Duplicacy: A New Generation of Cloud Backup Tool Based on Lock-Free Deduplication

Zonghui Li, Gilbert (Gang) Chen*, Yangdong Deng, *Member, IEEE*

Abstract—The pervasive deployment of cloud services poses an ever-increasing demand for cross-client deduplication solutions to save network bandwidth, lower storage costs, and improve backup speeds. However, existing solutions typically depend on lock based approaches relying on a centralized chunk database, which tends to hinder performance scalability. In this work, we present a new cross-client cloud backup solution, named Duplicacy, based on a Lock-Free Deduplication approach. Lock-Free Deduplication stores chunks to network or cloud storage using content hashes as file names. It then adopts a two-step fossil deletion algorithm to solve the hard problem of deleting unreferenced chunks in the presence of concurrent backups, without the need for any locks. Experiments demonstrate that Duplicacy enables significant performance improvement for backups over previous well-known backup tools. In addition, Duplicacy can work with many general-purpose network or cloud storage services which only support a basic set of file operations, and turn them into sophisticated deduplication-aware storage servers without server-side changes.

Index Terms—backup tool, lock-free deduplication, cloud storage, concurrent backup, cross-client deduplication

1 Introduction

As the cost of cloud storage continues to drop and the internet bandwidth keeps growing, more and more backups will be saved to the cloud [1]. According to the Cisco Global Cloud Index (2016-2021) [2], the data stored in cloud will grow 4.6-fold and reach 1.3 ZB by 2021. With the unprecedented growth of cloud storage, the large-scale deployment has been emerging as the tendency of a backup solution. Hundreds of computers may share similar sets of files, such as OS system files or a large code base. As a result, cross-client deduplication is extremely important as it saves network bandwidth and storage costs, and improves backup speeds.

Deduplication [3], [4] is the method to save multiple identical copies of the same data as one copy. There are actually two levels of deduplication. File-level deduplication means two identical files must be stored as one copy, even if they come from different directories or backup clients. Chunk-level deduplication means same parts from different files, or different versions of the same file must be stored as one copy. Chunk-level deduplication is more desirable as it occurs more frequently in practice and often implies file-level deduplication. Cross-client deduplication refers to the fact that deduplication can be exploited by multiple backup processes running concurrently on different computers.

Once it is clear how files can be split into chunks, we need to decide how to store chunks in the storage server. It is a common practice for previous deduplication-enabled backup tools [5], [6], [7], [8] to combine multiple chunks into a packed file which is then uploaded. The consideration here is to reduce the overhead of uploading files. A chunk database is then needed to map chunks to packed files in order to quickly locate a chunk. Such a centralized database leads to four performance-loss problems as follows.

- If there are multiple backup clients, the chunk database must be shared and synchronized by all clients, otherwise the chunk-level deduplication will be reduced.
- The chunk database is usually cached locally for performance improvement, but that makes it hard to delete a chunk in presence of multiple clients.
- Storage space can only be freed when all chunks belonging to a packed file are deleted.
- To download a single chunk the entire packed file must be downloaded if the storage service does not provide segmented downloads.

Moreover, the deleting procedure in deduplication further weakens the concurrent performance if it uses a locked or atomic operation to delete a chunk whose reference count becomes zero [9].

To solve these problems of the lock-based solutions above, we propose a new deduplication solution called Lock-Free Deduplication. First, the proposed deduplication takes a much simpler approach to manage chunks without a centralized database. That is, each chunk is uploaded to storage server individually, and stored in a file using the hash of the chunk content as the file name. Such a straightforward naming scheme leads to an important property: duplicate chunks can now be checked by a simple file-name lookup. With this property, many cloud storage services can now be made deduplication-aware by the Lock-Free Deduplication, as long as they provide an API method that checks whether a file with the given name exists or not. The absence of the chunk database significantly reduces the complexity, rendering the implementation less error-prone.

- *Z. Li is with the Beijing Key Laboratory of Transportation Data Analysis and Mining, School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China, 100044. E-mail: zonghui.lee@gmail.com.*
- *Gilbert (Gang) Chen is with the Acrosync LLC, 4 Annabel Pl, Clifton Park, NY 12065, USA. Gilbert (Gang) Chen* is the corresponding author. E-mail: gchen@acrosync.com.*
- *Y. Deng is with the School of Software, Tsinghua University, Beijing, China, 100084. E-mail: dengyd@tsinghua.edu.cn.*

Second, the proposed deduplication then employs a two-step fossil deletion to delete unreferenced chunks in the presence of concurrent backups, without using any locks. Instead of deleting unreferenced chunks immediately, the first step, called *fossil collection*, identifies and renames those chunks not referenced by all backups. A chunk that has been renamed is called a *fossil*. The second step, namely the *fossil deletion* step, conditionally deletes those fossils. If a new reference to a fossil is detected, this step will turn the fossil back into a normal chunk. Combining with the defined operation rules, we prove that the two-step fossil deletion is lock-free. Finally, we present the implementation of the proposed lock-free deduplication in a new cloud backup tool named Duplicacy [10]. Duplicacy has widely been combined with many commercial cloud storage services including Amazon S3, Google Drive, Microsoft OneDrive, etc., to enable deduplication-aware cloud storage. It is already open source in the github repository [10]. Furthermore, compared with previous well-known backup tools, namely Duplicity [5], Restic [6], and Attic [7], [8], Duplicacy achieves significant improvements for backup performance.

Our main contributions are threefold as below.

- First, we propose a new cross-client deduplication solution called Lock-Free Deduplication that eliminates the previous use of lock based solutions as well as a centralized database, and prove that the proposed techniques are lock-free.
- Second, we implement Lock-Free Deduplication in a new cloud backup tool named Duplicacy that can work with many general-purpose network or cloud storage services to enable deduplication-aware cloud storage, and open source for the new backup tool.
- Finally, we evaluate the backup performance for Duplicacy compared with the previous well-known backup tools, which demonstrates that Duplicacy achieves significant performance improvement in various datasets.

The rest of the paper is organized as follows. Section 2 describes the background and related terminology. Section 3 presents the technical details of Lock-Free Deduplication and gives the corresponding proof. Following these proposed techniques, a new cloud backup tool, Duplicacy is implemented in Section 4. Section 5 reports the evaluation for the backup performance of Duplicacy. Section 6 presents the related work. Finally, Section 7 concludes this paper.

2 Background

To understand the proposed lock-free deduplication better, this section presents the background for the related techniques and terminology.

2.1 Chunking

Splitting files into chunks is the first step towards chunk-level deduplication. There are two types of chunking algorithms: fixed-size chunking and variable-size chunking.

The fixed-size chunking algorithm simply splits a file into chunks of the same size. It has little overhead, with the downside being that deduplication is lost when a small portion of the file is deleted or new data is inserted. To fix this problem, Rsync [11], the popular Unix file synchronization and backup tool, makes use of the fixed-size chunking algorithm where the chunk size is determined by the file size (the square root of the file size), but adds a simple rolling hash algorithm to detect insertions and deletions, at the cost of one lookup every time the sliding window is shifted by a byte, thus incurring a significant amount of overhead.

The variable-size chunking algorithm, also called Content-Defined Chunking, has become well-known in the industry [3], [4]. The idea is to use a rolling hash, similar to that of Rsync, but only to identify breakpoints whose hash values follow a specific pattern, for example, ending with a given number of zeros [12]. These breakpoints serve as the boundaries between chunks, and the selected pattern controls the expected size of chunks. The main advantage is that a lookup to check duplicates is performed only after a breakpoint has been identified (which indicates the end of a chunk), rather than one per byte as in the case of Rsync.

2.2 Naming Chunks

The major concern with the naming-by-hash scheme is that it may create too many chunks if there are millions of small files. Lock-Free Deduplication addresses this concern by first packing together all files, small or large, in alphabetical order according to their names, as if it were creating one big tar archive, and then splitting the conceptual tar archive into chunks.

This pack-and-split approach makes the number of chunks only dependent on the total size of files, regardless of individual file sizes. Thus, a relative large chunk size can be selected (at the megabyte level), reducing the overhead with the chunk lookup and upload. As files are split into chunks, a backup is now no more than a list of chunk hashes as well as a few metadata fields. Each backup can now be represented by a so-called *manifest* file. The manifest file is to be uploaded to the storage after all chunks that compose the backup have been uploaded. To restore a backup, the manifest file is downloaded first, and all needed chunks can be identified by parsing the chunk hash list contained in the manifest file, and then be downloaded thereafter.

