A secure and flexible backup system in Haskell

DIKU

Keywords: DIKU, master's thesis, backup, deduplication, garbage collection, b-tree, crash-safety, security, implementation, Haskell.

Nøgleord: DIKU, speciale, backup, deduplikering, spildopsamling, b-træ, crash-sikkerhed, sikkerhed, implementering, Haskell.

Formalities:

Start date:	5th of September 2011
End date:	16th of April 2012
Defence date (latest):	30th of April 2012
Subject to:	We reserve all rights over developed design and/or software.

Johan Sejr Brinch Nielsen & Morten Brøns-Pedersen

Students:	Johan Brinch	<zerrez@diku.dk>,
	Morten Brøns-Pedersen	<mortenbp@diku.dk>
Supervisor:	Ken Friis Larsen	<kflarsen@diku.dk>

Dept. of Computer Science,
University of Copenhagen

Abstract

Hindsight is a backup system providing guaranteed safe backup to untrusted sources such as SSH, Amazon S3 or DropBox.

We employ concurrent b-tree for storing log data, as opposed to the more common transactional database. This choice has two beneficial side effects: *a)* more flexible handling of program crashes (which allows resuming progress); and *b)* compression and lazy retrieval of log data (which allows efficient snapshot inspection).

In this report, we present the design and a prototype implementation written in Haskell. Our prototype implements conservative garbage collection, but we give a comprehensive discussion of other techniques as well. We compare our prototype to several other systems and present benchmarks.

Additional material. In addition to this report, we have produced a poster and a series of slides. The poster was used at an open day event the 9th of Marts 2012, at the H.C. Ørsted institute at Copenhagen University. We used the slides during a presentation at the 13th annual Open Source Days conference.

The poster and slides are available at:

`http://hind.sight.dk/doc/poster.pdf`
`http://hind.sight.dk/doc/slides.pdf`

The source code as described in this report is available here:

`hind.sight.dk/src/hindsight.tar.gz`

The latest version of Hindsight can be found here:

`http://hind.sight.dk`

Acknowledgements. We would like to thank the following people who helped improve this project and the resulting documents:

Our supervisor Ken Friis Larsen for guidance. The good people of the Haskell mailing list for their elaborate answers. The GHC developers for responding quickly to bug reports. Thomas Conway for discussion and input on implementing a concurrent b-tree. Everyone involved with proof reading the report (especially Mathias Svensson and Lisbeth Brinch). The crew behind Open Source Days for letting us talk. Brian Vinter and Jesper Rude Selknæs for lending us a server with +5TB disk space. Fellow student Jesper Reenberg for commentary and LaTeX assistance. And Jesper Louis Andersen for being awesome as always.

Contents

Chapter 1

Introduction

In this report, we present a novel design for a snapshotting backup system along with a prototype implementation. While our design lends itself to general key-value storage, our main focus is file system snapshotting. Our prototype runs on Linux and handles most common file meta-data. It is implemented in Haskell and uses a process model that allows trivial exploitation of parallelism.

Our design features global deduplication through content-based indexing [62]; a means for eliminating copies of redundant data. To store log data (the entire internal state maintained by the backup system), we employ concurrent b-trees instead of the popular single-file approach, often in the form of a SQLite database (e.g. as used by S3QL [79] or Brackup [2]). This allows for efficient updates while drastically decreasing log data overhead – especially when storing large amounts of data. This decrease is achieved by reusing the system to store the newly generated log data, hence reapplying deduplication. Furthermore, the use of b-tree makes it possible to query the structure of backed up data (e.g search for a file) without all the log data being present locally.

For inspection, snapshots can be mounted as read-only filesystems through FUSE [47]. The ability to only download parts of a snapshot's file structure makes the filesystem quite responsive.

We discuss several solutions to the problem of deleting snapshots while retaining global deduplication in a crash-safe manner. This problem is often solved with reference counting, however this solution seems unpractical to combine with b-trees.

We propose using reference lists instead. Though not yet implemented, simulations suggest that the size overhead stemming from the reference lists is acceptable when encoded as bit-vectors (see section A.4 on page 102).

For our prototype, a simple conservative garbage collector has been implemented. Our tests show that it is able to clean most of the garbage introduced by deleted snapshots, in at least some use-cases. We are not aware of any other backup systems, which use conservative garbage collection. The technique can be combined with an exact garbage collector, which is run on rarer occasions. We leave this as a topic of further work.

The design assumes an untrusted (but reliable) back-end with a minimal API, making it suitable for a wide range of storage solutions. Our prototype can back up to a local file system, Amazon S3 [85], a remote file system via SSH, and CouchDB [45]. Privacy and data integrity are guaranteed through encryption and authentication provided by the NaCl high-level cryptographic library [19].

We compare our prototype to a range of other systems (see section 5.1 on page 78) and present benchmarks (see appendix A on page 97). Our benchmarks are based on versions of Linux kernel source code, a Linux virtual machine image and the home directory of one of the authors. The larger snapshots are around 60GB each.

Our results show that the benefit of content-aware chunking might not be as great as the wide application [42, 72, 73, 83, 93] of the technique suggests.

The main contributions of our work are:

- A Haskell implementation of eventually balanced concurrent b-trees. Our implementation is inspired by Larsen and Fagerberg [59] and based on software transactional memory (STM) [50, 86]. To our knowledge this is the first implementation of concurrent b-trees utilising relaxed balancing, although T. Conway has also done some work in the area (see appendix D on page 123 for a record of our correspondence).

  Our b-trees are stored on disk as one file per node, which makes them ideal for deduplicated storage; few operations lead to few changed files.

- We break with the trend ([79, 93] among others) of storing log data in a transactional database. We argue that a non-transactional data structure can lead to increased performance with regard to both time and space (see section 5.3 on page 84).

  In order to maintain global deduplication and crash-safety while featuring deletion of backed up data, we are forced to abandon reference counting.

  We propose the use of reference lists encoded as bit-vectors. We provide simulations showing that the resulting space overhead is not too restrictive (see section A.4 on page 102).

- Along with our prototype we give an implementation of a conservative garbage collector and an evaluation of its effectiveness. We are not aware of any prior work investigating conservative garbage collection in a backup system.

- Log data is stored in b-trees which are split across multiple files. It is possible to operate on parts of our b-trees without all the files being present. For example, this means that one can search for files or list/checkout a subdirectory in a snapshot without retrieving the full log data.

Combining this with the usually high redundancy between log data from consecutive snapshots, yields a speed-up in viewing the changes in a small subdirectory over a period of time. See section A.3.2 on page 100 for test results.

- An almost complete – and presently usable – prototype implementation of our design, in Haskell. Our prototype runs on Linux and maintains most file meta-data.

  We benchmark our prototype on real-life data and compare it to other backup systems.

- A record of our experiences implementing a prototype of our design in Haskell. We started out basing the prototype on the Haskell Enumerator library [66] (which is based on work by Kiselyov [56]). However we gave up on that idea, and wrote a small library for programming with processes using message passing instead. See section 4.2.1 on page 57 and section 4.8 on page 75.

1.1 Motivation

Several recent surveys show that few people take regular backups, and most people have lost digital data on one or more occasions [55, 81]. Yet the majority of the surveyed people are aware of the importance of regular backups.

In this thesis, we describe the design and implementation of a prototype backup system that we call Hindsight: secure, flexible and efficient. We have found ourselves annoyed with the available selection of backup systems, and this has been our main motivation.

Many commercial backup systems are bound to a specific – often proprietary – back-end and are closed source. That is a problem because *a)* different people have different needs, possibly also depending on the data; *b)* it promotes vendor lock-in; *c)* it is hard to assess the reliability and performance of an essentially unknown back-end; and *d)* without resorting to reverse engineering (which might even be illegal) it is often impossible to tell if data is stored responsibly. Many companies are quite vague when describing their security model, and even general functioning of their products.

Further, the open systems we have seen, all have shortcomings. We discuss a selection of systems and their properties in section 5.1 on page 78.

Thus it is important to us that our system is as flexible as possible with regard to back-end and that it is transparent, both in terms of functionality and security model.

Of course, we also want the system to be practical and usable. We elaborate on what exactly we mean by this in section 2.2 on page 12, but from the outset we felt that performance is a big part of it. Thus we devised the following simple "test" as a guideline during early development. While maybe not common, we want to be able to handle the following use case:

> The user takes a daily snapshot of her home directory. The directory contains about 1'000 gigabytes data (1 terabyte) spread over 500.000 files. There is at least one several gigabytes large file of which a small part changes often (e.g. a database or a virtual machine image). Some files, such as program caches or temporary files, change very often and may be written or deleted during a snapshot.
>
> We assume that the backup system runs in an unreliable environment, such as a laptop; the Internet connection may drop out, the battery may run out, or an impatient user may kill the system during a snapshot.

This scenario has helped us quickly gain an intuition about the feasibility of an idea; being it a solution to a problem or a new feature.

1.2 Scope

The use case we mentioned above should give some pointers to the scope of this project. Our main focus is the design and implementation of a backup system for personal computers.

Though the design focuses on general key-value storage, we do not aim to provide a general programming framework, ready for use for backups by other applications. Our prototype is rather a proof of concept which can act as a building block or guideline for such a framework.

While definitely important, cryptography is not a subject that we allot much time to discuss. Our approach in that regard is one of less-is-more: The simpler the model, the more likely we are to get it right. We do however discuss how the implementation could be extended to follow a more flexible model (see section 3.8 on page 47).

Our design as described and implemented does not support the sharing of data between different users. We discuss ways to include the feature, but it is not fundamental to our design.

The Hindsight prototype acts as a proof of the feasibility of our design, and hence there is a number of things that we have deemed less important.

In particular we:

1. only support Linux,
2. only provide a simple CLI interface, along with a FUSE-based mount tool,
3. only support standard Unix file permissions and ownership (e.g. not extended attributes),
4. are yet to conduct an in-depth source code review,
5. have not performed user tests,

6. do not micro-optimise for speed (in particular there appears to be a memory leak in our prototype, but this could just be due to the massive memory overhead of the b-trees; see section 6.5 on page 90 for details).

1.3 Background

Traditionally, backup systems (or tools) were designed to access the smallest amount of storage during a backup. They were designed to work well for tape backups, which not long ago was the only affordable mass storage available. Such tools (e.g Unix dump and cpio) write filesystem snapshots as a single stream of data to tape. During a backup old snapshots are not accessed, allowing old tapes to be archived elsewhere.

Snapshots can be *full* or *incremental*. A full snapshot is a self-contained copy of the filesystem, whereas an incremental one only consists of the files changed since the last snapshot. An incremental snapshot is based on earlier snapshots in that only files which have changed are backed up. When files need to be restored from an incremental snapshot, all the snapshots back to the latest full snapshot are needed. To deal with this chain of dependencies, a full snapshot is taken reguarly (e.g. once every weekend).

Amanda [36] is a backup system building on tools such as dump and tar.

The decreased cost of disk space has allowed systems to base new snapshots on file data (as opposed to just whether or not files have changed) of prior snapshots, which would be impractical using tapes. Duplicity [42] is such a system. When taking incremental snapshots, Duplicity will identify changed files and only save the parts where data has changed. This is fundamentally different from storing the files in full, because it requires access to the contents of the file's prior version. The process of identifying parts which are equal and only storing them once is called *deduplication*; see section 3.5 on page 28 for details.

While this approach yields more compact backups, it also suffers from the drawback that these backups cannot be read as-is since file parts are scattered.

A problem with Duplicity (and similar systems such as rsnapshot [83] and Apple Time Machine [5]) is that deduplication is not applied across files; if one file has much in common with (or is indeed the same as) another within the same snapshot, they will still be stored separately.

A solution is to look past files, and instead focus on data blocks. A file is then represented as a reference to the blocks it consists of. Applying deduplication to the data blocks now, gives deduplication within and across files "for free". Venti [77], which ships with Plan 9 from Bell Labs [76], is a system that stores data blocks, with deduplication. A front-end to Venti would traverse the filesystem, divide files into data blocks (in this report called *chunks*) and store those blocks through the Venti back-end. A snapshot would then be a collection of files with references to their chunks.

Since Duplicity keeps no log data, it is possible to rebuild the backed up data solely from the raw data stored on the back-end. With Venti, the log data

(i.e. the collection of files and references) is needed to give structure to the raw data. This makes checking out a snapshot's data more complicated.

Another negative side-effect of deduplication on a data block level is that deletion is difficult to implement. As a result, Venti does not support it. The problem arises because file chunks can be shared across multiple snapshots, and thus it is difficult to tell who references a given chunk. A solution to the problem should be found among garbage collection techniques; see section 3.6 on page 33 for details.

Cumulus [93] is an example of a system that *does* support deletion and deduplication (though not across files). Cumulus takes only full snapshots and uses deduplication on the log data to avoid an overhead, linear in the number of snapshots.

A technique referred to as *convergent encryption* [40] tries to combine safety with deduplication, by always encrypting the same data block with the same key. This can be useful in systems that handle data from different users. We discuss the consequences of this in section 3.8 on page 47 and revisit it as a topic of further work; see chapter 6 on page 87.

Pastiche [34] is a peer-to-peer backup system that uses convergent encryption [40] to enable deduplication of file blocks across multiple users. To furhter improve deduplication in a peer-to-peer system, each peer uses file *abstracts* to discover "buddies" with overlapping data. The local state consists of an append log, that maintains file meta-data and their block descriptors for each snapshot.

Hindsight goes one step further. It takes full snapshots and does data block deduplication across all files (in all snapshots), as well as log data. The size of local log data is linear in the size of unique data backed up. In our prototype, deletion is supported through conservative garbage collection, but we discuss other methods as well.

So to recap, we have described the following approaches:

Amanda mixes incremental snapshots, where changed files are saved in full, with full backups. Uses regular encryption.

Duplicity improves incremental snapshots, by only storing the changed parts of files. Full snapshots are still supported. Uses regular encryption.

Venti supports block-based deduplication among all data blocks. A front-end is responsible for applying deduplication to the log data.

Cumulus uses block-based deduplication among blocks belonging to versions of the same file. Log data is de-duplicated. Uses regular encryption.

Pastiche uses convergent encryption to de-duplicate data across users, while storing data safely.

Hindsight uses block-based deduplication among all data blocks, and deduplicates log data as well. Deletion is supported. Uses regular encryption.

1.4 Overview

The rest of the report is divided into the following chapters.

Analysis The problem of making a backup system is analysed. First, we clarify what we mean by a backup system. Secondly we list relevant properties. We conclude with a summery of how properties affect each other.

Design First we give an overview of the design of our system and summarise the terminology we use. We proceed with a more detailed description of the central data structure of our system: indices modelled by external b-trees. We give running times for a few search queries.

Four largely self contained sections follow. They are concerned with deduplication, deletion and garbage collection, crash safety and security.

The chapter concludes with a discussion of the storage back-end, the storage format and a summary.

Implementation We present our prototype, along with cherry-picked parts of the implementation, the problems we discovered during development and our solutions to them.

We begin by giving instructions for installing the prototype. Then we go into details of the implementation. Rather large sections are allotted to describing the process model we use, our b-trees and crash safety.

Somewhat shorter sections are concerned with back-end modules, deletion and security. We conclude with a short record of our experiences with the Haskell iteratee and enumerator packages, and how we arrived at the Erlang style process model we currently use.

Evaluation We start by giving a short record of the development of the design and our prototype.

Next we give a rather comprehensive comparison of backup systems based on a selection of properties. We briefly discuss each system.

We assess the quality of our prototype and describe the tests we have performed. Then we give a summary of our benchmarks. The raw benchmark results can be found in appendix A on page 97.

Further work We discuss areas that either need attention before the prototype could be called a mature product, as well as interesting topics for future study.

Conclusion We summarise the main results and conclude.

Chapter 2

Analysis

In this section, we discuss what a backup system is and what our expectations to such a system are.

2.1 Functionality

From a backup system, we expect the following functionality.

Storage: The ability to store the current state of the data, including meta-data and hierarchical structure. We refer to such state as *snapshots*. It should be possible to store multiple snapshots side by side without interfering with existing snapshots. A backup system should provide mechanisms for decreasing storage requirements when storing similar snapshots.

Inspection: After storage, one must be able to restore a snapshot. For practical purposes, it should be possible to limit the retrieval to the needed structure within a snapshot.

For example, with a file system backup tool it should be possible to retrieve the small directory `diku/speciale` without retrieving the much larger `downloads`.

The system should also limit the retrieval of structural information. In the example above, the system should not retrieve a directory listing for `downloads`, when inspection `diku/speciale`.

Deletion: It should be possible to delete old snapshots that are no longer needed, thereby reclaiming storage. Preferably, it should be possible to modify an existing snapshot by deleting only parts of it. And of course without interfering with other snapshots.

2.2 Properties

In addition to the basic functionality listed above, there are some properties relevant to backup systems. In this section, we present and discuss those properties.

2.2.1 Flexibility

Front-end: How easy it is to implement a new front-end. For example, is it possible to port the system to another OS without changing its core components?

Back-end: Which back-end one can choose from. Having multiple back-ends to choose from prevents vendor lock-in and gives freedom to pick what fits ones needs: the cheapest; the most reliable; or perhaps the most transparent.

Modularity: How easy it is to change parts of the system or move logic from the client to the server and vice versa.

There are two ways to ensure a broad support of back-ends, either *a)* select the back-ends to support and implement the required logic directly in the system; or *b)* provide the means to extend the system to support new back-ends.

2.2.2 Performance

The performance of a backup system must be good enough for it to practical. This applies to all functionality and operations the backup system can handle. We will consider the following performance measures:

Local space usage: The amount of local storage needed by the system for temporary and permanent data.

Remote space usage: The amount of remote storage needed by the system.

Time usage: The time the backup system needs, in terms of: computations, disk IO and network communication.

Transfer: The amount of data that needs to travel over the network during an operation. Here, we think of the actual bytes that need to be transferred. This influences time usage.

Scalability

Scalability goes hand in hand with performance. If a scenario forces the system into either an expensive computation or a large amount of data transfer, the system does not scale well with regard to that scenario. Typically, a *direction* can be identified and it is possible to describe how well the system scales in that direction. For filesystem backup we have identified the following directions.

1. The number of files in a snapshot – filesystems can grow very large: `ext4` [63] supports up to 4 billion files.
2. The amount of changed data in a snapshot compared to prior snapshots (changed and deleted files) – usually, only a few files change regularly, but occasionally a directory containing a huge project is moved or deleted, and a lot of files are updated as a result.

3. The total amount of data used by a snapshot; `ext4` supports volume sizes of up to 2^{60} bytes.

4. The size of an individual file – typically, files are small but then there are virtual machines and databases. Most files are small, but most bytes belong to large files [39, 65]. The maximum file size supported by `ext4` is $16 \cdot 2^{40}$ bytes.

5. The length of file paths. `ext4` supports file names up to 256 characters. Combine this with a deep directory structure and you may end up with file paths of several thousands characters.

6. The amount of meta-data of individual files (permissions, timestamps etc.). `ext4` limits the inode size to 256 bytes.

A general purpose backup system should scale in every thinkable direction to allow acceptable performance throughout all use cases. However, it is more likely that a backup system chooses a handful of these to focus on.

2.2.3 Reliability

For the backup system to work in practice it must be reliable. Obviously, you would want the software to do its work, but what about those special cases where something fails? What if the system crashes, the network is down, or a programming error simply breaks something?

As a minimum, we want the software to be able to recover the stored data in case of a disk crash. Software crashes should be handled without state corruption and preferably without hurting performance.

2.2.4 Transparency

It is good to know whether your backup tool is working correctly or not. One way to determine this is to follow the process of a backup and inspect the data stored from it. The more you know, the easier it is to determine whether the system does as it claims:

Black box inspection: Whether one can access and list data stored on the back-end at all. This provides some transparency as to how much data is stored and what it looks like, though the data itself may not be understandable. It further allows the user to look for traces of the original files; something that would not be expected if the backup system claims to be secure.

White box inspection: Whether one can understand the data stored on the back-end. If the files being backed up are stored in an easily accessible format on the back-end, the user will not have to rely on the backup system to restore them.

Open source: If the backup system is open source software, one does not need to take the author's word for its functionality. Here, one has the

opportunity to read the code or at least consult others, and hear what they have to say about it.

2.2.5 Security

Depending on which entities are considered trusted, it can be beneficial to apply some means of security:

Authentication: To detect intentional (or unintentional, e.g., bit-rot) modifications by the back-end, the data can be cryptographically authenticated. This procedure can be combined with privacy protection.

Privacy: The back-end can be shielded off from knowing the data stored as a protection against privacy intrusion. This can be achieved by a layer of encryption. Likewise, if the back-end is trusted, the connection from the client to the back-end can be encrypted to prevent other parties from eavesdropping.

In a multi-user system, it can be a concern whether other users can access or deduce information about the system's local state or work in progress.

Access Control: The ability to control access to the stored data, both fully and partially. This includes read, write and delete permissions as well as permission revocation.

Denial of Service: In a denial of service attack a malicious party can prevent the system from running or completing a backup. This could for example be carried out by a malicious user whose data is being backed up by the system administrator. Examples include system crashes on strange file names or tricking the system into processing an infinite special file (like `/dev/zero`).

2.3 Cost-benefit

It may not be possible to get all the wanted properties listed above, since some of them counteract each other.

Flexibility in support for back-ends influences everything. If the system ditches flexibility and moves logic to the back-end, it could gain better performance and reliability. However, unless the back-end is trusted, this would likely hurt transparency and security. More flexibility leads towards transparency: Being able to freely choose a back-end makes it easier to know what the system is doing.

Performance is hurt by all other properties. When new logic is introduced, performance is likely to suffer from it. On the other hand, the new logic may be a trade-off that hurts performance in one area, while improving it in another (e.g. time and space).

Reliability may hurt performance, by maintaining extra state or adding extra logic to survive a failure. But when the failure comes, this extra logic can make the system more efficient at recovering and cleaning up.

Security is bad for everything. It does not bring anything beneficial to the table except security itself. This could explain why some systems choose to skip this property completely. Others relax the property with methods such as convergent encryption [1, 3].

Transparency may hurt performance and security, by limiting the freedom to restructure and transform the stored data, when white box inspection is required.

In any case, compromise is inevitable.

Chapter 3

Design

When designing Hindsight we had the following keywords in mind: *flexible*, *secure* and *efficient*. In this chapter we describe the design of Hindsight, and discuss the choices and trade-offs that we have made.

These were our expectations of the Hindsight backup system, before we designed it and implemented it as the prototype:

1. Secure by default: Protects privacy and integrity.
2. Efficient:

   Small resource footprint: Low usage of CPU and memory.

   Small local space footprint: Low amount of local log data: at most 1% of the data it represents after deduplication.

   Low external space usage: Compression combined with global deduplication across all snapshots.

   When transferring: Parallel transfer and limited amount of back-end requests.

   During inspection: Only retrieve the needed data and log data.

   Crashes: Recover and reuse as much of the progress as possible.

   Scalability: Must handle many files and lots of data in tons of snapshots.
3. Transparent:

   In security: Well-described and simple security model.

   Open source[1]**:** The user can review the software herself.

   Local back-end: Makes it possible to inspect the stored data.
4. Flexible:

   Back-end: Easy to implement support for new back-ends as well as running with a local back-end for inspection.

[1] The Hindsight prototype is licensed under GPLv3

High-level API: An abstraction in form of a high-level API makes it possible to reuse the Hindsight backup system with other clients.

In the rest of this chapter, we will discuss how our design works and how it meets these goals. In chapter 4 on page 55, we discuss how we have tried to make our prototype implementation meet the design. It does not accomplish all the goals: it does not expose a high-level API (so far it just implements a traditional disk backup system) and its resource footprint is by no means *small*.

Outline. Our prototype implementation focuses on backup from a local filesystem; It can traverse a directory structure and store files either locally or remotely through a choice of several *back-end modules* (e.g. SSH or Amazon S3).

Figure 3.1 shows the three main components of the prototype. Each component knows about the one to its right, and the back-end module is custom tailored to the back-end of choice.

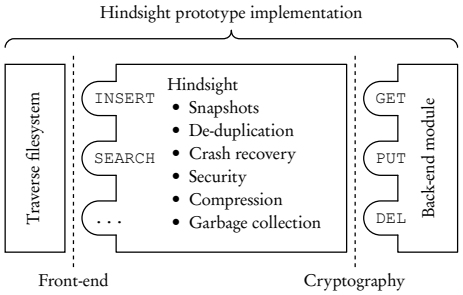

Figure 3.1: The main components of the Hindsight prototype.

The center component comprises our design and is what we call Hindsight. It is responsible for all the bookkeeping in connection with snapshots, deduplication and garbage collection. We will refer to the data used for book-keeping as *log data*. In addition, Hindsight is responsible for security such that no private data is sent to the back-end module.

We discuss the APIs of Hindsight and the back-end module in the next section and in section 3.9 on page 51 respectively.

The choice of a filesystem backup tool for our prototype implementation is quite natural because Hindsight needs such a tool to function. In section 3.2.2 on page 23 we describe how the system uses itself to save log data.

3.1 API

At the heart, Hindsight maps *keys* to *values*. Several such mappings can exist side-by-side. They are mutable and are handled in isolation by Hindsight (modifying one does not affect the others).

These mappings reside in the Hindsight *repository*, which is a directory on the system where Hindsight runs. Each mapping can be used to represent data being backed up. For example, the mapping named "home" could contain the file paths from `/home` as keys with their data as values, and the mapping named "etc" the file paths from `/etc` and so forth.

At any time a *snapshot* of a mapping can be taken. This copies the mapping's current state and captures it as it looked at that specific time. The new copy is partially frozen in that only deletion is allowed, and is hereafter stored on the back-end. The copy is named after its mapping, but with an additional version number.

We call the mutable mapping, or the in-progress snapshot, the *head*. We call a head and the frozen snapshots generated from it a *family*.

So to recap: In Hindsight, each name covers a list of earlier *snapshots*, and an in-progress one called *head*. Together they are called a *family*, and are located within a *repository*.

Next, we give an outline of the API of Hindsight. This is the exposed high-level API that is used by a front-end to manage a repository. The API is divided into three categories depending on the scope of the operations:

1. Head operations work on the head of a family.
2. Snapshot operations work on a specific snapshot within a family.
3. Global operations work on the full repository, and not on just one snapshot.

3.1.1 Head operations

`INSERT` $(\boldsymbol{key}, \boldsymbol{value}, \boldsymbol{meta}, \boldsymbol{version})$ Establishes the mapping

$$key \mapsto (value, meta, version)$$

in the head. A *version* is used to quickly discover if the mapping already exists, in which case it will be ignored. Thus it is important that the version changes whenever *meta* or *value* changes. Alternatively, the version field can be omitted, in which case the mapping will be inserted with no checking.

The *meta*-data is stored directly with its key, which makes retrieval fast. It should be kept small ($\approx$ 4KB) since many such values needs to fit in memory. Our prototype places file permissions, modification times, etc. in the *meta* field.

`DELETE` ***key*** Removes a key (if it exists) and its associated data from the head.

`SNAPSHOT` Takes a snapshot of the head, without changing it.

3.1.2 Snapshot operations

SEARCH *f* This operation gives a general means of searching for *key*s in a snapshot. The operation must be supplied with a function *f*:

```
f : (key, key) -> bool
```

Presented with two keys, a and b, the function must answer the question "Could there be any relevant keys in the interval $[a;b]$?". Using different functions, we can construct a range of different pseudo operations, of which a few are listed below. An in-depth discussion of how searching works is given in section 3.4.1 on page 26.

RETRIEVE ***key*** Retrieve from the snapshot, the value and meta-data associated with a specific key.

DELETE ***key*** Removes the a key (if it exists) and its associated data from the snapshot.

DISCARD Removes the whole snapshot. This not only removes every key in the snapshot, but also the snapshot itself.

In addition, our prototype supports the following *pseudo* snapshot operations. These are pseudo operations in the sense that they can be created by combining `SEARCH` and `RETRIEVE`.

Pseudo operations

LIST ***prefix*** Takes a key-prefix (which can be empty) and returns every key in the snapshot with the given prefix.

LISTDIR ***dir*** Regards the keys as filepaths. Returns all paths in a given directory, but not in subdirectories.

CHECKOUT ***prefix*** Takes a key prefix and retrieves the data and meta-data for every key in the snapshot with the given prefix. A variant of this operation which ignores subdirectories also exists.

3.1.3 Global operations

INIT Creates a new encryption key. This only needs to be done once. See section 3.8 on page 47 for details.

LIST Returns a list of every snapshot in the system.

SEAL Saves all log data on the back-end. This makes it possible to restore the system in case of data loss (e.g. disk failure). See section 3.2.2 on page 23 for details.

3.2 Overview

In this section we give an overview of the design of Hindsight, in order to give a context in which the rest of the chapter can be read.

As mentioned in the previous section, Hindsight supports multiple named families of snapshots. Global deduplication on a sub-file level is applied across all snapshots. See section 3.5 on page 28 for details on deduplication.

By insisting on these features, some parts of the design are already given. In particular the following must exist.

- An index of snapshots, called the *snapshot index*.
- For each snapshot an index listing the files with references to their contents is needed. We call these indices *key indices* because Hindsight to some extent resembles a key-value store.
- An index of all known data blocks, called the *hash index*. We use this name because data blocks are referenced by their fingerprint which is a cryptographic hash.

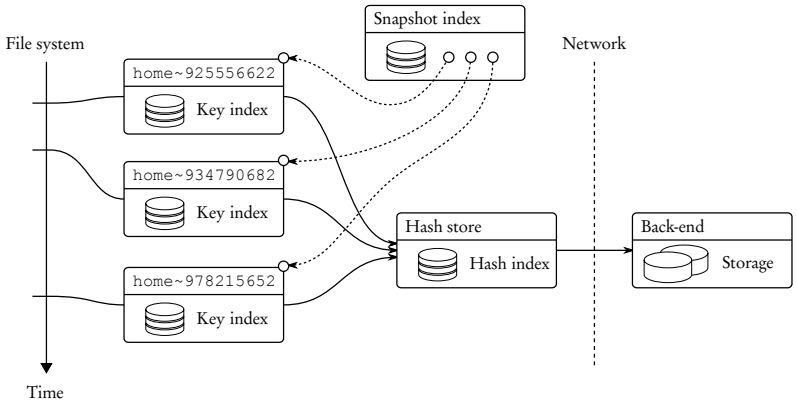

Figure 3.2: Hindsight's main components.

These components are shown in figure 3.2. The vertical line on the left represents the file system over time. Each snapshot represents a time slice of the file system. A snapshot is uniquely identified by a timestamp and a user-supplied family name.

