workload. The experiment consisted of running rsync at the receiver site, each time requesting one workload directory version (\(d_1, d_n\), etc) to a target directory in the sender file system. Therefore, successive synchronization sessions took advantage of similar contents that already existed in the target directory, belonging to the previously downloaded version directory.

- **TAPER**, a compare-by-hash-based distributed data deduplication protocol [10]. Although TAPER is not publicly available, their authors have published experimental results [10] taken with some of the sequential workloads that we consider. For some of such measures, we were able to replay the same experiments with the same exact workloads, therefore obtaining results that allow a meaningful comparison with TAPER.

The results used with TAPER were obtained with an expected chunk size of 4 KB and a maximum chunk size of 64 KB for TAPER’s variable-size chunk phase (Phase II [10]). TAPER further complements such a phase with three other phases.

- **dedup-lbfs**, our implementation of the compare-by-hash protocol with variable-sized chunks used by the Low-Bandwidth File Solution (LBFS) [14]. We do not directly evaluate LBFS because no public stand-alone implementation of the original solution was available. The experimental methodology with dedup-lbfs was analogous to the one we followed with dedupFS.

- **diff**, the popular Linux tool relying on delta-encoding. diff detects similarities between two text files, outputting a sequence of added, deleted and changed text extracts. Such extracts encode a space-efficient delta that is sufficient to reconstruct the second file from the first file.

---

6In fact, the implementation of LBFS is partially available inside the SFS [13] framework. However, such an implementation does not allow running LBFS as a stand-alone file solution, as described in the original paper. In particular, it uses a cryptographic transport, which adds a significant overhead to the synchronization performance of LBFS. For this reason, we chose to re-implement LBFS using dedupFS as a base framework.
diff does not support distributed data deduplication by itself. However, diff can be easily complemented with some simple synchronization protocol to offer distributed data deduplication. With measurements taken with stand-alone diff, we are able to evaluate the relevant aspects of one such hypothetical distributed solution.

We used version 2.8.1 of diff. We recursively applied diff between the root directories of $d_1$ and $d_n$ (-ruN options). We set diff for highest precision in similarity detection (-d option). Since diff only works with text files, we only consider diff with the workloads that exclusively include text files.

- **xdelta**, a Linux tool for archiving and compressing consecutive versions of a file [11]. Similarly to diff, xdelta uses the delta-encoding approach. It runs a different algorithm to detect similarities between two consecutive versions of a given file [11]. xdelta then encodes the second version of the file as a delta file, consisting of a sequence of **insert** and **copy** instructions. Similarly to diff, xdelta does not support distributed data deduplication if used stand-alone. We used the same procedure as the one used with diff to partially evaluate an hypothetical solution that leveraged a simple synchronization protocol with xdelta. One fundamental difference to diff is that xdelta is able to compute deltas for both text and binary files. Hence, we evaluate xdelta with all workloads that this section considers.

  We used version 1.1.3 of xdelta, both with and without zlib compression of deltas (again, we used the default zlib compression level). We disabled xdelta’s option of MD5 checksums in delta files for a fair data volume comparison with other solutions.

- **version-control-only**, which consists of a simple shell script that approximately emulates synchronization protocols such as CVS’s [3]. Essentially, the script recursively compares two workload directory versions (e.g. $d_1$ and $d_n$) file-by-file and filters out the files with similar relative pathnames under each workload directory version that have identical contents. If some version tracking took place, the synchronization protocol would only propagate the remaining files, and ignore the unmodified files (which would have the same version in both workload directory versions).

  This script is useful since it tells us the data volume that we would be able to avoid propagating during synchronization by tracking version histories, without resorting to data deduplication or compression.

- **plain**, plain transference of files. This alternative consisted of dedup-FS with the deduplication protocol disabled.

- **gzip**, direct transference of zlib-compressed files [5]. The same as above, with the difference that each file is compressed before being transferred.
2.2 Sequential Workloads

In a first experiment, we ran each sequential workload with every solution (and corresponding variants). Since out-of-band redundancy may only arise from concurrent work, sequential workloads have no out-of-band redundancy. Hence, the following results evaluate the effectiveness of the considered solutions when detecting in-band redundancy only.

Most of the sequential workloads that we evaluate next closely resemble the very workloads that recent literature on data deduplication uses as motivational scenarios and based on which their authors advocate their solutions. More precisely, gcc, emacs, linux and usrbin are either the same workloads or very similar equivalents of the ones used by [14], [10] and [7], for example.

Before actually experimenting with different solutions, we deduce some intrinsic compressibility and redundancy characteristics of each sequential workload we consider. Namely:

1. A first value is given by the zlib-compressed size of $d_n$, relative to its plain size; this constitutes a good estimate of the data compressibility of each workload.

2. A second value was obtained by dividing the file contents of $d_n$ into chunks, using LBFS’s algorithm with an expected chunk size of 128 B, and adding such chunks to a repository that already held the chunks of $d_1$. Then, we measured the size of the chunks of $d_n$ that didn’t yet existed in the repository, i.e. which were not redundant across $d_1$ and $d_n$. This value, obtained with an efficient algorithm using a relatively low expected chunk size, is a strong estimate of the data redundancy of $d_n$ across $d_1$ and $d_n$.

3. Finally, a third value indicates how much workload contents reside in files whose contents and names remain the same from $d_1$ to $d_n$. We calculated such a value by running diff between files with similar relative pathnames from both directories, $d_1$ and $d_n$. This value helps us understand the write patterns that produce $d_n$. From it, we can distinguish whether write activity is concentrated in a small number of files of spread across many files.

Table 3 presents the previous values for each sequential workload. It shows that the set of sequential workloads we study covers a wide space of workloads, whichever axis we consider. The majority of workloads are highly compressible, in contrast to course-docs and usrb in, which exhibit relatively low compressibility. Workloads such as student-projects and usrb in have no or almost no file that remains unmodified, while in linux only less than 40% of file content belongs to files that were actually written. Finally, most workloads have relatively high redundancy across their contents (42% to 55% in redundant chunks). usrb in is the negative exception, with less than 30% of redundant content, while linux sets the redundancy record with 88%.