The backup manifest file can become too large for a big backup. Uploading large manifest files may waste storage space and increase upload time, especially for incremental backups where only a few files have been changed. To address this issue, backup manifest files are also split into chunks using the same variable-size chunking algorithm. The final manifest file being uploaded becomes a much shorter list of hashes representing all the chunks that make up the real manifest file.

It is worth noting that the method of searching chunks by their content hashes is well-established [12], [13]. However, naming chunks by their content hashes and thus eliminating the need for a chunk database, as far as we know, has not been attempted before by any backup tool. A centralized chunk database actually makes it trivial to delete chunks not referenced by any backup, with synchronous locking operations or reference counters. This perhaps explains why such a chunk database is widely used in previous deduplication systems [3], [4], [5], [6], [7], [8]. But centralized databases and locking operations prevent concurrent access and thus hinder the performance of cross-client deduplication. Our proposed techniques build a practical deduplication-enabled backup tool on top of the naming scheme without a centralized chunk database or locking operations, which is fundamentally different from approaches adopted by other deduplication tools [5], [6], [7], [8].

3 Lock-Free Deduplication

Concurrent deduplication includes three basic concurrent situations: concurrent backup, concurrent deletion, and concurrent backup and deletion. In this section, we first present the technical details for backup and deletion procedures in the proposed lock-free deduplication, and then prove such three concurrent situations are lock-free.

3.1 Naming Chunks Based Backup

The backup procedure in Lock-Free Deduplication uses the naming chunk techniques described above. It eliminates a centralized indexing database for tracking all existing chunks and for determining which chunks are not needed any more. Instead, to check if a chunk has already been uploaded before, one can just perform a file lookup via the file storage API using the file name derived from the hash of the chunk. More importantly, the absence of a centralized indexing database means that there is no need to implement a distributed locking mechanism on top of the file storage.

The lock free approach in the backup procedure has two implications. First, each chunk is saved individually in its own file named by content hash, and once saved there is no need for modification. Data corruption is therefore less likely to occur because of the immutability of chunk files. Any modification in source files will lead to new named chunks due to the changes of content hash. The backup procedure simply uploads these new chunks and original chunks will be deleted by the deletion procedure. Second, after one client creates a new chunk, other clients that happen to have the same source file will notice the existence of the chunk with a simple lookup, and therefore will not upload the same chunk again. We defined a policy below for the backup procedure.

Policy 1 (Checking before Uploading)**.** *Before uploading a new chunk, the backup procedure always performs a lookup with a filename determined only by the chunk content hash.*

Thus, after a chunk has been uploaded by one backup client, if another client happens to encounter the exact same chunk, such a policy will find the existing chunk, and avoid uploading the chunk again. By eliminating the chunk indexing database, the backup procedure can achieve the highest level of deduplication. That is, clients without actively communicating directly with each other can share identical chunks with no additional effort.

3.2 Two Step Fossil Deletion

The deletion procedure is to delete those chunks unreferenced by any backups. To implement lock-free deletion, we propose a two-step fossil deletion procedure, namely *fossil collection* step and *fossil deletion* step. The procedure is inspired by the optimistic approach to Parallel Discrete Event Simulation (PDES) [14], [15], [16]. There, events can be processed out-of-order as fast as possible, for maximum performance, but processed events are still kept in the memory. When a causality error occurs, processed events are rolled back to guarantee the correctness of simulation. A Global Virtual Time (GVT) algorithm [14], [17] is used to reclaim memory associated with processed events by the timestamps and simulation-constrained policies of these events. Our approach follows the same idea: the fossil collection step aggressively identifies unreferenced chunks based on only existing backups, ignoring backups that may be still in progress. However, unreferenced chunks are renamed rather than deleted, offering the fossil deletion step an opportunity to correct the mistake if some of them are in fact needed by new backups.

Algorithm 1: Fossil Collection

```
Step 1: Download all backup manifest files.
Manifest[] allManifests = downloadAllManifestFiles();
Step 2: Divide manifest files into two groups, one to keep
  and the other to prune, based on user-specified retention
  options;
Manifest[] keptManifests, prunedManifests =
  applyRetentionOptions(allManifests,
  retentionOptions);
Step 3: List chunks referenced by manifest files to keep
List[] referencedChunks =
  getChunkList(keptManifests);
Step 4: List chunks referenced by manifest files to prune
List[] unreferencedChunks =
  getChunkList(prunedManifests);
Step 5: Remove any chunk from the set of unreferenced
  chunks if it referenced by other backups.
for chunk in unreferencedChunks do
    if chunk in referencedChunks then
        unreferencedChunks.remove(chunk);
    end
end
Step 6: For each chunk in the set of unreferenced chunks,
  turn it into a fossil by renaming it (such as either adding a
  suffix to its file name or moving it to a different location).
Fossil[] fossils;
for chunk in unreferencedChunk do
    Fossil fossil = getFossilByRenaming(chunk);
    fossils.addFossil(fossil);
end
return fossils;
```

Step One: Fossil Collection is illustrated in Algorithm 1. It first downloads all known backup manifest files. It then lists chunks from manifest files to keep, and chunks from manifest files to prune, and then compares these two lists to identify chunks that will become unreferenced after deleting manifest files that will be removed. Instead of deleting unreferenced chunks immediately, this step performs a renaming operation on these chunks which can be rolled back later if need be. A chunk that has been renamed is called a *fossil*, and thus this step is called the *fossil collection* step. To clarify the difference between fossils and chunks, we define the access policy for fossils as follows.

Policy 2 (Fossil Accessing)**.** *The following two rules present the access policy for fossils.*

1) *A restore, list, or check procedure that reads existing backups can read the fossil if the original chunk cannot be found.*
2) *A backup procedure cannot access any fossils. That is, it must upload a chunk if it cannot find the chunk, even if a corresponding fossil exists.*

Step Two: Fossil Deletion will permanently delete fossils, which is presented in Algorithm 2. This step will not run unless the defined policy is met.

Policy 3 (Fossil Deleting)**.** *The following two conditions should be met before deleting fossils permanently.*

1) *For each backup client, there is a new backup that was not seen by the fossil collection step.*
2) *For each backup client, the new backup must have finished after the fossil collection step.*

The backup clients are concurrent backup procedures. Each computer may run one or multiple backup clients and each client can be identified by a unique id. The first condition defines a new backup as a backup not seen by the *fossil collection* step. A backup not seen by the *fossil collection* step implies that the *fossil collection* step did not use its manifest file to collect fossils. The second condition emphasizes that the finish time of a new backup must be after the end time of the *fossil collection* step. If a backup satisfies the first condition but not the second condition, the *fossil deletion* step will not be activated.

When both conditions in Policy 3 are met, the *fossil deletion* step lists all chunks referenced by these new backups. For each fossil from the *fossil collection* step, if it exists in the list, it is turned back into the original chunk by recovering its original file name. Finally, remaining fossils will be deleted permanently.

The two steps are integral parts of the algorithm and together they guarantee that no fossil needed by new backups can be deleted. The *fossil collection* step collects fossils and generates a local fossil collection file that records the list of collected fossils as well as the finish time of this step. The *fossil deletion* step deletes these recorded fossils conditionally according to Policy 3. As a result, both steps can be repeatedly invoked at regular intervals. The local fossil collection file is only visible locally, and will be deleted when the *fossil deletion* step finishes.

Algorithm 2: Fossil Deletion

```
Step 1: Download all known manifest files and then check
  whether Policy 3 is met. If so, then proceed to Step 2;
  otherwise exit and wait for next invocation.
Manifest[] allManifests = downloadAllManifestFiles();
Manifest[] previousManifests =
  getPreviousManifests();
Manifest[] newManifests = diff(allManifests,
  previousManifests);
for client in allClients do
    Boolean met = false;
    for backup in newManifests do
        if backup.id == client and backup.finishTime >
          lastFossilCollectionFinishTime then
            met = true;
            break;
        end
    end
    if !met then
        exit;
    end
end
Step 2: Recover fossils referenced by new manifests and
  permanently delete others.
Fossil[] fossils = getLocalFossils();
for fossil in fossils do
    if $isReferenced(fossil, newManifests)$ then
        recoverFossil(fossil);
    end
    else
        deleteFossil(fossil);
    end
end
```

3.3 Guaranteed Lock Free

We now show that concurrent backups, concurrent deletions, and concurrent backup and deletion are all lock-free.