The references from the snapshot index to the key indices are dashed because of an indirection. This is necessary to support deduplication of log data. We return to this in section 3.2.2 on page 23.

Data is de-duplicated on a sub-file level. This means that files are divided into several chunks which are then de-duplicated. In the key indices, each file path is mapped to a list of references to the chunks that the file has been divided into.

The fingerprint of each chunk (its hash) and where the chunk is saved on the back-end is saved in the hash index. This allows Hindsight to perform *online* deduplication, where data chunks are de-duplicated as they are saved (as opposed to some point later).

Before they are transferred to the back-end, chunks are collected into blobs. The blobs are encrypted and authenticated (see section 3.8 on page 47 for details on security) before entering the network.

3.2.1 Taking a snapshot

Let us focus our attention on the snapshot process for a minute. Our prototype traverses the filesystem and inserts each file in the head of the chosen family (the API was discussed on page 18).

Figure 3.3 on the next page shows how Hindsight stores[2] files.

To the left three files (`foo`, `bar` and `baz`) are shown. Each file consists of meta-data (represented by the circles to the left) and data which is divided into chunks. We have given the chunks numbers to make them easier to track through the diagram.

For each file the insertion $(key, version, meta, value)$ is constructed by letting *key* be the file's path, *version* its inode change time, *meta* its meta-data and *value* its contents. If the file is already present with the same version, then it will be skipped.

The leftmost vertical line represents the mapping given by the key index. The meta-data is stored directly in the index (there are several good reasons to do that; see section 3.10 on page 52), but only the fingerprints of the chunks are stored.

The next vertical line represents the mapping given by the hash index. Each chunk's fingerprint is looked up in the hash index, and if it is determined that it has already been stored by another snapshot (the shaded chunks and dashed lines) nothing else happens.

If, however, the chunk is new it is put into a blob and its location (which blob and where in it) is recorded in the hash index. Finally, when a blob is full (our prototype uses blobs of up to 2MB, but this number is configurable) it is compressed, encrypted and authenticated before being saved on the back-end. Blobs are identified by an ID which is randomly generated when a new blob is created (our prototype uses 128bit numbers[3] for the IDs).

The fact that the same hash index is used for every file in every snapshot is what gives *global* de-duplicity. Also notice that data is de-duplicated within the same snapshot; in the diagram some chunks have the same fingerprint (they point to the same place on the line representing the hash index). Of course this is also true for chunks within the same file.

We have deliberately omitted some details of the process to ease the reading of this overview. In particular, care has to be taken in order to make the system

[2] Or as Brackup [2] puts it "slices, dices, encrypts and sprays across the net."

[3] Generated by AES in counter mode.

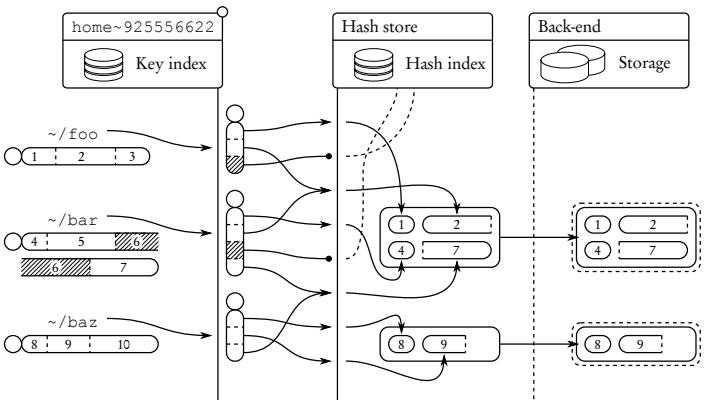

Figure 3.3: How data is stored with Hindsight.

resistant to crashes. We will revisit the process of taking a snapshotting in further detail in section 3.7 on page 43.

3.2.2 Preserving log data

Now that all the user data has been saved, we need to save the log data generated by Hindsight during this process. Specifically, we need to store the new indices that represent the snapshots and the updated hash index.

As we will see in section 3.4 on page 25 our indices consist of multiple small files. This enables us to reuse the system itself to save its own state. We do this using what we call the *secondary* run. Here, the Hindsight system is reused to snapshot the log data from the *primary* run (that was used to snapshot user data). The result is a new pair of indices that we call the *secondary* indices. These constitute the log data of the *secondary* run.

In short, the system runs twice: once to snapshot user data and then again to snapshot the log data from the first run.

This process is illustrated in figure 3.4 on the next page. On the left side of the figure, we have the primary indices referencing user data. From these, we use Hindsight to generate the secondary indices that reference the primary indices. These are small enough to simply store them as tarballs on the back-end, along with their secondary hash index. The tarballs are randomly named and the names are stored in the snapshot index. Note that this hash index is used to reduce redundancy in log data across snapshots.

All the stored tarballs are referenced by the snapshot index, and now we see why we needed that extra indirection: instead of referencing snapshots directly by their key index, they are referenced through their secondary key index. Lastly the snapshot index is saved as a tarball under the name `snapshots`, which is the only non-random name on the back-end.

Since the hash index is global and used across all snapshots, we do not have to store it right away. Instead, we delay this action, and split the saving of log data into two phases:

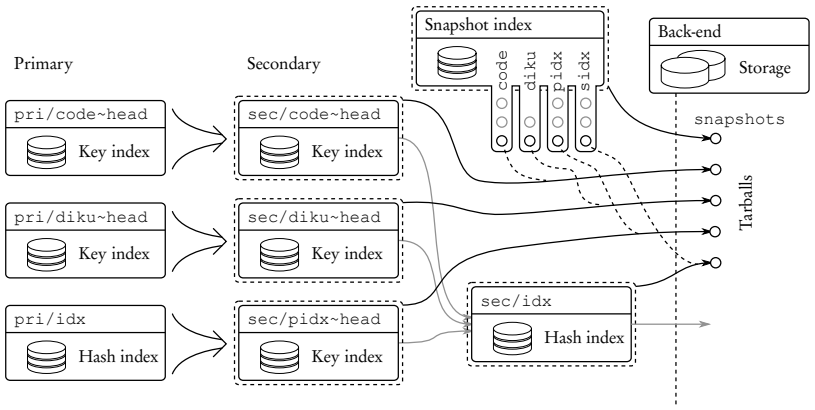

Figure 3.4: Preserving log data

1. Right after a snapshot has completed, its key index is saved. Meanwhile, the global hash index remains local. This allows the system to recover files as long as the internal state is not corrupted.

2. At some point, the global hash index and the snapshot index are saved. The system can now withstand system and hardware crashes by recovering all internal state from the back-end.

This makes it possible to complete several snapshots before committing the updated hash index, which makes sense for example when using isolated snapshots, say one per user on the system. Here, the hash index could be committed after snapshotting all the user home directories.

3.3 Terminology

We give a short recap of the terminology we use:

Chunks: A part of a file. Files can be divided into roughly evenly sized chunks to promote deduplication effectiveness.

Blobs: A collection of chunks.

Log data: Data kept for bookkeeping. In Hindsight's case the local log data consists of

- Hash indices, one primary and one secondary.
- Key indices, two for each head and snapshot, one primary and one secondary. Additionally one for the primary hash index.
- Snapshot index.

Of the key indices, only the ones belonging to the heads are normally stored locally (the others might be stored in a cache, but is not needed for normal operation). Due to deduplication of the primary indices the space usage is not linear.

The primary and secondary indices are generated by two isolated instances of the system, and so a discussion often applies to either. In those cases we will ommit "primary" and "secondary".

When we just write "the key index" we refer to the key index of some specific head, given by the context.

Hash, chunk hash: The fingerprint of a chunk (see section 3.5 on page 28).

Snapshot: A frozen mapping from keys to values, from some point in the past. A snapshot belongs to a family.

Head: The current mutable mapping from keys to values. Each family has exactly one head (which can be empty).

Family: A list of snapshots and a head. A family has a name chosen by the user.

Repository: A collection of families.

3.4 Indices

Hindsight maintains three kinds of indices as its state (the key, hash and snapshot indices). These represent its world view and are updated periodically (using the `snapshot` and `seal` commands, see section 4.1.2 on page 56).

Snapshots: The snapshot index lists all snapshots. Each snapshot is stored with a creation time stamp and a reference to its key index (next). With this index, Hindsight can efficiently find a named snapshot taken at a specific point in time.

Keys: Maintains all file paths with their meta-data and references to their respective data chunks. The references are hashes that reference blobs through entries in the hash index (next). Hindsight uses this index during checkout and searching. When a new key index is to be built and used for a snapshot, the prior version of the index is used as a starting point (head); thus allowing efficient skipping of unchanged files.

Hashes: Maps hashes of data chunks to their respective blob IDs and offsets. This index is used for deduplication, and works as an indirection between files and their data chunks.

Additionally, this index can be extended with extra data needed for garbage collection, such as the list of snapshots referencing each chunk (we discuss reference lists in section 3.6 on page 33).

To ease comprehension, we have left out the indirection between snapshots and their respective key indices (through the secondary indices). This was discussed in detail in section 3.2.2 on page 23.

The data structure used for indices is an external b-tree [33] (see section 4.3.6 on page 63 for implementation details). The main advantage of

using a b-tree is that the tree structure – unlike e.g. a SQLite database – splits naturally into several files. This allows querying against the index, residing on a back-end, only retrieving relevant parts.

This goes for updating as well: a small change in the index gives small changes in the b-tree node files, which again gives us a means for eliminating redundancy across snapshot log data.

3.4.1 Search

For each index, we want to support searching, and in the key index we want to enable inspection of the hierarchical structure reflected in its keys (e.g. filesystem directory structure). At the same time, we want to allow our indices to perform these queries efficiently.

As a compromise, we use a search API that limits the searching operation in a way that allows the b-tree implementation to execute the search efficiently, while remaining agnostic towards the actual query:

`SEARCH` f The operation is given a function `f`, that defines the query:

```
f : (key, key) → bool
```

For a pair of keys (a,b), f answers the question "Could there be any relevant keys in the closed interval $[a;b]$?", with `true` meaning "Yes", and `false` "No". Only keys that can be located using `f` and for which

```
f(k, k) ⇝ true
```

are returned (the interval $[k;k]$ contains a single element, k).

By describing the query as a predicate on intervals, the b-tree indices can cut away irrelevant sub-trees where the predicate cannot be satisfied, directly in the branch nodes. Thus giving us efficient queries.

Here we show example uses of the search API that can be used to express various interesting queries (we make use of *curried* functions, as known from Standard ML and Haskell). We assume an alphanumeric ordering of keys, but the exact details are defined by the implementation[4]. We give worst case running times as a function of the total number of keys t, and the number of returned keys r:

$O(t)$ Filter all keys according to a predicate function p:

```
filter : (k → bool) → (k, k) → bool
filter p (lb, ub) = lb <> ub orelse p lb
```

Example:

```
hsFiles k = isSuffix ".hs" k
let files = search (filter hsFiles)
```

[4]Further note that in the implementation, the search function returns a monadic value. However, f is pure as in the examples.

$O(\log t)$

Lookup a specific key k:

```
lookup : k → (k, k) → bool
lookup k (lb, ub) = lb <= k andalso k <= ub
```

Example:

```
case search (lookup "thesis/code/Backup.hs") of
  []  -> Nothing
  [x] -> Just x
```

$O(\log t + r)$

We give an argument for this running time in a moment.

Find all keys sharing a chosen prefix s:

```
prefix : k → (k, k) → bool
prefix s (lb, ub) =
  (lb <= s andalso s <= ub)
 orelse isPrefix s lb orelse isPrefix s ub
```

Example:

```
(* Emulate Unix "ls --recursive" *)
let folder_ls_r = search (prefix "thesis/code")
```

$O(\log t + r)$

The number of inspected tree nodes never exceeds that of `prefix`.

Find the contents of a directory *dir* without recursing into subdirectories:

```
listdir dir (lb, ub) =
  (* key is file in dir *)
  (lb = ub andalso isPrefix dir lb
          andalso not (isSubstring "/" (stripPrefix dir lb)))
  (* dir is between lower bound and upper bound *)
 orelse (lb <= dir andalso dir <= ub)
  (* range includes dir: cut if lb and ub are in same subdir *)
 orelse (isPrefix dir lb andalso
         not (isPrefix dir ub andalso
              getSubDir dir lb = getSubDir dir ub))
```

Example:

```
(* Emulate Unix "ls" *)
let folder_ls = search (listdir "thesis/code")
```

Running time of `prefix`: Keys sharing a common prefix always inhabits contiguous leaf nodes. Figure 3.5 on the next page shows a small (eventually balanced) b-tree. The two leaves placed on a gray background share the chosen prefix.

The green nodes are the nodes that must be inspected. In a b-tree each branch node holds lower and upper bounds of the keys in each of its sub-trees. Therefore it can immediately be determined if the subtrees under a branch node needs to be inspected further. The sub-trees that do *not* need further inspection are emphasised with a dashed line.

Below the tree is the full range of keys; the parts marked with red are the parts that are cut off during the search.

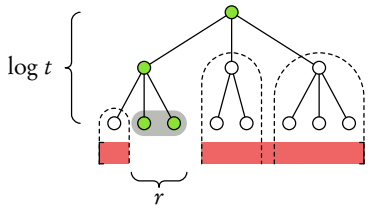

Figure 3.5: Searching for keys with a chosen prefix.

In this example we were lucky that the interesting leaves both resided in the same sub-tree. But that might not always be the case; in the worst case the leaves are placed in different sub-trees of the root node, adding roughly another $\log t$ to the running time. But to include a third sub-tree in the search there must be enough leaves of interest to populate a full sub-tree, in which case the r is the dominating term. Therefore we conclude that the running time is $O(\log t + r)$.

For details on our implementation of b-trees see section 4.3.6 on page 63.

3.5 Deduplication

Deduplication is the process of discovering and eliminating duplicated data. The goal is to save space by only storing each data object once. In backup systems, objects are usually either whole files or individual file chunks.

A short and to-the-point introduction to deduplication is given in [46]; a more in-depth discussion is given in [87].

3.5.1 Online and offline deduplication

Deduplication can work in an *online* or *offline* fashion (also referred to as "inline" and "postprocessing" deduplication, respectively).

Online: With online deduplication whether an object is a duplicate or not is determined at the moment it arrives, and it is only saved if it is not. This has the advantage that duplicated objects are never stored, at the cost of the need to identify them immediately.

Since the deduplication mechanism is in the data path, throughput can be hindered by it. On the other hand, if the rest of the data path is slow (e.g. disk or network), then the removing of duplicates early can improve throughput.

Offline: With offline deduplication objects are saved whether duplicates or not. Another process then (continuously or at timed intervals) looks through all the stored objects and performs deduplication on them.

The advantage of this is that throughput is not lowered by deduplication. However, deduplication must happen at some point so the higher throughput

cannot be held up indefinitely. Further, if the data path is slow, the larger amount of data that needs to be transferred can hurt performance.

3.5.2 Local vs. global (vs. universal) deduplication

The amount of data objects considered naturally affects the effectiveness of deduplication. The more objects taken into consideration the larger is the probability that the next one will be found to be a duplicate.

Local: Only a subset of the known objects are taken into account. The reason for that is performance; the fewer the objects, the less the work. A common way to choose the objects to consider is based on file names. When a file is saved it is de-duplicated against objects from files of the same name (on the assumption that these files are either the same file, or an earlier version of it with relatively few changes). A similar method is used by for example rsnapshot [83]. Another method described by Broder [28] is to compute a sketch from each file and use them to find similar files.

By only considering a subset of the known objects, not all duplicates can be identified, and thus local deduplication is often less effective than global deduplication (next). As an example, if a file is renamed it may not be de-duplicated.

Global: All the known objects are considered for deduplication. This can be computationally infeasible if there are extremely many objects, however deduplication is certain to detect all duplicates within the system.

Universal: If the data-object storage is shared with other users, it is possible to de-duplicate data not only against one's own objects, but everyone else's as well. However, this comes at the price of lesser security (see section 3.8.4 on page 50.

3.5.3 Granularity and chunk size

Data is divided into objects. The granularity of these objects is important for the effectiveness and performance of deduplication.

If the objects are smaller then there will be more duplicates, and the effectiveness will improve – the chance of identifying equal objects will rise. On the other hand the increase in the number of objects can hurt performance.

File level: One approach is to regard each file as an object. This has the advantage of simplicity, and for typical workstation computers it still offers reasonable performance [65].

Chunking: With file level deduplication, parts of files that are common cannot be de-duplicated, and a file of which a small part changes over time will be copied in full each time it is saved. To overcome this problem each file

is split into several *chunks*.

How exactly a file is divided into chunks also determines effectiveness. Generally, smaller chunks are better[5]. But the boundaries of chunks can also be important:

If chunks have a fixed size and alignment, and data is inserted or deleted in a file, it can lead to much of its data being unaligned, thus resulting in new chunks.

3.5.4 Chunking

In the simplest chunking method a file is split into chunks of a fixed size, and a fixed alignment (if the file cannot be evenly split, the final chunk will be smaller). Consider this scenario:

> Alice has a large file which she backs up. It is split into fixed-sized chunks as seen in figure 3.6 on the following page. Now, Alice inserts her name in the beginning of the file, thus extending it. As a result, all the data is copied once more.
>
> This happens because the method is using fixed offsets. It only inspects the chunk at offset 0, not the chunk at offset 5, which happens to be known ("Alice" being five characters long).

With variable sized chunks, it is possible for them to shrink or grow such that alignment can be restored. A solution is to base the chunking boundaries on the content within a data object, rather than the position. Such techniques have earlier been used to segment data for copy detection [27].

A commonly used method presented by Manber [62] is to compute a *rolling hash* of the data, and place a boundary when this hash satisfies some predicate ([68] and [72] both consider the value of the 13 low-order bits of the hash to determine boundaries). This works by moving a window over the data – byte by byte – while updating the hash for each offset to fit the change in the window. The hash function can be varied, but for performance, it should allow the rolling hash to be updated efficiently. Examples include Rabin fingerprints [78] (used in the Low-Bandwith Network File System (LBFS) [68]), cyclic polynomials [31] and Adler checksums (used in rsync [91]).

Now Alice's scenario looks a little different:

> Almost all of her two files are the same data. With content-aware chunking, only the three first chunks differ; the rest match up. The chunking algorithm has quickly recovered to the sequence of known chunks (see figure 3.6 on the next page).

[5] Within reasonable limits; references to each chunk must be saved, and so the overhead from references should be considerably smaller than the chunks.

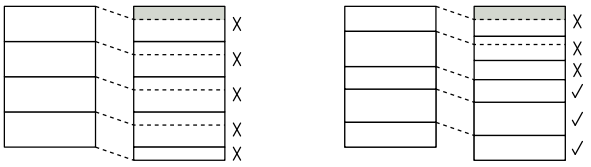

Figure 3.6: Chunking strategies

3.5.5 Hashing

Until now we have avoided the question of how data objects are compared. The simplest approach is just to compare the objects one byte at a time. But this is impractically slow.

The solution is to *fingerprint* each object, and compare fingerprints instead. But since the fingerprints are much smalller than the objects themselves there will exist collisions. If a collision occurs it will lead to data corruption because references to different data will in fact reference the same data.

This could be avoided by using fingerprints to detect *candidate* objects only, and then compare to the candidates. To perform such comparison, and test whether the found candidate was in fact the sought object, the objects would either have to be available locally, or be transferred from the back-end for comparison. This would either require a lot of local storage, or a lot of data transfer, whenever known data is being de-duplicated.

If a cryptographic hash [43] is used, the probability of collisions is so low that it is generally accepted to regard objects with the same hash – i.e. fingerprint – as equal (for further discussion see section 3.8 on page 47). This custom is opposed by Henson [52], who advises caution.

3.5.6 Improving performance

To decide if an object is a duplicate, it must be checked whether its fingerprint is known. Since deduplication systems are generally geared towards massive amounts of data, it is unlikely that an index of all known fingerprints can fit in memory [96]. Therefore at least part of the index must reside on disk.

Accessing an on-disk index is an expensive operation. If most of the index is located on disk, and no other measures are taken, performance is likely to be impaired.

Zhu et al. [96] discuss techniques for avoiding index lookups. The most important are:

Summary vector: The purpose of a summary vector is to reduce the number of times fingerprints are looked up in the index, when they don't exist; a summary vector is a conservative summary of the index in that when a fingerprint is not found in the summary, then it is definitely not in the index either.

In [96], Bloom filters [24] are used to model summary vectors. Systems that use Bloom filters for this purpose include [72, 77] and [90].

Locality preserved caching: A cache fills the opposite role of the summary vector; it provides access to fingerprints which *are* in the index, without a lookup.

A traditional cache will have a very high miss rate because fingerprints are essentially random, and thus it is impossible to predict future accesses. The solution given in [96] is to preserve the locality of fingerprints from the same file. To do this, there must exist a mapping from fingerprints to a collection of fingerprints with high locality (e.g. other fingerprints from the same file).

3.5.7 Discussion

We discuss the choices we have made with Hindsight in the light of deduplication.

Online vs. offline: We expect back-ups to be stored remotely, thus needing to be transferred over the Internet. Therefore it is reasonable to assume that the network is the weakest link in the data path.

The main reason for supporting offline deduplication would be to take some load off the client, but it would also require a somewhat intelligent back-end. This reduces flexibility and security.

For these reasons we think that online deduplication is the better choice.

Local vs. global: Several systems such as bup [72] and LBFS [68] show that global de-duplicity is computationally feasible for large systems running on regular workstations.

Furthermore, depending on which garbage collection technique is chosen (see section 3.6 on the next page), local access to a global hash index may be needed anyway.

Combining those reasons with the higher effectiveness of global deduplication, we believe that global deduplication is the better choice when possible.

While universal deduplication has obvious use cases we are hesitant to choose this method because of the requirement to lessen security (see section 3.8 on page 47). Providing universal deduplication as an opt-in is a subject of further work (see chapter 6 on page 87).

Chunking: While deduplication on file-level is reasonably effective [65] it leads to quite bad worst case scenarios. For example, the system will be near-useless for backing up large databases or virtual machines which are stored in massive files. Therefore we use chunking of files.

The choice is therefore between fixed and content-aware chunking. While content-aware chunking can do no worse than fixed chunking, it is also somewhat computationally expensive. We could not judge which would be the better trade-off and so our prototype implements both.

Perhaps somewhat surprising, fixed chunking performs almost as good as content-aware chunking in our use cases. See appendix A on page 97 for details.

Hashing: If only hashing is used for checking equality of objects there is a risk of collisions. While Henson [52] and Freeman [46] point out this issue, the risk is generally considered negligible if a cryptographic hashing algorithm is used [13, 43, 84]. See section 3.8 on page 47 for details. Furthermore, it would be impractical to check whether a collision was in fact due to a fault in the hash function. This would require retrieval of the chunk responsible for the collision for comparison. If this was the practice, one may as well store the entire data as is and perform offline deduplication.

Performance tweaks: Our prototype does not implement summary vectors, but we discuss the possibility in section 6.5 on page 90.

We have not implemented locality preserved caching in our prototype, but it does preserve locality of chunks when packing them. For that reason we don't think it will be too hard to add in later.

3.6 Deletion

We want Hindsight to be able to delete snapshots when they are no longer needed. However, deduplication complicates deletion. This is because of shared chunks between different files. When a file is deleted it must be determined whether or not other files reference the same chunks. Only the chunks that are no longer referenced by anyone – across all snapshots – can be deleted.

When a chunk is referenced by no snapshots we call it "dead". We give a general discussion of techniques for finding dead chunks, that is: garbage collection. Then we proceed to a discussion of garbage collection specifically in Hindsight.

3.6.1 Garbage collection

Garbage collection is the act of finding objects which are no longer needed (referenced), and reclaim them in order to reuse resources. The usual setting is programming languages where the objects are data structures, the ones referencing them are variables and other data structures, and the resources are memory.

This is relevant to backup systems that split files into smaller chunks and de-duplicate them. Here, files are represented by the chunks they use, and a garbage collector is needed to remove the chunks that no one uses.

There are backup systems where this is not the case. If a clever back-end is assumed, deduplication can be performed there, as we will discuss in section section 6.7 on page 93. However, in this discussion we will assume a thin back-end as discussed earlier.

In Hindsight the objects are data chunks and their hashes which are referenced by snapshots, and the resources come in terms of space on the back-end. In this section we discuss garbage collection in general and finally we give a short discussion of how it relates to Hindsight.

For brevity we use GC as short for garbage collector and garbage collection.

GC techniques can be classified by the amount of additional information stored about references. At one extreme of the scale are *tracing* methods [94], where garbage is collected periodically by tracing references and discovering live objects. No information needs to be stored with the objects themselves (except a livelihood-flag during GC).

Further up the scale is *reference counting* [8] and at the far right *reference listing* [61]. With reference counting, the number of references to each object is stored and on the far right of our scale, we have reference listing which for each object, keeps the full set of references to it.

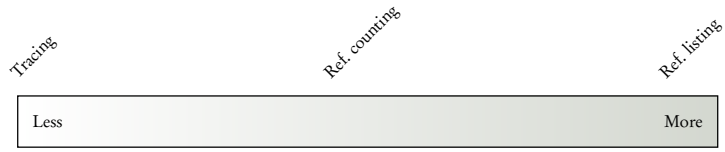

Figure 3.7: GCs classified by the amount of information they store.

The amount of extra data is quite important in our setting because the number of objects in a backup system can be huge.

GCs can be *conservative* [26] or *exact*. Conservative GCs are in contrast to exact GCs because they cannot reclaim garbage with full certainty, meaning that objects which have become garbage might not be reclaimed.

Finally GCs can be compacting or non-compacting [94]. Compacting GCs will try to rearrange live objects in order to minimise fragmentation. This can lead to higher utilisation of resources. Compacting GCs are usually tracing based because the compacting routine must either halt the system or run concurrently with it. But other GCs can be combined with a compacting routine as well.

Trace based methods

An example of a trace based method is the classic *mark-sweep* algorithm. It works in four steps:

1. Pause the system.
2. Trace all references and mark objects as they are encountered. This is called the *mark* phase. Objects are *only* marked during GC.
3. Iterate through all objects and reclaim dead ones. This is the *sweep* phase.

4. The system is restarted.

Variations of the algorithm include *mark-compact* and *copying GC* [94]. In the former, objects are rearranged to decrease fragmentation while in the latter objects are copied to a new location (known as an arena). With copying GC the sweep-phase is not needed because every object in the old arena has become garbage and therefore the whole arena can be reclaimed. A refinement to this scheme is generational GC where objects are grouped depending on their age. Younger groups are then garbage collected more often, as they tend to die faster than older objects [12].

Pros: Trace based algorithms are generally space-efficient because no extra information needs to be stored with the objects (except a live-flag during GC). Another advantage is that they are able to reclaim circular structures.

These algorithms are also fault tolerant because a trace can be aborted and restarted if anything goes wrong.

Cons: In a trace based algorithm every object is visited, and live objects are visited twice. Additionally, generated garbage is not collected until the next time the algorithm is run.

The ordinary mark-sweep algorithm needs to stop the system during garbage collection. While improvements are possible, every trace based algorithm must have a synchronisation point, and therefore will not scale well [30, chapter "Harder stuff"].

There have been numerous attempts at improving the mark-sweep algorithm to run concurrently with the rest of the system [37, 53]. But they usually bring a large overhead in terms of extra work the system has to do to work with the GC [21].

Reference counting

With reference counting the number of references to an object is stored with it. When a reference is deleted, a signal is sent to the object and the counter is decremented. When the counter reaches zero the object can be reclaimed.

Pros: Since objects get a signal when references are deleted, they can be reclaimed the moment they become garbage. A big advantage is that it is not necessary to stop the system to run the GC.

Cons: Objects participating in a circular structure will all have a positive reference count, and thus they cannot be reclaimed. The problem can be mitigated by using a cycle collector. Collectors which can run concurrently with the system exist [71], but bring an overhead none the less. Another solution is to use a hybrid GC which uses tracing for structures which can contain pointers and reference counting for objects which cannot (e.g. strings) [8].

Naïve reference counting does not work in a distributed setting where order of delivery is not guaranteed. Furthermore, each duplication or deletion sends an increment or decrement message, which can result in a large overhead. A solution to both problems is weighted references [21].

Finally, reference counting is not fault tolerant. If something goes wrong while sending a message, the message cannot simply be resent because incrementing or decrementing the counter is not an idempotent operation.

Reference listing

This technique adds fault tolerance to reference counting. Instead of saving the number of references to objects, the actual set of references is saved.

This technique shares most of its properties with reference counting.

Pros: The algorithm is fault tolerant because adding or removing an element to or from a set is an idempotent operation. Thus, like with tracing methods, garbage collection can just be restarted in the event of a fault.

Cons: A set of references naturally takes up a lot more space than a single number. This leads to a larger space overhead than reference counting.

Conservative methods

Conservative GCs reclaim garbage with some probability less than one. This allows them to have a smaller overhead or work in unusual environments. One example of the latter is the Boehm-Demers-Weiser GC [25, 26], which is a mark-sweep based GC for C and C++.

Later in this section we give an example of the former. Our method uses Bloom filters to approximate the livelihood of objects. We are not aware of any implementations using this method, but Rhea et al. [82] mention it briefly.

Pros: In the case of `malloc`/`free` replacements for C/C++ and the likes, a conservative GC is the only choice, due to pointer arithmetic.

With our method, only the Bloom filters are needed for GC, not the actual references. This allows us to store the references remotely while performing GC locally, yielding a more transfer efficient method than the classic mark-sweep (which has to retrieve all the references before the mark phase).

Cons: Objects cannot be reclaimed with certainty. This problem can be solved by coupling the method with an exact GC which runs less frequently. In that case, there is an increase in complexity from having two GCs. Additionally, some overhead is added to the remote storage, since it must store a Bloom filter for each snapshot.

Discussion

The concerns of GC in Hindsight is a little different from usual GCs:

Cyclic structures: There are no cyclic structures in Hindsight. Snapshots reference data chunks through a hash index, but this is just an indirection. A data chunk cannot reference other data chunks. Even if Merkle trees are used for this indirection (see section 6.3.1 on page 88) there can still be no cyclic structures.

Therefore the biggest disadvantage of reference counting and reference listing is a non-issue in this context.

Sweeping and compacting: If data chunks are stored separately on the back-end then no compacting is necessary, because they can be deleted individually. Thus fragmentation cannot occur.

However in order to better utilise the network connection, data chunks can be collected into blobs which are then what is stored on the back-end. When a chunk dies, its blob cannot be deleted from the back-end unless all the other chunks in it are also dead. So in this case sweeping cannot reclaim all the storage used by dead blobs.