Before proceeding with the evaluation of each solution, it is pertinent to discuss the effective meaning of an analysis of sequential workloads. In theory,
Workload | Compressed Size | Literal Chunks | Size of Untouched Files |
---|---|---|---|
**course-docs** | 69.7% | 55.3% | 5.9% |
**student-projects** | 31.5% | 57.1% | 0.0% |
**gcc** | 24.0% | 44.3% | 11.6% |
**emacs** | 28.7% | 46.4% | 12.4% |
**linux** | 25.5% | 11.1% | 61.5% |
**usrbin** | 42.8% | 71.6% | 0.2% |

Table 3: Analysis of evaluated sequential workloads, regarding compressibility, touched files and chunk redundancy. Compressibility assumes zlib’s data compression algorithm, while chunk redundancy assume LBFS’s algorithm with an expected chunk size of 128 B, applied locally.

...the conclusions that we take from the results we present next are inevitable incomplete. For, in an optimistically replicated system, concurrency across replicas can always occur and, consequently, out-of-band redundancy may arise.

However, provided that, in practice, such out-of-band redundancy is negligible when compared to in-band redundancy, the present analysis is completely valuable to evaluate dedupFS. For the moment, we assume such a premiss: that out-of-band redundancy only arises at negligible levels (when compared to the in-band redundancy). Section 2.4.1 will then try to validate such a premiss with real concurrent workloads.

### 2.2.1 Transferred Volumes

We start by analyzing how much data and meta-data each solution needs to transfer to bring a reader site, initially holding $d_1$, up-to-date with a writer site holding both $d_1$ and $d_n$. Figures 2 to 4 present such measures for each sequential workload and solution.

For most solutions, we are able to distinguish between file content data and meta-data such as directory entries, version vectors or chunk references. In the case of diff and xdelta, we were not able to make such a distinction, and hence we do not differentiate their transferred volume. Furthermore, in the case of TAPER, the meta-data only considers information exchanged by the deduplication protocol, lacking the overhead of directory meta-data of TAPER.

In what follows, we follow a bottom-up analysis of dedupFS. We start by focusing on dedupFS only, comparing the different its evaluated instances. We then turn our attention to compare-by-hash and delta-encoding solutions, comparing them with dedupFS. Finally, we consider plain and gzip-compressed transference.

**dedupFS with varying expected chunk sizes, with and without data compression.** We start by focusing our attention on the different dedupFS instances, before comparing dedupFS with other solutions. The evaluated dedup-FS instances include dedupFS with different expected chunk sizes, namely 128 B, 2 KB and 8 KB, both with zlib data compression turned on and off.
As expected, the results confirm that smaller chunk sizes yield higher precision in detecting redundancy and therefore achieve lower transferred data volumes. In fact, such a decrease is substantial for all workloads except for the binary workload, `usrbin`. This suggests that redundancy across binary executables mostly occurs at coarse-grained chunks; as decreasing expected chunk size from 8K to 2K, and then to 128 B, detected only 2.4% and 2.9% more redundant data, respectively.

On average (over all workloads), dropping from 8 KB to 2 KB expected chunk size results in more 17.1% data detected as redundant, while further decreasing to 128 B expected chunk size finds more 15.0% redundant data (relative to 2 KB expected chunk size). Turning data compression on attenuates such differences to 15.1% and 11.0%, respectively. Such an effect is natural as, with finer-grained chunks, less literal data is compressed, hence less gains are achieved from data compression.

Evidently, literal data volume is not sufficiently meaningful to evaluate dedupFS, as it neglects the overhead with chunk references that dedupFS introduces. If we take such an overhead into account, the gains in transferred volume, relative to the gains in data volume, inevitably drop.

Figure 5 takes a closer look at such gains for every workload, confronting them with the number of chunk references that dedupFS transmits. From it, we can see that reducing expected chunk size from 8K to 2 KB, and then to 128 B, yields more 15.4% and 12.4% reductions in transferred volume, respectively (in contrast to 17.1% and 15.0% reductions in data volume, respectively).

Most importantly, reducing chunk size is always beneficial in terms of transferred volume. The reference coalescing step of dedupFS is key in ensuring such an important condition. An interesting observation is that, when data compression comes into play, reference coalescing may no longer ensure the previous condition. The student-projects workload exposes such an evidence: when going from 8K- to 2 KB expected chunk size, with data compression active, the latter actually transfers more 9% volume than the larger chunk alternative.

The reason behind this pathological effect is the difference in compressibility between chunk references and the chunk contents that the former replace. Since reference coalescing does not take data compressibility into account, it ignores the fact that dedupFS does not zlib-compress chunk references, while it does compress chunk contents.\(^7\) Hence, opting for a smaller volume of chunk references instead of a larger volume of chunk contents may not always be the best choice when we enable data compression.

Although not as evident as in the student-projects workload, the previous effect is present in all the other workloads. With data compression on, when we drop from 8 KB expected chunk size to 2 KB, and from 2 KB to 128 B, the gains in transferred volume become 10.2% (in contrast to 15.4% without data compression) and 3.0% (in contrast to 12.4% without data compression), respectively.

\(^7\) Even if dedupFS did compress chunk references, the attained compression gains would be small, as chunk references are poorly compressible.
Comparison with compare-by-hash solutions. We now proceed to compare dedupFS with the remaining solutions, starting at those that adopt the compare-by-hash approach. Let us recall Figures 2 to 4. Unless where noted, we use dedupFS with 128 B expected chunk size, which we abbreviate as dedupFS-128, as the reference in what follows.

With few exceptions, which we address next, dedupFS attains significantly higher reductions in transferred volume than any compare-by-hash solution, for any workload. Essentially, the key reason for such an advantage is the fact that dedupFS matches similar chunks by running a highly accurate local algorithm on local file contents. Consequently, dedupFS is able to be highly accurate in redundancy detection while keeping meta-data volume exchanged during synchronization to low levels. This contrasts with the considered compare-by-hash solutions, which employ a distributed redundancy detection algorithm. Hence, they cannot avoid the trade-off between higher accuracy and more overhead with transferred deduplication they transfer across the network meta-data (namely, hash values and chunk references).