Theorem 1. *The proposed backup procedure is concurrently lock-free.*

Proof. In the proposed backup procedure, each chunk is saved individually in its own file named by the content hash. When multiple backup clients runs at the same time, only new chunks and new manifest files will be generated due to new or changed content, and then uploaded. In case of different backup clients generating the same chunk, multiple copies of the same chunk are usually resolved by the storage server (which tends to keep the most recent copy). In any case, each backup process runs independently of each other. Therefore, the proposed backup procedure is concurrently lock-free. □

Theorem 2. *The proposed two-step fossil deletion procedure is concurrently lock-free.*

Proof. Since the deletion procedure includes two steps, namely *fossil collection* and *fossil deletion*, when considering the concurrent deletion, there are three cases to be considered.

Case one: multiple processes are performing *fossil collection*. They respectively download all manifest files and compute the set of fossils. They may attempt to rename the same chunk into a fossil at the same time. Only one renaming operation will be successful, and others will report an error which can be safely ignored. So, multiple processes performing the *fossil collection* step are lock-free.

Case two: multiple processes are performing *fossil deletion*. They respectively check whether each known backup client generates a new backup and meets Policy 3. If so, the collected fossils will be deleted. They may attempt to delete the same fossil at the same time. Similar to case one, only one deletion will be successful and others can be ignored.

Thus, multiple processes performing *fossil deletion* are lock-free.

Case three: some processes are performing *fossil collection* while other processes are performing *fossil deletion*. *fossil collection* operates on chunks and *fossil deletion* operates on fossils. The only link between them is the local fossil collection files. *fossil collection* generates this file and *fossil deletion* deletes fossils recorded in the file. As this file is kept locally, if we only allow one deletion process on each computer, then we can guarantee that the *fossil collection* step and the *fossil deletion* step cannot run at the same time. Thus, the processes performing *fossil collection* and the processes performing *fossil deletion* will not affect each other, and are therefore lock-free.

Considering all these 3 cases, the proposed two-step fossil deletion procedure is concurrently lock-free. □

Theorem 3. *The interplay of the backup procedure and the deletion procedure is concurrently lock-free.*

The backup procedure generates referenced chunks while the deletion procedure collects fossils and then deletes unreferenced fossils. The proof of Theorem 3 is to prove that there is no deletion of chunks referenced by backups in the concurrent interplay of the backup procedure and the deletion procedure.

Proof. Assuming there exists a referenced chunk that is deleted by the deletion procedure, the backup generating the reference to the chunk is a new backup that was not seen by the *fossil collection* step. Otherwise, the chunk will not be collected as a fossil and then not be deleted by the *fossil deletion* step. We denote the backup as backup A. So, A is a new backup.

For example, Fig. 1 illustrates an interplay instance between the backup procedure and the deletion procedure. The *start* and *end* are the positions making timestamps. For the case (a), although the end time of the backup was before the start time of the *fossil collection* step, due to the delay such as networking delay, the time of the manifest file reaching *Storage* was later than the start time of the *fossil collection* step. Fortunately, the arrival time was earlier than the time of the request to download manifest files. As a result, the backup was seen by the *fossil collection* step. That is, the chunks referenced by the backup will not be collected as fossils and will not be deleted. For the case (b), the time of the manifest file arriving at *Storage* are later than the time of the request to download manifest files. As a result, the backup is not seen by the *fossil collection* step and thus the chunks only referenced by the backup are collected as fossils.

So, backup A is like the case (b) not seen by the *fossil collection* step in Fig. 1. If A is also not seen by the *fossil deletion* step, the step will delete the fossil referenced by A as a result that a referenced chunk is deleted. Otherwise, the *fossil deletion* step will rename the referenced fossil into the chunk according to the manifest file as a result that the deletion of a referenced chunk will not happen. Furthermore, the start time of A is earlier than the end time of the *fossil collection* step. Otherwise, the *fossil collection* step has collected the chunk as a fossil before the start time of A. According to Policy 2, when A looks up the chunk, it cannot find the chunk, even if the corresponding fossil exists, and thus it uploads the chunk. As a result, the deletion of the fossil will not lead to the deletion of a referenced chunk.

In fact, according to Policy 3, the *fossil deletion* step will not delete fossils until each backup process generates a new backup whose end time is after the end time of the *fossil collection* step. So, the backup process also generates a new backup called backup B whose end time is after the end time of the *fossil collection* step. The B is not the A because B is seen by the *fossil deletion* step before the step start deleting fossils according to Policy 3. In addition, the same backup process will not start another backup until the current backup successfully ends just like *Backup1* following *Backup2* in Fig. 1 (d). Now, we discuss the order of A and B as follows.

1) A is before B. Since B is seen by the *fossil deletion* step, A is also seen by this step, which is a contradiction that A is not seen by the *fossil deletion* step.
2) A is after B. Since the end time of B is after the end time of the *fossil collection* step according to Policy 3, the start time of A is also after the end time of the *fossil collection* step, which is a contradiction that the start time of A earlier than the end time of the *fossil collection* step.

Above all, the interplay of the backup procedure and the deletion procedure is concurrently lock-free. □

In addition, the second condition in Policy 3 is very important for the lock-free interplay. Ignoring the second condition will lead to the deletion of a referenced chunk. For example, in Fig. 1, cases (b), (c) and (d) are all new backups from the same backup process. If the *fossil deletion* step immediately checks Policy 3 after the end time of the *fossil collection* step, the first condition is met due to case (b). If the second condition is ignored, Policy 3 is met. And then, the *fossil deletion* step starts deleting fossils. If case (c) is not seen by the *fossil deletion* step, since the chunks only referenced by case (c) are collected as fossils by the *fossil collection* step, the *fossil deletion* step will delete these fossils as a result that these referenced chunks are deleted. If the second condition in Policy 3 is not ignored, even if the *fossil deletion* step immediately checks Policy 3 after the end time of the *fossil collection* step, since the second condition is not met, the *fossil deletion* step will not delete fossils until case (d) is seen by this step, which is illustrated in Fig. 1. When case (d) is seen by the *fossil deletion* step, the step also sees case (c). As a result, the deletion of a referenced chunk will never happen.

Concurrent deduplication includes three basic concurrent situations: concurrent backup, concurrent deletion and the concurrent interplay of backup and deletion which are all proved to be lock-free by Theorem 1, Theorem 2 and Theorem 3, respectively. So, the proposed deduplication is concurrently lock-free.

4 Duplicacy

In this cloud age, with a wide variety of competing cloud storage services to choose from, users willing to spend efforts and resources to protect their data always expect that