Over time half filled (or half empty) blobs can crop up. To reclaim storage, these blobs can be retrieved and repackaged into a smaller amount of filled blobs. After repackaging the new blobs must be stored, and the old ones deleted. So compacting can reclaim all excess storage, but is an expensive operation.

One advantage of reference counting and reference listing is that no explicit marking and sweeping is needed; objects can be reclaimed the moment they become garbage. But if chunks are collected into blobs then sweeping is expensive;

It needs to be determined whether the blob to which a garbage-chunk belongs is dead. To do that one needs to find out which all chunks in the blob are dead. There are two options:

1. Maintain a reverse index mapping blobs to the chunks in them. Then look up the blob in question and check whether it contains live chunks.
2. Go through all live chunks and see if any lives in the blob in question.

Both options add an overhead in performance and complexity. For this reason it is better to just mark the dead chunks when they become garbage and postpone sweeping or compacting until more garbage can be cleaned.

3.6.2 Garbage-collection in Hindsight

In the following discussion, we consider this scenario[6]:

> Alice has 101 snapshots with a million files in each. From one snapshot to the next, 1% of the files (10.000) has changed and introduces new data. All the files are small; each consists of 1 data

[6]Inspired by the use case we presented in section 1.1 on page 7.

chunk and thus reference just 1 hash (meta-data is stored directly in the key index).

The hash index thus contains 2.010.000 entries, while each key index contains 1.000.000 entries. For illustration, we will assume 50 bytes per hash index entry (hash and blob id) and 100 bytes per key index entry (file path, meta-data and hash after compression[7]). Thus the hash index is 100,5 MB, while each key index is 100 MB.

Alice now deletes the oldest snapshot.

3.6.3 Copying GC and mark-sweep

The conceptually simplest form of garbage collection is a copying GC. In its simplest implementation, we rebuild the repository on a new back-end with the live data only. This approach retrieves all external data, with the purpose of deleting some and storing back the other. In the given example, this requires retrieval and storage of $1.000.000 + 100.000$ chunks of data.

This figure can be improved a lot with mark-sweep by only retrieving what is necessary in each phase: During the marking phase all references must be visited. References are stored in the key indices, so these need to be retrieved from the back-end. But the data itself is not needed. The amount of data that must be retrieved is decreased further by deduplication across log data (only the differences between snapshots are retrieved).

But even with deduplication the *least* amount of data to be retrieved is still $100 \cdot 10.000$ key index entries of 100 bytes each yielding 100 MB[8].

In the worst case there is no gain from deduplication, and thus all 100 snapshots (excluding the 101th which might be the head) each containing 1.000.000 entries, totalling 10.000 MB, must be retrieved.

The amount of data transfer can be lowered, by introducing an indirection between the entries in the key index and the chunks they use. This could allow us to retrieve the references of a snapshot without retrieving the rest of the key index (file paths and meta-data). This new index could be just 32 bytes per entry (an ID for the indirection and the chunk reference). In the example, this gives us a best case of 32 MB and a worst case of 3.200 MB.

But the amount of data transferred is still $O(\textit{chunks in live snapshots})$.

Pros: When deleting many snapshots and leaving few alive, the log data of those few alive only needs to be retrieved. There is no dependency on extra information in the log data in order to know which chunks are alive.

Cons: When deleting few snapshots and leaving many alive, the log data from the many live snapshots needs to be retrieved. Deduplication is required on the log data level for the method to be efficient. The same log data may be retrieved multiple times across different deletion rounds.

[7] 32 byte file path, 144 byte stat meta-data and hash – 50% compression ratio.

[8] The changes between snapshots account for 10.000 entries each. The reader might find it odd that we don't account for the "first" 1.000.000 entries, but these might be present in the head.

3.6.4 Reference counting

With a reference counting GC each chunk has a number tracking how many snapshots reference it.

When a file is stored, every new chunk it introduces starts with a reference count of one. Existing chunks have their reference count incremented and when deleting a snapshot, its chunks have their reference counters decremented. When the reference count of a chunk reaches zero, the chunk can be reclaimed or marked for reclaiming (sweeping or compacting).

The advantage is that all GC analyses can be performed solely using the global hash index, annotated with reference counts; Each hash would keep the reference count next to the chunk's blob-location.

As a disadvantage, when deleting a snapshot, we would have to retrieve its log data to decrement the counters of the chunks it references. This is comparable to the copying GC, except that here we retrieve the log data of all *dead* snapshots. Thus, the best-case and worst-case scenarios from section 3.6.3 on the preceding page are equally valid for reference counting, except that now the worst case is when only a few snapshots survive.

Extra care must be taken for this technique to be crash safe. The problem arises due to a race condition when incrementing the counters. Consider the following scenario:

> A file referencing an existing chunk is being snapshotted. Either the file is recorded in the snapshot before incrementing the reference counter, or the reference counter is incremented before the file is recorded. In both cases, a system crash may leave the log data inconsistent.
>
> In the first case, the reference counter is one too low, while in the latter it is one too high.

The underlying problem is that updating the reference counter is not an idempotent operation, meaning that the results gained from performing the operation one time or many differ. Thus recording a file and updating the relevant counters must be performed atomically.

Though possible (e.g. with transactional databases) it requires the hash index and the key index to be tightly coupled – either by storing these in the same database, or at least in two structures that allow transactions to span them both. This gives a less flexibility in the design.

Additionally, reference counting is fragile: It requires a lot of extra bookkeeping logic to be executed safely [30].

Pros: When deleting a few snapshots, leaving few dead, only the log data of the dead ones needs to be retrieved. The same log data is never retrieved multiple times (the snapshot and its log data is deleted after retrieval). The liveliness analysis is fast, since only the chunks from snapshots that have been deleted need visiting.

Cons: When deleting many snapshots, leaving many dead, the log data of all the dead ones needs to be retrieved. The implementation depends on extra GC log data in the form of counters. Updating the counters is not simple to get right without transactions. Deduplication is required at the log level to be efficient.

3.6.5 Reference lists

Reference "counting" can be made idempotent by keeping a full list of references[9] instead of merely a reference count. For each chunk, its reference list keeps track of all snapshots referencing that chunk.

When a new snapshot references a chunk, the snapshot is added to the chunk's reference list. Conversely, it is removed from the list when the snapshot no longer references the chunk (e.g. the snapshot was deleted). A chunk is dead when its reference list is empty.

During a snapshot, the reference lists of the snapshot's chunks are updated.

Because removing a snapshot from a reference list is an idempotent operation, a deleted snapshot can simply be removed from every reference list; No log data needs to be retrieved. And if this process crashes, it can merely be restarted (the reference lists for which the snapshot was removed in the first run will stay unchanged during the second).

As opposed to the tracing GCs and the reference counting GC, this method allows liveliness analysis based solely on local data. This makes the GC routine more efficient, but introduces an overhead in storing the reference lists.

On the downside, a reference list is a more complex structure than a simple counter. But the requirement of atomicity is avoided, due to the idempotent insert and remove operations. Our tests suggest that reference lists can be represented efficiently when encoded as bit-vectors and compressed in bulk. See section A.4 on page 102.

Pros: It is possible to find dead chunks with local data only. The actual deletion is quick, since the snapshot can be removed from reference sets locally (its log data is not needed).

Cons: Deletion requires traversal of the local index, to remove the snapshot from the reference lists of all its chunks. A reference list needs to be stored for each chunk.

3.6.6 Conservative garbage collection

It is possible to perform a conservative garbage collection (see section 3.6.1 on page 33), where not all references to dead chunks are known precisely. Thus, not all garbage is necessarily removed, but it is guaranteed that what *is* removed *is* garbage.

[9] Technically a set, but the word "list" is used in the literature [30, 61].

This can be achieved by adding a Bloom filter [67] to each snapshot, containing its chunk hashes. Rhea et al. [82] lightly touches the possibility of using a Bloom filter to enable conservative garbage collection, but we are not aware of any real analysis of this idea.

A Bloom filter is somewhat like the opposite of a cache, in that it performs approximate membership testing. But unlike the cache its certain answer is the *positive* one. When testing whether (or not) a chunk is needed by a snapshot that *does* reference it, the Bloom filter will always answer *positively*. However, if the chunk is not in the snapshot (and therefore not in the filter), the Bloom filter *may* answer negative, or with probability ϵ falsely positive.

A Bloom filter consists of an m sized bit-vector, and k hash functions which each map a value to a bit position in the vector ($v \to [0; m-1] \in \mathbb{N}$). The empty set is a bit-vector of sole zeros. When a value is added to the filter, its k hashes are computed (one for each hash function) and the resulting bit positions are set to 1. To check whether a value may have been added, its hashes are computed, and its bit positions checked: if one of them is 0, the result is a definite *no*. On the other hand, if all of them are set to 1, the result is an uncertain *maybe*. Elements cannot be removed from the filter once inserted. The trick is to choose a configuration given a false positive rate $0 < \epsilon < 1$ and the number of elements the filter must contain, in terms of the bit-size m and the number of hash functions k.

This can be used to implement a conservative GC. When creating a new snapshot, a new Bloom filter containing all the snapshot's chunk-hashes is created. To test whether a chunk is in use, it is checked against all the snapshot Bloom filters, and if one claims to use it, it may be alive and must be considered to be so. If no filter claims to use it, it is definitely dead and can be deleted. Whenever a snapshot is deleted, its Bloom filter of chunk-hashes is deleted too, thus removing its chunk references (positives as well as false positives).

If the same configuration of hash functions is used for all Bloom filters, they can be merged together to form a single Bloom filter describing all the snapshot's references, by simply or'ing the bit-vectors. This turns checking a hash's liveliness into a constant-time operation. However, it gives a higher false positive rate, since the new combined filter contains the elements from all the merged filters, and thus we cannot advise this as anything but a heuristic to avoid checking all filters individually for every hash. Our prototype does not employ this heuristic, and we do not discuss it further.

The false positive rate ϵ of a Bloom filter is approximated as [67]

$$\epsilon \approx (1 - e^{-kn/m})^k$$

where m is the size of the Bloom filter in bits, n is number of elements inserted and k is the number of hash functions used. However, since every Bloom filter from all snapshots has to reject a hash, before it can be deleted, the false positive rate of the GC is $1 - (1 - \epsilon)^s$, where s is the number of snapshots.

From this, we can calculate that a Bloom filter using 10 bits per element and 7 hash functions has a false positive rate of $\epsilon \approx 0.082$. This would result

	GC time	GC transfer	Extra log data
MS-GC	$O(a+d)$	$O(a)$	$O(1)$
RC-GC	$O(d)$	$O(d)$	$O(a+d)$
RS-GC	$O(a+d)$	$O(1)$	$O(a+d)$
C-GC	$O(a+d)$	$O(s)$	$O(s)$

Table 3.1: Garbage collection methods.

MS-GC is the mark-sweep GC, RC-GC is the reference counting GC, RL-GC is the reference listing GC and C-GC is the conservative GC. Here, d is the number of chunks that are used by snapshots marked for deletion, while a are all other chunks. And s is the number of live snapshots. Note that the running time of RS-GC requires constant-size reference lists with constant-time operations.

in a conservative GC that would remove 92.1% garbage from 10 snapshots or 43.9% from 100 snapshots (assuming no data overlap between snapshots).

Mitzenmacher [67] further investigates compressed Bloom filters, and show that these may yield an even lower false positive rate per stored byte.

Pros: Uses a small amount of extra log data, and likewise uses a small amount of data transfer during the GC routine.

Cons: Is likely to leave some percentage of garbage behind. Requires a liveliness check on every known hash (dead or alive).

3.6.7 Discussion

Table 3.1 lists each GC method along with its expected performance. The mark-sweep GC requires the least extra log data, the reference counting GC has the cheapest liveliness analysis, the reference listing GC transfers the smallest amount of data and the performance of the conservative GC depends on the number of live snapshots.

We consider data transfer to be the largest penalty, and find reference lists the most appealing GC method on large repositories. We expect a compact representation to yield a small overhead and make the reference listing GC's extra log data an acceptable trade-off. As mentioned in section 3.5 on page 28, global and online deduplication already needs an index of hashes, so much of the log data is present already. The actual reference lists are the only added overhead.

Another interesting method could be a combination of mark-sweep and conservative GC. Frequently, the repository is cleaned with the conservative GC, which is cheap, but cannot remove all the garbage. Periodically, mark-sweep is invoked to clean whatever the conservative GC left behind. This method is especially efficient on small repositories, where the Bloom filters needed by the conservative GC are small. For a million chunks, filters of just

1 MB per snapshot could clean out 98% of the garbage. However, when using the same filters with 10 million chunks, the effectiveness drops to 30%.

In the prototype, we have implemented a conservative GC, partly because this required the fewest changes in the implementation, and partly due to our failure in finding any other project that has evaluated the performance of a conservative GC.

In section 4.6 on page 70, we present the details of our conservative GC and in chapter 5 on page 77 we test and discuss its performance.

3.6.8 Garbage collection of log data

Since Hindsight reuses its own backup system for saving the log data, it can also reuse the garbage collection method to reclaim log data storage. Thus the prototype naturally uses its conservative GC to reclaim log data.

3.7 Crash Recovery

We want the system to be able to recover from a system crash. If a snapshot is interrupted, it should be possible to recover the local log data to a consistent state. The alternative is to discard the partial snapshot, recover the latest version saved on the back-end and start over.

When considering how to make the system crash safe it should be taken into account in which environment it is running. If it runs on a server in a basement that virtually never fails it may be acceptable if it takes a very long time to recover from a crash. On the other hand, if the system crashes often it might not. We expect Hindsight to run in a variety of environments and it is simply not acceptable to be faced with a long recovery process because the laptop's battery has run out.

We have designed Hindsight with an "expectation of crash". This is inspired by Candea and Fox [29] who discuss crash-only design, where a piece of software is designed to crash as the correct and only way of stopping it. In Hindsight, crashing is not the preferred way of stopping, but the penalty from doing so should be small.

We distinguish between two kinds of crashes; program crashes and disk crashes. In a program crash, the program is terminated and any data residing in memory is lost, but data already written to disk is safe. In a disk crash all local data is lost and only the back-end remains. Because of checksumming we can detect bit rot in local files, and we treat such data corruption as a disk crash.

3.7.1 Program crash; no data loss

A naïve approach to handle program crashes is to copy the affected log data before taking a snapshot. If the snapshot failed for some reason, the log data can simply be rolled back. This approach, however, involves several problems:

- The blobs that were transferred to the back-end during the failed snapshot are not referenced by the saved log data. When the log data is

rolled back, the blobs are no longer used but they still take up space.

This could be improved by only copying the key index and treating the hash index specially. That way, the chunks uploaded during the partial snapshot could be referenced, and need not be rolled back.

- The log data may be large (extrapolating from the results given in appendix A on page 97 suggests several gigabytes for a large system) and so taking a full copy can be expensive. This copy must be taken every time the log data is modified.
- After the log data has been rolled back, the failed snapshot must be restarted from the very beginning. Of course the log data can be copied at several points during a snapshot, but this adds even more overhead.

In Hindsight, we handle crashes a bit differently. The hash index keeps track of blobs that are in-transfer, while the key index keeps track of keys whose chunks may not have been stored.

So the first point is easily addressed; for each blob that is transferred its ID is written to a log file. After a crash all transferred – but unreferenced – blobs can be removed from the back-end, preventing dead blobs from cropping up.

The second and third points are addressed by not copying the log data at all and instead starting directly with the key index left by the prior snapshot. This makes it crucial, that the system is always able to bring the indices back to consistency. What it boils down to is that the hash index must not reference a non-existing blob and the key index must never reference a non-existing chunk.

The recording of chunks cannot be delayed until the blob in which they are contained has been transferred to the back-end, because the system should be able to transfer several files in parallel. It is the whole point of deduplication that a chunk is only saved once, so one part of the system must immediately be aware of the chunks saved by another.

Appending to a log each time a chunk is inserted is not feasible because

- There are too many chunks. Every file gives rise to at least one chunk.
- An expensive disk sync must be performed after each chunk is appended to the log, making the approach even more prohibitive.

We solve this problem by having an in-memory chunk cache, and writing chunks (to the index, and to the log) in bulk; see section 4.4 on page 66 for details.

Logs of the inserted files must also be kept. After a crash we have the choice of *a)* reinsert each file; or *b)* for each inserted file check if its chunks have been inserted in the hash index, and only remove the file path if there are chunks missing.

Depending on the garbage collection strategy (see section 3.6 on page 33), it may be necessary to adjust the hash index before a file path can be removed from the key index.

This makes option *b)* the better choice because it sometimes saves files, and has roughly the same overhead as option *a)*[10].

When files are very small it can be quite limiting to write and sync the log files. Since all the files that are to be backed up are known in advance (they are right there on the disk), an obvious solution is to bulk-write several file paths to the log at a time. This is the solution we use in our prototype.

In section A.2 on page 98 we present benchmarks of our prototype while varying the amount of file paths written to the log at a time.

3.7.2 Disk crash; data is lost

If the log data is lost it must be possible to restore it from the back-end. So it must have been saved there at an earlier stage. To exploit redundancy in the log data we use Hindsight itself to save its own log data. The resulting meta log-data is saved directly.

The log data must be saved periodically. A problem arises if the disk crashes when blobs have been transferred since the log data was last saved; when the state is restored it no longer references those blobs, which are now dead.

There are three ways to fix this: *a)* the snapshot's log data must be stored incrementally on the back-end, so that there will never be an "old" version; *b)* it must be possible to remove the dead blobs from the back-end, even though they are not referenced; or *c)* the log of in-transit blobs must be kept remotely to survive the crash.

The first option will decrease performance because no blob can be transferred before the hash index referencing it has been saved. In the extreme case all the blobs are queued before being transferred, and in the others the hash index will be transferred several times. In either case the time it takes to complete a snapshot will increase.

To be able to remove dead blobs from the back-end there must exist a way to discover which blobs are stored there. The solution is to extend the back-end API with an operation, *LIST*, which returns a list of stored blobs. As an example Cumulus [93] requires a *LIST* operation of its back-end.

3.7.3 Back-end failure

Even though we assume a reliable back-end, that does not loose data, we have still thought about what would happen if it did. It would improve transparency and confidence in the system, if a separate tool could restore data from the back-end, even when it is partly corrupted.

In this section, we discuss the consequences of losing a single blob. Naturally, the type of data stored within the blob is essential to the amount and nature of the information lost. When discussing indices we will assume the

[10] The reinsertion of files in option *a)* leads to as many index lookups as checking in option *b)*, if no chunks are missing.

lost blob contained the root of the index and hence that all of the index has been lost.

Data blob: We lost some of the data, and all we can do is to skip the chunks stored in this blob when extracting. This will leave holes in the extracted files that reference the missing blob, but it does not affect other files than these.

Hash index: If the hash index is broken, we can rebuild it by inspecting all data blobs and recomputing the chunk hashes. This is possible, because the length of each chunk is stored in the data blob, and not in its reference in the hash index.

If a reference counting or reference listing garbage collector is used, we would have to recompute the reference counts or lists, by scanning each key index and update the hashes it uses.

Key index: If the key index is lost, we have lost the directory listing for the corresponding snapshot. But even though we have deduplication on log data across snapshots, we can still recover the missing files in their other versions.

If a file and its "neighbours" in the key index have never changed they are lost forever, because that part of the key index will not have changed, and thus there is no newer or older version of it.

But if a file has never changed, chances are that it is still available on disk.

Tarballs: If a tarball describing log structure is lost, then either a hash index or a key index is lost. In each case, the situation resembles that of losing one of these indices, and in both cases the required logic is the same as just discussed, but at the secondary level.

Snapshot index: The snapshot index can be seen as the "root" of the system: the main entry point. If we loose it, we loose the descriptions of all snapshots. However, this data is stored redundantly in the tarballs themselves, in a file named `info`. Thus, we can rebuild the snapshot index by finding the tarballs.

This discussion can be extended somewhat to loss of multiple files. If we lose many data blobs, we can skip more data when checking out. With the indices, we assumed that all of the index was lost. And if multiple indices are lost, we can repeat the actions taken, as long as one valid version exists.

Even if the meta-data turns out completely broken, we can still restore the data blobs, for user inspection. We cannot say much about how the chunks fit together, except that the original version of the file was stored continuously.

3.7.4 Discussion

Our prototype is (almost[11]) fully resistant to program crashes. To strengthen our argument we have devised a test in which the system is crashed very often. See section 5.2.1 on page 83 for details.

The prototype can live through a disk crash but it cannot reclaim unreferenced dead blobs. We find that the unlikelihood of leaving unreferenced blobs does not warrant introducing a fourth operation to the back-end API. However it can easily be introduced if one wishes.

We have not done anything to recover from blobs lost by the back-end. However, we think that the prospects of a tool that can do that puts our design in a positive light.

3.8 Security

In this section we discuss security issues, and what Hindsight does to address them.

3.8.1 Chunk hashing

Internally, Hindsight uses cryptographic hashes [43] to identify equal chunks of data (i.e fingerprinting). A cryptographic hash function takes an input of arbitrary length and produces a fixed-sized number from it (usually just referred to as a hash[12]). Since the resulting output is fixed-sized, there is a risk of producing equal hashes from two data chunks that are not the same.

If this happens, Hindsight would discard the latest chunks, and thus loose data. On the other hand, comparing full chunks is impractical both with respect to time and local space usage (we do not want to store the full data chunks locally).

The size of the hashes needs to be chosen, and it seems consensus says 256 bit hashes are secure and that the risk of collisions with these are negligible [13][43][84], so this is what the prototype uses. On average the first collision occurs with roughly $2^{n/2}$ hashes, where n is the bit-size of the hashes, so in our setting the first collision would occur after roughly 2^{128} hashes [43]. For comparison, the popular revision control system Git uses 160-bit hashes to identify content [4].

Further note, that we mainly need protection from *accidental* collisions; In order to abuse a collision, one would have to first create a malicious file with a hash that matches some different data on a user's system, send the malicious file to the user, and then have Hindsight register this malicious file *before* the user's own data. Creating a chunk of data with a hash that matches some other *specific* chunk requires 2^{n-1} hash computations on average.

This attack could be avoided by using a Hash-based Message Authentication Code (HMAC) instead of a regular hash. HMAC is like a hash function that besides the input data takes a secret key. Each key will produce a different

[11] See section 6.1 on page 87.

[12] But beware: a hash function is not necessarily a *cryptographic* hash function.

output. With this construction 2^n hash computations are needed to produce a collision [43], and thus it may be safe to use a smaller hash size.

To recap, the prototype uses regular 256-bit hashes to identify content. For information on the choice of algorithms, see section 4.7 on page 73.

3.8.2 Threat model

We want Hindsight to safely store data on an untrusted, yet reliable, back-end. This must occur with a minimum of information leakage. In this section, we present a threat model and discuss how the threats relate to Hindsight, and what we do to limit them.

We use a life-cycle threat model based on the one presented in Hasan et al. [51], where threat models for storage systems are investigated. Here, the life-cycle of a piece of data is divided into 7 parts: *a)* creation at the client and transmission; *b)* reception at the back-end; *c)* output preparation at the back-end; *d)* retrieval; *e)* backup and redundancy at the back-end; and *f)* data deletion. Throughout the life-cycle, four properties must be protected:

Confidentiality: Others cannot read the data.

Integrity: Others cannot modify the data.

Availability: The data stays available for its rightful owner to retrieve (this category includes avoidance of denial-of-service attacks).

Authentication: By requiring authentication from those accessing the data and verifying their permissions, confidentiality and integrity can be maintained to some extent.

Here, "others" refer to other users of the back-end, hackers and insiders.

We assume a reliable back-end, which means the back-end is responsible for points *b*, *c* and *e* of the life-cycle. This implies protection from other users of the back-end and hackers, regarding data integrity, authentication and overall availability. In short, the back-end must maintain the data stored and be able to bring it back for later retrieval. Only Hindsight shall be able to put and delete data.

Since we think of the back-end as untrusted, we do not want to share our data with it. In fact, we assume a malicious insider who wants to read and modify the data silently. Naturally, we cannot protect the data from an insider who wants to corrupt or delete it. The parts of the life-cycle that Hindsight is responsible for is then *a*, *d* and *f*:

Confidentiality: To prevent others, including insiders, from reading the data Hindsight stores, we apply client-side encryption of all blobs. Additionally, the blobs are given non-sense names in the form of unique random IDs (except the snapshot index).

The data is encrypted locally before transmission and likewise decrypted locally after retrieval. Thus, the back-end never sees the unprotected data, and confidentiality is ensured throughout transmission, retrieval

and deletion. At creation, the data resides in memory at the client, which we assume is safe.

Since the back-end never sees the unprotected data, neither will a back-end insider or hacker.

Integrity: We prevent data tampering from going unnoticed by protecting all blobs with a cryptographic MAC. As with encryption, this process is done locally, before any transmission. When decrypting a blob the MAC is first verified; if it fails the blob is rejected and considered dead.

Since the back-end never sees the data without a MAC it cannot modify it without being noticed when the data is retrieved. Thus, we prevent silent modification by a back-end insider or hacker.

Both encryption and authentication requires a key. Our prototype generates a random 256-bit key once. This key is then used for the whole repository. For encryption, a 192-bit random nonce is used to ensure, that a blob will always result in a unique ciphertext. For more details, see section 4.7 on page 73.

3.8.3 Information leakage in Hindsight

The design of Hindsight is leaking information:

New blobs: By monitoring new blobs, the back-end may monitor the amount of new information added to the system. This may leak information about when system updates are run or when working hours occur. This leak is inevitable, since we only want to transfer the new information to the back-end. This is a minor issue and expected when using a backup system.

Total data size: The total byte count of all blobs leaks a minimum for the amount of data stored (after compression). This is a minor issue and expected when using a backup system.

Blob sizes: Compression and content-aware chunking yield blobs whose sizes depend on the data they contain. By knowing the size of the blob, we know something about the data. This can be used for fingerprinting and proving the presence of some specific data. A back-end maintainer may be able to find evidence for the existence of a Wiki Leaks tarball solely by looking for blobs of specific sizes.

While the first two leaks are minor issues that one would expect from a backup system, the last one is worse. It allows an outsider to gain insight in the stored data. This can be lessened by removing compression and content-aware chunking. But the last chunk of a file could still leak information. A complete solution is to add padding to blobs and only store blobs of a fixed size.

3.8.4 Convergent Encryption

A drawback from this simple, yet safe, security model is that it breaks the possibility of deduplication across systems (universal deduplication), instead of merely deduplication across files within one system (global deduplication).

This is because the file chunks from one user are encrypted differently than the file chunks from another. As a result, the same chunk will look different to the back-end each time it is stored.

A simple way to solve this, is to always use the same key on the same chunk. This is achieved by encrypting the chunk with a key derived from the chunk itself (typically its hash). This process is often referred to as convergent encryption[40].

When the same chunk always yields the same encrypted ciphertext, it is possible to implement a deduplication routine on the back-end and get deduplication across all systems. This could save a lot of data, since the systems may share system libraries and applications.

However, this efficiency comes from a compromise in privacy. When the chunk is encrypted with a key derived from it, the back-end can prove whether or not a user has a certain chunk of data. This is a serious leak of information, and the reason why Hindsight does not support it.

It would none the less be an interesting *optional* feature, allowing the user to choose some paths to back up with a lower security requirement (e.g. `/usr/lib` on a Linux machine). This feature would allow the user to make a choice between space usage and privacy.

We have aimed for "security by default" and therefore did not go with convergent encryption. In section 4.7 on page 73, we discuss what one would need to do to extend our prototype with convergent encryption.

3.8.5 Access Control with Hindsight

Hindsight has no built-in support for access control. With the master key, one can read everything and generate new authenticated files.

We did however consider what could be done, in order to introduce some form of access control. A motivation for these scenarios could be:

Read-only: You may want to allow others access to your backups (e.g. your company), while still being the only one writing new content.

Write-only: If a thief steals your laptop, you do not want that person to access your backups. Your laptop needs write permissions only.

Revocation: If you ever loose your key, you want to be able to revoke its permissions.

The first one cannot be done through cryptographic protection. We could prohibit someone from writing authenticated files, but the person could still delete or corrupt the data.

In the second case, an asymetric cryptosystem could allow the system to encrypt the data, without being able to decrypt it. However, if the thief has

access to the current state of the data, then information has already leaked. And in any case, this does not prevent the thief from deleting or corrupting the data.

Therefore, access control is better resolved on the back-end. A key could be used for authentication, and if that key is ever lost (laptop is stolen) its permissions could be revoked. If the back-end supports access control, then Hindsight supports it inherently.

But even then, a thief could still have the master key, which is stored on the client. Though the access to the back-end has been revoked, this is still a major breach. To avoid this, an indirection between the key used by the client and the master key is required:

1. When the repository is initiated, a master key *and* a client key are generated. The master key is encrypted with the client key and stored on the back-end. The client key is stored locally as part of the Hindsight state.
2. When Hindsight starts an operation, it retrieves the encrypted master key and decrypts it, but never stores it locally.
3. When a new client is introduced, the master key is retrieved, decrypted and re-encrypted with the new client key. Hence, clients can create new clients.
4. To revoke a client, its permissions are revoked and its encrypted copy of the master key deleted. The client is now left with a useless client key.

This setup only works if the revocation happens before the client is overtaken and used to retrieve the master key. Ironically, the client protects itself from the back-end with cryptography, while depending on the back-end to protect the data from other hostile clients.

3.9 External Storage

In order to accommodate as wide a range of external storage solutions as possible the back-end API must be simple. We require back-ends to implement this minimal API:

$PUT(k,v)$**:** stores the blob v with key k. If a blob with key k already exists, it should be overwritten. The operation must be atomic: It should either store the data correctly, or inflict no change. This allow us to ignore the problem of half-stored blobs and enable us to overwrite our snapshot index root directly, without worry.

$GET(k)$**:** retrieves the blob stored with key k.

$DEL(k)$**:** deletes the blob stored with key k along with the key.

Hindsight overwrites stored blobs in one case only: the snapshot index is always stored with key `snapshots`.

This makes it easy to implement new back-end modules for new types of external storage, may it be an external harddisk or a cloud-based key-value store. It also simplifies the task of applying a secure encryption layer, since the server is left with as little knowledge as possible – it does not need to know the semantics of the data in order to store it, and that is all we require.

We have left out *INIT* and *UNINIT*, which means the back-end must be setup and ready for use when Hindsight is run. The reason for leaving them out is that they was not strictly needed by our prototype. Adding the functions will not change the design fundamentally. In short: we judged our time spent better elsewhere.