Needless to say, the present evaluation leaves out the unmeasurable advantage of dedupFS of not being prone to data corruption due to hash collisions. By definition, all compare-by-hash solutions we consider are vulnerable to such a possibility.

We take rsync as a first example. On average over all workloads, rsync transfers 35% more (data and meta-data) volume than dedupFS-128 when we disable data compression. Turning data compression on worsens rsync efficiency, as it transfers, on average, 48% more volume than dedupFS-128. Such a value suggests that rsync’s current implementation applies zlib in a less effective manner (e.g. compressing individual chunk, instead of chunk streams).

Three main factors contribute to the difference in volume efficiency between rsync and dedupFS-128. First, the increased deduplication overhead of rsync’s distributed redundancy detection algorithm. Second, rsync’s algorithm works with larger chunks than dedupFS-128; otherwise, previously mentioned overhead would grow to unacceptable levels. Third, rsync’s algorithm works exclusively on between pairs of files, matched by pathname. Hence, rsync is only able to detect redundancy across files with a similar pathname.

The third factor is probably the crucial factor for the differences in rsync’s efficiency with each workload. On the worst extreme, rsync drops to 66% and 67% more volume than dedupFS-128 with the course-docs and student-projects workloads (without data compression). This is easily explained by the frequent file renaming activity in these workloads.

On the other hand, in workloads where pathnames are practically stable (gcc, emacs, linux), rsync is able to reach closer to the dedupFS-128’s levels of transferred volume. In particular, we found one exception, linux with compression enabled, where rsync actually transfers -17% volume than dedupFS-128 with compression enabled. The previous results of Jain et al. [10] seem

---

8Unfortunately, rsync does not output the size of such an overhead and, therefore, we are not able to quantify it.
to explain such an exception. Jain et al. show evidence that, for cases of small changes randomly distributed in a file that does not change its pathname, rsync’s similarity detection algorithm is more efficient than a variable-size chunk algorithm (on which dedupFS is based).

Note, however, that dedupFS may also employ rsync’s algorithm for local similarity detection, or even combine it with dedupFS’s current algorithm. More generally, dedupFS may adopt any distributed redundancy detection algorithm, employing it locally across the local contents at each site.

Another interesting analysis is to compare dedupFS with the highly optimized, 4-tiered distributed deduplication scheme of TAPER. Such a comparison is possible with the gcc, emacs and linux workloads, for which TAPER results are publicly available [10]. Both solutions transfer comparable data volumes: on average over the three workloads, TAPER is only 0.29% better than dedupFS-128. However, distinguishing transferred volume between content and deduplication meta-data volume allows more meaningful conclusions.

On one hand, both solutions exhibit very similar accuracy in redundancy detection with some workloads. On average over the three workloads, TAPER detects more 0.14% more redundant data than dedupFS-128 (with a maximum of more 4.0% redundant data detected with gcc, and a minimum of less 2.9% with emacs).

On the other hand, TAPER’s more intricate distributed algorithm generally requires transmitting more bytes of deduplication meta-data (more 21.3% than dedupFS-128, on average over the three workloads). Figure 6 illustrates such an advantage of dedupFS. It presents the overheads with deduplication meta-data for dedupFS and TAPER, as well for other solutions. For gcc and emacs, TAPER imposes substantially more deduplication overhead than dedupFS-128 (21.2% and 73.9%, respectively), in contrast to linux, where TAPER’s scheme has 31.0% less overhead.

More importantly than the differences in deduplication overhead, TAPER’s more intricate distributed algorithm generally requires transmitting more bytes of deduplication meta-data (more 21.3% than dedupFS-128, on average over the three workloads). Figure 6 illustrates such an advantage of dedupFS. It presents the overheads with deduplication meta-data for dedupFS and TAPER, as well for other solutions. For gcc and emacs, TAPER imposes substantially more deduplication overhead than dedupFS-128 (21.2% and 73.9%, respectively), in contrast to linux, where TAPER’s scheme has 31.0% less overhead.

More important than the differences in deduplication overhead, TAPER’s more intricate distributed algorithm generally requires transmitting more bytes of deduplication meta-data (more 21.3% than dedupFS-128, on average over the three workloads). Figure 6 illustrates such an advantage of dedupFS. It presents the overheads with deduplication meta-data for dedupFS and TAPER, as well for other solutions. For gcc and emacs, TAPER imposes substantially more deduplication overhead than dedupFS-128 (21.2% and 73.9%, respectively), in contrast to linux, where TAPER’s scheme has 31.0% less overhead.

Finally, we compare dedupFS with our implementation of LBFS’s scheme, dedup-lbfs. dedupFS and dedup-lbfs employ practically similar algorithms for similarity detection, with some important differences. The results expose three such differences.

Firstly, dedup-lbfs, following LBFS’s original algorithm [14], does not exploit recursive redundancy, i.e. redundancy that may exist exclusively across the contents to send. dedupFS incorporates such an optimization. Its effects are significant: they are evident on the differences between the data volume that each solution considers as redundant. On average over all workloads, such an optimization detects 10.8%, 14.4% and 20.8% more redundant chunk content for expected chunk sizes of 8K, 2K and 128 B.

We should note that the recursive redundancy optimization is not exclu-
sively applicable to dedupFS. One can directly incorporate it LBFS’s original algorithm, hence eliminating the above differences in detected chunk contents.

The second relevant difference is that dedupFS applies the similarity detection algorithm locally across the local contents at each site. In contrast, dedup-lbfs applies the algorithm in a distributed manner across the new versions at the sender site and the contents that the receiver site holds. Consequently, dedup-lbfs needs to exchange the hash values of the chunks that comprise the versions to propagate. This is not the case with dedupFS.