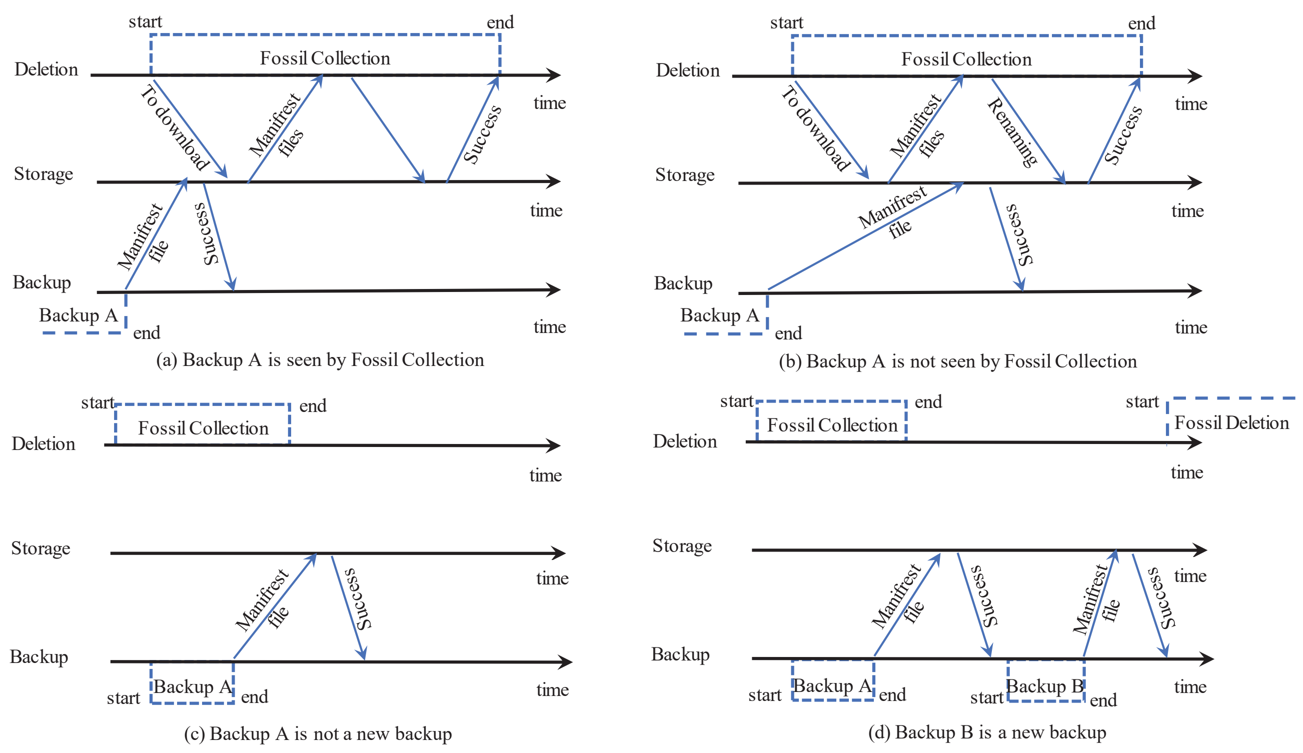
start
end
Fossil Collection
Deletion
time
To download
Manifrest files
Success
Storage
Manifrest file
Backup
Backup A
(a) Backup A is seen by Fossil Collection
Renaming
(b) Backup A is not seen by Fossil Collection
(c) Backup A is not a new backup
Fossil Deletion
Backup B
(d) Backup B is a new backup

Fig. 1. An instance of the interplay between the proposed backup procedure and the proposed deletion procedure. Timestamps are made at these positions pointed by *start* and *end*.

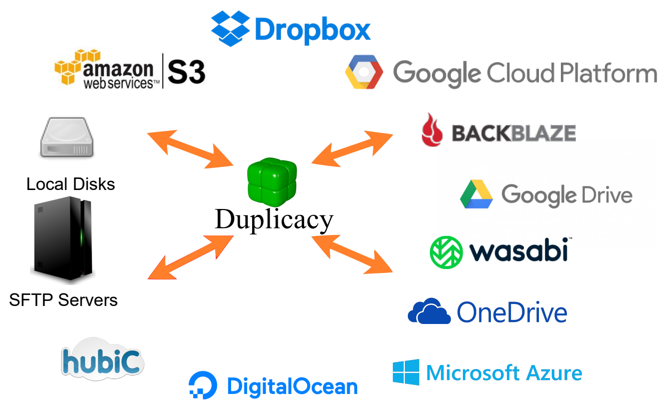

Fig. 2. The wide range of cloud storages and local or networked drives that are all supported by Duplicacy.

there is a single backup tool to support all those different storage services.

This requires that a backup tool should be built on top of only 6 basic file access APIs: upload, download, deletion, lookup, list, and renaming. Besides, to provide high storage efficiency and to enforce privacy, we argue that the following 6 essential features are expected of a state-of-the-art backup tool:

- **Incremental backup:** only back up what has been changed.
- **Full snapshot:** even if each backup is incremental, it must appear to be a full snapshot independent of others for quick restoration and easy deletion.
- **Deduplication:** identical parts from the same or different files must be stored only once.
- **Encryption:** encrypt not only file contents but also file paths, sizes, timestamps, etc.
- **Deletion:** every backup can be deleted individually without affecting others.
- **Concurrency:** multiple clients can back up to the same storage at the same time.

Among these 6 essential features, deduplication is perhaps the most important one. Deduplication not only reduces bandwidth consumption and storage space, but also speeds up backup operations due to identical parts stored only once. Furthermore, deduplication is also the basis for enabling other features. For instance, deduplication only stores the changed parts and does not store other identical parts repetitively, which naturally enables incremental backup.

Duplicacy [10] is the first backup tool built on the techniques of the proposed lock-free deduplication. Since the lock-free deduplication only depends on the 6 basic file access APIs, Duplicacy is capable of supporting a wide range of cloud storage systems including Amazon S3, Google Cloud Storage, Backblaze B2, Microsoft Azure, Google Drive, Microsoft OneDrive, Drop-box, and hubiC, as well as local or networked drives, SFTP servers like Linux computers or NAS devices (Fig. 2). The command line version of Duplicacy is written in the Go language, so it runs on all platforms supported by Go. It is free for personal use and the source code is available on github.com [10]. The GUI version of Duplicacy is a web-based wrapper around the command line version to provide additional features such as easy configurations, email notifications and so on. There is also a special edition, called Vertical Backup [18], which is specifically developed and optimized for VMware vSphere Hypervisor (ESXi) to back up virtual machines.

We also declare that Duplicacy supports all 6 aforementioned essential features. Since lock-free deduplication allows for concurrent backups and deletions, Duplicacy natively support 3 of them, namely, deduplication, deletion and concurrency, which have been detailed in Section 3. Here we present how other features are made possible by the lock-free deduplication, as well as some efficiency considerations in the design of Duplicacy.

4.1 Incremental Backup and Full Snapshot

Duplicacy adopts the variable-size chunking algorithm by default. Specifically, the cyclic polynomial hash [19] is chosen. What is different in Duplicacy from other implementations is that Duplicacy uses a relatively long sliding window for two reasons: 1) it can lower the chance of chunk breakpoints occurring too frequent and 2) the size of the slicing window becomes the minimum chunk size, reducing the number of configuration parameters.

Duplicacy also supports fixed-size chunking algorithm. If the minimum chunk size, the average chunk size, and the maximum chunk size are set to the same value when initializing a storage, Duplicacy will switch to the fixed-size chunking algorithm. This eliminates the overhead of computing the rolling hash and is therefore preferable when backing up large files unlikely subject to insertions and deletions, such as databases and virtual machine disk images. The ESXi edition, Vertical Backup [18], uses only the fixed-size chunking algorithm in order to achieve the maximum performance.

A chunk-based approach provides native support for incremental backup and full snapshot. A backup is composed of a number of chunks, so for each new backup, only new chunks are needed to be uploaded. Backups are independent of each other, even if they may share some chunks. To restore a backup, only chunks referenced by this backup are needed to be downloaded. Any backup other than the latest one from each backup client can also be deleted individually, by the two-step fossil deletion algorithm.

4.2 Caching

Lock-Free Deduplication requires that a file lookup is performed before uploading each chunk, which means two API calls per chunk. To alleviate this overhead, it is preferable to maintain an in-memory cache to store hashes of chunks referenced by the last backup from the same client. As a backup finishes, a copy of the backup manifest file can be saved in a local disk. The next backup will load this backup manifest file and construct the chunk cache accordingly. Only chunks that are not in the chunk cache will trigger a lookup on the storage. With a cache set up this way, the *fossil collection* step should not attempt to remove the last backup for each backup client. This is mainly to avoid race condition between the *fossil collection* step deleting a backup manifest file and the backup procedure reading the same one to build the chunk cache.

Theorem 4 illustrates the feasibility of caching the last backup in each backup client, that is, caching does not break the availability of the two-step fossil deletion algorithm.