Our prototype comes with back-end modules for storage in a local file system, via SSH, at Amazon S3 or in an Apache CouchDB database. But it is almost trivial to write a new Hindsight back-end module to support another key-value store.

On rare occasions, unreferenced blobs can be left on the back-end (see section 3.7 on page 43). With the suggested API it is not possible to remove these blobs. A solution can be implemented if the API is extended with a *LIST* operation, which returns all keys stored on the back-end. This would allow Hindsight to compare the list of keys it knows to the actual keys, and remove unreferenced keys from the back-end.

3.10 Storage format

In this section, we describe what Hindsight stores and where. This includes file paths, data and meta-data.

3.10.1 Chunks and blobs

When backing up, we take the data present locally and replicate it on the back-end. Because of deduplication (see section 3.5 on page 28), files are divided into chunks, and to hide latency and use network bandwidth more effeciently they are collected into blobs.

An example of a system that does not collect chunks into blobs is the S3QL file system [79]. As a result, the system may store many small files on the back-end, giving an excessive amount of back-end operations. If each operation comes with a long latency (authentication etc.) this may be inefficient, compared to storing fewer large blobs. Further, S3QL is designed for the Amazon S3 key-value store where the user is charged per operation.

To avoid this scenario, Hindsight groups the chunks into blobs, until some predefined size criteria has been met. However, the blobs should not be too large either: since the back-end API only allows retrieval of whole files, the blobs should be small to allow a low minimal data retrieval. The default in Hindsight is 2 MB. Other systems that group chunks into blobs include bup [72], Cumulus [93] and Venti [77][13].

[13] A blob is equivalent to a *segment* in Cumulus, an *arena* in Venti and a *packfile* in bup.

When Hindsight groups chunks, it guarantees that the new chunks of a file (those that are not known beforehand) are stored contiguously; that is, they are stored in a sequence of blobs where only the first and last blob may contain data from other files. This preserves locality between chunks, making it more likely that retrieving a blob to get a chunk of a file gives other chunks from the file along with it. This is something Venti cannot do [96].

A beneficial effect of grouping chunks into blobs is that this allows for compression of whole blobs – and thus many chunks – at a time. This gives the compression algorithm more information about the data and is likely to yield a better compression ratio; specifically in the case where the chunks are from the same file. At least Cumulus authors (Vrable et al. [93]) noted this benefit.

Likewise, encryption and authentication can be performed once per blob, thus lowering the overhead of securing the data.

Splitting file data into smaller chunks gives better deduplication, whereas grouping the resulting chunks gives better performance.

3.10.2 File paths and meta-data

Hindsight stores both file paths and meta-data directly in the key index. File paths needs to be stored here, since this index is used for querying the directory structure. However, meta-data could be stored in blobs on the back-end like the file data.

There are two reasons for storing the meta-data up front:

1. The meta-data is likely needed whenever the file data is needed, and it may even be needed when the file data is not.

   When restoring a file, its meta-data is restored as well. However, in our FUSE front-end, the meta-data is used to construct the file structure, before the file data is retrieved.

   By avoiding the indirection, the meta-data is more accessible and faster to retrieve.

2. The meta-data is likely better compressed when stored in the index alongside other meta-data, than when mixed in blobs with file data. Further, file meta-data can rarely be de-duplicated like file data chunks, since the time stamps they contain are monotonically increasing; they do not repeat themselves and so neither does the meta-data as a whole.

On the other hand, storing the meta-data directly in the key index is likely to make it larger (but not certain; it might be possible to compress the meta-data to less than the size of its fingerprint). If the size of the key index grows, querying for keys will be slower because more data needs to be retrieved.

3.11 Summary

In this section, we discuss the limitations and flexibility of the design.

Snapshots per repository: If reference lists are used for garbage collection, the number of snapshots is limited by the lists' ability to scale. We discussed this in section 3.6.5 on page 40.

Since our prototype uses a tracing garbage collector it has virtually no limit on the number of snapshots; The only limiting factor is that the snapshot index is read into memory, but it is very small, and millions of snapshots are still possible.

Files per snapshot: There is no limit on the number of files in a snapshot as long as the disk can store the resulting log data. Hence, the only limit is whether we can store the file names, meta-data and chunk hashes. This log data is typically less than 1% of the original data (see appendix A on page 97).

Bytes per file: We maintain the chunk hashes of each file in a list in memory. Thus, we cannot snapshot files that result in more hashes than can be kept in memory. This is limited further by the fact that we need to be able to keep a leaf from the key index in memory – we may be in trouble if many very large files are snapshotted together.

We discuss how this problem could be solved with hash trees in chapter 6 on page 87 on further work.

Deletion: The design is not limited to deleting full snapshots only. It is possible to delete just some keys inside a key index, by copying it with the wanted keys, and then deleting the original.

Front-end: Due to the decoupling of front-end and backup system, it is possible to write new front-ends without changes to the core system.

Remarks on resource usage: The design does not rely on any large in-memory data structures. It uses external b-trees which allows most data to stay on disk, and any space-time trade off decissions are left to the implementation.

As a note on time usage, each chunk is stored in $O(\log \mathit{hashes})$ time, which comes from inserting its hash in the hash index. Further, each key requires $O(\log \mathit{keys})$ to store it in the key index. As a result, each key is stored in $O(\log \mathit{keys} + \mathit{chunks} \cdot \log \mathit{hashes})$ time, where *chunks* is the number of chunks in the key's contents.

In practice b-trees will never grow very deep because of the large amount of splits in each node. In other words, the "log"s will be quite small.

We believe it possible to write an efficient backup system that implements this design.

Chapter 4

Implementation

4.1 Installing and using the prototype

In this section we briefly explain how the Hindsight prototype can be installed and used. This program is *not* stable yet, however it will *probably* not harm the host system.

To ensure safety, it is possible to run the prototype with the Ubuntu 11.10 live CD, or in a modern Ubuntu virtual machine.

4.1.1 Installation

To build Hindsight the following is needed: the Glasgow Haskell Compiler in its newest stable version (`http://haskell.org/ghc`), the Haskell package tool `cabal-install` and the Hindsight source code (`http://goo.gl/Vl7ho`).

Before Hindsight can be built, a configuration file must be created. An example is located in `Config.hs.example`. This file should be copied to `Config.hs` and edited as needed.

The most important configuration is the back-end module, and its location on the file system. The default is to use the back-end module in `~/.hindsight-modules/local` for local storage. If this is used, a link to the `modules` directory in `src` should be placed in `~/.hindsight`:

```
ln -s src/modules ~/.hindsight-modules
```

We have written a project file for the Cabal package tool, which should make installation as simple as running `cabal install` in the `src` directory.

Prebuilt binary. As an alternative to building Hindsight, we provide a 64-bit prebuilt binary. This executable was built to use the module path `~/.hindsight-modules/current`. The `current` module can be setup as a link to the actual module. To install the prototype with this configuration, a `setup.sh` script has been included. It copies the back-end modules to the aforementioned location and links the `local` module to `current`.

Since we cannot ask the reader to simple download and run an executable from a strange website, the tarball containing the files has been signed by one

this report's authors' public key using GPG. Details on how to download and verify the prebuilt binary can be found in appendix C on page 122.

4.1.2 Using the prototype

In this section we shortly present how to use the prototype, and how to inspect snapshots using the FUSE mount tool.

Table 4.1 gives an overview of the available commands. First, the repository is initialised. Snapshots are added to the system with the `snapshot` command. These can be inspected with the various list and checkout commands.

Description	Command (`hindsight [...]`)
Documentation and help	`help`
Initialise repository	`init`
Save all log data on back-end	`seal`
Recover log data from back-end[1]	`recover`
Garbage collect	`gc`
Take a snapshot[2] (named "`home`")	`snapshot home ~`
List snapshots	`list`
List contents	`list homedir~1`
List files matched by prefix	`list homedir~1:code/hs`
List directory	`listdir homedir~1:code/hs-tree`
Recursive checkout	`checkout -r homedir~1`
...by prefix	`checkout -r homedir~1:code`
...and without recursion	`checkout homedir~1`
...files and meta-data only	`checkout --nodata homedir~1`
Delete snapshot	`delete homedir~2`

Table 4.1: List of Hindsight commands

Playing with cache: Hindsight caches the blobs it retrieves locally to avoid retrieving the same blob multiple times. To clear its caches, the user has to delete the cache folder from the repository directory, as well as all the locally stored key indices:

```
> rm -r ~/.hindsight/{cache,pri,sec,snaps}
> hindsight recover
```

The recover is needed because the command also removes the `idx` folders containing the hash index. The recover command recovers the hash index, making the repository ready for use again.

[1] Caution: This command should only be used if the local log data is corrupted, as it *will* be overwritten.

[2] More precisely the command takes the name of a family, modifies the head to reflect the filesystem under the given path, then takes a snapshot of the head.

Mounting a snapshot: To mount a snapshot, the mount tool located in `src/tools/fuse` is used. The tool creates a cache folder named `dir.cache` to store the actual directory structure.

A snapshot is mounted in an empty directory with the command:

```
> ./mount.py [snapshot] dir
```

The `dir` directory now mirrors the snapshot as it looked when it was taken. To unmount it, use the `fusermount` utility and remove the cache directory:

```
> fusermount -u dir
> rm -r dir.cache
```

4.2 System design

In this section, we first give an overview of the process model that drives the Hindsight prototype. Then we describe the prototype's individual processes.

4.2.1 Process model

The implementation is structured with a process model inspired by Erlang and the actor model. This means that the system is written as various independent and isolated parts that run in their own dedicated thread.

The model is inspired by the Haskell packages eprocess by Benavides [14], remote (also known as Cloud Haskell) by Epstein [41] and Combinatorrent by Andersen [10].

In our model, a process consists of:

A message API which describes the messages that this process can handle. When the process is first created, a new input channel that accepts such messages is created and assigned to it. This channel acts as the mailbox of the process.

A message handler that takes care of executing the needed action whenever a new message arrives. Messages can either be sent synchronously (i.e., the caller blocks and waits for a reply) or asynchronously.

A flush handler that runs when the system flushes its state (e.g. when the last file has been inserted). When a process receives a flush message it should pass it on to the processes it delivers data to, and then prepare to be shut down.

Sending a flush message is a blocking action, and so it can be used to empty the system of data. A flush message is sent just before shutdown to make sure that all data has been stored on the back-end.

A runner to setup any initial state needed by the process.

The model allows us to reason about a single part of the system in isolation, and to easily replace one part with another that follows the same API. Since every process can maintain arbitrary state, it gives us a way to implement

global state. An example use of this is the statistics process that collects and displays progress information while the system is running.

One of the big advantages of this model is the inherent concurrency which makes it easier to exploit parallelism. But there is another interesting observation: We can insert arbitrary logic at the boundary of processes, since this logic is defined in the process library and not by each individual process. There are some interesting use cases for this:

Exception handling: By catching all exceptions directly in the process model, we could report on messages whose execution failed and implement a retry mechanism for this scenario.

We could further re-throw the exception in the calling process to get error tracing on a per-process level, somewhat like stack traces known from imperative languages.

Throttling: By limiting the maximum number of queued messages in a channel, the system can auto-throttle itself. When the system is stressed, new messages will meet a full input channel and will have to wait. This concept is already used in the prototype.

Profiling and debugging: By measuring the resources used to process a message, we could collect statistics to see what kind of messages are the most expensive and in which processes. This could lead to more detailed profiling. Combined with a system for detecting anomalies (e.g., Surveilr [49]), this could be used to detect malfunctioning and strange behaviour in corner cases.

4.2.2 Processes in the prototype

The Hindsight prototype consists of the following processes (illustrated in figure 4.1 on the next page):

Statistics: Keeps track of how much data the system has processed so far, along with the most recently processed file. In general, this process takes care of all status communications.

Key store: The outer, high-level API for adding keys to a head. This process uses the key index and the hash store processes.

Hash store: Calculates fingerprints and determines whether a chunk is already known. New chunks are sent on to the blob store.

Blob store: Responsible for merging chunks into blobs and retrieving these again later. This process uses the hash index and the external processes.

Key index: Maintains a b-tree of the keys in the current snapshot and their respective chunk hashes, without relying on other processes.

Hash index: Maintains a b-tree of all hashes available to the repository, without relying on other processes.

External: Takes care of all external communication. This is the only place where the logic needed for using the external storage (e.g. ssh, S3) is implemented, and hence all such actions have to go through this process. This process ensures that all data is compressed and encrypted before leaving the client.

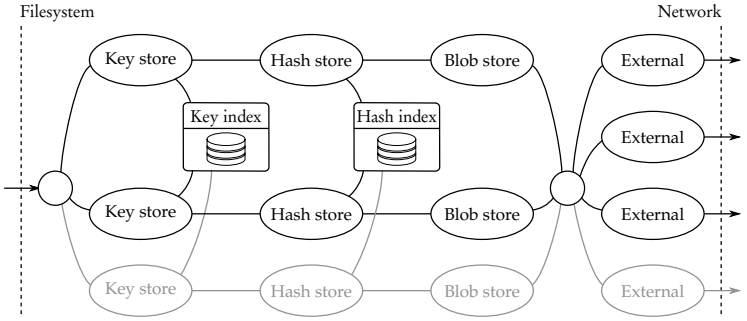

Figure 4.1: Hindsight's processes.

As illustrated in figure 4.1, the main *chain* consists of the three processes: key store, hash store and blob store. These processes use the two[3] system-wide processes: key index and hash index. This chain can be replicated to support parallelism across files, but the system-wide processes can only exist as singletons due to invariants in the b-trees they maintain[4].

Each of these chains outputs to a multiplexer, which takes care of load-balancing (based on mailbox sizes) between multiple external processes. A dedicated multiplexer-process is needed here to support the special flush-message.

Every process supports flushing in its own way: The index processes sync their b-trees to disk, the blob store empties the buffer no matter the size, and the external process empties its mailbox and saves everything on the back-end. Each process flushes the next process in its chain:

key store → hash store → blob store → multiplexer → external.

The index processes are flushed specially at last, to make sure the log data is updated *after* the external-process is flushed (the hash index is updated by a callback in that process).

4.3 Indices

4.3.1 Overview

The indices used in Hindsight are implemented as mutable concurrent b-trees with relaxed balance, as described by Larsen and Fagerberg [59]. The tree varies from regular b-trees [33] in that:

[3]Three actually; but here we disregard the statistics process as it is not paramount to the system.

[4]The index processes could employ worker threads, but the prototype does not do this.

1. It allows periods of imbalance, where one or more nodes contains either too few (less than *order* elements) or too many (more than $2 \cdot order$) keys.

2. It decouples the update operations from re-balancing: An insert operation will leave a fully updated leaf, and let the re-balancing process take care of it instead. This allows for simple implementations of update operations, while collecting all the complex re-balancing logic in one place.

   In the implementation, a `modifyLeaf` function for locating and modifying a leaf is used to unite logic from `insert`, `modify`, `delete` and `lookup`. For example, `lookup` is implemented as[5]:

   ```
   lookup key = modifyLeaf (findChild key) $
     \_ _ _ values -> return $ M.lookup key values
   ```

   The function `findChild` is used by `modifyLeaf` to locate the correct leaf. When found, the key is looked up in the `Map` describing the leaf.

To avoid working directly on the persistent storage, the implementation is built using three structures:

1. A transactional hash table, used to cache nodes in-memory.

2. A transactional cache that combines said hash table with a persistent storage (e.g. a hard disk).

3. Finally, the concurrent b-tree that uses the cache for storing nodes. Active tree nodes are held in-memory in the hash table. Updated nodes are periodically synced to persistent storage, and flushed out of memory.

.

Caching nodes in-memory improves performance, but it is also a necessity for working with Haskell's STM library which cannot be combined with IO operations.

4.3.2 Concurrency in Haskell

We use the "software transactional memory" (STM) [86] model for concurrency. In this model, a computation can be run as a transaction – either completely and without interference from transactions running in other threads; or not at all.

In Haskell, STM is implemented using an optimistic strategy [50]: The transaction is run with no checks. When ready to commit, the STM system verifies that no other thread has modified the active state and commits the new state. When unable to commit, the transaction is safely rolled back.

To guarantee safe rollbacks, all transactions must run inside the restricted STM monad. The main advantage of this model is the ability to compose STM actions safely. If `a` and `b` are safe STM actions, so is their composition `{ a; b }`; this is *not* the case for *explicit locking* [54].

See Discolo et al. [38] for a simple queue example using STM.

[5]The other operations are slightly more complicated, as they need to guide the re-balancer.

4.3.3 STM and IO

The restricted STM monad does not permit IO actions. However, the indices will eventually need to reside on persistent storage, and thus we need to execute IO actions.

To get around this limitation, our transactional computations can either:

1. Complete and return the expected result.
2. Return an IO action that needs to be performed. The IO action is performed and the transactional action is retried until successful. These IO actions will usually mix IO and STM actions, such as fetching a node from persistent storage (IO) and storing its value in the cache (STM).

When a transaction returns early due to needed IO, it is *not* rolled back. Rather it is committed with the IO action as the result. It is the programmer's responsibility that this is safe with respect to the STM state (`TVar`, `TChan`, etc.). This is achieved by computing actions that can request IO *before* actions that affect critical state. For example, when splitting a leaf, it is important to create the two new leaves *before* storing any of them in the cache (otherwise the cache may store unreferenced nodes in the persistent storage).

We denote STM computations that can be rolled back without side-effects as *pure* STM actions. Any STM computation that can return an IO request is thereby an *impure* STM action. Note that this affects composition of STM actions: A pure STM action composed with an impure STM action will yield an impure STM action, and thus it is now the programmer's responsibility that composing a sequence of STM actions is safe.

4.3.4 Transactional hash table

The hash table used for storing the cache is implemented as pure STM. It has no IO parts and can therefore never trigger an early return. It is implemented as a fixed size `Array` of buckets; each of which resides inside a `TVar`. Having one `TVar` per bucket allows several threads to modify separate buckets concurrently without conflict. Having a fixed number of buckets simplifies the logic and allows the bucket `Array` to reside outside of a `TVar`, since the value is never modified.

A bucket is a regular `Map` from keys to values. Keys are distributed evenly among the buckets by hashing. `Map` allows the table to scale well, even with few buckets and many values and further avoids unnecessary exploit risks [35, 58].

4.3.5 Transactional cache

The cache is implemented as pure STM computations where possible. But since the cache is the link between in-memory storage and the persistent storage, not everything can be pure STM.

For example, when a requested key is not present in the cache it will need to be fetched from the external storage. This requires an IO computation and hence the `fetch` function may return with an IO request. On the other hand,

`store` and `remove` are both written as pure STM, since these are ignorant to the previous value of the key they are changing.

The cache takes care of fetching values from external storage transparently, just as it takes care of retrying operations that request IO actions. The latter functionality is implemented directly in `runCache`; an IO action.

It does this while remaining back-end agnostic (back-ends are defined as an instance of our `KVBackend` type-class) and while providing a mechanism for consistently syncing the updated values in the cache to external storage (to ensure crash safety).

Consistent sync. To restrict memory usage, the values in the cache can be periodically synced to the persistent storage and flushed from memory by a background process. When a value is synced, it changes state from a *pending update* (`Write`) to a *cached value* (`Read`).

Only values that do not represent pending updates are flushed – these are the values that are in sync with the external storage and are therefore safe to flush.

However, there is a problem with syncing a tree while it is being updated: What happens when a thread updates the tree during a sync process? We want the result of a sync to always yield a consistent tree in the persistent storage.

One way to do this, is to lock the tree from updates while syncing it. This would be inefficient, since syncing requires expensive IO operations (like disk writes and syncs). Instead, we want to run these expensive IO operations in the background while still using the CPU for other operations.

In order to provide a syncing facility that can run concurrently with updates as well as maintain a consistent state in the external storage, we have implemented the following two mechanisms:

Generations: The ability to freeze the current state of the cache. The frozen state can then be queried concurrently with updates to the newest state. This is implemented using copy-on-write and only supports *one* frozen state. When the frozen state is no longer used it is released for garbage collecting by the flush process.

Sync dependencies: The ability to define the write order used by the sync process. This allows updates (`store` and `remove`) to include a key, specifying some value (e.g. a parent node) that must be written *after* the updated value. This allows the user of the cache to define a dependency graph of values, where if $A \rightarrow B$ then A is synced before B. Values with such dependencies are synced before values without, and thus the process becomes:

1. Values with dependencies are ordered and updated according to the provided guidance (in the b-tree, these are the nodes whose value has changed).

2. Values with no guidance are updated (in the b-tree, these are the deleted nodes).

The b-tree implementation uses this to achieve bottom-up syncing, by including an updated node's parent as the dependency. This ensures that the node is written *before* its parent. When deleting, the dependency tag is omitted. This ensures deletion *after* writing, which means the worst that can happen in a crash is leaving garbage on the disk (which could be cleaned up later).

Sync is implemented by iterating over all values in the tree and syncing the updated ones. This could be done more efficiently by adding updated values to a queue as they are modified. Since syncing occurs only rarely, we consider this a low priority optimisation.

We have now established a concurrent hash table, and a concurrent cache. In the following section, we describe how they are used to implement a concurrent b-tree.

4.3.6 Concurrent B-Tree with relaxed balance

A k order b-tree is a tree that allows up to $2k$ keys in each node. Leaf nodes consist of keys and their respective values, while branch nodes (internal nodes) are used solely for routing [32]. Keys are ordered to enable efficient searching.

One of the main advantages of having multiple keys in each node is that the *fat* nodes enable efficient read and write operations when the b-tree is used as an external data structure. Here, it is convenient to read and write more keys at a time as a single sequential operation.

We have chosen to implement the concurrent b-tree described by Larsen and Fagerberg [59]. The tree uses relaxed balancing: it will experience periods where it looses balance, but will eventually regain it. The motivation for allowing this is higher concurrency; update operations can be reduced to updating leaf nodes only. This reduces the need for locking and thereby congestion.

The balance is restored by a special background re-balance process. This process is responsible for merging and splitting branch nodes, as well as merging small leaf nodes. Figure 4.2 and figure 4.3 on the following page give two examples of this process.

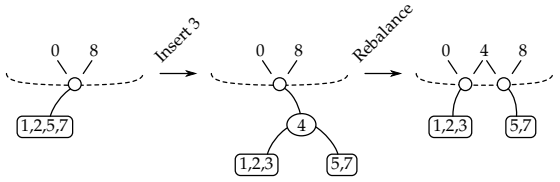

Figure 4.2: B-tree example: Inserting the key 3 overflows the leaf (order is 2), which is first split by the *insert* operation and then merged back into the parent by the *re-balance* process.

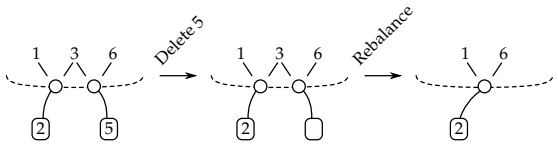

Figure 4.3: B-tree example: The *delete* operation results in an empty leaf, which is later cleaned by the *re-balance* process.

A pleasant side effect of decoupling re-balancing from queries and updates is simpler operations and improved reuse of logic. Finding the right path from the root node to the leaf is shared between `insert`, `modify`, `delete`, and `lookup`.

As a consequence of concurrency, routing must be handled specially: The re-balance process may merge two branch nodes into one, just when the insert operation was about to read one of them. Thus, routing may fail.

Though a rare occurrence, this case needs special treatment. Our solution is to simply retry the operation n times[6]. If the operation still fails, something must be wrong (e.g. the tree is in an invalid state) and the exception is re-thrown. Due to the decoupling of re-balance logic and tree operations, this routing logic is shared among all tree operations.

Iteration. For safe iteration in a concurrent environment, the cache feature "generations" (see section 4.3.5 on page 62) is used to freeze the current version of the b-tree. Thus, `toList`, `foldli` and `foldri` can all be used while updating the tree concurrently. In the prototype however, only `toList` is implemented to support this.

An alternative way of implementing iteration is to add a reference from each leaf node to the next. This would require some extra book-keeping, but may yield a performance gain in case of intensive iteration. We have chosen to simply freeze the tree, since this feature was already present and used when syncing.

Consistent sync. To maintain a consistent state of the tree in the external storage, it is important to never write out a node before the children it references. This is achieved with bottom-up syncing, using sync dependencies in the cache (see section 4.3.5 on page 62). Whenever a node is updated with a new value, a sync guide is included to tell the cache to sync this newly updated node *before* its parent. When removing a node, no guide is included. The result is a two-pass sync: *a)* nodes are updated from leaf to root; and *b)* nodes no longer referenced are removed.

Since all writes are done atomically[7] this guarantees a consistent tree. In case of sudden crash, the worst scenario leaves unused nodes. The tree will remain valid.

[6]Here, n is an arbitrary large number compared to the height of a b-tree. A limit is required in case the tree is actually corrupted.

[7]Atomicity is achieved by first writing the value to a temporary file and then moving this file onto the final location.

Limitations of the implementation: We have not implemented all of the required b-tree logic. Specifically, we have accepted the following shortcomings:

1. Re-balance logic to handle empty leaves is missing. There is currently no logic in the re-balancer to merge under-full nodes.
2. Nodes in need of re-balancing are only discovered when created. This works as long as the tree is in use, but if it is synced to disk and restored later it may stay out of balance forever.
3. `findMin` and `findMax` are incomplete. Current implementations of these functions only work partially: they fail in the non-trivial case, where the value queried is not located in an outermost leaf. This case emerges after deletion and is never resolved due to item 1.

   If the re-balancer could handle deletion, these functions could simply tell it about the empty leaf and retry from the root until succeeding.
4. `foldli`, `foldri` and `search` may fail if run in parallel with a re-balance process, but `toList` is safe since it is using the generation system to freeze the tree.

Item 1 and Item 2 is a matter of implementing the required functionality. There are no limitations in the design to prevent it.

Item 3 is trivially solved when the re-balancer is complete: when the troublesome case occurs, simply tell the re-balancer about the empty leaf and retry from the root of the tree.

Item 4 could be solved by freezing the b-tree in all these functions (using the generations feature from the cache) like with `toList`. However, such operation can be expensive in terms of memory, since all changes during the iteration have to be copied. Another solution is to change the leaf structure to include pointers to the previous and next leaf. This would introduce further complexity when splitting a leaf as well as in the re-balancer. But it is unlikely to hurt performance, since re-balancing and splitting are rare actions.

4.3.7 Locality

As discussed in section 3.2.2 on page 23 we use the system to save its own log data. To improve inter-blob locality of saved b-trees we save them according to the Van Emde Boas layout [92].

4.3.8 Summary

Above, we have presented a concurrent b-tree with relaxed balance, implemented in Haskell using software transactional memory. To summarise, the implementation supports:

Isolation: The operations `insert`, `delete`, `lookup`, `modify` and `toList` all execute atomically and in isolation, without interference from other threads.

Consistent sync: When the b-tree is stored on persistent storage (e.g. a local disk) it is stored bottom-up – from leaves to root – to ensure consistency in the event of failure.

Online re-balancing: A re-balance thread takes care of re-balancing the tree, and does so concurrently with other operations.

Back-end agnostic: The back-end used for persistent storage can be changed. In the prototype, the b-tree is used both with a local back-end (when taking a snapshot) and with an external back-end (when inspecting a snapshot).

4.4 Crash recovery

As mentioned in section 3.7 on page 43 we distinguish program crashes where data in memory is lost, but data written to disk is safe (e.g., power failure), and the more rare disk crashes where data stored on disk is lost.

4.4.1 Program crash

In the event of a program crash, the local log data is what the program managed to store before the crash, and thus the key index and hash index may be out of sync. Thus we must be able to bring the indices back to a consistent state, as well as remove blobs that have been transferred to the back-end but whose references have not yet been committed to the hash index.

Upon beginning a snapshot, two directories named `rollback` are created in the hash and key indices' directories. They act both as a marker to show that the system is in the process of taking a snapshot, and as a place to save log files to allow the system to recover itself to a consistent state.

To aid recovery in the event of a crash, several log files are maintained. To get an overview of the measures we take, let us follow the data flow from a file being backed up and till it is on the back-end.

`KeyStore`. When a file is inserted, the first thing that happens is that its path is appended to a log in the key index's `rollback` directory. This makes it possible to later track down files of which not all chunks were registered. The file entries are logged in files in the `rollback` directory, before entering the `KeyStore` processes.

To lessen the penalty of syncing the logs to disk, we log many files at once. This is possible because the files that shall be backed up are known in advance. The number of file paths to write each time is chosen to be 128. Benchmarks suggest that this is a good number (see section A.2 on page 98).

`HashStore`. When a chunk is inserted, it is not added to the hash index right away. It first resides in an in-memory cache as a "promise" until the blob containing the chunk is safely stored on the back-end. A callback

function to be triggered when this is certain, is created to insert the chunk's hash and blob position into the hash index, removing it from the in-memory cache.

`BlobStore`. Chunks and their respective callbacks are grouped into blobs. When a blob is full and ready for transfer, a log file is written to the hash index's `rollback` directory. The name of the file reflects the ID of the blob, while the contents describe a list of the hashes whose chunks are in the blob. A single combined callback function is created and sent along with the blob to the `External` process.

`External`. Only after a blob has been stored safely on the back-end, the final callback function – and thereby all the callbacks from the hash index – is evaluated. The effect goes all the way back to the `HashStore` which commits the relevant chunks to the hash index and removes them from the in-memory cache.

Thus we know that chunks that are referenced by the hash index have been uploaded.

Now that we have seen the elements of the crash recovery strategy, we show how the system recovers from a crash. We are concerned with three things: *a)* unreferenced blobs on the back-end (those that are not mentioned by the hash index); *b)* chunk hashes that reference non-existing blobs; and *c)* entries in the key index that reference chunks that have not yet been transferred.

These are the only structures involved in taking a snapshot, and so it is enough to ensure that they are brought back to a working state.

a) We need to remove blobs that were transferred, but whose chunk hashes are not yet in the hash index. We can do this by running through the log files in the hash index's `rollback` directory. These files give us the blob IDs along with their respective hashes. If one of the hashes is in the hash index, the blob has a reference and is kept; if not, it is deleted from the back-end.

b) Because chunk hashes are kept in memory until they are certain to have been transferred to the back-end, it is impossible for a chunk hash in the hash index to reference a blob that does not exist on the back-end.

c) Log files show the keys inserted in the `KeyStore`. In the event of a crash, we check for each one whether all the referenced chunks is in the hash index. If any chunk is missing, the file is deleted from the key index. Notice however, that due to deduplication the missing chunks only will be transferred when the snapshot operation is resumed.