It is easy to show that, theoretically, the number of hash values that dedup-lbfs transmits grows linearly as the expected chunk size decreases. Hence, as one decreases the expected chunk size from 8 KB to 2 KB or 128 B, the extra accuracy comes at the expensive cost of a linear growth of overhead with hash values. This is a well known result [14, 10, 6, 2].

In practice, this cost is worsened by the fact that the small dimensions of most files in typical file system workloads [16] (including the ones we consider). Our evidence shows that small files bias the actual chunk sizes to at most 2 KB, no matter how high we set the expected chunk size. For example, the highest average chunk size that we found while dividing chunks using an expected chunk size of 8 KB was 2.5 KB (student-projects workload).

As a consequence, dedup-lbfs introduces a substantially larger overhead with deduplication meta-data that one might predict theoretically. Recalling Figure 6, such an increase is evident and grows as expected chunk size decreases. On average over all workloads (including course-docs and student-projects, not present in Figure 6), dedup-lbfs-8K exchanges 147% more meta-data than dedupFS-8K, fundamentally due to hash values. If we step down to dedup-lbfs-2K, such a difference rises to 387% more deduplication overhead than dedupFS-2K. Finally, dedup-lbfs-128 increases deduplication overhead by over an order of magnitude, relatively to dedupFS-128.

The impact of the higher overhead on the overall transferred volume (i.e. literal data plus deduplication meta-data) is relevant. This is true even if we assume that dedup-lbfs incorporates the optimization of exploiting recursive redundancy (by replacing the data volume measures of dedup-lbfs by dedupFS’s). Under such an assumption, dedup-lbfs-128, dedup-lbfs-2K and dedup-lbfs-8K would transfer 35%, 7% and 2% more bytes than dedupFS-128, dedupFS-2K and dedupFS-8K, respectively, on average over all workloads.

Thirdly, since dedup-lbfs has no mechanism as dedupFS’s reference coalescing step, it may easily transfer a larger volume than the plain approach. This is the case of the usrbin workload, where the low redundancy does not compensate for the overhead of exchanging hash values (see Figure 4). Note that, despite unobserved with our workloads, such a situation may also arise with the remaining compare-by-hash solutions we have evaluated. As noted before, such a situation cannot arise with dedupFS (except in rare cases if data compression is enabled, as explained earlier in this Section).
Comparison with delta-encoding solutions. Interestingly, the delta-encoding solutions, diff and xdelta, proved to be strong contenders of dedupFS, with regard of transferred volumes. In half the workloads, delta-encoding was significantly more efficient in exploiting redundancy than any other solution, including dedupFS.

Namely, these are the workloads with high redundancy between the same files of $d_1$ and $d_n$ (whose pathnames remain the same in both versions). More precisely, the most efficient delta-encoding solution, xdelta, would transfer 29%, 39% and 44% less volume than dedupFS-128 in gcc, emacs and linux, respectively.9

Delta-encoding solutions are specifically tailored to detect similarities between pairs of versions of the same file, which explains such an advantage. They employ special purpose similarity detection algorithms that are limited to running locally, and between two versions only. Furthermore, the delta encoding format [9] is lighter than the chunk references that dedupFS (and compare-by-hash) exchanges. For instance, the former takes advantage of the fact that the referenced version identifier is already implicit; hence, it needs not be explicitly encoded.

When file renaming and copy become frequent, or redundancy across multiple versions/files becomes significant, delta encoding looses its efficiency to dedupFS. Namely, for the remaining three workloads, course-docs, student-projects and usrbins, xdelta transfers 68%, 66% and 33% more volume than dedupFS-128, respectively. Moreover, in such workloads, xdelta performed worse than any compare-by-hash solution.

We may regard delta encoding as a particular instance of the versioning-based deduplication approach that dedupFS follows. The difference being that delta encoding is a more constrained solution, which may only exploit redundancy between pairs of consecutive versions of the same object (i.e., files in this context). If most redundancy occurs with such a pattern, delta encoding can be very efficient, as some of our results suggest. However, dedupFS’s ability to exploit multiple cross-version and cross-object redundancy relations allows dedupFS to adapt well to general patterns of file system usage.

As a final remark, the previous conclusions suggest that a possible direction for future work would be to leverage dedupFS with xdelta’s algorithm and encoding format in situations where redundancy occurs between consecutive versions of the same file.

---

9Such an advantage drops to 15%, 27% and 44% when we enable data compression in both solutions. However, the most recent version 3 of xdelta claims to have a more efficient, self-contained data compression algorithm, specifically designed for the VCDIFF encoding format [12].
Figure 2: Transferred data volumes for collaborative document editing workloads.
Figure 3: Transferred data volumes for software development workloads.
Figure 4: Transferred data volumes for the executable binaries workload.
Figure 5: Transferred volumes vs. chunk size in dedupFS
Figure 6: Transferred volume overheads due to deduplication (i.e. transferred meta-data such as hash values and chunk references).
2.3 Synchronization Performance

Measuring and comparing transferred volumes gives us a valuable estimate of the performance of each solution. However, low data and meta-data volumes does not necessarily mean good performance. Factors such as number of network round-trips and local processing time may also strongly affect its overall performance.

As a second experiment, we set out to measure the time that each solution takes to complete synchronization. This experiment does not include some solutions, namely: TAPER, for which a running executable was not publicly available; diff and xdelta, which do not support distributed synchronization in their current implementation.

The experiment used a 100Mbps local ethernet connection between the sender and receiver sites. Such a bandwidth is one or two orders of magnitude higher than the typical bandwidth in the environments that dedupFS mainly targets, namely the Internet and mobile networks. From the viewpoint of dedupFS, we may regard such a setting as a near-worst-case scenario. It conceals the positive impact of the reductions in transferred volume (which we have already evaluated in the previous section), while amplifying the performance overhead of the additional processing.

Figures 7 to 9 present the performance results for every sequential workload. For all solutions except rsync, we were able to differentiate two components of the total synchronization time: a pre-download time, which includes all protocol steps before the actual download of the contents of the new versions starts; and the remaining, download time.