Theorem 4. *Caching the last backup in each backup client does not invalidate the two-step fossil deletion algorithm.*

Proof. As a fact, a backup starting before the end of the *fossil collection* step will always have their chunks taken into account in the *fossil deletion* step according to Policy 3. Suppose that there is a backup that starts after the *fossil collection* step finishes but before the *fossil deletion* step makes a reference to a chunk. If the chunk is not in the cache, it will issue a lookup and the chunk will be uploaded if the chunk has been made a fossil. If the chunk is in the cache, it must be referenced by the previous backup as well. And the previous backup may issue the lookup, or it may inherit the chunk from the other backup before it. Therefore, there must be a chain of backups that trace back to either a backup that is known to the *fossil deletion* step, or to a backup that is still unknown to the *fossil deletion* step. If it is the latter, since the backup is unknown to the *fossil deletion* step, the start time of the backup is later than the finish time of the *fossil collection* step. As a result, the lookup of the unknown backup for the already fossilized chunk should have failed, which caused the same chunk to be uploaded again according to Policy 2. □

4.3 Compression and Encryption

A chunk-based approach also lends itself to compression and encryption since each chunk can be individually compressed and encrypted. It is apparent that compression must be done before encryption so as to reduce the encryption cost, but should the hash used as the chunk file name be calculated before or after compression and encryption?

Duplicacy calculates the hash before applying the compression and encryption algorithms because the hash is directly used for the chunk lookup and if the chunk turns out to be identical to an existing one, there will be no need to perform the compression and encryption. However, if the hash of the chunk content in plaintext is to be used as the file name, then it becomes susceptible to the identifying files attack [20], i.e., an attacker can determine if a given chunk exists in the original file. To avoid this vulnerability, Duplicacy applies an HMAC function to the chunk hash, and uses the result as the file name of the chunk. Since the HMAC secret is shared by all backup clients but not known publicly, it is impossible to deduce the chunk hash from the chunk name without knowing the HMAC secret.

When encryption is enabled, Duplicacy will generate four random 256 bit keys as follows:

- **Hash Key:** for generating a chunk hash from the content of a chunk.
- **ID Key:** for generating a chunk id from a chunk hash.
- **Chunk Key:** for encrypting chunk files.
- **File Key:** for encrypting non-chunk files such as manifest files.

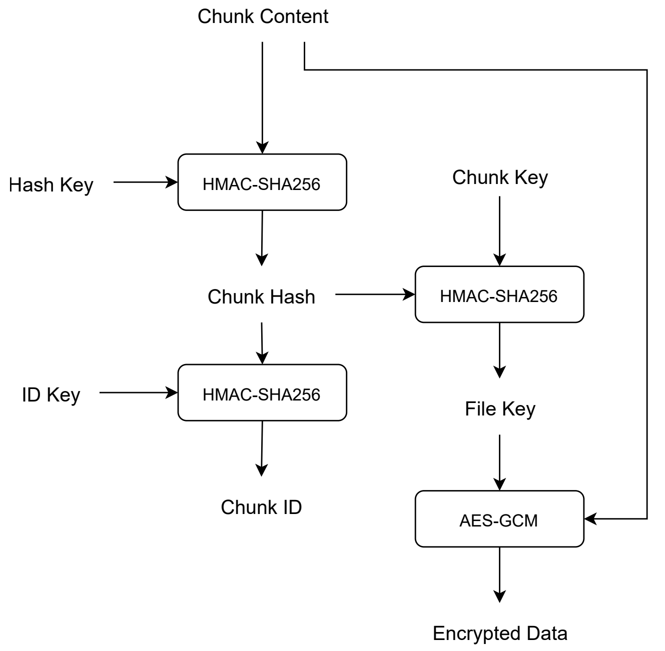

Fig. 3. The process flows of chunk encryption with four random 256 bit keys, namely *Hash Key*, *ID Key*, *Chunk Key*, *File Key*.

Fig. 3 shows how these keys are used. Both the chunk hash and the *ID Key* are used to generate the chunk id which becomes as the file name for the chunk. So, chunk hashes are never exposed unless the manifest files are decrypted. Chunk content is encrypted by AES-GCM, with an encryption key that is the HMAC-SHA256 of the chunk Hash with *Chunk Key* as the secret key. The manifest files are also encrypted by AES-GCM, using an encrypt key that is the HMAC-SHA256 of the file path with *File Key* as the secret key. These four random keys are saved in a file named 'config' in the storage, encrypted with a master key derived from the PBKDF2 function on the storage password chosen by the user.

5 Evaluation and Analysis

In this section, we demonstrate the advantages of Duplicacy in the following three aspects.

- **Backup Performance:** It compares the execution times of backup and restore, the two most frequent use cases of a backup tool, among Duplicacy and other well-known backup tools [5], [6], [7], [8].
- **Storage Efficiency:** It illustrates the amounts of storage space used by Duplicacy and other backup tools.
- **Ubiquity Integration:** It presents the wide support of Duplicacy for cloud storage systems and local or networked drives.

5.1 Experiment Setup

Tool Selection for Comparison: Three popular backup tools selected for comparison are Duplicity [5], Restic [6], Attic [7], [8]. Both Restic and Attic, like Duplicacy, employ a chunk-based approach, while Duplicity is a more traditional backup tool selected only for its popularity. Table 1 lists main configuration parameters for these tools used in experiments. We believe that these configurations have significant impacts on the overall evaluation, so similar options were chosen whenever possible to ensure an unbiased base.

TABLE 1
The main configuration of four backup tools used in experiments.

Configuration	Duplicacy	Restic	Attic	Duplicity
Version	2.1.1	0.9.2	1.1.7(Borg)	0.7.12
Average Chunk Size	2 MB	1 MB	2 MB	25 MB
Hash	Blake2	SHA256	Blake2	SHA1
Compression	LZ4	\	LZ4	ZLIB Level 1
Encryption	AES-GCM	AES-CTR	AES-CTR	GNUPG

Dataset Selection: Two typical datasets [21], [22] are chosen in these experiments. The first dataset is the Linux code base [21] which is the largest github repository that we could find. It has frequent commits and its total size is 1.76 GB with many small files about 58K files, so it is good for testing incremental backups. It also represents a popular use case where a backup tool runs alongside a version control program such as git to frequently save changes made between checkins. The second dataset is the VirtualBox virtual machine file [22], a 64 bit CentOS 7 disk image. Its size is about 4 GB and is widely used nowadays. It is selected to target the other end of the use case spectrum – a dataset with fewer but much larger files.

All tests were performed on an ESXi virtual machine running on a Dell XPS 8700 with an i7 4-core processor and 16 GB memory. The virtual machine was assigned 2 virtual CPUs and 4GB memory. No other virtual machines were running during the experiments. Backups were all saved to a storage directory on the same hard disk as the source directory, to emulate the situation with unlimited network bandwidth, and also to eliminate the performance variations introduced by different implementations of networked or cloud storage backends. The scripts to run these tests are available from our github page [23].

5.2 Backup Performance

Linux Code Base: To test incremental backup, a random commit on July 2016 was selected, and the entire code base was rolled back to that commit. After the initial backup was finished, other random commits were chosen such that they were about one month apart. The code base was then moved forward to these commits one by one to emulate incremental changes.

Fig. 4 shows the elapsed time (in seconds) as reported by the *time* command for each backup. Duplicacy was clearly the best performer by a comfortable margin. It is interesting to note that Restic, being the second fastest, did not implement compression so it had an unfair speed advantage (at the cost of using larger storage space), and yet it was still considerably slower than Duplicacy. The performance fluctuations are caused by each commit having different

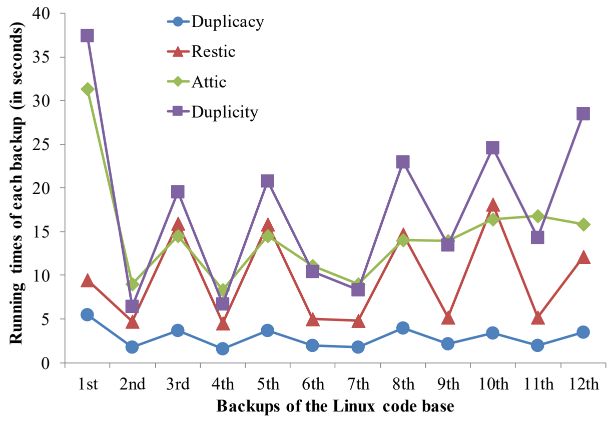

Fig. 4. The comparison of the incremental backup time cost of the Linux code base.

sizes and numbers of updated files. The first commit, the initial backup, has the most new files, and thus takes the most running time. Unlike other chunk-based backup tools where chunks are grouped into pack files and a chunk database is used to track which chunks are stored inside which pack file, Duplicacy takes a database-less approach where every chunk is saved independently using its hash as the file name to facilitate quick lookups. Fig. 5 shows the elapsed times of each restore. The destination directory was emptied before each restore. Again, Duplicacy was not only the fastest but also the most stable.