Flushing log files. The observant reader will notice that we have not discussed removal of log files, but surely this is needed to prevent the system from rolling back all its progress. To remove individual file entries from the log files in the key index's `rollback` directory, we need to be able to tell when

all chunks from a particular file have been saved, and then delete the file from the log.

To prune the log files in the hash index's `rollback` directory we must be able to tell when the hash index is safely stored on disk and which "in flight" hashes it contains. One approach is to force syncing of the hash index to disk for each blob that is transferred. This would ensure that the blob's chunks were referenced, and the log file could be removed. But this has a large impact on performance.

If we could know the progress of each chunk throughout the system, we could prune the logs precisely, however this holistic worldview does not fit well with our process model where each process has limited knowledge of the system. And so we have not chosen this option.

Our solution to both problems is to periodically flush the whole system (by sending a flush message, see section 4.2.1 on page 57). This action is similar to what happens when the snapshot has finished and the system is shut down, except in this scenario the system continues the snapshot afterwards. During flushing all data is forced out of the system to the back-end, and the indices are synced to disk.

After flushing all the log files can be deleted. So the worst case scenario is that the files inserted between flushes have to be processed again; however the stored blobs that are referenced from the hash index need not be stored again.

As of such, our worst case scenario yields performance comparative to other backup systems' best case.

4.4.2 Disk crash

Above we assumed that no data was lost from the disk. Now we see how to recover from a crash where data stored on the disk is lost. Since we use checksums on all external data structures, we can detect damaged data, such as bit-rot, and treat it as lost.

As discussed in section 3.7 on page 43, it is necessary for Hindsight to save its log data periodically.

When a snapshot is taken (with the `snapshot` command) its key index is backed up as described in section 3.2.2 on page 23, leaving the hash indices (one primary and one secondary – section 3.2.2) locally.

The `seal` command takes care of this. First the primary hash index is backed up in a secondary run; a regular snapshot is taken, using the secondary hash index for deduplication, and the resulting key index is saved as a tarball. Then the updated secondary hash index is saved, also as a tarball. References to these two tarballs are saved in the snapshot index under the special names `__pidx` and `__sidx` (for primary and secondary index).

The snapshot index is saved as a tarball under the special name `snapshots` as usual.

If a disk crash happens, it is enough to reconstruct the hash indices; the

next snapshot might take longer to complete because the head has been lost, and so the full key index must be constructed, and every file inspected. But that is only a (very rare) performance issue.

The command `recover` is used to reconstruct the hash indices. It performs the following steps:

1. Download the snapshot index tarball (saved under `snapshots`) and unpack it.
2. Use the snapshot index to retrieve the tarballs of the secondary hash index and the key index resulting from the snapshot of the primary hash index. Unpack them.

   We now know the files needed by the primary hash index (from the key index), as well as their positions within blobs (from the secondary hash index).
3. Use the two indices just retrieved to checkout the primary hash index.

Of course, everything backed up since the last `seal` command is lost. Furthermore all references to blobs transferred in that time are lost. This means that it is possible to leave "dead" blobs on the back-end in this rare event.

If the back-end API is extended with a `LIST` command, we can find and delete these blobs, but since the case in which we leave them there is so rare, we have decided against it; at least in the prototype.

4.5 Back-end modules

In order to ease the process of implementing support for a new back-end, we have decided that such functionality does not need to be in Haskell, like the rest of the implementation. This leaves the choice of programming language to the developer implementing the module and hence supports the idea of using "the best tool for the problem".

We have implemented support for four different back-ends using two different languages, none of which is Haskell:

`bash` scripts: The modules for local storage, SSH and Amazon S3 are all written as `bash` scripts. The Amazon S3 module invokes the `s3cmd` CLI tool.

Python: The module for CouchDB is written as python scripts that rely on the Python `couchdb` library.

To implement a module, one has to write three programs, all of which take the relevant blob name as their only argument and exit with either return code zero (success) or non-zero (failure):

`get`: retrieves the requested blob and writes its data to `stdout`. If the blob does not exist, the program exits with a non-zero return code.

Example: `get B4X62bMOaPqIMHhKiirc70 > data`

put: takes the blob data on `stdin` and puts it on the back-end. If the file exists, it is overridden. If the command exits with return code zero (success), it must be possible to retrieve the data with `get`.

Example: `put B4X62bMOaPqIMHhKiirc70 < data`

del: deletes the blob. If the blob does not exist, the program must ignore the command and exit normally, with return code zero.

Example: `del B4X62bMOaPqIMHhKiirc70`

Back-end modules are ignorant of the encryption and authentication used to protect the data. All this is applied *before* the module ever sees the data.

An advantage of implementing back-end modules as small programs is that one can use the language that provides the best bindings, or merely shell scripting when a CLI tool is available.

A disadvantage is that if the best bindings happen to be present in Haskell we still have to call programs and make sub-processes. However, we do not think that this overhead is significant compared to transferring data to the back-end.

Furthermore, the described API does not show how to handle errors. If something goes wrong, the program can exit with a non-zero return code, but it cannot inform Hindsight of what the error was. As a result, there is currently no way of distinguishing a "blob does not exist" error from a "no network connection" error.

We leave designing a more expressive framework for implementing back-end modules as a topic for further work (see chapter 6 on page 87).

4.6 Deletion

To reclaim data that is no longer needed by the live snapshots, the prototype implements the conservative garbage collector described in section 3.6 on page 33. It operates in four phases:

Post snapshot: After a snapshot has completed, a Bloom filter is generated to represent the hashes referenced by it.

The Bloom filter is compressed and saved on the back-end along with the key index.

GC init: Old versions of hash indices are deleted from the back-end (there is one for each `seal` operation), leaving just the newest versions.

GC mark: The purpose of this phase is to mark the dead hashes for deletion. This is done in two steps:

1. Retrieve the Bloom filter for each live snapshot.
2. Check for each hash whether it is present in one of the Bloom filters. If not, mark it as dead.

GC sweep and compact: This phase reclaims space based on the dead hashes from the mark phase. It consists of two steps:

1. Locate the blobs that only dead hashes refer to. These blobs are deleted along with the hashes referencing them.
2. Locate the blobs that only *few* living hashes refer to. These blobs are retrieved and merged to form new blobs, with no dead data in them.

Splitting the mark and sweep phase in two allows us to reuse the sweep-compact implementation with other garbage collectors' mark phase, if the prototype is ever extended with more collectors.

In the following sections, we discuss crash safety and the expected performance of our conservative GC. In chapter 5 on page 77 we evaluate it through a series of tests.

4.6.1 Crash safety

Each step of the GC mark and sweep routines are idempotent; if the process is aborted, it can be safely restarted. However, the implementation does not yet ensure that no blobs are not left on the back-end without reference. This can happen if the process is aborted while rewriting small blobs into larger ones. To solve this, a recovery procedure should be implemented.

Further, the implementation is not safe against hardware failure: if the disk crashes, the updated hash index is lost, along with the references to new blobs introduced by rewriting. To make the process safe, one would have to seal the updated hash index *after* writing the new blobs, but *before* deleting the old (thus rendering both the new and old hash index valid, in case of a crash).

4.6.2 Filter configuration

The configuration garbage collector uses Bloom filters to identify live hashes, which means not all dead hashes can be found. Thus it is important to configure the Bloom filters to yield an acceptable accuracy in live hash identification, while not using too much space.

Our conservative GC uses 4 hash functions and 16 bits per element (chunk hash). A Bloom filter with this configuration, covering the hashes referenced by the key index, is added each snapshot. This gives each filter a false positive rate, in terms of chunk hashes, of:

$$p \approx (1 - e^{-4/16})^4 \approx 0.002394$$

Adjusting this, by the number s of filters – one per snapshot – gives a *chunk reclaim rate* of

$$r \approx (1 - p)^s$$

This configuration of the GC can reclaim 99.8% of the dead chunks, with one snapshot, 95.3% with twenty snapshots and 78.7% with one hundred snapshots. The 50% mark – where only 50% of dead hashes can be marked as such – is reached at 289 snapshots.

This reclaim rate is per snapshot, and so the total amount of garbage will accumulate when more snapshots are deleted. However, the reclaim rate improves with the total amount of garbage, since garbage from older deleted snapshots (which gave false positives) can also be reclaimed. We investigate this effect during evaluation in chapter 5 on page 77.

Our configuration is inspired by Mitzenmacher [67], who proposes the use of fewer hash functions to allow for better compression ratios in the stored filters, giving a lower false positive rate per stored byte. This could also open up for exploitation of redundancy across filters from the same snapshot family. Our prototype does not yet exploit such redundancy, and simply compresses the filters like any other data.

We have identified a problem with using varying Bloom filter sizes in this kind of GC: If we start with some small snapshots (with a small number of chunks) we get small Bloom filters, which is fine due to the small number of chunks. If we then introduce a lot of new data in a new snapshot, that snapshot gets a significantly larger Bloom filter, so it can identify its many chunk hashes accurately.

But all filters are used by the GC to identify which chunks are alive, and so the small filters introduced by the first snapshots are now used against more hashes than they were designed for, thus yielding a smaller reclaim rate than expected.

In the implementation, this problem is *relaxed* by having a minimum filter of 16.000 bits and by doubling the Bloom filter sizes on each resize. This gives most filters some extra room for expansion.

4.6.3 Merging Bloom filters

Bloom filters of equal length can be trivially merged by *or*'ing their bit-vectors. This gives a new Bloom filter of the same size, but containing elements from both filters. The advantage is that checking each hash against the filters can be done using a single filter which is faster. But the false positive rate has increased – the size is the same, but the number of hashes is the sum of that from the two merged filters – so less chunks will be reclaimed.

To stay effective, the GC in the prototype does not merge filters. However, a merged filter could be used as a heuristic in front of the full filters to check whether a hash has any chance of being alive. If the snapshots share most of their hashes, their combined hashes will not be that many extra elements, and the combined filter's false positive rate not that much higher. In this case, this heuristic may prove effective.

4.6.4 Compacting

Since the mark phase can only identify some of the dead hashes, it is important that the sweep phase can reclaim space, even when blobs are only partially dead.

If a blob contains h hashes, its probability of being completely reclaimed is just r^h, where r is the reclaim rate from section 4.6.2 on page 71. Thus if 1000 hashes reference dead chunks stored in the same blob, there is a 9.1% chance that all the hashes will be marked as dead and thus a 90.9% risk that one of them will remain alive, and that the blob cannot be deleted[8]. But on average, 997 of the 1000 hashes will be marked dead, and this space can be reclaimed by rewriting the blob.

The way this is implemented in the prototype is to check whether a blob looks *small*. Only blobs that are certain to contain fewer chunks that what a blob has room for (compared to the maximum chunk size) are rewritten. The prototype cannot discover all small blobs, since it does not maintain any chunk size information locally and thus cannot compute how big a part of a blob is in use without downloading it.

In the prototype, rewriting blobs are done after the garbage collector has run, and is thus performed *eagerly*. An alternative method is employed by Cumulus [93], which never rewrites blobs (called *segments* in Cumulus). Instead hashes in blobs with low utilisation are marked "do not use". If a new snapshot needs to reference a chunk whose hash is "do not use", it must store the chunk on the back-end as if it was new. If old snapshots are continually deleted then blobs with low utilisation will eventually perish.

This method has the advantage of rewriting the small blobs in a way that preserves locality. The live chunks are written with other chunks referenced by files that contain them.

The disadvantage is that small blobs are only rewritten if new snapshots reference the chunks inside them.

The prototype rewrites small blobs eagerly, as they are discovered, as a simple yet efficient strategy.

4.6.5 Garbage collection of log data

Since the primary indices are deduplicated through a secondary instance of the system, the same garbage collector is used. The secondary indices are saved as tarballs and can be deleted directly.

There is one extra concern: When the `seal` command is run, a new snapshot of the primary index is taken. Before garbage collection, every snapshot but the latest is deleted in that family.

4.7 Security

We want the prototype to be safe, yet we feel we are not fully qualified because: *a)* it is not the main focus of the project; and *b)* plenty of examples exist where implementing cryptography logic has gone wrong in spite of being the main focus. Here, we list some such examples:

[8]The statistics presented in this paragraph assumes a single live snapshot. The risk is higher with more snapshots.

AES: Advanced Encryption Standard [9] – have several implementations that suffer from timing attacks [15].

Google Keyczar: A high level crypto library – suffered from a timing attack when comparing hash values [60].

OpenSSL: An open source implementation of cryptographic routines – had a timing attack in their RSA implementation [70].

SSL: Secure socket layer used in HTTPS – was vulnerable due to a flaw in the renegotiation logic [80].

Tarsnap: Secure backup – didn't use unique nonces (after a refactoring of the code) which rendered the encryption insecure [74].

We understand that getting security right is extremely difficult. And even when encryption works correctly it is not possible to verify the security guarantees.

In Hindsight we have a very small and simple crypto-system. We want encryption and authentication of all stored data. No more; no less. But we do not trust our ability to implement cryptography logic securely. Instead, we minimise the amount of code that can impact security by using the high-level crypto-library NaCl [19] (pronounced *salt*) and by limiting the number of processes that know about cryptography.

4.7.1 Fingerprinting

For chunk fingerprinting, the prototype uses 256-bit Skein hashes [44]. Skein is one of 5 finalists in the SHA-3 competition [69], and we use this algorithms solely as a placeholder while waiting for the new standard (the SHA-3 winner).

As with the AES selection process, SHA-3 is another year-long selection process, this time with the goal of finding a secure hash function to replace SHA-2 as the new standard. During the process, cryptographers world-wide collaborate in an attempt to break the submitted candidates. The process started with 64 entries in November 2007 and is now down to 5 finalists: BLAKE [11], Grøstl [48], JH [95], Keccak [20] and Skein [44].

4.7.2 NaCl: a high level crypto library

NaCl gives us a high level `encrypt` function which handles encryption and authentication given a key and a random nonce. NaCl is designed to be easy to use, secure and fast [18, 19]. It uses standard algorithms with high security margins. Specifically, the NaCl-routines called by our prototype uses Salsa20 with 256-bit keys for encryption and poly1305 for authentication. The eSTREAM project showed some confidence in the 12-round Salsa20/12. NaCl is conservatively using the 20 round variant XSalsa20/20 that uses a nonce of 192 bits instead of Salsa20/20's 64 bits [17, 19]. The authentication algorithm, poly1305, was originally published with a security proof depending on the security of the underlying encryption (here XSalsa20/20) [16].

NaCl and Google Keyczar [60] seems to serve the same purpose. We went with NaCl due to its experienced authors and its fine Haskell bindings. Keyczar does however have the advantage of built-in versioning of the configuration used. This allows backward compatibility when outfacing an algorithm. We have to maintain this ourselves when using NaCl.

On the other hand, NaCl provides asymmetric encryption (based on elliptic curves) in case the prototype needs to be extended with this at some point.

4.7.3 Crypto-aware parts of the prototype

Due to the process model used in the prototype, we can point out the specific processes that are aware of cryptography. These are:

Initialisation: The code that initialises a new repository needs to generate a new client key and store it locally.

External The process that communicates with the back-end needs to encrypt and authenticate all data before transmitting it.

Since the logic for communicating with the back-end is only available through the external process, this is the only place that needs to perform encryption and authentication.

We wrap the Haskell `salt` library in our own very small crypto library. This gives us stricter typing on nonces, which were originally of type `bytestring`. In addition, we have a very simple API for maintaining the client key:

`newMasterKey path`: Creates a new master key and writes it to the specified file.

`readMasterKey path`: Reads the master key from the specified file.

There is no way of modifying the key outside this module. This makes it less likely to accidentally swap the secure key for something predictable.

4.7.4 Lacking features

The prototype does not support the master key indirection. Instead, the master key is stored locally as part of the Hindsight repository (where the client key should be stored). There is nothing in the prototype that prevents implementation of this.

The prototype does not store information about the cryptographic scheme used. To implement this, one would have to add a version to all blobs. We believe a single byte would prove sufficient, since their are usually many years between new versions of cryptographic standards (the widely used Advanced Encryption Standard is from 2001).

4.8 Iteratee

Before settling on the process model (described in section 4.2.1 on page 57) we looked at the iteratee and enumerator packages for Haskell.

The iteratee package by Kiselyov and Lato [57], introduced an abstraction for efficient and composable data processing based on left folds. Though a conceptually simple approach, the library was more than just difficult to understand, which is seen by the number of questions it got on the Haskell Cafe mailing list from frustrated developers.

Millikin [66] tried to clean things up with the enumerator package, in an attempt to embrace the community. He succeeded, in that this package is now more used than the original (29 versus 11 packages on Hackage), but still the types and terminology gives usage of this package a steep learning curve.

When we started our project, enumerator was the most popular package for this kind of thing. So we felt it natural, to investigate whether we could implement Hindsight's data flow within this model. We thought about having an enumerator for each step of the process, as a data pipeline from left to right:

file reading → chunking → blob store → external storage

But we ran into some issues, that we never managed to solve:

- How can we handle concurrency, so that more than one chunk can be processed at the same time?
- How can we maintain global state, as needed by b-trees in the form of caches?
- How can we handle bi-directional information flow, as when the chunker needs to ask the hash index whether or not the chunk's hash is known?

Instead, we found that the Erlang-style process model, employed by the prototype, allowed all this, and in a much simpler way. And it even contributed with some things that the enumerator library may not have: A simple way to collect statistics, a global means for error handling and automatic throttling due to limited mailbox sizes (see section 4.2.1 on page 57 for details).

During this project, the authors of the Yesod Web Framework [6] released the conduit package [88]; yet another approach to efficient stream processing. While it implements roughly the same concepts as the other two packages, it does so with a different approach. We have not carefully examined whether this package makes it easier to implement something like Hindsight.

What we have done, is to use the conduit package for one of the iteratee package's original purposes: Strict IO. Even though the Hindsight prototype handles arbitrarily large files, it does not rely on lazy IO, which is the classic way of doing this in Haskell. Instead, it employs the conduit package as a means for efficient, yet encapsulated, strict IO.

Chapter 5

Evaluation

Our original idea for the design of the Hindsight backup system was a lot simpler than what we have presented in the previous sections:

> We wanted to chunk files and store the chunk hashes in b-trees. The unique chunks were to be grouped in blobs – to lower the number of transfers – and sent to the back-end.
>
> We were aware, that deletion *should* be possible and that b-trees *could* allow lazy querying. We also wanted a simple abstraction for back-ends, and knew that the needed operations were few.

While this is still the foundation of the current design, a range of additions have been made since we started.

Security and compression was trivial to add in.

Crash safety with recovery was added by making the b-trees robust with bottom-up syncing and log files were added to aid recovery.

Deletion was something we always thought possible, however exactly how it could be done was under continuous discussion (especially regarding crashes). The current conservative garbage collector was implemented as an experiment, because we could not find any other system using this method. As it turns out, it works quite well (though not in all circumstances).

Deduplication of log data was a feature we got for free by invoking the system recursively; something we discovered gradually through the design phase.

Back-end agnosticism was gained through *back-end modules* with minimal requirements.

Statistics and progress is a pleasant feature for impatient users. It was a quickly introduced feature due to the process model used in the implementation.

We think this shows that the design is at least somewhat flexible: whenever we have found a problem, or wanted to add a feature, it was never too hard to add in. Of course, every change takes time of thought and consideration, but the design itself has not yet resisted.

5.1 Comparison

To get a feel for the rôle Hindsight fills among backup systems we give an overview of existing systems. In table 5.1 on page 80, we compare our system, Hindsight, with eleven other systems.

We compare the systems on the following points, inspired by the properties listed in the analysis (section 2.2 on page 12).

Multiple snapshots. Whether the system can keep several versions of the stored data. Multiple snapshots can be emulated by explicitly storing several versions of the same data in the same snapshot, which is what S3QL [79] does. We will not regard this technique as "multiple snapshots", because of the problems of copying the data and marking it read-only.

Simple back-end. We regard a back-end as "simple" if it can do its job without any semantic knowledge of the data it stores. A regular key-value store is an example of a simple back-end. We discussed back-ends in section 3.9 on page 51.

If the system uses a simple back-end, it is easier to extend it to support new back-ends, thus making the system more flexible.

Sub-file deduplication. To avoid storing the same data over and over all backup systems apply some form of *deduplication*. The granularity of deduplication is important for its effectiveness.

We classify deduplication scope and granularity:

L/file (Local scope and file-level granularity): Files are only compared with their former version, and if they have changed they are stored in full.

L/sub (Local scope and sub-file granularity): Files are still only compared to their former version, but only changed parts are saved.

G/sub (Global scope and sub-file granularity): Sub-file data blocks are deduplicated across all files and snapshots.

Fingerprint size. The size of the used fingerprint. When using deduplication, data objects must be compared. For performance reasons, it is a common practice to compare objects by their *fingerprint*. The fingerprint function is usually a cryptographic hash [43].

Since a fingerprint has a fixed size, which is almost always less than the size of the data it identifies, there is a risk of collisions. The larger the fingerprint, the lesser the risk.

Content-aware chunking. How a file is split into chunks (which implies sub-file deduplication) also decides the effectiveness of deduplication. Content-aware chunking as opposed to fixed-size chunking gives better deduplication in some cases. Refer to section 3.5 on page 28 for details.

Using blobs. Whether the system collects chunks into blobs. Two marks (✓✓) are given when blobs are locality-preserving.

Storing many objects on the back-end can be problematic for several reasons: *a)* for each object sent over the Internet there is an overhead; and *b)* storing many objects can be more expensive than storing few (for example, this is reflected in the pricing model for Amazon S3 [85], in which a small amount is charged for each request). To keep the number of stored objects low, they can be collected into *blobs*.

If files are split into chunks (implying sub-file deduplication) it can be beneficial to preserve the locality of chunks in the blobs; for example, the amount of blobs which must be downloaded to reconstruct a file will be low if most of the chunks in the blobs belong to that file.

Security. Whether the actual data stored on the back-end leaks critical information about the data being backed up. Here, we do not count the time of backup or the amount of data stored as critical information. See section 3.8 on page 47 for details.

Log data deduplication. In all but the simplest backup solutions there will be some form of *log data*; data giving structure to the data which is backed up (e.g. which chunks came from what file?). It is thus crucial to preserve log data, for without it the original data is hard or impossible to retrieve. The "Log data deduplication" column tells whether deduplication is applied to lower the overhead of the log data when storing it.

Full snapshots. Whether each snapshot reflects the full system being backed up, or only the parts of it that have changed since earlier snapshots; that is full snapshots versus incremental snapshots.

Deletion. Whether the system supports deletion of snapshots. Supporting deduplication complicates deletion of stored objects (see section 3.6 on page 33), and thus not all systems supporting deduplication supports deletion.

Transparency. A measure of how easy it is to verify the functionality of the software.

We give marks for each of these points:

1. Technically well documented, such as research and white papers. Of course the word "well" implies a subjective judgement on our part. We disregard user documentation, such as `man` pages.
2. Open-source software. An advantage is that the system can be verified to implement the claimed design. Additionally, if lacking proper documentation, one can resort to reading source code.

	Multiple snapshots	Simple back-end	Deduplication	Content-aware chunks	Fingerprint size (bits)	Uses blobs	Security	Log data deduplication	Full snapshots	Deletion	Transparency	Compression	Efficient queries
Venti [77]	–	–	G/–	–	160	✓	✗	–	–	✗	❶❷❸	✓	–
bup [72]	✓	✗	G/sub	✓	160	✓✓	✗	✓	✓	✗	❶❷$\overline{❸}$	✓	✗
S3QL [79]	✗	✓	G/sub	✗	256	✗	✗	(✓)	–	✓	❷❸	✓	✗
Duplicity [42]	✓	✓	L/sub	✓	128	✗	(✓)	–	✗	✓	❷$\overline{❸}$	✓	✗
SAM[2] [90]	✓	✗	Hybrid	✓	160	✗	✗	✗	✓	✗	❶❸	✗	✗
rsnapshot [83]	✓	✗	L/file	–	–	✗	✗	–	✓	✓	❷$\overline{❸}$	✗	✗
Tarsnap [73]	✓	✗	G/sub	✓	256	?	✓	✗	✓	✓	(❷)	✓	✗
Cumulus [93]	✓	✓	L/sub	✓	160	✓	✓	✓	✓	✓	❶❷❸	✓	✗
Amanda [36]	✓	✗	L/file	–	–	✗	(✓)	✗	(✓)	✓	❶❷$\overline{❸}$	✓	✗
Time Machine [5]	✓	✗	L/file	–	–	✗	✗	–	✓	✓	$\overline{❸}$	✗	✗
Brackup [2]	✓	✓	G/sub	(✓)	160	✗	(✓)	✗	✓	✓	❷❸	✓	✗
Hindsight	✓	✓	G/sub	✓	256	✓✓	✓	✓	✓	(✓)	❶❷❸	✓	✓

Table 5.1: Systems overview. We write "–" when a property is irrelevant for the system in question. See remarks for details.

❸ Black-box inspection. The data can be retrieved as it is stored on the back-end.

$\overline{❸}$ White-box inspection. The data is stored in an open and easily accessible format.

Compression. Whether a compression algorithm is applied to the data before storing. We do not distinguish between the various compression schemes (zlib, bzip2, lzma etc.), since changing from one scheme to another is usually a trivial task[1].

Efficient queries: Whether it is possible to query the hierarchical structure of a snapshot without retrieving its entire log data. An example of a query could be listing a directory within a snapshot.

Remarks

Venti: Venti is intended as a back-end for storage applications. It uses magnetic disks for storage. Therefore it is meaningless to classify the system by the first four points (multiple snapshots, sub-file deduplication and method of chunking are the responsibilities of the backup application and there is no back-end because Venti *is* the back-end). In Venti an *arena* roughly corresponds to a blob.

bup: bup is a back-up system based on Git [4]. It stores data in git *packfiles*. One such roughly corresponds to a very large blob (up to 1 GB). bup does not yet store any file meta-data.

[1] A trivial task in the design. Not if existing snapshots need to be migrated.
[2] Semantic-Aware Multi-Tiered Source Deduplication Framework for Cloud Backup.

A quote from the bup developers on deletion:

> Because of the way the packfile system works, backups become "entangled" in weird ways and it's not actually possible to delete [...]
>
> We'll have to do it in a totally different way. There are lots of options. For now: make sure you've got lots of disk space :)

In order to feature global sub-file deduplication, bup maintains an index of all known chunks (called midx – "merged index" we guess).

It would not be too hard to implement more efficient queries. Git (and therefore bup) saves the directory structure as separate directory listings, thus it would be possible to just retrieve the relevant listings. But if all files are stored in the root, even this would be inefficient.

S3QL: It is meaningless to talk about full snapshots because S3QL does not support multiple snapshots. A limited sort of deduplication is applied to the log data: files are referenced by an ID while the actual filenames are stored in another index. This saves space if the same files are stored many times (e.g. when simulating snapshots by creating full copies).

Duplicity: Supports full and incremental backups. Incremental backups are stored as rsync delta files. Duplicity maintains no log data. Security is supported through GnuPG, but is not the default.

SAM: A hybrid deduplication is used in SAM; Files are deduplicated on a sub-file basis in the client, and on a per-file basis in the server. Notice that Tan et al. [90] writes "local deduplication" to mean deduplication among all data being backed up from a single client, and "global deduplication" to mean deduplication among data backed up by all clients. In contrast we write "global deduplication" and "universal deduplication", respectively. See section 3.5 on page 28 for details on deduplication.

rsnapshot: The rsync algorithm is used for transfer, but deduplication is only applied at file-level on the back-end. Hard links are used to create full snapshots.

Tarsnap: We are not certain as to whether Tarsnap collects the file chunks in blobs or not. What happens after deduplication is not described. When it comes to transparency, Tarsnap has a an open-source front-end, a proprietary back-end prohibiting black-box inspection, and no technical documentation.

Cumulus: The object fingerprint hash is SHA-1 but can be replaced. A *segment* in Cumulus corresponds to what we call a blob.

Vrable et al. [93] writes that Cumulus can be extended to perform global deduplication. However, we fail to see how this can be done without some compromises:

If deduplication is not to be overly slow, fast access to the known chunks is needed. Thus a global fingerprint-index must be maintained (à la bup's midx-files).

For each snapshot, Cumulus saves a summery of the segments used by that snapshot. This summery is used to enable garbage collection. But if a global index is maintained then the chunks in deleted segments should be removed from it. Therefore a list of these chunks must be kept for each segment; this is essentially a reverse fingerprint-index.

Therefore global deduplication comes at the cost of not one, but two, global indices.

Amanda: Native system tools are used for the actual backup. For the table we have assumed that GNU tar was used. Security is supported through GnuPG, but is not the default. Amanda can be used to restore full snapshots, but using GNU tar they will be stored incrementially on the back-end.

Time Machine: The system is closed source and undocumented, but it seems to work roughly as rsnapshot, possibly with the exception of how data transfer is performed.

Brackup: Fixed sized chunking is used, with the exception of MP3-files which can be chunked based on their contents. Brackup uses mark-sweep for garbage collection (see section 3.6 on page 33 for details on garbage collection).

5.1.1 Discussion

The point of this overview is not to give a thorough survey of each backup system. Therefore, the points listed are those that can be determined from technical documentation, as opposed to reading source code and testing each system.

This is why we have left out modularity and reliability. To estimate modularity would require a deep understanding of both design and implementation, whereas assessing reliability would require testing each individual backup system.

What can be seen from the comparison is that Hindsight is one of 7 systems that apply global deduplication, and one of 4 that extend this with deletion. Among the remaining systems is Tarsnap with its proprietary back-end, the file system S3QL and Brackup which uses mark-sweep garbage collection. We list Hindsight's ability to delete in parenthesis due to its immature garbage collector (see section 3.6 on page 33 and section 4.6 on page 70), however the design does support the feature.

Further, Hindsight is the only system to support efficient queries against the file paths in a snapshot. While bup could implement this feature (as we mentioned earlier), it would be nullified in the worst case. Hindsight does not suffer from such a worst case scenario.

Not shown in the table is that only SAM, Venti and Hindsight are designed to support multiple front-ends. However, our prototype does not yet expose a

simple API. Amongst these backup frameworks, Hindsight is the only one to support security and deletion.

Hindsight is designed to handle crashes well. In particular our prototype is able resume after a crash without redoing most of the work done before the crash. In general, the other backup systems we have looked at do not pay much attention to system crashes. As a result, they cannot skip files that were processed as part of an uncompleted snapshot without data inspection.

5.2 Quality

Our prototype does not meet standard quality measures. In the following, we list some tasks that should *at least* be completed before the software can be called stable.

5.2.1 Testing

We would like to test the full code base properly. The prototype is written in Haskell, which makes the popular and well-tested QuickCheck tool an obvious choice for verifying functionality.