On average over all workloads, dedupFS-2K with no compression exhibits the best performance among all dedupFS variants. However, turning data compression on does compensate on some highly compressible workloads: dedupFS-8K+zlib takes 40.2% less time to synchronize than dedupFS-2K with no compression in the student-projects workload, while dedupFS-8K+zlib and dedupFS-128+zlib achieve marginal performance gains in the gcc and linux workloads, respectively.

In spite of the high bandwidth, the download times of dedupFS dominate pre-download times (download time takes more than 80% of the total synchronization time) for the majority of workloads.

Some pathological cases of excessive pre-download time occur, which expose a weakness of dedupFS’s current design and implementation, which we discuss as follows. dedupFS’s optimization of exploiting recursive redundancy is, in its current implementation, an inefficient step. Currently, such a step implies, for every chunk reference of every version to send, traversing a singly-linked list that holds the identifiers of the versions to send that the pre-download phase has already handled.

Hence, the performance penalty of such an optimization grows quadratically with the number of versions/objects to send at a single synchronization session. Notwithstanding the significant reductions in transferred data volume of such an optimization (see Section 2.2.1), such a penalty has a strong impact
in the workloads where \( d_n \) comprises a large number of files. This is the case of the gcc and linux workloads (for any variant of dedupFS) and the student-projects workload with dedupFS-128, where dedupFS exhibits excessive pre-download times that range from 38% to 89% of the total synchronization time.

Disabling the detection of recursive redundancy considerably reduces the pre-download times in the above pathological cases. For instance, the pre-download time of dedupFS-128 (with no compression) drops from 7.476 ms to 441 ms. As an inevitable consequence, transferred data volume rises from 17.2 MB to 22.9 MB. Such an increase, however, has no significant impact on the overall performance with 100Mbps. It may, of course, become relevant if one considers environments of lower bandwidths.

Eliminating the performance penalty of recursive redundancy detection is, therefore, a key goal of future work. A potential solution must consider more efficient membership inclusion tests, rather than the current list traversal test.

When compared to the other solutions, dedupFS achieves better performance, with a small number of exceptions in some of the pathological workloads that we mention above. The advantage over rsync is substantial. On average over all workloads, rsync almost doubles dedupFS-2K’s total time (91.7% and 105.8% slower with zlib compression off and on, respectively). Even in the worst workloads for dedupFS, rsync is at least 40.8% slower than dedupFS-2K. It is worth noting that rsync never accomplishes better performance than the plain transfer alternative.

Concerning dedup-lbfs, its performance degrades considerably as we decrease the expected chunk size. With an expected chunk size of 128 B, dedup-lbfs reflects the strong penalty of a large network overhead with hash values and chunk references and, most importantly, the delay of numerous lookups to the chunk hash table. Its synchronization time is, on average over all workloads, 17 times slower than plain transfer and 26 times slower than dedupFS-2K.

Its performance rises substantially as we increase expected chunk size to 8 KB. In that configuration, dedup-lbfs is, on average over all workloads, 29.5% slower than dedupFS-2K. More notably, dedup-lbfs-8K outperforms dedupFS in the pathological workloads that we mention above: dedup-lbfs-8K is 11.6% and 11.7% faster than dedupFS-2K in gcc and linux, respectively. Nevertheless, if we disable recursive redundancy detection, dedupFS-2K becomes faster than any deduplication solution for every workload.

As a final remark, we compare dedupFS with the plain and compress-only alternatives. In general, dedupFS substantially outperforms the previous solutions, in almost all workloads with relatively high redundancy (on average, 25.6% and 27.8% faster than plain and compress-only, respectively). The exception is the gcc workload, where the performance penalty of the pathological cases we describe previously is sufficient to make dedupFS a worst alternative than the plain and compress-only alternatives (16.8% and 17.8% slower, respectively).\(^{10}\) Again, disabling recursive redundancy detection eliminates such

\(^{10}\)Another exception is dedupFS-128 and dedupFS-128+zlib in the student-projects work-
Figure 7: Synchronization times for collaborative document editing workloads.

performance losses.

When the redundancy degree is low, which is the case of the `usrbin` workload, dedupFS (with no data compression) is still able to attain better performance than the plain and compress-only solutions. However, the low compressibility of this workload (see Table 3) causes dedupFS+`zlib` to be slower than plain transfer, due to the additional processing time the former spends with `zlib` functions.

load. In this workload, however, all other variants of dedupFS outperform both plain and compress-only.
Figure 8: Synchronization times for software development workloads.
Figure 9: Synchronization times for the executable binaries workload.
2.4 Asserting Our Results with More Realistic Assumptions

The advantages of dedupFS over other solutions will only be effective as long as two conditions hold: (i) out-of-band redundancy, which dedupFS cannot exploit, is not significant; and (ii) the synchronizing sites are able to maintain sufficiently old version logs.

As we discuss before, the promising results in Section 2.2 do not completely evaluate dedupFS, as they exclusively consider two-version sequential workloads where out-of-band redundancy does not arise, and both sites’ logs are assumed sufficiently deep to hold a common ancestor version \(d_1\). Hence, we can only accept the results from Section 2.2 as meaningful in situations where the two conditions above hold.

This section aims at understanding whether the scenarios that this thesis considers verify the two conditions. In the next two sub-sections, we gradually weaken the assumptions of the previous section. In Section 2.4.1, we evaluate concurrent workloads, where out-of-band redundancy finally occurs. Then, in Section 2.4.2 evaluates \(n\)-version workloads \((n > 2)\) with version logs of limited size.

2.4.1 Out-of-band Redundancy

We measured the amount of out-of-band redundancy using real-world concurrent workloads, course-docs and student-projects. Both workloads result from collaborative activity among different teams of users, as we explain in Section 2.1.2. In both workloads, two sites started with a replica of a common, initial version. During a considerable period of time – 6 months for course-docs and 3 months for student-projects –, both replicas evolved independently with frequent concurrent updates (typically, more than one update per day).