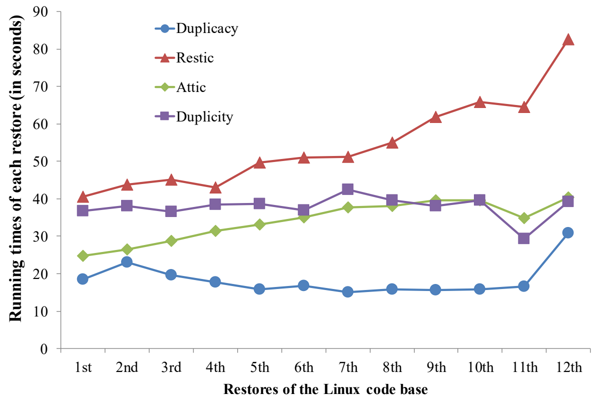

Fig. 5. The comparison of the restore time cost of the Linux code base.

VirtualBox Virtual Machine Image: For this experiment, the fixed-size chunking algorithm is enabled in Duplicacy but not in others (Duplicacy is the only one to support both fixed-size and variable-size chunking algorithms). Three typical backup scenarios are evaluated in the experiment. The first backup was performed right after the virtual machine had been set up without installing any software. The second backup was performed after installing common developer tools. The third backup was performed after a power on immediately followed by a power off. Fig. 6 shows the incremental backup time cost and Fig. 7 shows

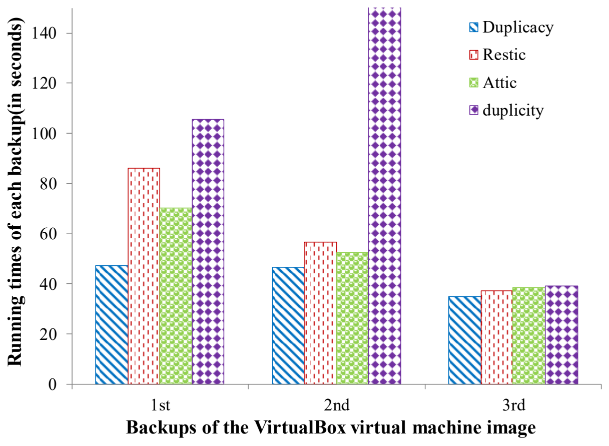

Fig. 6. The comparison of the incremental backup time cost of the VirtualBox virtual machine image.

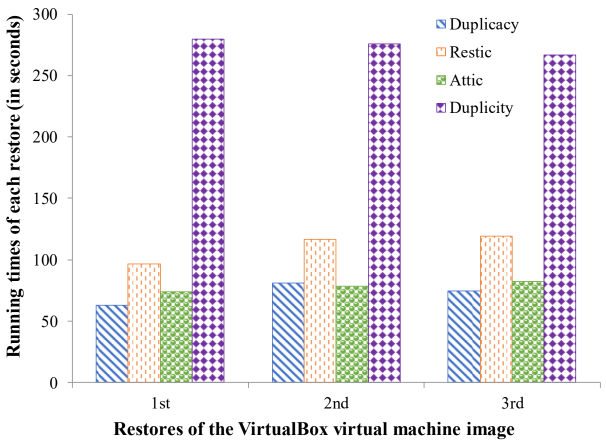

Fig. 7. The comparison of the restore time cost of the VirtualBox virtual machine image.

the restore time cost. Duplicacy is still the fastest in all cases.

5.3 Storage Efficiency

Fig. 8 shows the comparison of storage usage cost for each backup of the Linux code base. Our Duplicacy gets the second highest storage efficiency and lower than that of Duplicity because Duplicity benefits from compressing the entire source directory as opposed to individual chunks used by ours.

We also ran another experiment to determine deduplication efficiency with multiple source directories concurrently backing up to the same storage, for Duplicacy and Restic only, as concurrent backups are not supported by Attic and Duplicity. In this experiment, the Linux code base was duplicated to another directory, and both directories with identical contents were backed up to the same storage concurrently. As indicated by Fig. 8, Duplicacy was capable of exploiting cross-client deduplication on the fly to maintain the roughly same level of storage usage, while Restic failed to do so and its storage usage literally doubled.

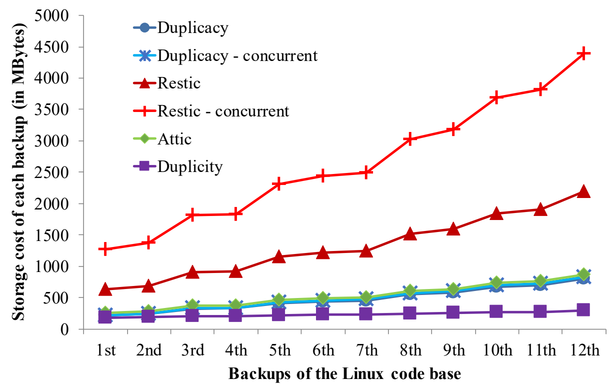

Fig. 8. The comparison of storage usage for each backup of the Linux code base.

This clearly demonstrates one major advantage of not using a chunk database. The chunk database in Restic cannot be concurrently shared by multiple backup clients, so the information regarding new chunks uploaded by one client can only become available to other clients after the end of the backup. As a result, different clients running in parallel will upload their own versions of identical chunks led to the doubled storage usage.

The experiment for storage efficiency of the VirtualBox virtual machine image has almost the same results as that of the Linux code base. So we omit the result report of the VirtualBox virtual machine image here and the integral report is available from our github page [23].

5.4 Ubiquity Integration

Here we test and compare the performances of major cloud services when used as backup storage systems for Duplicacy. All tests were performed on a Ubuntu 16.04.1 LTS virtual machine running on a dedicated ESXi server with an Intel Xeon D-1520 CPU (4 cores at 2.2 GHz) and 32G memory. The server is located at the east coast of USA, so the results may be biased against those services who have their servers on the west coast. The network bandwidth is 200Mbps. A local SFTP storage is also included in the test to provide a base line for the comparisons. The SFTP server runs on a different virtual machine on the same ESXi host. All scripts to run the tests are available in our github page [24].

Fig. 9 and Fig. 10 present the backup performance comparison of cloud storage systems for Duplicacy based on the Linux code base and the VirtualBox virtual machine image, respectively. They demonstrate similar results. These results indicate that the performances of different cloud storage systems vary a lot. That is, while S3-compatible ones (Amazon, Wasabi, and DigitalOcean) and Azure can back up and restore at speeds close to those of the local SFTP storage, others are much slower. S3-compatile ones and Azure all support Simple Storage Service API that has become the de-facto standard for create, read, update, and delete operations for object storage. Storages designed to be primarily accessed via an API are generally faster than storages that are offered as cloud drives, because the latter are perhaps more optimized for their own clients with the API access merely being an addon. In additions, most cloud storage systems support simultaneous connections, so we can keep increasing the number of threads to improve performance, until the local processing or the network becomes the bottleneck. The bottom charts in Fig. 9 and the right charts in Fig. 10 are the results with four threads. Compared with the single-thread charts in Fig. 9 and Fig. 10, most of them demonstrate the significant improvement for performance. Google Drive was the only cloud storage that didn't benefit from the use of multiple threads, possibly due to strict per-user rate limiting. Such results suggest that cloud backup can be as fast as local backup, with only modest network bandwidth, especially if we can use multiple threads. Dropbox doesn't support simultaneous writes, so it was missing from the multiple-threads charts.

6 Related Work

Deduplication is a technique to eliminate file redundancy, and may incur significant benefits with regard to storage efficiency, network bandwidth and even backup speeds. According to [25], data deduplication achieves storage reduction by more than 50% in standard file systems and by up to 90% to 95% for backup applications, which is superior to data compression and dominates storage efficiency. Many research works including some recent surveys [1], [3], [4] renewed the interests in data deduplication. Most of them focus on two directions of data deduplication, namely secure data deduplication and accelerated deduplication. The former often uses encryption based privacy strategies [4], [26], [27] that can be integrated into our backup tool, Duplicacy, as illustrated in Section 4.3. The latter comes from popular backup tools for general purpose [5], [6], [7], [8], [28], [29] and custom-built proprietary [30], [31] storage systems.