Currently, we use QuickCheck to test the our concurrent b-tree. Before the test runs, an empty b-tree is created and its root stored in a mutable `MVar`. An iteration of the test consists of:

1. Open the b-tree from the current root.
2. Build an in-memory `Map`, reflecting the tree.
3. Start a re-balance process, a cache-flush process and a disk-sync process that periodically saves the tree and updates the root `MVar` with the possibly new root.
4. Execute a randomized sequence of operations on both the `Map` and the tree.
5. Kill the processes spawned in step 3.
6. Compare the map with the current state of the b-tree. If the items do not match, the test has failed.

Note that the b-tree is not synced after the iteration. Instead, the next iteration must use whatever the disk-sync process managed to store before being killed. Thus the test can check whether the bottom-up sync phase works.

However, the test does not fully check the b-tree's concurrency features, since it needs determinism to be able to verify the result. One way to test this would be to make another test where the sole purpose is to try to break the tree, and not to test whether the operations executed have the expected results.

Processes: The next step in QuickCheck'ing the full functionality of the prototype could be to check all the processes through their API. This is possible by checking them one at a time, starting with a process setup of 1 and then extending it. So in order, one could write tests for: *a)* External on its own; *b)* BlobStore using External; *c)* Index on its own; *d)* HashStore using Index and BlobStore; and *e)* Hindsight's main API: KeyStore using Index and HashStore.

These tests would be able to locate mistakes in each individual process. Unfortunately, we did not have time to write them.

Back-box: Additionally, black-box testing should be used to check the code base at its highest level of abstraction. An example could be to make a series of snapshots, delete some, restore the data and verify the result.

We have written a simple black-box test to check Hindsight's recovery system. We call it the "die test". It works in three steps:

1. Start a `snapshot` operation.
2. Wait N seconds.
3. If the operation has completed: stop; else kill it and go to 1.

This checks whether Hindsight can survive in a hostile environment, where it is constantly killed and resumed. Here, "killed" means the Linux signal KILL – also known as signal 9 – that kills a process brutally and without asking. After the test has finished[3] we restore the data and verify it.

See appendix B on page 121 for an example run of this test.

5.2.2 Documentation

We have already released the prototype under GPLv3, and of course a sensible open source project calls for documentation.

In Haskell, the standard documentation system is Haddock. We have used this system to document the API of our b-tree, but have not yet come around to document the API of the processes, and the rest of the system.

The Haddock documentation for the b-tree can be found at

```
http://hind.sight.dk/doc/Data-BTree-BTree.html
```

5.3 B-trees

Hindsight's use of a multi-file structure in the form of b-trees for indices allows it to retrieve parts of log data as it is needed. This makes inspecting part of a backed up file structure more efficient by retrieving the needed log data only.

In section A.3.2 on page 100, we present the results trying partial retrieval on a snapshot with a million files. The results show retrieval of around 33%

[3] Which is not guaranteed; for example bup never completes the snapshot.

of the log data when listing 0.1% of the snapshot and less than 50% when listing 1% of the snapshot.

When listing the top-level directory non-recursively, some excess transfer was avoided. The synthetic data set was not particularly fit for prototype's `listdir` command, in that the files were equidistributed. To be effective, the command must be able to cut away large subdirectories, but no such existed. We believe this command could be more effective on other data sets.

5.4 Benchmarks

In this section, we evaluate our prototype with respect to space and time usage and garbage collection.

5.4.1 Space and time usage

In section A.6 on page 110, we present our results of making 10 snapshots of data collected from one of the author's home directory and of the Linux kernel sources respectively. We compare our Hindsight prototype to four other backup systems: Brackup, bup, Cumulus and Tarsnap.

From our benchmarks we see that our Hindsight prototype is 2-3 times slower than the competition (except for Brackup). We would expect it to scale well with lots of data (being based on b-trees) and the tests give some credence to this claim, in that the prototype does not increase in time usage with each snapshot. It does however spend twice as long on its 10th snapshot of the Linux kernel sources, compared to its first.

Another observation is that the prototype looks more reasonable in its time use, when storing data on S3. This could be because it can do some of its computations in parallel with network communications.

The prototype performs quite well with respect to local and external space usage. Excluding Brackup who stores its hash index remotely, our prototype uses fewer local bytes than any of the tested systems. The data it sends to the back-end corresponds to that of other `gzip` based backup systems (Cumulus stores fewer bytes, but uses a more aggressive compression method).

5.4.2 Deduplicated log data

In appendix A.3 on page 99, we have tested the effect of applying deduplication on the log data. While it does save some space externally, and thus saves network transfer, it is likely to also increase the transfer when retrieving back the data (due to the indices becoming entangled).

The prototype uses the same blob size of 2 MB for backing up user data, as for log data. This implies a worst case of retrieving 2 MB to unpack one b-tree node. This worst case could be lowered by either lowering the blob size for the secondary run, or by putting a limit on the indices considered when deduplicating.

When retrieving many consecutive key indices, deduplicating yields a lower data transfer.

5.4.3 Conservative garbage collector

In section A.5 on page 105, we test our conservative garbage collector. The results show that the collector can provide acceptable performance for small repositories where only a small percentage of data is rewritten between snapshots.

Specifically, an overhead of just 10% garbage is present with 32 live snapshots and 12% rewrite, which we think could make for a realistic setting of snapshotting a personal computer. On the other hand, our own home directory data from section A.6 on page 110 varies a lot more in the amount of data, and thus counteracts this notion.

Further, this garbage collector is only as efficient as the smallest snapshot family permits (due to its small Bloom filters). Thus, keeping small snapshot families in the same repository as large ones will result in a large garbage overhead.

We have not looked at how much time our garbage collector spends on analysis, due to its immature implementation.

Chapter 6

Further work

In this chapter we list some topics for further work.

6.1 Crash safety

The prototype has a recovery routine to bring its log data back to consistency after a program crash (see section 4.4 on page 66). It works by employing a number of log files, which is then used to rollback to a valid state.

However, the directories containing the log files are currently not synced. So even though the prototype syncs the files it creates, they may not have been registered as part of their parent directory.

This gives us two race conditions:

1. The log file describing the contents of a blob could disappear, and an unreferenced blob could be left on the back-end as a result. This is possible because the hash index' rollback directory is not synced.
2. The log file describing the newly inserted keys could disappear, and a key referencing a non-existing hash could avoid rollback. This can happen because the key index' rollback directory is not synced.

These issues could be solved by adding logic to sync the parent rollback directories. It's definitely possible, but the prototype does not yet do this (there was no obvious way to open a file descriptor for a directory in Haskell at the time).

We do not believe there is a problem with the way the b-trees write data: all updated nodes are written and synced in bottom-up order, before removing anything. Removing deprecated nodes occurs without explicit syncing.

No matter when the directory is synced, it will register some split of this sequence, and all splits are safe (a program crash would leave an arbitrary split as well, and this is what the routine was designed for).

6.2 Asymmetric encryption

Instead of the simple security model employed by Hindsight, a more advanced model could be designed. With asymmetric encryption (also known as public-

private key encryption), the encryption and decryption processes rely on different keys: The *public* key is used for encryption, and the *private* key is used for decryption.

Instead of encrypting blobs with a single master key, each one could be encrypted with a personal key. This key could then be encrypted with a public key and stored in the hash index, along with the reference to the blob. It would now be impossible for the client to retrieve any blob data (assuming it does not have the private key). Only the meta-data is available to it.

If we then repeat this change at the secondary level, and thus employ another *secondary* public-private key-pair, we get the following situation:

Snapshot and seal require both public keys, but no private keys. Hindsight does not retrieve information from the back-end during a snapshot; all the needed log data is already local.

Checkout requires both private keys. One for retrieving log data and one for retrieving file data.

Search and recover require just the secondary private key to access the log data.

GC requires all the keys, since it needs to scan log data and rewrite blobs. Without blob rewriting, it would only need the secondary key-pair.

The cryptographic library used by the Hindsight prototype, NaCl [19], supports asymmetric encryption with its `crypto_box` API.

6.3 Indices

Here we discuss further work related to our hash and key indices. For a discussion of the b-tree implementation, see section 6.5 on page 90.

6.3.1 Merkle trees

As mentioned in section 3.11 on page 53, Hindsight currently stores all hashes used by an entry in the key index as a list. This leads to a scalability issue, when backing up large files.

This problem could be solved using Merkle trees [64] (also known as hash trees). A Merkle tree is a binary tree, where each branch node contains a hash of its children's values. The value of a leaf node is the hash of the data block it represents (the sequence of leaf nodes correspond to the list of hashes the prototype maintains now).

Thus, the root node is sufficient to safely verify the data contents of all the leaf nodes put together, and we could identify the contents of an entry in the key index (the contents of a saved file) by the root of a Merkle tree (a single hash). To make this change we would have to:

1. Extend the hash index to allow for hashes that reference other hashes along with the current hashes that reference blobs. Thus, we can model a tree inside the hash index[1].

2. An entry in the key index now needs to reference just a single hash, that is the root of the Merkle tree that describes its contents.

This would make the hash index slightly larger – by adding the Merkle tree branch nodes – and the key index a lot smaller. This will give a lower space usage by eliminating some of the current redundancy from storing all hashes in *both* the hash index and in *at least* one key index.

Of course, this will introduce more lookups and inserts in the hash index per file backed up. Now, both the data hashes and the tree structure need insertion if not present. On the other hand, it can be used to efficiently identify known parts of a file. Szydlo [89] investigate methods for efficient traversal of Merkle trees.

6.3.2 Hash Index

In the prototype, the hash index is implemented using our concurrent b-tree. But unlike the key index, it does not need efficient range queries or lazy retrieval from the back-end (the hash index is always stored locally). Thus it could use a completely different data structure, such as an external hash table or a fractal b-tree, as long as it would provide the same level of deduplication as the b-tree.

It could be interesting to try other data structures and see how they affect performance, both in terms of time usage and local space usage.

6.4 Garbage Collection

We have only implemented a conservative garbage collector in the prototype. It would be interesting to implement and experiment with a classic mark-sweep garbage collector as well as a garbage collector based on reference lists.

The prototype clearly distinguishes between the mark phase and the sweep-compact phase, so other collectors can implement their own mark phase and reuse the existing sweep-compact phase.

For example, the classic mark-sweep garbage collector needs only to retrieve each key index and mark the hashes used. When this is done, it can run the existing sweep-compact implementation. Likewise, a garbage collector based on reference lists needs only to maintain the references and mark the dead hashes.

[1] The approach is similar to the one used by Git where a tree object can reference either blob objects or other tree objects.

6.5 B-trees

As mentioned in section 4.3.6 on page 63, the re-balancing logic used is incomplete, and some of the API is either partially broken or unsafe. Apart from finishing this work, we have other ideas for what could be interesting to investigate, which we present in this section.

6.5.1 Summary vector

As discussed in section 3.5 on page 28, expensive hash index lookups can be avoided by maintaining an in-memory summary vector (implemented by a Bloom filter) of the hashes.

In an earlier implementation of our prototype every hash resulted in an index modification, which is why we didn't implement summary vectors. However, this limitation is not present in the current version, and so summary vectors can be implemented. A summary vector is used to identify hashes which are *not* in the index. In that case, the hash can be inserted in the memory cache of to-be-inserted hashes. The result is a halving of index accesses.

Zhu et al. [96] note that throughput can be increased using Bloom filters as summary vectors, and that the Venti backup system achieves a 16% increase in performance in this way. Another system which uses summary vectors is bup [72].

6.5.2 Types and performance

We conjecture that our current b-tree implementation could be a lot faster and that it is using a lot more memory than needed.

The current b-tree accepts arbitrary Haskell types as keys and values, as long as instances for the needed Haskell type classes are available. This is sufficient to build the tree and write its nodes to disk. While it provides for a very flexible b-tree that will accept any type with little work, it results in a lot of overhead.

First, serialisation is expensive. In the current b-tree, serialisation is done on a per-node basis. To lookup the value of a key, the nodes on the path through the tree are deserialised along with all of the keys and values in the sought-after leaf. As a result, many values are deserialised, just to be discarded.

This could be avoided if keys and values were forced to be of the `Bytestring` type. Keys and values are now already of a serialised type which allows nodes to always remain as `Bytestrings`. Thus they could be written directly to disk when needed.

Of course, this makes searching for values inside nodes trickier, which is why the implementation uses Haskell data structures: It was easier to get working.

Which leads us to the second part: The data structures used in the b-tree has a significant memory overhead. A leaf in the b-tree is implemented as a map that points keys to values. An entry in the map takes up 6 machine words in which it stores: a pointer to the data constructor (`Bin`), the size of

the sub-tree, a pointer to the key and one to the value, a pointer to the left sub-tree and one to the right.

Furthermore, keys we use are Bytestrings which come with another 5 words of overhead (Data constructor, `ForeignPtr` [two words], length and offset). In the case of the hash index, the values are pairs of `Word64` with no pointer overhead.

In the key index, the keys are also Bytestrings, but the values consist of another two Bytestrings (version and meta-data) as well as a list of bytestrings (the chunk hashes). A list element comes with an overhead of 3 words, while each of the Bytestrings has an overhead of 5.

With these observations, we estimate the memory overhead of the b-trees on 64-bit architecture:

	Branch entry	Leaf entry
Hash index	88 B ($550 - 978\%$)	88 B (157%)
Key index	88 B ($100 - 978\%$)	$176 + 64c$ B (200%)

Estimating the overhead percentage of branch entries is a bit difficult, since the data stored depends on how short a prefix was picked when splitting the nodes in the sub-trees. In the hash index, a prefix of length 8 (fairly long) gives 550% while prefix of one char gives the worst case of 978%. In the key index, the branch prefixes are expected to be longer (file paths are longer than hashes), though the worst case remains the same.

We are under the impression that a significant speedup could be gained by exploiting that keys and values are Bytestrings, and the shown memory overhead could be eliminated completely.

6.5.3 Inter-node deduplication

Assume that we have changed keys to Bytestrings, as proposed in the last section. We can now eliminate redundancy in the keys, by removing common prefixes. A simple way to do this, is to allow each node to reference back to the parent node, to reuse prefixes from it.

The PostgreSQL developers have discussed a similar feature [75] and the Oracle database already employs a similar scheme for their key-compressed index [7].

Example leaf with four keys, surrounded by its two parent routing keys (`p` is parent and `l` is leaf):

```
p:  /home/alice/code
 l:   /home/alice/code/data/tree.hs
 l:   /home/alice/code/algo/mergesort.hs
 l:   /home/alice/docs/wishlist.csv
 l:   /home/alice/work/theboss.dat
p:  /home/alice/work
```

Now, the leaf key can reference the parent routing keys. The three tags are *L* for prefixing the left routing key (first line in the listing), 0 for no extra prefix and *R* for prefixing the right routing key (last line in the listing):

```
p:  /home/alice/code
 l:  L/data/tree.hs
 l:  L/algo/mergesort.hs
 l:  0/home/alice/wishlist.csv
 l:  R/theboss.dat
p:  /home/alice/work
```

Here, the two first keys and the last key in the leaf can strip a long prefix, by referencing the routing key from their parent. One value cannot use any of the prefixes in their full form, and is instead written as is. An improvement could be to allow a key to reference only part of the prefix.

In practice, the leaf boundaries will not match folders so well. We used a folder boundary here for readability.

We are not yet certain that this approach is compatible with safe re-balancing in the concurrent b-tree used by the prototype. This needs further study.

6.5.4 Order preserving compression

As another means for lessening the b-tree's space usage, the keys could be compressed with an order preserving compression method. See Binnig et al. [22] for a discussion of the advantages of such methods in the context of database systems.

Order preserving compression could be used the compress the keys of the b-tree, while still allowing operations such as lookup and insert to run without decompressing the keys.

Compression would only be beneficial in the key index, since the hash index's keys are pseudo-random. However, compressing the file paths in the key index could lead to a significant gain in performance from lower memory usage and comparison of shorter elements.

6.5.5 Hackage package

As with Hindsight itself, we would like to release the b-tree implementation as its own independent package on Hackage (the Haskell package system).

This would allow other developers to use the b-tree in their own projects, and to help improve it (e.g. by implementing some of the suggestions we have presented in the prior sections).

6.6 Alternative Front-ends

It could be interesting to look at other types of front-ends to use with the Hindsight API. So far, we have presented a front-end for backing up POSIX file systems and a front-end for mounting a snapshot for inspection through FUSE.

6.6.1 Inspection

To extend on the ability to mount a snapshot, a feature for cycling through the history of a specific file or folder could be made. Thus giving an interface with features inspired by Apple's Time Machine.

This kind of inspection could benefit from the deduplication Hindsight applies to log data, for more efficient transitions from one snapshot to the next or previous one.

6.6.2 Snapshotting

We have ideas for a couple of alternative snapshotters:

Data inspection: A front-end can pick and choose which data to back up, which allows for different rules for different kinds of data. For example, a front-end for a database could open the tables and insert each row as a key in the snapshot; or a front-end could open a virtual machine image and insert each file individually.

Of course, it would be more efficient to read the data contiguously and for example insert the virtual machine image as *one* entry, however opening the image for inspection gives flexibility.

Log rotation: Another front-end for snapshotting could be a service for log rotation. It accepts messages that are added to the log, and periodically rotates the log, by archiving the current log and starting over with a fresh one. This process could use Hindsight's head as its current log. When a message arrives, it is added to head with some unique key. When the log is rotated, the snapshot is committed and the head is cleared. Thus, entries of the log are preserved securely as snapshots.

File monitoring: Another useful fron-end is a file monitoring service that monitors the file system and processes changed files real-time. When a file is changed, it is inserted into a queue. Workers pick files from the queue and adds them to Hindsight's head. Periodically, the workers are paused, and the head is archived as a snapshot. This operation should be quick, since all file processing has already occurred. The daemon could exploit kernel services such as the Linux `inotify` API or the Windows `FileSystemWatcher` API.

6.7 Alternative back-ends

In this project, we have focused mainly on an untrusted back-end. If we introduce a trusted back-end – or perhaps a trusted back-end proxy – that sits in front of an untrusted back-end, we could move some logic to this entity.

In the following, we consider a scenario with a client and two back-end parties P and B, trusted and untrusted respectively. The design discussed earlier contained only one *untrusted* back-end, which is the same as running P at the client.

Everything at P: We could move the entire Hindsight API to *P* and do backups through a hosted service. *P* could then use *B* for storage of protected blobs. This would move all backup logic to *P*. While we could still query *P* to ask whether a file has changed, once such change occurs we would have to resend the entire file, since *P* is responsible for file chunking and deduplication.

Local chunking: If instead we move file chunking to the client, along with a hash index to identify known chunks, we could send only new chunks to *P*. We could still have *P* take care of snapshots, chunk references and garbage collection. Note that since the hash index is not used for garbage collection, it is allowed to drop hashes, even though a key index needs them. We could just ask *P* for the hash. In this setting, the hash index works like a *cache*.

We could protect the chunks with convergent encryption, to hide them from *P*, but we would still disclose meta-data and file paths.

This could be achieved by extending the Hindsight API to allow insertion of keys in the head by inserting their hashes instead of the actual contents. Additionally, it must be possible to add hashes to the key store.

Local snapshots: Let us extend the client a bit more, and keep snapshot maintenance and key indices local. We now know locally which files each snapshot consists of, their meta-data and what their content hashes should match. Depending on whether we keep all hashes ever seen in the hash index, or put a limit on its size, we can perform full or partial, global deduplication locally. *P* can extend this with *universal* deduplication, across all users that go through *P*. Some of which can occur either *online* (we can still ask *P*, whether the chunk hash is known), *offline* after the fact or with any mix of these.

Combined with convergent encryption, *P* need not be fully trusted. *P* would no longer be able to read the chunks, the meta-data or the file paths. All *P* gets are the encrypted chunks, and unless *P* knows the contents of the chunk, it cannot decrypt it.

The key indices can be stored at *B* without going through *P*, with the approach described in our "single back-end" design. This means they are fully protected with encryption and authentication and can thus be used to verify the file data when received from *P*, by comparing its hashes with the key indices.

In the prototype, this could be done by running local `KeyStore` and `HashStore` processes. The latter in a modified version that does not use a `BlobStore`, for storing chunks, but instead speaks directly to an `External` process. This process protects the blobs with convergent encryption and transmits them to *P*.

6.8 Encoding format

In this section, we briefly discuss further work regarding the encoding format the Hindsight software uses to serialise its structures.

6.8.1 Backwards compatibility

Hindsight relies on two data structures which are stored on the back-end: b-trees and blobs. If at some point, one of these structures are replaced, or their encoding is changed, it would be beneficial if the implementation could migrate old data to the new structure or format.

This could be achieved by adding a version to all stored files, indicating its type, serialisation format and encoding (e.g. compression and encryption).

6.8.2 Transparency

To increase transparency, we would like to write a technical report documenting the bits and bytes of the encoding format used. This would allow anyone to understand the format used by Hindsight and a user could inspect the data stored on the back-end herself.

Chapter 7

Conclusion

During this master's project we have designed and implemented a full featured backup system in Haskell. We started out with a small core idea, and has since developed that idea into the prototype and design described within this report.

We do not yet consider our system mature, but on some points it fares rather well. Specifically, it uses fewer local bytes than any of the other similarly featured system we have encountered.

The landscape of backup systems is a diverse one. On some points we mimic what is already known and on some points we explore new ground. Specifically we've investigated conservative garbage collection and lazy log data retrieval (using b-trees). Furthermore we perform log data deduplication elegantly by running the system recursively.

So what have we learned from this? Lazy log data retrieval is a very usable feature, and our prototype wouldn't be the same without it. Conservative garbage collection is an interesting idea. We implemented it because we were curious and it seems no one else have done it. So does it work? The answer to that question is a definite "maybe"; there are certainly usage patterns which fit the garbage collector well, but there are also some that utterly breaks it.

While log data deduplication does decrease the amount of transferred log data when taking a snapshot, it can increase the transfer during inspection. On one hand, it may decrease data transfer when inspecting multiple consecutive snapshots, and one the other hand it may increase it when inspecting a single or few snapshots due to their log data being entangled with one another.

We are hesitant to conclude on whether our idea for log data deduplication is a good compromise or not. We think that it would be a shame if we are forced to abandon this idea, in favor of a deduplication method which relies on the semantics of the log data (i.e., knows about the filesystem).

Though not directly visible in this report, a significant amount of time was spent improving our skills in system design and Haskell programming. The prototype is clear evidence of our progress.

We enjoyed it.

A Benchmarks

A.1 B-tree compared to SQLite3

Table A.1 shows the results of a time and space comparison of our b-tree and an SQLite3 database when inserting 200,000 keys.

	Time usage	Space usage
Our b-tree:	24.4s,	497 KB
SQLite3:	16.9s,	2,413 KB

Table A.1: Comparison between our b-tree and a SQLite3 database when inserting 200,000 keys-value pairs

We see that our b-tree is 44% slower than the SQLite database, but that it uses just 20% of the space (mainly due to compression of the tree nodes). The SQLite database maintains a b-tree index over the key values just as we do, but does so more efficiently through optimised c-code and efficient IO.

We find it likely, that our b-tree can at least match a SQLite database in performance after fixing the serialisation problems discussed in section 6.5 on page 90.

A.2 Writing log files in bulk

Figure A.1 shows the time used to take a snapshot of a few files with various configurations of keys per log file. The number of keys per log file is doubled for each run. Many keys per log file gives less precise recovery, while fewer is less efficient and take up more space.

As guessed, writing just one key per log file and syncing it is very slow. At 2^3 keys things looks more reasonable and the saved time hereafter is low. But there is another trade off in play here: Every log file allocates a disk block which is usually 4096 bytes, and thus a lot of space may be unnecessarily allocated if each log file is small.

Assuming an average path length of 100 (which is probably a bit pessimistic), we can fit 40 keys in a single block. In this scenario, 8 keys per log file would only utilise around 20% the disk blocks.

This is the reasoning behind our choice of $2^7 = 128$ keys per log file. Though somewhat arbitrary, we guess that this is a good trade off between utilising disk blocks and enabling efficient recovery.

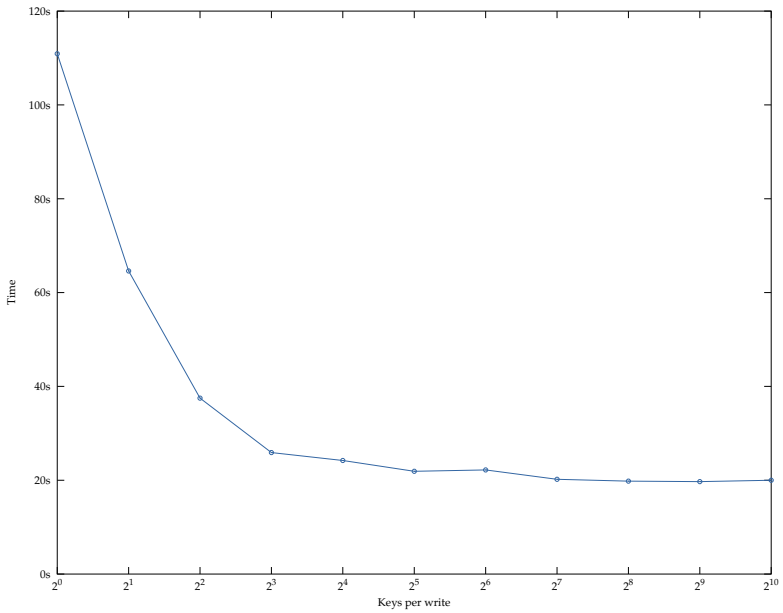

Figure A.1: Writing many file paths in bulk improves performance.

A.3 A million files

Using the tool `genbackupdata` we created a million small files, each containing 20 random bytes. Then we took a snapshot with the Hindsight prototype. It took about 45 minutes. The test data was generated with the following command:

```
genbackupdata --seed=0 --file-size=20 --create=20000000 testdir
```

files	1,000,000
directories	7,876
unique files	19,614
file size	20 B
file path lengths	26,27,28
Hindsight log data	48.5 MB
Key index	44.0 MB
Hash index	0.8 MB
External space usage	44.8 MB

Table A.2: Test with a million files.

Table A.2 lists some statistics of the data in the snapshot. Only around 19,000 of the files had unique contents. This is due to the `genbackupdata` program that apparently uses a biased method to generate the "random" contents of the files (its `man` page does not mention this). A *fair* pseudo random number generator does not yield collisions on a million samples of 20 bytes.

In any case, this explains the very small hash index, which only contains 19,614 hashes (one per unique file). This tells us that the hash index is using around 41 bytes per entry. An entry consists of a hash of 32 bytes and a blob ID of 24 bytes, 56 bytes in total. Since the files are so small, only 47 blobs were generated during the snapshot, which implies many repeated blob ID's in the hash index, and thus a lot of redundancy for the compression to eliminate.

We can use this to reason about the hash index size, if all files had been unique. This would have given 1 million entries, and thus would likely have taken about 41 MB. Thus giving a total amount of log data of around 90 MB, about twice as much as now.

The key index contains 1,007,876 entries, one for each file and directory. Thus, it uses around 45 bytes per entry. Since many of the files contain the same contents, the same content hashes will be repeated in the index, which leads to a higher compression ratio. Due to the short time span of generating the files, a lot of files contains the same meta (time stamps etc.), which leads to even more redundancy that the compression can eliminate. The file paths are well suited for compression too.

All in all, compression is very effective on this log data. But even then, the log data is 48 MB while the original data is just 81 MB. Because the files contain so little data, their key index entries take up a lot of space compared

to the original files. However, the data blocks used to store the data takes up 3.9 GB and compared to this, the log data is just 1.2%. If all files had been unique, this would likely have been around 2.5%.

This is higher than our goal of 1%, however this is a synthetic test that does not resemble regular use (it is unrealistic that a filesystem should not include at least some larger files).

We feel that 2.5% would be an acceptable result for this data (assuming that our extrapolation is accurate).

A.3.1 Log data deduplication

To test the effect of data deduplication, we made 5 snapshots of the million files from the previous section: *a)* an initial snapshot; *b)* a snapshot where a selection of around 11.5% semi-ordered files were updated (115,591 files)[1]; *c)* a snapshot where the first[2] 11.5% ordered files were updated; and *d)* two snapshots where a random selection of 5% and 1% percent of the files were updated respectively.

Table A.3 on the following page, shows the amount of log data retrieved when listing the files of each snapshot by itself, and when listing it after listing the one just before it (2 after 1 and 3 after 2). In the latter case, the blobs referenced by the key index of the previous snapshot are cached.

The first snapshot retrieves its own key index only, which is 44.5 MB. The second snapshot's log data has been deduplicated and is thus entangled with that of the first snapshot. Listing it on its own retrieves 79 MB, which is a lot more than the key index it extracts (80% overhead). The same effect shows when listing the third snapshot, where 107 MB are retrieved to unpack a 44 MB key index (143% overhead).

This is the down side of deduplication: it fragments the data – breaking chunk locality in the stored blobs – which results in retrieval of irrelevant data blocks.

On the other hand, the second and third snapshots have only contributed with 35 and 28 MB log data respectively. That is 72.3% of their key indices, or a saving of 27.7%. The second key index by itself has a saving of 35.6%. However, when we start to introduce random changes, the percentage of saved space usage drops to almost zero.

A.3.2 Lazy log data retrieval

Table A.4 on the next page and Table A.5 on the following page, show the amount of log data retrieved when listing a directory with 129 and 16,513 files respectively. As can be seen, deduplication leads to less data retrieval when the previous key indices are already present, but more if not due to entanglement. Additionally, listing fewer paths leads to less retrieval.

[1] The number of files was chosen as around 10%, and as something that fit the structure generated by the `genbackupdata` tool. It corresponds to the files of 7 directories.

[2] Following the order of the key index' b-tree.

Table A.6, shows the amount of log data retrieved when listing the top-level directory non-recursively (using the prototype's `listdir` command). In the first listing, it seems the full key index is retrieved. Hereafter, it seems that the b-tree is able to cut away irrelevant sub-trees when listing without cache, and that deduplication can improve on this result.

Snapshot	Key index	By itself	With the previous
1	44.0 MB	44.5 MB	–
2	43.9 MB	79.1 MB	35.2 MB
3	43.8 MB	106.6 MB	28.2 MB
4	44.6 MB	59.7 MB	45.0 MB
5	44.8 MB	94.5 MB	42.0 MB

Table A.3: Data retrieved during five full snapshot listings.