The experiment assumes that, during such periods, the underlying file system can only trace the local evolution from the initial version to the final version of each replica. As Section 2.1.2 details, considerable information was potentially exchanged between both replicas via channels that are untraceable by the underlying file system (e.g., dedupFS). Inevitably, such out-of-band exchanges may lead to out-of-band redundancy, which dedupFS cannot detect.

We measured the amount of out-of-band redundancy that each workload collected at the end of its lifetime with a fairly simple experiment. Recalling Figure 1 from Section 2.1, each replica holds, after such a period, version \(d_a\) and \(d_b\), respectively (besides \(d_1\)). For each workload, we synchronized both divergent replicas, the receiver replica holding versions \(d_1\) and \(d_a\), and the sender replica holding versions \(d_1\) and \(d_b\). Firstly, we synchronized using dedupFS-128\(^{11}\), where the sender replica could only exploit in-band redundancy; i.e., the chunks that are redundant across \(d_1\) and \(d_b\). We then repeated the same experiment using

\(^{11}\)We disabled recursive redundancy detection. Without such an optimization, dedupFS attains exactly the same efficiency in detecting in-band redundancy as dedup-lbfs, for the same expected chunk sizes.
dedup-lbfs-128. In this case, the distributed redundancy detection protocol can
detect both in-band and out-of-band redundancy; i.e. the sender can exploit
chunk redundancy across any workload versions, $d_1$, $d_a$, and $d_b$. Therefore,
by subtracting the redundant data volumes\(^{12}\) of dedupFS from dedup-lbfs, we
obtain an estimate of out-of-band redundancy in the workload.\(^{13}\)

Figure 10 shows the results for both concurrent workloads, considering different
instances of dedupFS and dedup-lbfs, as well as the plain transfer solution.
For simplicity, and without lack of generality, we disabled data compression in
all solutions.

Perhaps surprisingly, the redundant data volume that results from out-of-
band data exchanges during several months is almost insignificant when com-
pared to in-band redundancy. In the course-docs workload, dedup-lbfs-128
detects just 1.01% more (out-of-band) redundant contents than dedupFS. The
student-projects workload exhibits more out-of-band redundancy, but still at
insignificant levels: dedup-lbfs-128 detects 4.37% more redundant data.

It is certainly easy to identify scenarios where out-of-band redundancy is
substantial. Nevertheless, the results from the above two real concurrent work-
loads are very meaningful. They capture collaborative scenarios that are very
representative of the ones that the present thesis mainly addresses. Further-
more, in both workloads there are numerous evident sources of out-of-band
redundancy, which Section 2.1.2 enumerates; in fact, our results show that out-
of-band redundancy does occur. However, our results confirm that almost all
redundancy that we can exploit when synchronizing the new contents at each
divergent replica results from contents copied from the initial version, $d_1$; in
other words, in-band redundancy.

The above small advantage in detected redundant data volume of dedup-lbfs
over dedupFS does not eliminate the overall advantage of the latter over the
former; either in terms of transferred volume or synchronization performance.
In order to synchronize the concurrent replicas of either course-docs or stu-
dent-projects, every dedup-lbfs variant transfers more bytes and takes more
time than any dedupFS variant. More concretely, the best variant of dedup-lbfs
(dedup-lbfs-8K) transfers 2.9% more volume the most network-efficient dedup-
FS variant, dedupFS-128, and takes 29.8% more time than the fastest dedup-
FS variant, dedupFS-2K. With student-projects, in spite of the higher out-
of-band redundancy, dedup-lbfs-8K still transfers 1.1% more than dedupFS-128,
and is 8.6% slower than dedupFS-2K.

2.4.2 Space Requirements of Version Log

A second condition to dedupFS’s effectiveness is that the version logs at each
synchronizing site span for a sufficiently large time period. Since dedupFS can
only detect redundancy across the versions to send and the intersection of the

\(^{12}\)I.e., without accounting for exchanged meta-data.
\(^{13}\)Of course, the previous values are estimates because neither dedup-lbfs’s nor dedupFS’s
redundancy detection algorithms are optimal. Nevertheless, using small chunk sizes (namely,
128 expected chunk size), we obtain an accurate estimate.
version logs of both sites, the larger such an intersection is, the more redundancy dedupFS may exploit. In the worst case, where both logs do not intersect at all, dedupFS can no longer exploit any redundancy that may exist across both sites.

Figure 11 illustrates such a condition with an example of two sites, each holding a divergent version of concurrent workload. As long as both sites are able to maintain sufficiently long logs (in this case, as long as both logs include the old version \(d_1\)), dedupFS will be able to exploit redundant chunks \(c_1\) and \(c_2\). However, excessive log truncation can strongly hamper dedupFS’s ability of detecting in-band redundancy across both sites, even when such redundancy exists. In the example, truncating the oldest version in the log of at one of the sites will result in an empty intersection between both sites’ logs, hence disabling dedupFS from detecting any in-band redundancy across both sites.

Evidently, it is key that, as a site’s log incorporates newer versions, its space overhead remains at acceptable levels, so as to delay version truncation to as late as possible. This section studies such space requirements. We analyze the two collaborative workloads, course-docs and student-projects. However, we do not consider the same two-version workloads as in the previous sections. Instead, we have obtained daily CVS snapshots of each such workload\(^\text{14}\).

We consider sequences of multiple workload versions, \(d_1, d_2, \ldots, d_{n-1}, d_n\). Each sequence spans across the whole duration of each workload: in course-docs, \(n = 168\) (days), while \(n = 74\) (days) in student-projects. Each workload version \(d_i\) exclusively includes the files whose contents have been modified relatively to the previous day’s version, \(d_{i-1}\).

In this experiment, we place ourselves at the last day of the workload, starting with a single version log (holding \(d_n\) only). We then add \(d_{n-1}, d_{n-2}, \ldots\) to the log and measure how much additional space each deeper log requires. We consider different schemes for log storage: plain storage, redundancy compression and partial pruning, with different expected chunk sizes.