Duplicity [5], Attic [7], [8], Obnam [29] and Duplicita [28] are all well-known and widely used backup tools for general storage systems. Duplicity [5] works by applying the rsync algorithm (implemented by the librsync library [32]) to find the differences from previous backups and then uploading the differences. A backup is either full or incremental, but an incremental backup becomes dependent on previous back-ups. As a result, frequent full backups are required, otherwise the dependency chain would be too long led to rendering deletion impossible and restoration slow. The deficiency in this incremental model is evidently reflected by its poor performance. Attic [7], [8], Obnam [29] and Duplicita [28] embrace the chunk-based approach. They all got the incremental model right in the sense that every incremental backup is actually a full, independent snapshot. But, their common disadvantage is the lack of support for concurrent backups – they all assume that only one client can back up to the storage exclusively. This requirement makes them less likely to be adopted for large-scale deployment where cross-client deduplication can be the main source of deduplication.

Very few backup tools are designed to support concurrent backups to cloud storages. Restic [6] is the closest to our Duplicacy in terms of the ability to exploit cross-client deduplication. However, as shown by Fig. 8, for concurrent

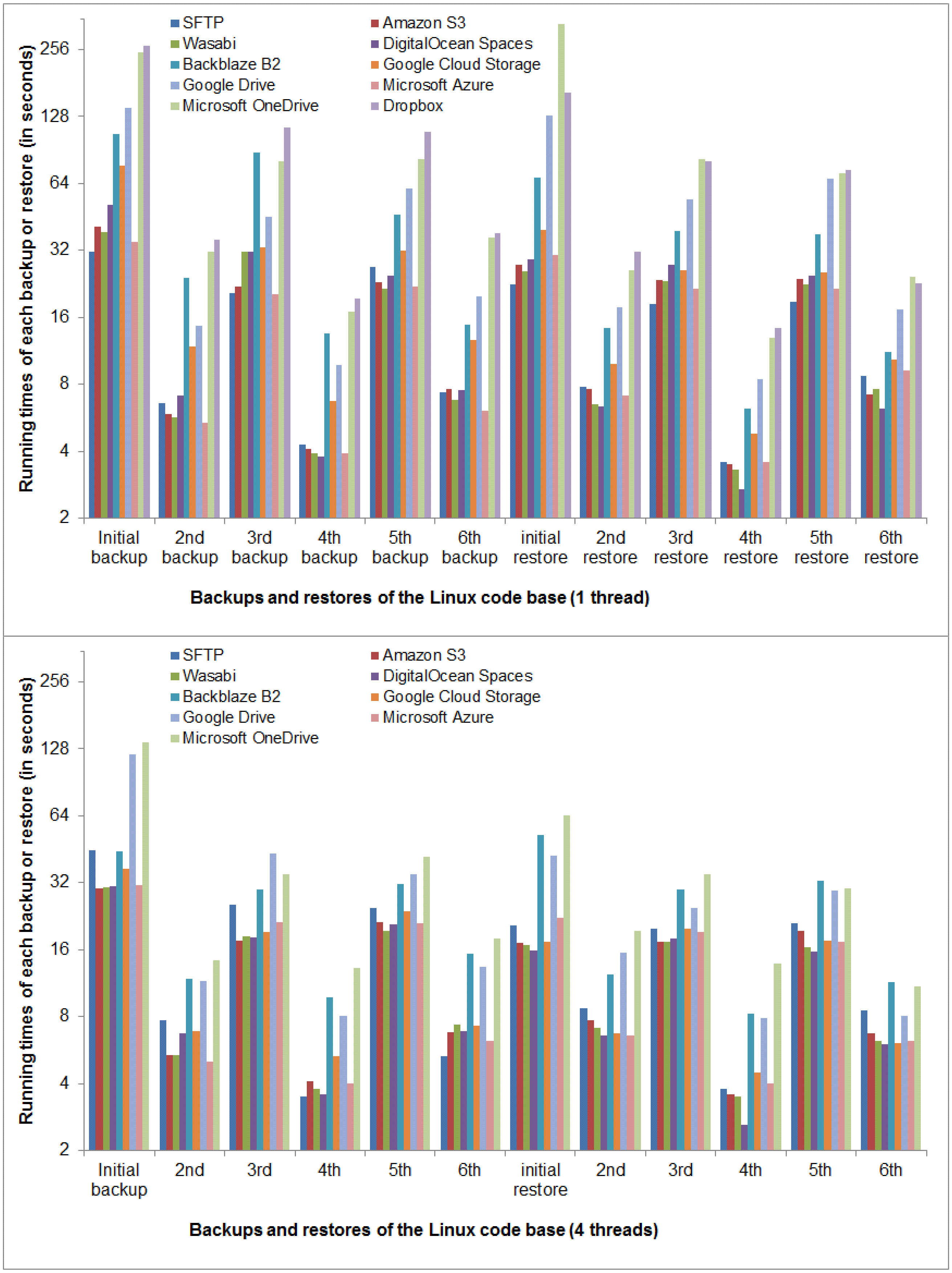

Fig. 9. The performance comparison of major cloud services when used as backup storage systems for Duplicacy based on the Linux code base.

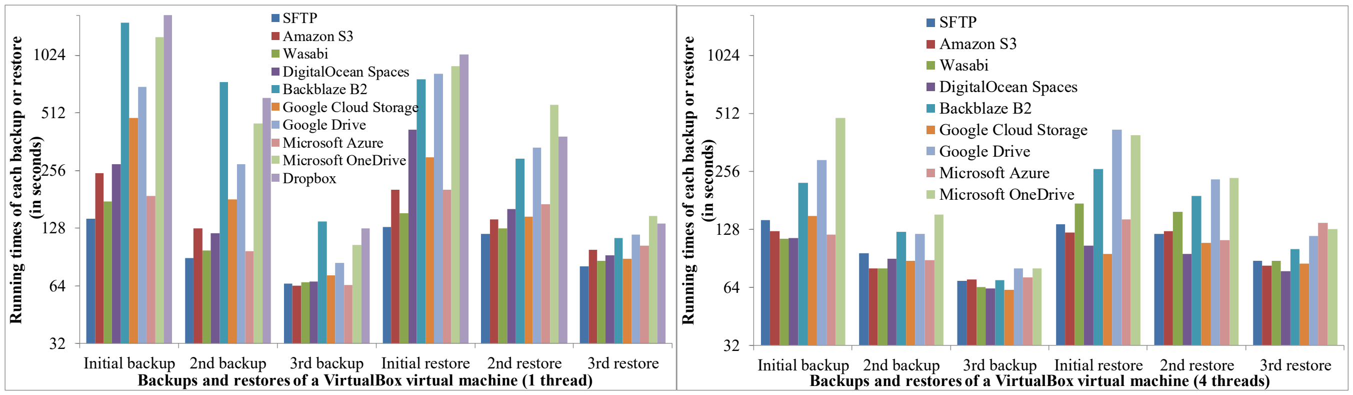

Fig. 10. The performance comparison of major cloud services when used as backup storage systems for Duplicacy based on the VirtualBox virtual machine image.

backups it loses this ability due to the use of a chunk database. Moreover, Restic relies on locking and a prune operation will therefore completely block all other clients connected to the storage from doing their regular backups. More fundamentally, as most cloud storage services do not provide a locking service, the best effort is to use some basic file operations to simulate a lock, but distributed locking is known to be a hard problem [33] and it is unclear how reliable Restic's lock implementation is. A faulty implementation may cause a prune operation to accidentally delete chunks still in use, resulting in unrecoverable data loss.

There are also custom-built proprietary storage systems that work as concurrent deduplication servers, where either reference counting is available [30] or there are special OS primitives to record writes during garbage collection [31]. However, these techniques are not applicable to cloud backup storage systems which are distributed in nature and often come with limited APIs. [34] handles the conflicts of editing collaboration on the same file. It can solve the conflicts that happen in different parts of the same file to avoid a lock while it cannot solve the conflicts that happen in the same part of the same file. In such a case, a lock is still needed to synchronize different editing operations. On contrast, our work targets the lock-free deduplication for backups. Any update to a chunk file will lead to a different hash value causing a new chunk file to be uploaded. Our method can create lock-free backups even if multiple clients concurrently operate the same part in the same source file.