Snapshot	By itself	With the previous
1	13.1 MB	–
2	14.2 MB	12.1 MB
3	15.2 MB	6.9 MB
4	13.6 MB	11.5 MB
5	14.3 MB	10.1 MB

Table A.4: Data retrieved during five snapshot listings of the directory path `0/32/64` containing 129 paths.

Snapshot	By itself	With the previous
1	21.5 MB	–
2	26.8 MB	14.2 MB
3	38.3 MB	15.3 MB
4	17.8 MB	15.7 MB
5	24.8 MB	14.3 MB

Table A.5: Data retrieved during five snapshot listings of the directory path `0/32` containing 16,513 paths.

Snapshot	By itself	With the previous
1	44.3 MB	–
2	56.1 MB	35.2 MB
3	68.6 MB	26.7 MB
4	45.0 MB	42.9 MB
5	58.7 MB	42.0 MB

Table A.6: Data retrieved during five non-recursive snapshot listings of the top level directory path `0` containing 62 paths.

A.4 Bit-vector encoded reference lists

In section 3.6.5 on page 40 we discussed garbage collection using reference lists, where the set of snapshots referencing a fingerprint is saved along with it. If snapshots are given numbers $1\ldots$ then a fingerprint's set of referencing snapshots can be saved as a bit-vector where a 1 in position n signals that snapshot n references the fingerprint. In order to avoid updating every list when a snapshot is added, the bit-vectors are interpreted as ending in an implicit infinite stream of zeros. Note however, that the bit-vectors would still have to be updated whenever the order of snapshots changes (e.g. when a snapshot is deleted).

In this test we show the effect of concatenating several vectors together and LZMA compressing them.

We have prepended a 2-byte size field to each bit-vector, but we think that an arbitrary precision encoding (á la Git's object size fields) should be used in an implementation. With such encoding, a single byte to indicate length would be sufficient for a reference set of several hundred snapshots, which we think includes most use cases.

The bit-vectors were synthesized using the following algorithm, parameterised by α, β and N.

Start by setting bit $b = 0$, and vector $v = []$. Repeat these steps N times:

1. Append b to v.
2. Draw two numbers x and y from the uniform distribution over $[0;1)$.
3. Flip[3] b if $y < \alpha$.
4. Flip the last bit of v if $x < \beta$.

The rationale behind the algorithm is this:

1. Chunks are long lived, that is if a snapshot introduces a chunk then the next snapshot will probably also use that chunk. The lower the value of α, the longer the lifetime of chunks.
2. Most snapshots will have a small amount of data that is unique to them (e.g., temporary files, log files, etc.). The β controls the amount of "noise" in the synthesized vectors.

We have included trailing zeros, even though they are redundant[4]. For this test we have used values $N = 1000$, $\alpha = 0.1\%$ and β varying from 0.0% to 0.5% (shown in the legend).

Thus, the expected number of noise-bits is $0,\ldots,5$, and b is expected to flip once [23]. But since we disregard vectors which are all zeros, the number

[3] So 1 becomes 0 and 0 becomes 1.

[4] Because of the implicit infinite stream of trailing zeros.

of flips has a skew towards higher numbers (judging from the synthesised vectors the mean seems to be around 2).

In figure A.2 on the next page we show the effectiveness of LZMA compression as a function of the number of vectors concatenated together. Notice that the number of vectors is dictated by the order of the b-trees (see section 4.3.6 on page 63). The b-tree order used in the prototype is 128, so between 128 and 256 vectors can be concatenated. With noise ratios between 0.0% and 0.5% this yields a $\sim 90\%$ compression or better.

In other words there is an overhead of around 0.1 bit per snapshot plus 8 bits for the size field in most cases.

Comparing this with 32 bits used for reference counting, we think that it is plausible to use reference lists, and the possibility should at least be investigated.

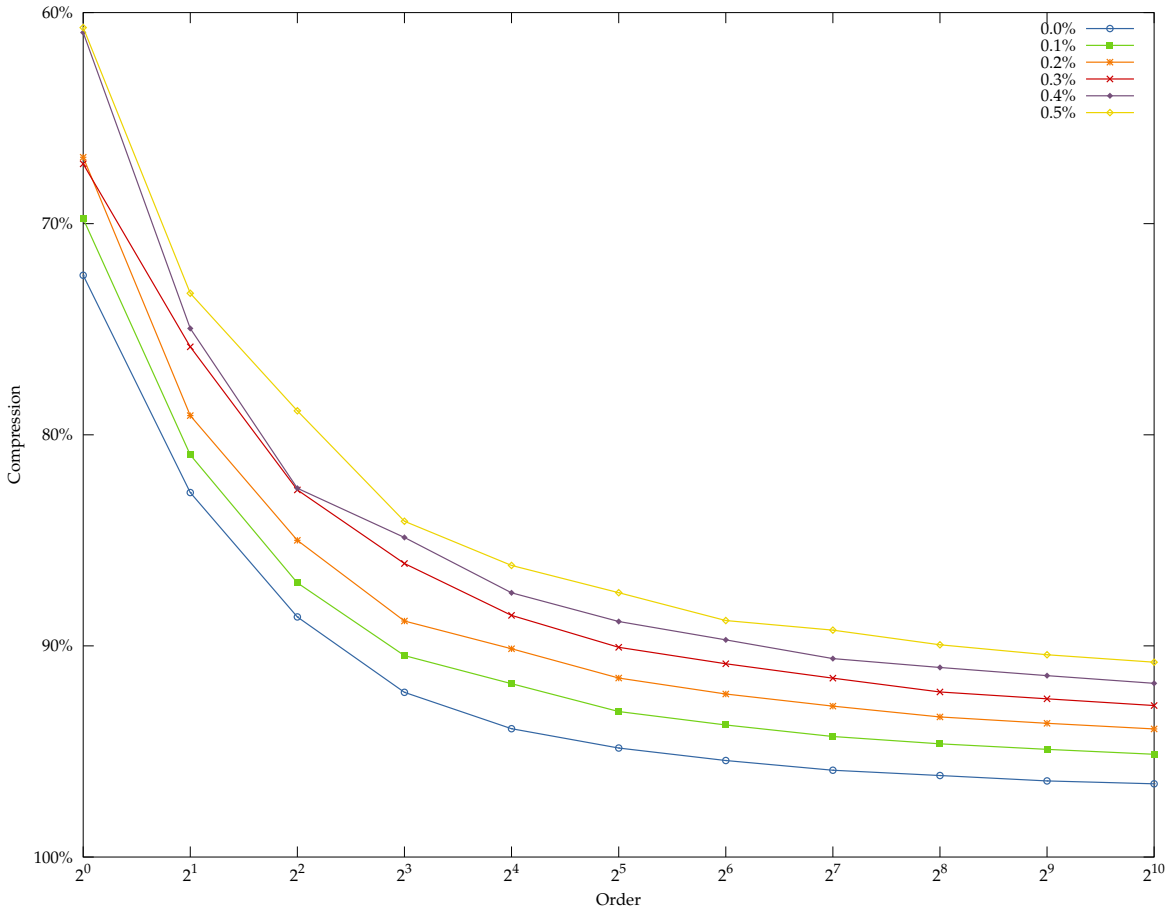

Figure A.2: Effectiveness of LZMA compressing bit-vectors as a function of vectors concatenated together. The graphs represent different amounts of noise.

A.5 Conservative garbage collection

In this section, we investigate the effectiveness of our prototype's conservative garbage collector (see section 3.6.6 on page 40 for details).

In the test we vary the amount w of snapshots and changed data Δ between each snapshot. To limit noise, we have emulated a perfect compacting garbage collector by adjusting the blob size, such that each blob contains exactly one chunk, thus the results are not skewed by the effectiveness of blob rewriting.

We have synthesised a range of data sets using this algorithm:

- Generate C random files each of size B. These constitute data set 1.
- To get data set $n+1$, copy every file from data set n, except the $C\Delta$ oldest which are replaced by new randomly generated files of size B.

It is clear that each data set has the same size (CB) and consists of the same amount of files (C). We have named the files `0`, `1`, ..., $w-1$ in these benchmarks. For these tests we have used values $C = 2048$ and $B = 4096$.

In each test we start by taking w snapshots of data sets 1,...,w. Then we proceed by repeatedly taking a snapshot of the next data set, deleting the oldest snapshot, and running the garbage collector. Finally we delete all snapshots, oldest to newest.

A graph of the amount of unique data backed up will start off by increasing rapidly for w snapshots, then stay flat while data is being replaced, and then finally rapidly decrease and end in zero. The three phases are: *a)* introduction of data (increasing); *b)* modification (flat); and *c)* deletion (decreasing).

Varying Δ will scale the graph along the y-axis but not change its shape. In this test we scale all graphs by the size of the backed up data (including log data) after w snapshots. Deviations from the shape just described represent sub-optimality of the garbage collector. Because of fluctuations in the deduplication of log data, the graphs can sometimes go below their value at w, especially when Δ is small (which means log data is responsible for a large part of the backed up data).

We have run the test for four values of w: 32, 64, 128 and 256. The results are shown on the following pages in figure A.3, A.4, A.5 and A.6, respectively.

Picking an acceptable overhead of $\sim 10\%$ we obtain these upper bounds for the rewrite percentage Δ as a function of the amount of live snapshots w:

Live snapshots (w)	Rewrite percentage (Δ)
32	$\sim 12\%$
64	$\sim 6\%$
128	$\sim 3\%$
256	$\sim 1.5\%$

We think that there are many use cases, in which these values makes for a practical system.

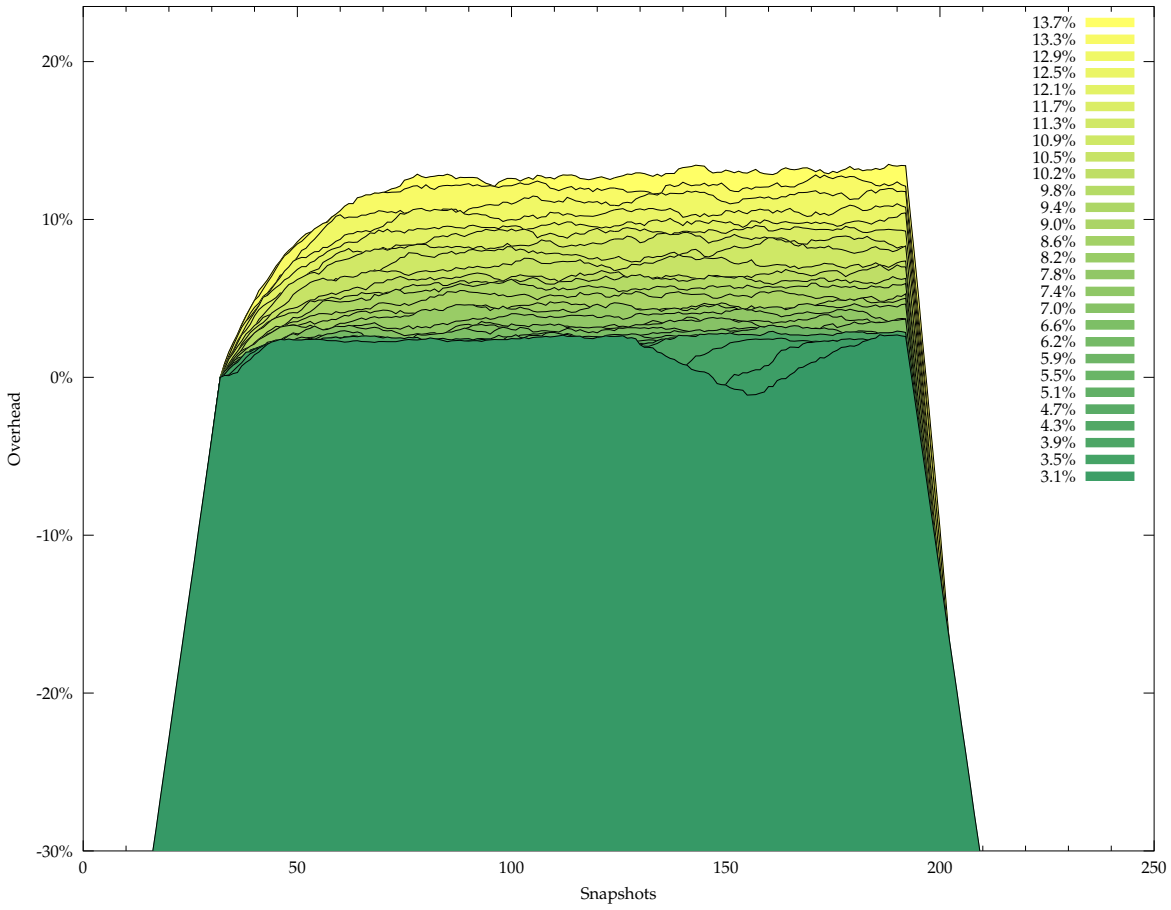

Figure A.3: Conservative garbage collection with 32 live snapshots.

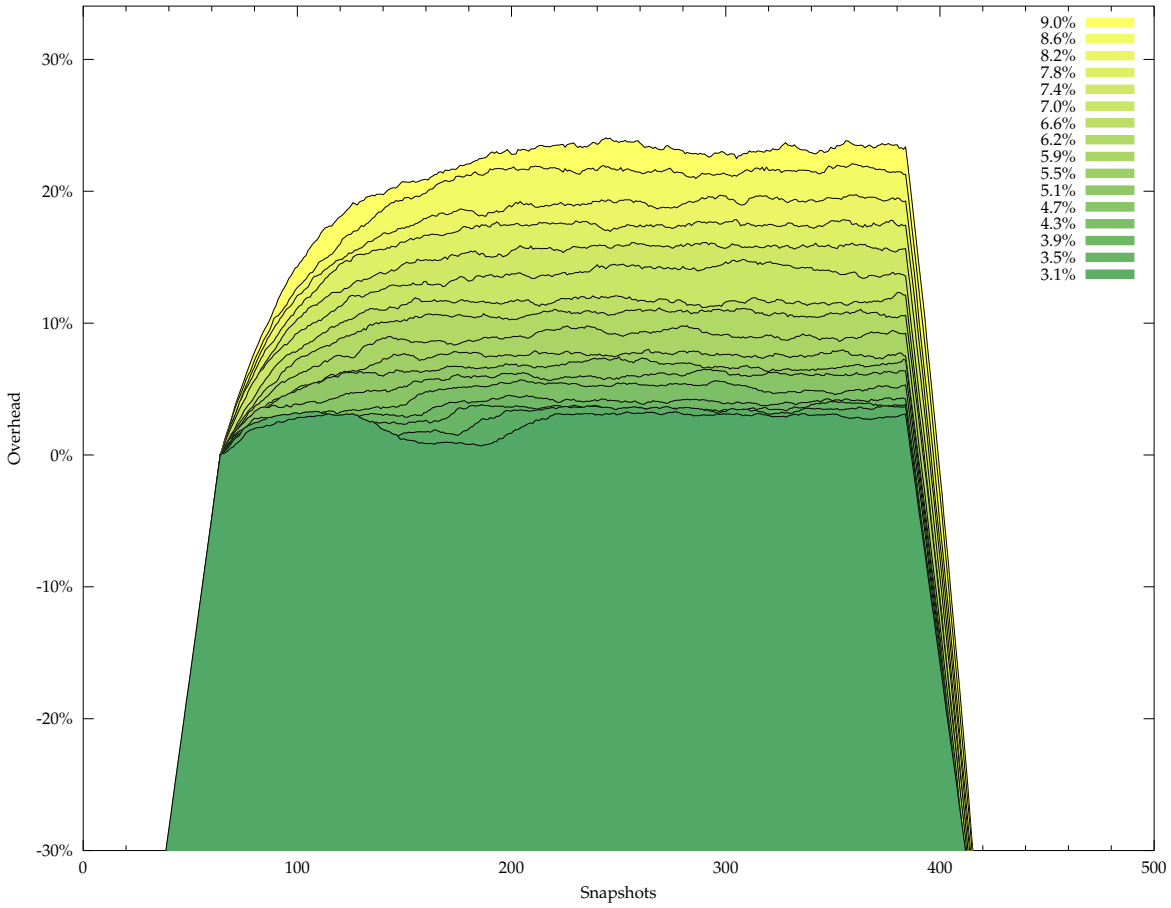

Figure A.4: Conservative garbage collection with 64 live snapshots.

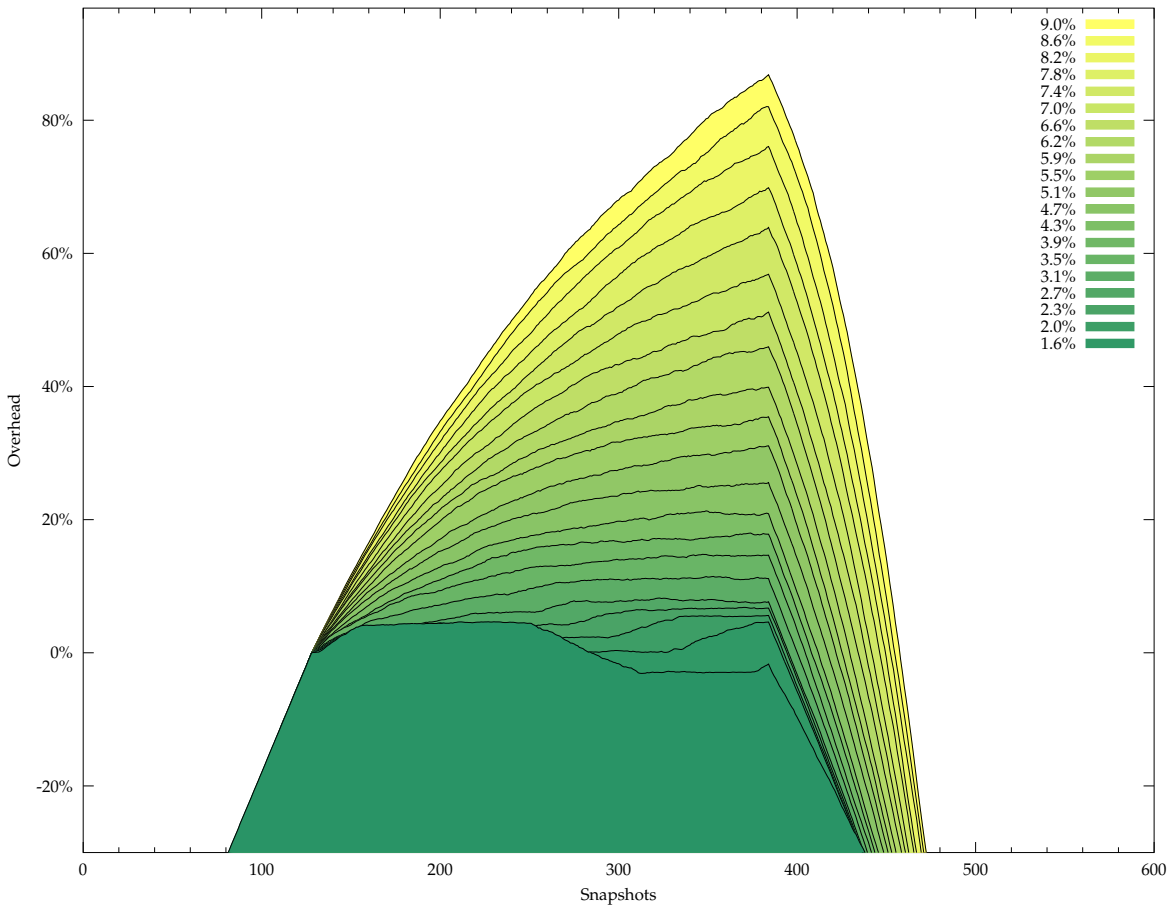

Figure A.5: Conservative garbage collection with 128 live snapshots.

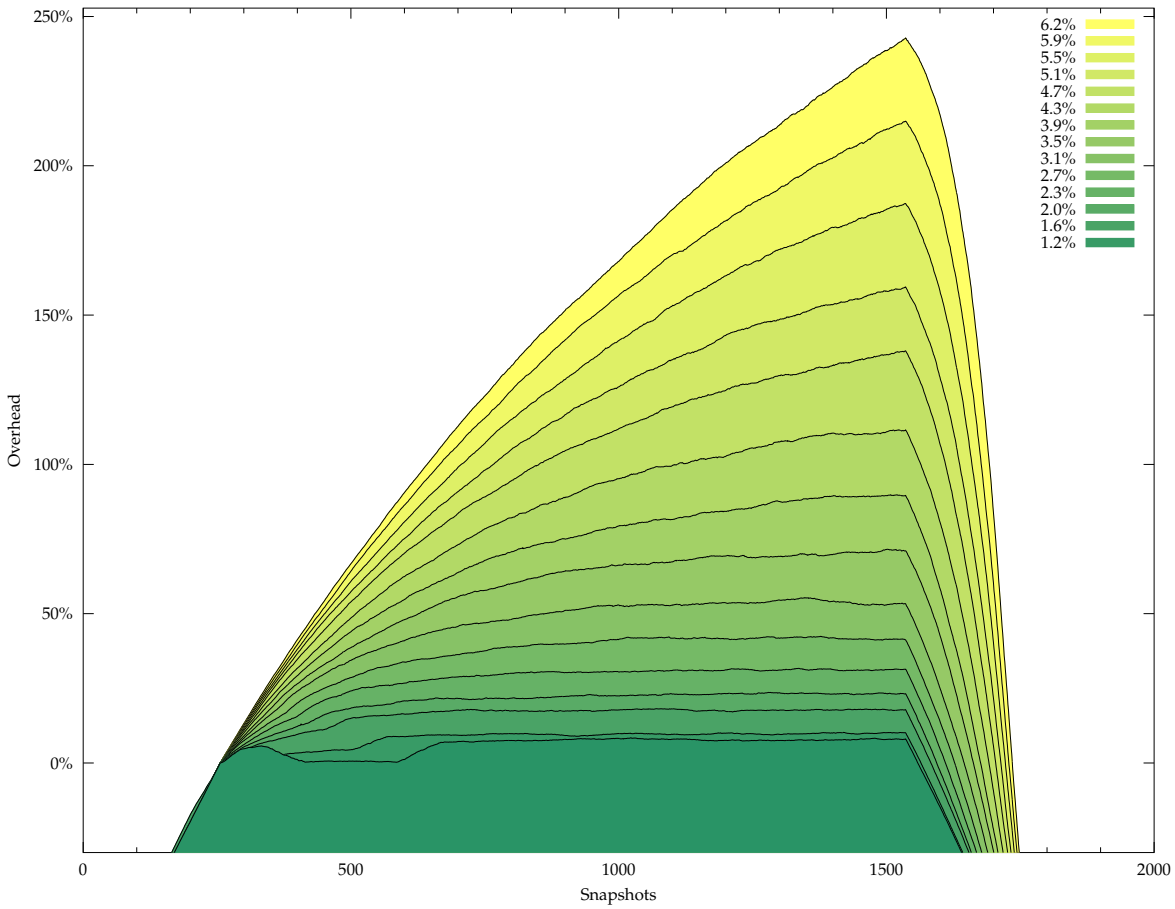

Figure A.6: Figure

A.6 Test on real data

To test the Hindsight prototype on some real data, we have made 10 snapshots of *a)* an authentic home directory (one of the author's) spanning a 2 months period; and *b)* the Linux kernel source code.

Since the home directory contains private information we cannot disclose it and instead present statistics describing its contents.

Figure A.7 on the following page shows a histogram of the files in the home directory (averaged over all snapshots), categorised by size. As reported by Meyer and Bolosky [65], most files are small, however large files contribute with most bytes.

Table A.7 lists the snapshots along with their number of files, directories, soft links and accumulated size. As shown, the snapshots vary between 12.9GB and 70GB.

Table A.8 presents the same statistics but for the Linux kernel sources.

Snapshot date	Files	Directories	Links	Size
9th Feb	138,373	17,167	111	12.9 GB
7th Mar	178,626	17,812	124	39.2 GB
8th Mar	183,596	18,049	125	39.5 GB
9th Mar	197,268	21,169	126	40.6 GB
13th Mar	201,800	21,272	129	41.1 GB
16th Mar	188,556	21,372	121	70.0 GB
20th Mar	153,592	21,458	123	68.8 GB
7th Apr	123,414	21,455	122	13.8 GB
8th Apr	123,754	21,414	121	13.9 GB
9th Apr	183,593	29,842	137	42.8 GB

Table A.7: Files, directories and links of the home directory snapshots.

Kernel version	Files	Directories	Links	Size
2.6.27.62	24,363	1,524	0	280,8 MB
2.6.32.59	30,489	1,878	1	366,0 MB
2.6.33.20	31,568	1,942	1	377,9 MB
2.6.34.11	32,299	1,982	1	387,1 MB
2.6.35.13	33,314	2,028	1	394,7 MB
3.0.27	36,788	2,265	1	431,0 MB
3.1.10	37,083	2,285	1	434,2 MB
3.2.14	37,619	2,345	1	440,4 MB
3.3.1	38,082	2,366	1	445,2 MB
3.4-rc2	38,560	2,389	1	451,3 MB

Table A.8: Files, directories and links of the Linux kernel sources.

In the following, we compare the Hindsight prototype's performance against the following backup systems: Brackup, bup, Cumulus and Tarsnap. Where relevant, we add rsnapshot and `cp` for reference.

We did not manage to run Cumulus on the home directory data, since we could not install it on the server hosting the data.

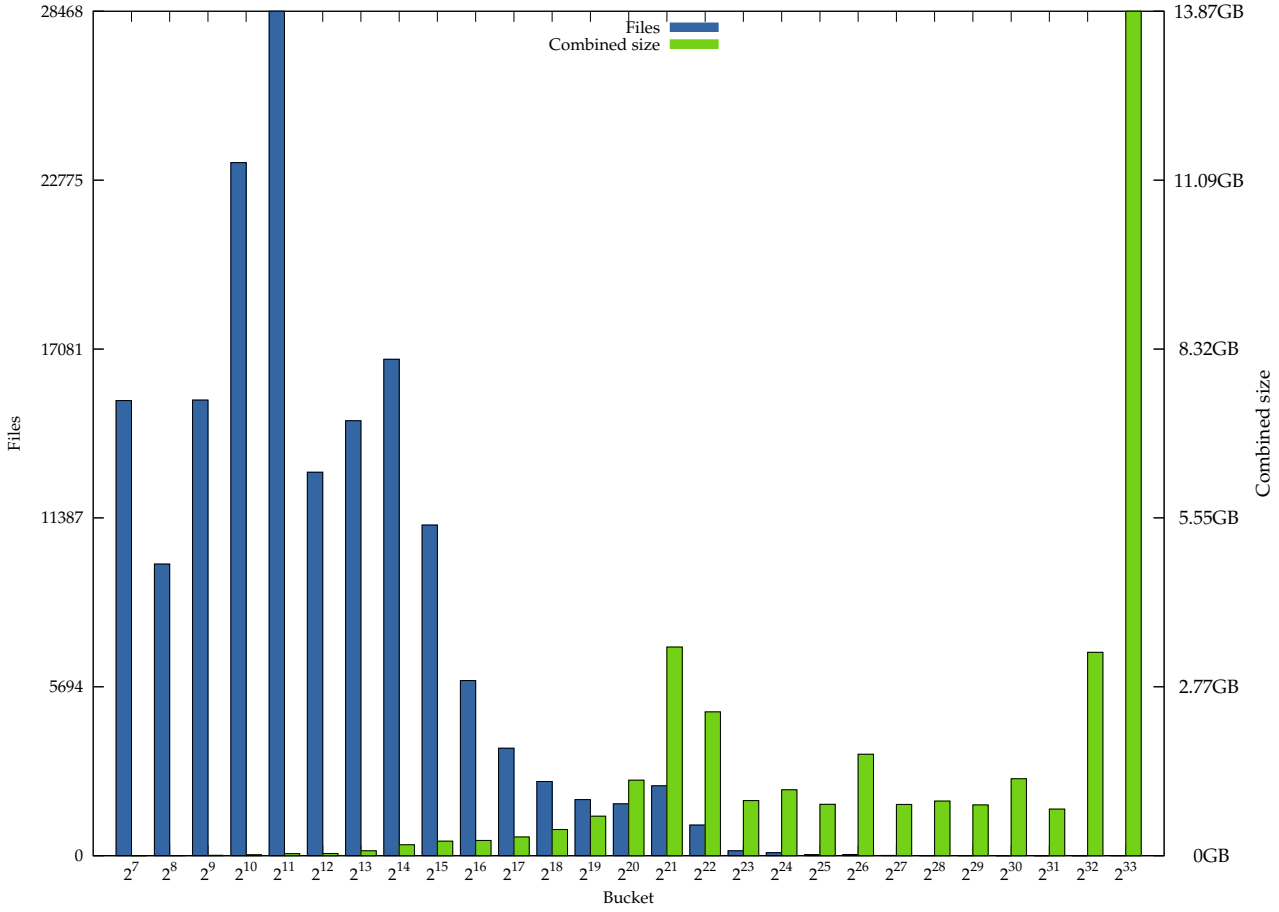

Figure A.7: Number of files and their accumulated sizes, power-of-two buckets. The buckets express upper limits; e.g, the 2^{18} bucket shows the number of files between 2^{17} and $2^{18} - 1$ bytes, and their combined size.

A.6.1 Linux kernel sources

Time usage. Figure A.8 on page 113 shows the time usage of the systems when snapshotting the Linux kernel sources. Our prototype does not do particularly well here, ending up slower than Tarsnap which stores data on the Amazon S3 cloud storage. The fastest is bup, with speeds comparable to `cp`. At its worst, our prototype is 3 times slower than bup (the fastest). The

time usage of Brackup is omitted because it is going off scale (between 16 and 31 minutes).

Time usage when storing via network. Figure A.9 on page 114 shows the Hindsight prototype versus Tarsnap when using Amazon S3 cloud store [85] as the back-end. The network link used was tested to perform around 98 Mb/s download and 58 Mb/s upload just before the benchmarks. Here, our prototype performs closely to Tarsnap, the two being just 17 seconds apart at the widest.

But the prototype is cheating and is exploiting two cores, where Tarsnap is only using one. This gives it an advantage of at most a factor of 2 (we are measuring wall clock time to include network and disk usage).

Another advantage is that our prototype can perform computations in parallel with network transfer. In the last snapshot, our prototype spends 126 seconds in the CPU, with a wall clock time of 111 seconds. Tarsnap, on the other hand, spends just 58 seconds in the CPU, during a total of 93 seconds.

It would be interesting to see how our prototype would perform, if its computations were more efficient, and how much it could gain from performing computations in parallel with network communications.

Note that this benchmark cannot be compared to the other Hindsight benchmarks, as they ran on different systems (one was chosen for its high availability and the other for its Internet connection).