Figures 12 and 13 present the results of such an experiment. A first observation is that, even with plain log storage, space cost grows moderately with log depth. With course-docs, the space overhead of maintaining the entire 6-month history of daily collaborative activity implies storing less than the size of the current version. More precisely, maintaining the most recent version requires 102 MB, while storing the entire history of 168 versions requires 98% more space. With student-projects, maintaining the entire 3-month version history requires 49% more space than the 37 MB of the most recent version. With most devices, for which secondary memory is cheap and abundant, such overheads are usually acceptable.

Furthermore, the more efficient log storage schemes are still able to substantially reduce such a space overhead. Redundancy compression allows logging the entire history of versions with a 25%-31% space overhead in course-docs, depending on the expected chunk size (128 B and 8 KB produce the lowest and\(^\text{14}\)course-docs and student-projects are concurrent workloads. In this section, we consider one of the divergent branches only, for each workload. More precisely, we consider daily versions from \(d_1\) to \(d_a\). The results for the \(d_b\) branch yield comparable conclusions.
highest overhead, respectively). In student-projects, such an overhead drops to 11.1%-12.6%. Such low overheads are clearly important when we consider memory-constrained devices, such as most mobile ones. Achieving such low overheads per day of logging means that even memory-constrained devices will be able to maintain knowledge of significantly long version histories.

Finally, partial pruning offers almost negligible overhead. However, exploiting the redundancy footprints at the receiver side is not necessarily advantageous for synchronization performance (see main paper). Hence, we restrict our analysis the case of sites that always act as senders. With this scheme, a sender-only site may store the redundancy footprints of the entire histories of course-docs and student-projects at the cost of 1.94%-0.08% and 2.06%-0.09%, respectively; as explained in the main paper, such footprints hold sufficient information to completely replace the whole versions when the site acts as sender during synchronization. It is worth noting that, in contrast to redundancy compression, partial pruning is less efficient with smaller expected chunk sizes, as more chunk references exist.

2.5 Local Similarity Detection

In contrast to a synchronization session, the local similarity detection algorithm runs in background. Hence, its performance is neither the central concern of dedupFS’s current implementation nor of the present evaluation. Nevertheless, despite not being a critical process, it is still desirable that local similarity detection consumes as less resources as possible (e.g. battery and CPU) during the shortest time possible. Therefore, this section devotes some attention on the performance of the local similarity detection algorithm of dedupFS.

During the experiments in Section 2.2 we gathered some measures concerning local similarity detection. We obtained such measures both for the local similarity detection algorithms dedupFS and dedup-lbfs. Recall that dedupFS performs a byte-by-byte comparison between any pair of chunks that it finds with similar hash values. On one hand, this step (which dedup-lbfs discards) imposes a strong performance penalty on dedupFS, as we shall analyze next. On the other hand, the reliance on byte-by-byte comparison makes it possible for dedupFS to use a faster, non-cryptographic hash function. Consequently, dedupFS is faster in determining hash values, but may have to byte-by-byte-compare more chunks, due to more frequent hash collisions. If comparing one byte per redundant byte found has a significant performance overhead, hash collisions may amplify it by causing multiple unnecessary byte-by-byte comparisons when false positives occur.

One first relevant measure is, for each byte that dedupFS found redundant, the number of bytes that, on average, dedupFS compares against such a byte. From Table 4, we can see that such a measure varies considerably across workloads. While emacs and linux remain close to the optimal one byte compared 15In the case of dedup-lbfs, we refer to the local similarity detection that dedup-lbfs performs for exploiting chunk redundancy in locally stored contents. Obviously, we are not considering the distributed similarity detection algorithm, used for synchronization.
Table 4: Number of byte-by-byte comparisons per each byte found redundant by dedupFS, for different expected chunk sizes.

<table>
<thead>
<tr>
<th>Workload</th>
<th>128 B</th>
<th>2 KB</th>
<th>8 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>course-docs</td>
<td>2.5</td>
<td>1.1</td>
<td>1.0</td>
</tr>
<tr>
<td>student-projects</td>
<td>48.0</td>
<td>21.6</td>
<td>13.3</td>
</tr>
<tr>
<td>gcc</td>
<td>1.2</td>
<td>1.8</td>
<td>1.5</td>
</tr>
<tr>
<td>emacs</td>
<td>1.9</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>linux</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>usrbin</td>
<td>2.7</td>
<td>1.1</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 5: Local similarity detection performance of dedupFS, both absolute and relatively to dedup-lbfs, for different expected chunk sizes.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Running Time (s)</th>
<th>Rel. Performance (vs. dedup-lbfs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128 B</td>
<td>2 KB</td>
</tr>
<tr>
<td>course-docs</td>
<td>201.9</td>
<td>30.5</td>
</tr>
<tr>
<td>student-projects</td>
<td>5,384.1</td>
<td>443.7</td>
</tr>
<tr>
<td>gcc</td>
<td>340.0</td>
<td>157.4</td>
</tr>
<tr>
<td>emacs</td>
<td>57.0</td>
<td>15.6</td>
</tr>
<tr>
<td>linux</td>
<td>342.8</td>
<td>118.8</td>
</tr>
<tr>
<td>usrbin</td>
<td>1,056.8</td>
<td>142.8</td>
</tr>
</tbody>
</table>

Per redundant byte, course-docs and usrbin require more than 2.5 comparisons per byte. Surprisingly, student-projects exhibits a highly pathological case of frequent hash collisions, reaching 13 to 48 byte comparisons per byte, depending on whether expected chunk size is 8 KB or 128 B, respectively.

Inevitably, the byte-by-byte comparison and the impact of hash collisions has a considerable performance penalty on dedupFS’s local similarity detection step. To quantify such a penalty, we compare dedupFS’s local similarity detection time with dedup-lbfs’s, as we show in Table 5. Clearly, dedupFS takes substantially more time to detect local redundancy than dedup-lbfs. Such a performance gap increases as we decrease chunk sizes. On average over all workloads (excluding student-projects’s pathological case), dedupFS takes 48% more time with an expected chunk size of 8 KB; decreasing such a parameter to 2 KB and 128 B causes dedupFS to become 2.6 times and 6.1 times slower than dedup-lbfs, respectively. We mainly attribute such a performance decrease to the higher number of chunks, which are directly proportional to the number of disk accesses and chunk hash table lookups during the algorithm execution.