7 Conclusion

In this paper, we propose a new cross-client deduplication solution called Lock-Free Deduplication to improve backup speeds at the cloud storage era. Its key idea can be summarized as follows:

- Use a fixed-size or variable-size chunking algorithm to split files into chunks.
- Store each chunk in the storage using a file name derived from its hash, and rely on the basic file system APIs to detect duplicates.
- Apply a two-step fossil collection algorithm to remove chunks that become unreferenced after a backup is deleted.

These proposed techniques are implemented in a new cloud backup tool called Duplicacy. It highlights lock-free concurrency and achieves significant improvements for backup performance compared with those other backup tools. Furthermore, with the only requirements of a basic set of file APIs, Duplicacy achieves a wide range support for cloud storage systems and local or network drives, turning them into sophisticated deduplication-aware storage servers.

Acknowledgments

This research is sponsored in part by the NSFC Key Scientific Instrument and Equipment Development Project (No.20151310834), in part by the Fundamental Research Funds for the Central Universities (No.2019RC046) and the Project funded by China Postdoctoral Science Foundation (No.2019M660439), and in part by the National Key Research and Development Program of China (No.2018YFB1702602). The authors would like to thank anonymous reviewers for comments.

References

[1] Y. Mansouri, A. N. Toosi, and R. Buyya, "Data storage management in cloud environments: Taxonomy, survey, and future directions," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 91:1–91:51, dec 2017. [Online]. Available: http://doi.acm.org/10.1145/3136623

[2] Cisco. (2019, 11) Cisco global cloud index: Forecast and methodology, 2016-2021. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html

[3] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, Sep. 2016.

[4] Y. Shin, D. Koo, and J. Hur, "A survey of secure data deduplication schemes for cloud storage systems," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 74:1–74:38, jan 2017. [Online]. Available: http://doi.acm.org/10.1145/3017428

[5] Savannah and Launchpad. (2019, 05) Duplicity: Encrypted bandwidth-efficient backup using the rsync algorithm. [Online]. Available: http://duplicity.nongnu.org/

[6] A. Neumann. (2018, 08) Restic: Fast, secure, efficient backup program. [Online]. Available: https://restic.net/

[7] J. Borgstrm. (2015, 05) Attic: an efficient and secure way to backup data. [Online]. Available: https://attic-backup.org/

[8] T. Waldmann, A. Beaupr, R. Podgorny, and Y. DElia. (2018, 08) Borg: an efficient and secure way to backup data with compression and authenticated encryption. [Online]. Available: https://borgbackup.readthedocs.io/

[9] I. Dropbox. (2019, 08) Dropbox: Simple & secure cloud storage. [Online]. Available: https://www.dropbox.com

[10] L. Acrosync. (2019, 08) Duplicacy: A new generation cross-platform cloud backup tool with client-side encryption and the highest level of deduplication. [Online]. Available: https://duplicacy.com/

[11] A. Tridgell, P. Mackerras *et al.*, "The rsync algorithm," *Technical Report*, 1996.

[12] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 174–187. [Online]. Available: http://doi.acm.org/10.1145/502034.502052

[13] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," in *Proceedings of the 2002 USENIX Conference on File and Storage Technologies (FAST 02)*. Montery, CA, USA: USENIX, 2002, pp. 89–102. [Online]. Available: https://www.usenix.org/legacy/events/fast02/quinlan/quinlan_html/

[14] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404–425, Jul. 1985. [Online]. Available: http://doi.acm.org/10.1145/3916.3988

[15] R. M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990. [Online]. Available: http://doi.acm.org/10.1145/84537.84545

[16] G. Chen and B. K. Szymanski, "Lookback: A new way of exploiting parallelism in discrete event simulation," in *Proceedings of the Sixteenth Workshop on Parallel and Distributed Simulation*, ser. PADS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 153–162. [Online]. Available: http://dl.acm.org/citation.cfm?id=564062.564087

[17] G. G. Chen and B. K. Szymanski, "Time quantum gvt: A scalable computation of the global virtual time in parallel discrete event simulations," *Scalable Computing: Practice and Experience*, vol. 8, no. 4, 2007.

[18] L. Acrosync. (2019, 08) Vertical backup: A new network and cloud backup tool for vmware vsphere (esxi) fast, deduplication, encryption, live backup, schedule. [Online]. Available: https://www.verticalbackup.com/

[19] J. D. Cohen, "Recursive hashing functions for n-grams," *ACM Trans. Inf. Syst.*, vol. 15, no. 3, pp. 291–320, jul 1997. [Online]. Available: http://doi.acm.org/10.1145/256163.256168

[20] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Security Privacy*, vol. 8, no. 6, pp. 40–47, Nov 2010.

[21] L. Torvalds. (2019, 08) Linux kernel source. [Online]. Available: https://github.com/torvalds/linux

[22] Umair. (2019, 08) Centos virtual machine images for vmware and virtualbox. [Online]. Available: https://www.osboxes.org/centos/

[23] L. Acrosync. (2019, 08) Experimental benchmarks and executable scripts. [Online]. Available: https://github.com/gilbertchen/benchmarking

[24] L. Acrosyn. (2019, 08) Cloud storage comparison. [Online]. Available: https://github.com/gilbertchen/cloud-storage-comparison

[25] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *Trans. Storage*, vol. 7, no. 4, pp. 14:1–14:20, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2078861.2078864

[26] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch, "Libseal: Revealing service integrity violations using trusted execution," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 24:1–24:15. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190547

[27] J. Dave, P. Faruki, V. Laxmi, B. Bezawada, and M. Gaur, "Secure and efficient proof of ownership for deduplicated cloud storage," in *Proceedings of the 10th International Conference on Security of Information and Networks*, ser. SIN '17. New York, NY, USA: ACM, 2017, pp. 19–26. [Online]. Available: http://doi.acm.org/10.1145/3136825.3136889

[28] Duplicati. (2019, 07) Duplicati:free backup software to store encrypted backups online. [Online]. Available: https://www.duplicati.com/

[29] Obnam. (2017, 08) Obnam: An easy, secure backup program. [Online]. Available: https://obnam.org/

[30] P. Strzelczak, E. Adamczyk, U. Herman-Izycka, J. Sakowicz, L. Slusarczyk, J. Wrona, and C. Dubnicki, "Concurrent deletion in a distributed content-addressable storage system with global deduplication," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX, 2013, pp. 161–174. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/strzelczak

[31] F. Douglis, A. Duggal, P. Shilane, T. Wong, S. Yan, and F. Botelho, "The logic of physical garbage collection in deduplicating storage," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, feb 2017, pp. 29–44. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/douglis

[32] Librsync. (2018, 02) Librsync: Remote delta-compression library. [Online]. Available: https://github.com/librsync/librsync

[33] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. OReilly, March 2017. [Online]. Available: https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

[34] J. Chen, M. Zhao, Z. Li, E. Zhai, F. Qian, H. Chen, Y. Liu, and T. Xu, "Lock-free collaboration support for cloud storage services with operation inference and transformation," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 13–27. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/chen

Zonghui Li received the B.S. degree in computer science from the Beijing Information Science and Technology University in 2010, and the M.S. and Ph.D. degree from the Institute of Microelectronics and the School of Software, Tsinghua University, Beijing, China, in 2014 and 2019, respectively.

He is currently an assistant professor in the School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China. His research interests include embedded and high performance computing, real-time embedded systems, especially for industrial control networks and fog computing.

Gilbert (Gang) Chen received the BE and ME degrees in Electrical Engineering from Tshinghua University in China, and the PhD degree in Computer Science from Rensselaer Polytechnic Institute.

He is currently an independent software developer focused on building reliable and robust cross-platform software solutions for efficient storage backup and synchronization. He is the founder of Acrosync LLC.

Yangdong Deng received the BE and ME degrees from the Electrical and Electronics Department, Tsinghua University, Beijing, China, in 1998 and 1995, respectively, and the PhD degree in electrical and computer engineering from the Carnegie Mellon University, Pittsburgh, PA, in 2006.

He is currently an associate professor at the School of Software, Tsinghua University, Beijing, China. His current research interests include industry data analytics, brain inspired computing, and computer architecture. He is a member of the IEEE. He received a best paper award from ICCD 2013.