Table 6 presents the overall throughput values of dedupFS’s local similarity detection step for each workload. Despite the unfavorable relative performance, the throughput of dedupFS’s local similarity detection achieves ranges from 746 KB (with 128 B expected chunk size) to 6,600 KB (with 8 KB expected chunk size) of new content handled per second, on average over all workloads (including the problematic student-projects).

In typical interactive collaborative applications, we expect such a throughput
Table 6: Throughput of local similarity detection in dedupFS. Throughput is given by the volume of new content that the similarity detection step can handle per second. Throughput is given in KB per second, for different expected chunk sizes.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Throughput (KBps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128 B</td>
</tr>
<tr>
<td>course-docs</td>
<td>555</td>
</tr>
<tr>
<td>student-projects</td>
<td>10</td>
</tr>
<tr>
<td>gcc</td>
<td>713</td>
</tr>
<tr>
<td>emacs</td>
<td>1,708</td>
</tr>
<tr>
<td>linux</td>
<td>928</td>
</tr>
<tr>
<td>usrbin</td>
<td>507</td>
</tr>
</tbody>
</table>

range to be higher than the typical rates at which users input new contents and synchronize. Hence, we consider the measured throughput acceptable, though subject to improvement. The use of fast, yet less collision-prone hash functions, and the use of massively available stream processor hardware such as graphics processing units (GPUs) to boost chunk local similarity detection (as proposed in [1]) are certainly interesting directions for future improvement.
Figure 10: Differences in detected redundant data volume in the concurrent workloads (course-docs and student-projects), between a solution that can only detect in-band redundancy (dedupFS) and a solution that can detect both in-band and out-of-band redundancy (dedup-lbfs).
When synchronizing A → B:

\[
\begin{align*}
\text{If } & \log_A = (d_{a,k}, \ldots, d_{a,1}) \quad \text{and} \quad \log_B = (d_{b,m}, \ldots, d_{b,1}) \quad \text{then dedupFS will neither send } c_1 \text{ nor } c_2. \\
\text{Instead, dedupFS sends chunk references to } c_1 \text{ and } c_2 \text{ in } d_1.
\end{align*}
\]

\[
\begin{align*}
\text{If } & \log_A = (d_{a,k}, \ldots, d_{a,1}) \text{ or smaller } \quad \text{or} \quad \log_B = (d_{b,m}, \ldots, d_{b,1}) \text{ or smaller} \quad \text{then dedupFS needs to send both } c_1 \text{ and } c_2.
\end{align*}
\]

Figure 11: Example of impact of log size in redundancy detection in dedupFS. Two sites, A and B, hold divergent replicas of a concurrent workload (for simplicity, the figure depicts each workload version as a single file). Both divergent replicas originate from a common, initial version, \(d_1\). When B synchronizes with A (i.e. synchronization from A to B), dedupFS’s ability to exploit redundancy depends on the depth of the logs that each site holds in that moment. Firstly, if B’s log no longer holds \(d_1\) (due to log truncation), then chunk \(c_1\) is not redundant across A and B, thus A must transfer its contents. Secondly, such a situation (B’s log without \(d_1\)) also means that dedupFS will not be able to detect chunk \(c_2\) as redundant across both sites, since both logs do not intersect; although \(c_2\) is, in fact, redundant across both sites.

Figure 12: Space requirements of version log for multi-version course-docs workload.
Figure 13: Space requirements of version log for multi-version 
student-projects workload.
Conclusions

This complementary document evaluates dedupFS, a prototypal implementation of our data deduplication solution for efficient replica storage and synchronization. Using dedupFS and other state-of-the-art solutions from each relevant approach to data deduplication, we have replayed real workloads covering a broad set of collaborative activities.

The obtained results complement previous evidence [8] that, to some extent, contradicts a general conviction that is subjacent to most literature proposing data deduplication through compare-by-hash [14, 4, 10, 6, 2]. Our results show that, for the workloads that the latter works consider and evaluate, there does exist a non-probabilistic solution that performs at least as well as compare-by-hash. Even if applications are willing to tolerate the non-null possibility of data corruption due to hash collisions.

In a first experiment, we consider similar workloads as those that motivate and serve to evaluate recent related work on data deduplication. Such workloads are sequential, hence do not allow out-of-band redundancy. Our results show that, with very few exceptions, dedupFS detects more redundant contents than relevant compare-by-hash solutions (rsync, TAPER and LBFS), while keeping meta-data overhead at low levels. In result, on average over all workloads, the volume that dedupFS transfers across the network during synchronization is substantially lower than any compare-by-hash solution that we consider. Accordingly, such an advantage is also reflected on synchronization performance.

With respect to synchronization schemes that rely of delta encoding, our results show that, for general workloads, dedupFS is significantly more efficient. However, for workloads with a sufficiently stable directory/file tree and where in-object redundancy dominates cross-object redundancy, delta encoding may substantially outperform dedupFS. Hence, an interesting direction for future improvement of dedupFS is to complement dedupFS’s similarity detection algorithm and reference encoding with a variant based on delta encoding. Such a hybrid approach is clearly compatible with dedupFS.

Finally, we assert the above promising results under more realistic assumptions. We do that by presenting strong experimental evidence that: (i) out-of-band redundancy does occur rarely in computer-supported collaborative scenarios; and (ii) the space requirements of maintaining sufficiently long version logs are, in general, acceptable even in the environments that the present thesis addresses. We support the first statement with an analysis of two concurrent workloads, taken from real users that actively made use of shared CVS repositories. In both workloads, after 3-month and 6-month periods, the amount of out-of-band redundancy accounts to less than 5% of the total redundancy that one may find in the new contents that each workload has received. Furthermore, we back up the second claim by showing that, in the previous workloads, dedupFS can maintain the entire version history of 3 and 6 months, respectively, at the acceptable cost of approximately 4% space overhead per logged month of update activity.
References