Local space usage. The prototype looks better when comparing space usage of local log data. Figure A.10 on page 115 shows the amount of log data used by the backup systems, and here our prototype is on par with Tarsnap and beaten only in the first snapshots and by Brackup which stores its hash index remotely. When using fixed size blocks, the prototype is consistently lower than Tarsnap, while it is higher with content-aware chunks (rsync-based). This is due to the latter yielding more chunks and thus more hashes to store in the hash index. In this test, the Hindsight prototype seems to follow Tarsnap's development.

Remote space usage. Figure A.11 on page 116 shows the remote space usage of each backup system. Cumulus stores the fewest bytes, which is likely contributed to its use of the aggressive `bzip2` compression scheme, where the other backup systems that apply compression (including our prototype) use `gzip`. The prototype does a bit better than bup and Tarsnap in this test, however all of them are quite close. The prototype configured to use content-aware chunks performs only slightly better than the one using fixed sized chunks. Brackup is clearly falling behind, which is likely due to its lack of compression.

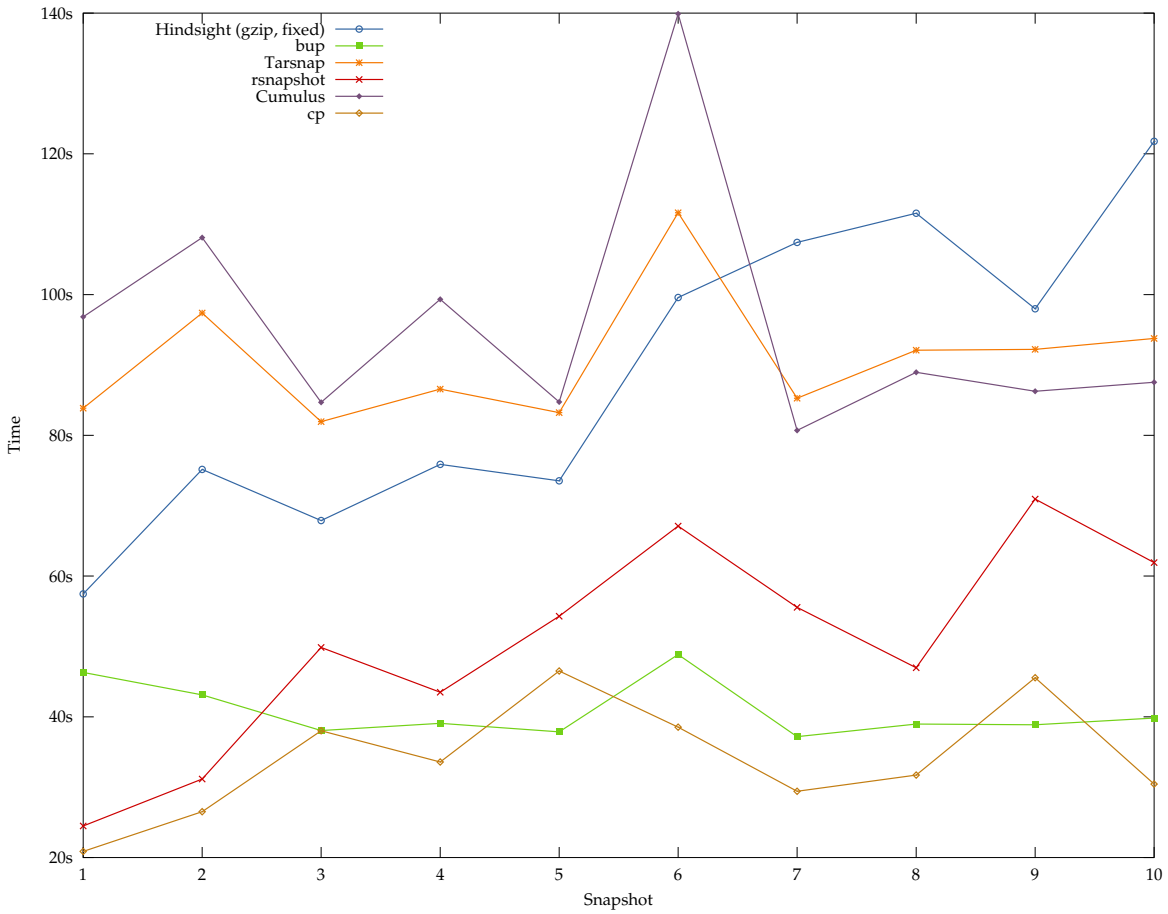

Figure A.8: Time usage during 10 snapshots of the Linux kernel sources.

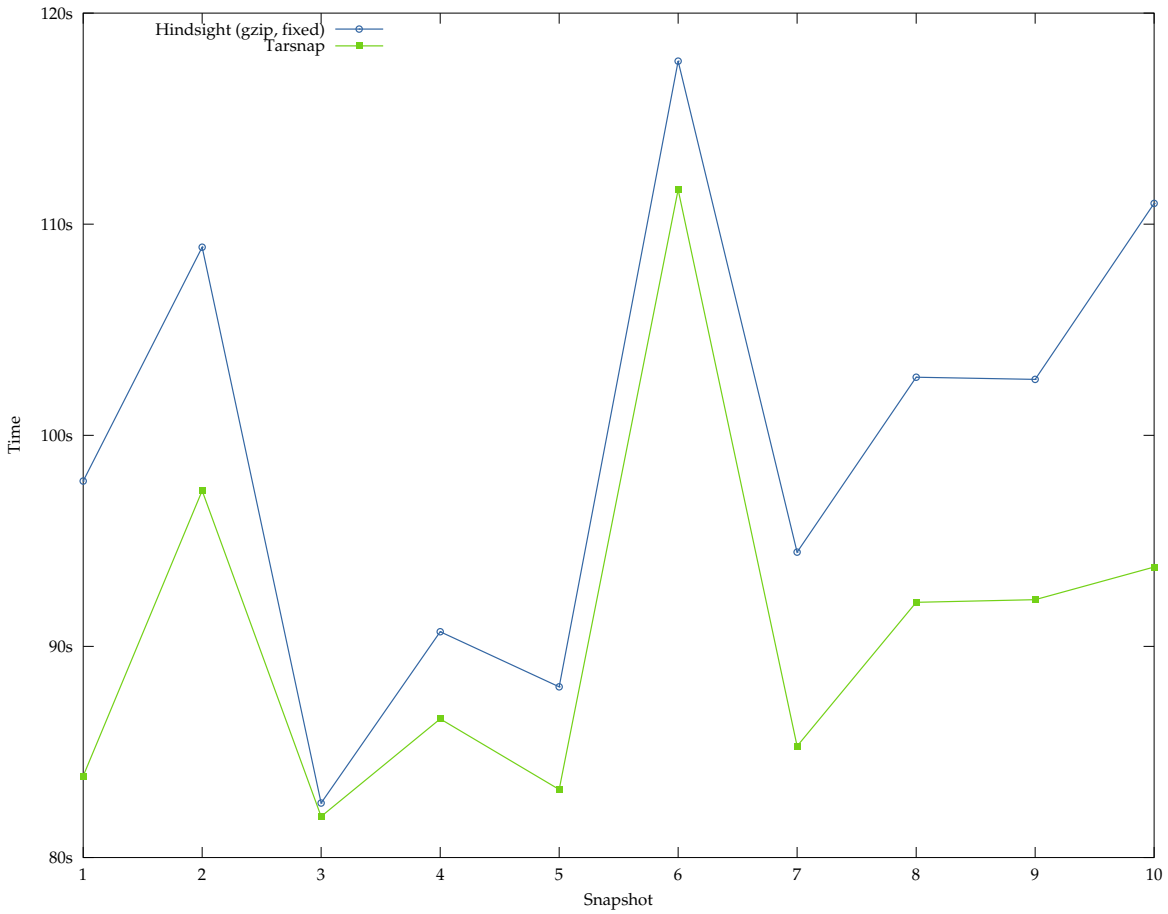

Figure A.9: Time usage during 10 snapshots of the Linux kernel sources.

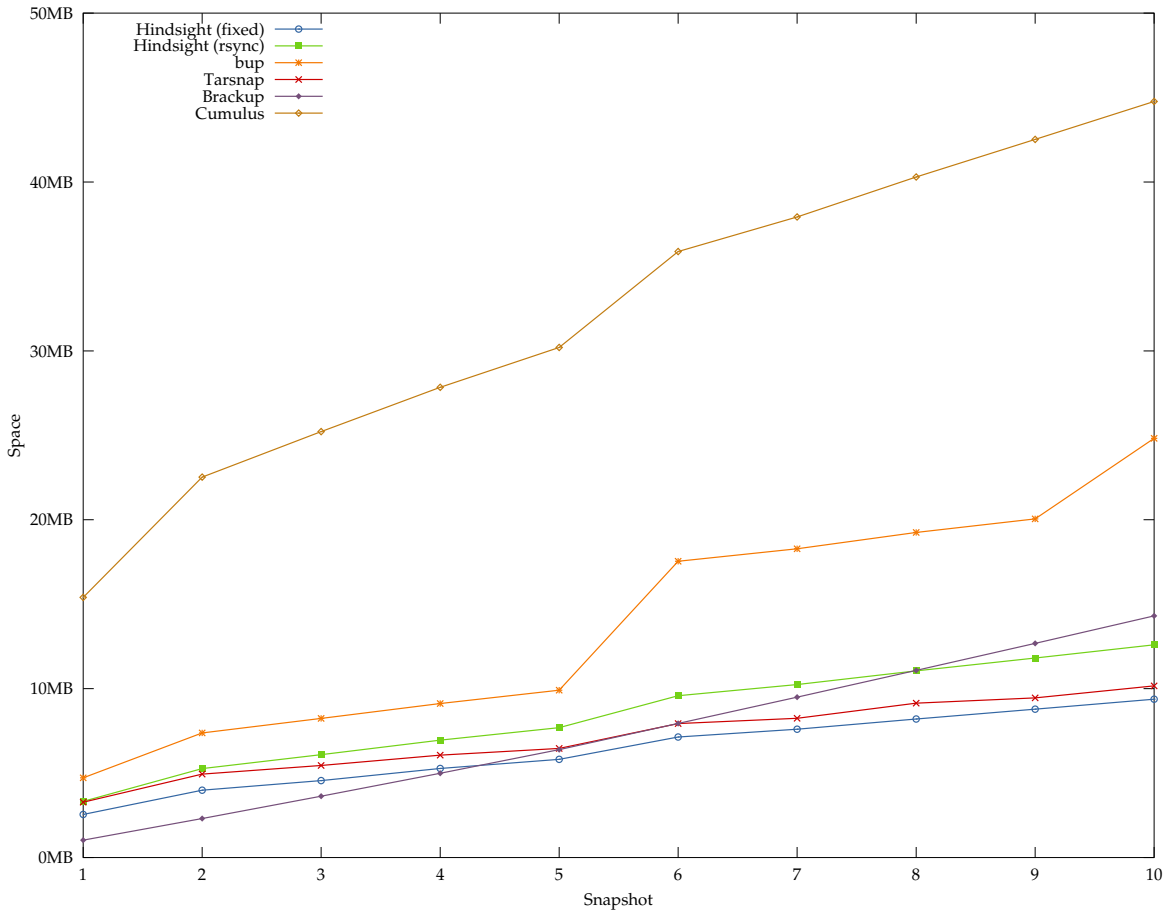

Figure A.10: Local space usage during 10 snapshots of the Linux kernel sources.

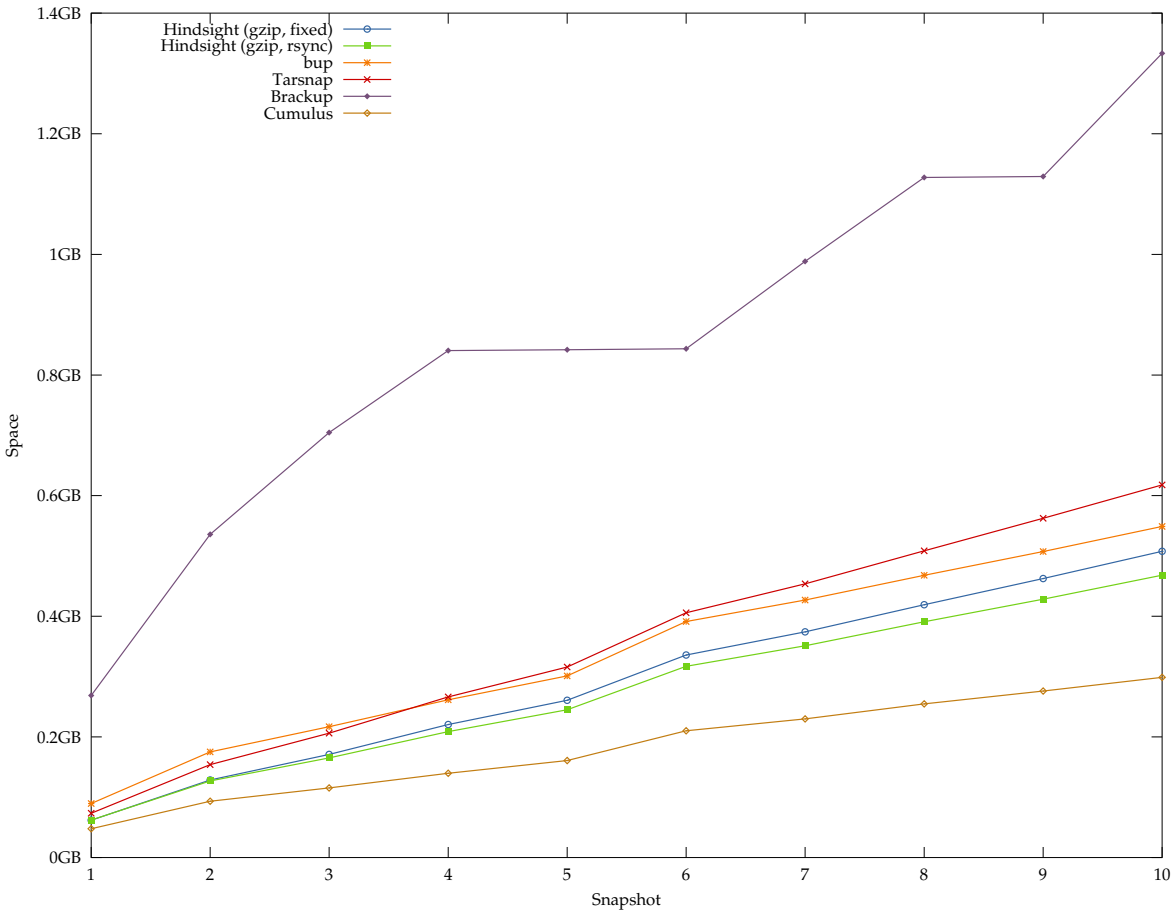

Figure A.11: Remote space usage during 10 snapshots of the Linux kernel sources.

A.6.2 Home directory data

We do not compare time usage spent on the home directory data, since we were not able to measure the timings reliably (the tests ran on two systems differing in hardware and setup).

Local space usage. Figure A.12 on page 118 shows the local log data used by the backup systems when making snapshots of the home directory data. Brackup leads, by storing its hash index remotely, but our prototype is the second best, and seems to win in the long run by only storing a single local key index, and thus avoiding the linear increase in space usage. Tarsnap quickly uses around twice as much local space than our prototype, which we believe

could be caused by less efficient compression. The prototype configured to use content-aware chunks is once again using more local space due to maintaining more chunk hashes. Worst is bup which keeps every key index local.

Remote space usage. When it comes to remote usage, every system seems to follow the same pattern. Tarsnap, bup and our prototype are lowest due to compression, followed by Brackup and rsnapshot (both storing uncompressed data). We guess that Cumulus would have performed even better in this test, had it been included, since it uses the aggressive `bzip2` compression method (as opposed to the faster `gzip`).

The Hindsight prototype. Finally, we compare our prototype in its various configurations. Figure A.14 on page 120 shows the remote space usage of the Hindsight prototype, when using fixed sized and content-aware chunks combined with `gzip` and `snappy` compression. The four graphs follow each other closely, with the `gzip` based graph slightly lower than the `snappy` based ones.

A.6.3 Summary

From our benchmarks we can see that our Hindsight prototype is slow. Not unusable (its seems to scale with more data), but significantly slower than the other systems tested. Not shown in the graphs is its memory use, which make it impractical to use at the moment. The speed of the prototype is likely hurt by its excessive memory usage.

However, the prototype does perform very well with respect to both local and remote space usage. It seems to be on par with the best of the competition, and even better when it comes to local space usage; likely because it can compress where others cannot.

We do not see a lot of gain from using content-aware chunks as opposed to fixed size chunks. Rather we see that the fixed size chunks yield less local space usage, due to there being fewer hashes (fixed size chunks are 64 KB, while our content-aware chunks vary from 4 KB to 64 KB).

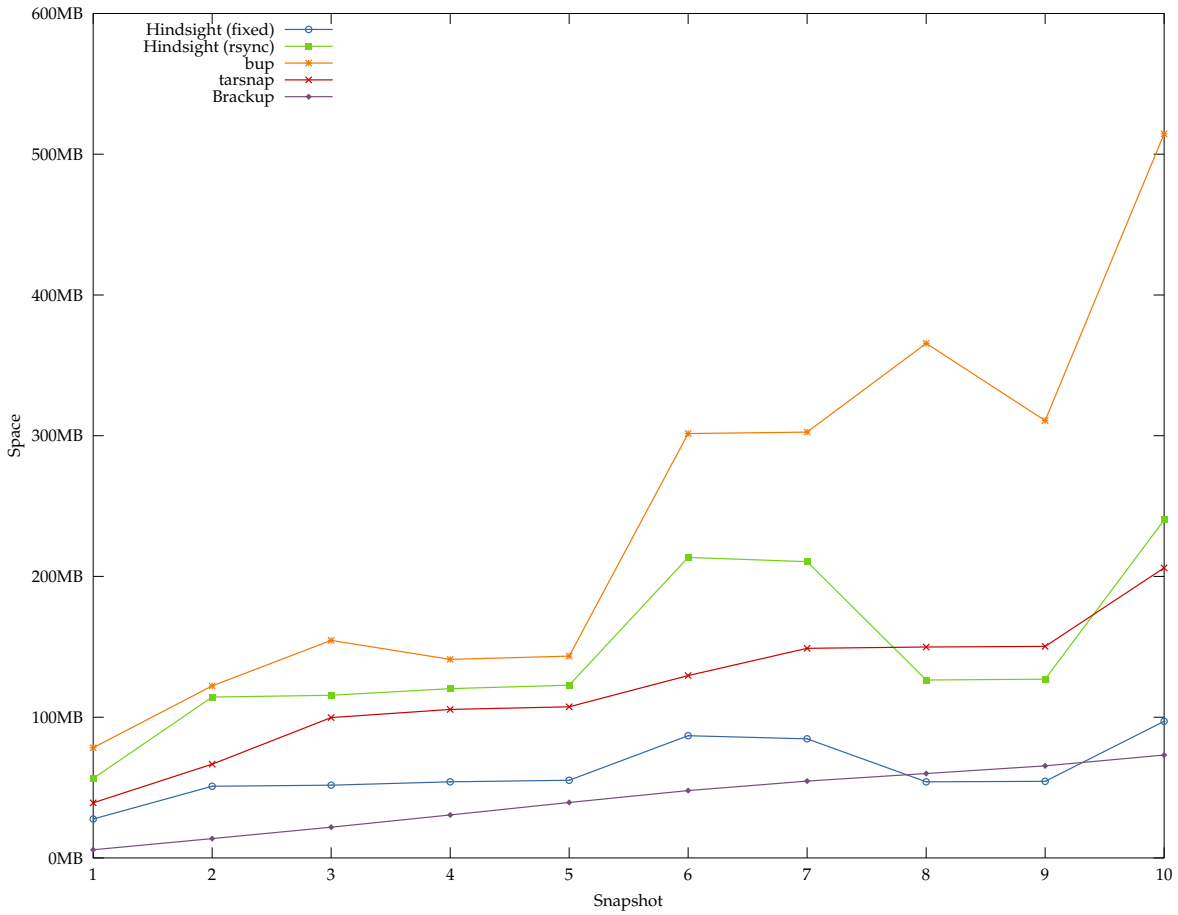

Figure A.12: Local space usage during 10 snapshots of home directory data.

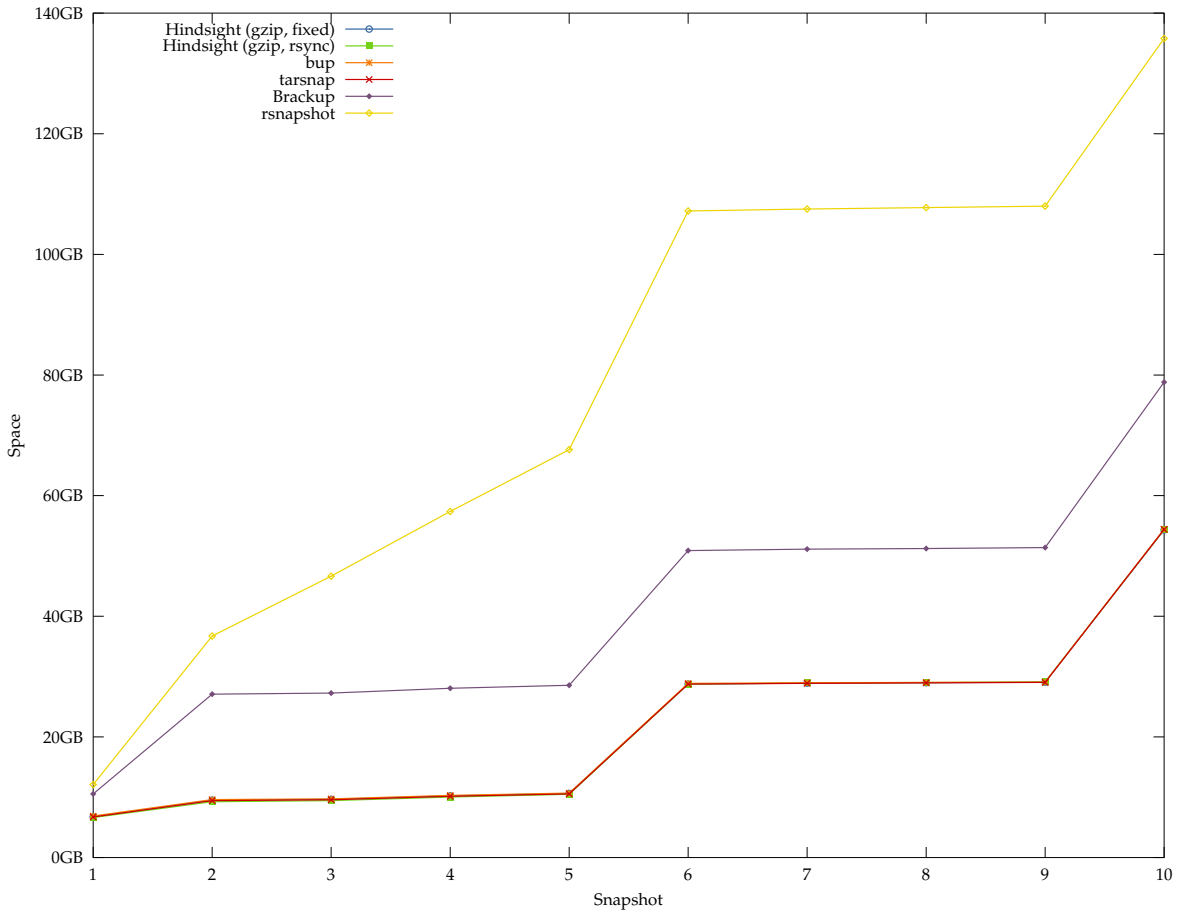

Figure A.13: Remote space usage during 10 snapshots of home directory data. The two versions of Hindsight, bup and Tarsnap are on top of each other.

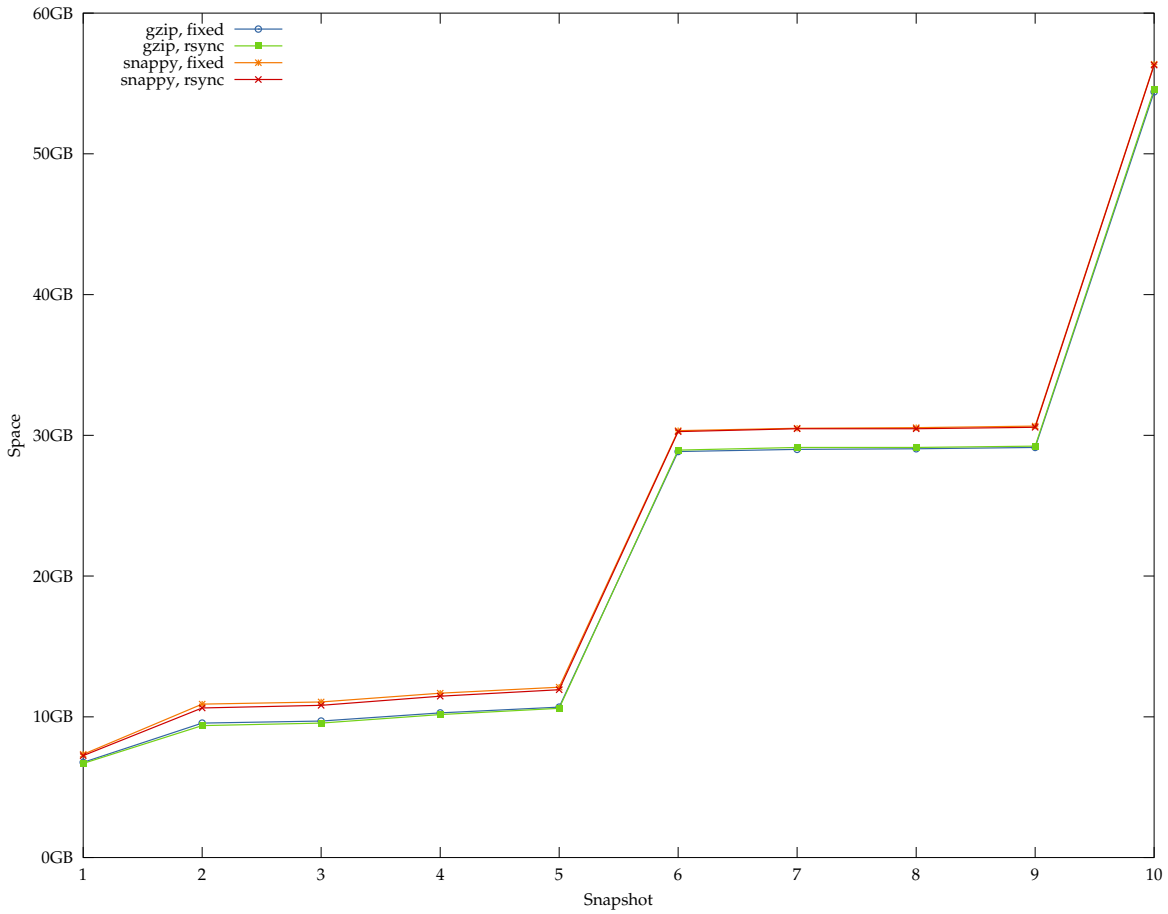

Figure A.14: The Hindsight prototype's space usage during 10 snapshots of home directory data and in four different configurations.

B Die test

The following lists the output from a die test where the Hindsight prototype was set to snapshot a directory with 25,887 files (the 2.6.27.62 version of the Linux kernel sources), while being interrupted with a KILL signal every 70 seconds (the output has been modified slightly for viewing):

```
Taking snapshot
  Calculating size
  Transferring
> 62.64% [3.06 MB/s => 720.92 KB/s]: drivers/net/wireless/b43legacy/sysfs.h
../bin/die-test.sh: line 7: 28708 Killed
Taking snapshot
  Rolling back dangling hashes
  Checking index consistency
  Calculating size
  Transferring
> 94.48% [4.02 MB/s => 873.44 KB/s]: Documentation/video4linux
../bin/die-test.sh: line 7: 29123 Killed
Taking snapshot
  Rolling back dangling hashes
  Checking index consistency
  Calculating size
  Transferring
> 99.76% [6.23 MB/s => 276.80 KB/s]: CREDITS
../bin/die-test.sh: line 7: 29405 Killed
Taking snapshot
  Checking index consistency
  Calculating size
  Transferring
Saving internal state
  Calculating size
  Transferring
```

C Getting and verifying the prebuilt Hindsight 64-bit binary

For convenience, we host a 64-bit binary of our prototype on our website. The following commands retrieves, verifies and unpacks it:

```
# download the tarball and its sum file
wget http://hind.sight.dk/bin/hindsight-64bit.tar.bz2
wget http://hind.sight.dk/bin/hindsight-64bit.tar.bz2.sum.asc

# verify signature file and checksum (see below for expected output)
gpg --verify hindsight-64bit.tar.bz2.sum.asc
sha256sum -c hindsight-64bit.tar.bz2.sum.asc

# unpack it
tar xf hindsight-64bit.tar.bz2

# setup with default configuration
./hindsight/setup.sh

# take first snapshot (named ``foobar'' and of directory ``/some/path'')
./hindsight/run.sh snapshot foobar /some/path
```

If this does not work, consult the `README` file for information on the needed shared libraries (We tested the binary on the Ubuntu 11.10 live CD).

The GPG output from verify must look like (key id `BF122588`):

```
gpg: Signature made Sat 14 Apr 2012 15:27:38 CEST using RSA key ID BF122588
gpg: Good signature from "Johan␣Sejr␣Brinch␣Nielsen␣<brinchj@gmail.com>"
gpg:                 aka "Johan␣Sejr␣Brinch␣Nielsen␣(DK,Denmark)␣<zerrez@gmail.com>"
gpg:                 aka "Johan␣Sejr␣Brinch␣Nielsen␣(DIKU,Denmark)␣<zerrez@diku.dk>"
gpg:                 aka "Johan␣Sejr␣Brinch␣Nielsen␣(DIKU)␣<zerrez@diku.dk>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: DF2D C8D0 AD9F 357E A1ED  C027 89EB E1CB BF12 2588
```

You may need to download the public key used:

```
gpg --search-keys BF122588
```

The `sha256sum` check must give the following output (GPG lines are skipped):

```
hindsight-64bit.tar.bz2: OK
sha256sum: WARNING: 14 lines are improperly formatted
```

For completeness, we also list the SHA-256 checksum of the tarball here:

```
b191420513397d89d239b895ed16637c432e81f05ef0835ac0ef79eec4231320
```

D Correspondence with Thomas Conway on concurrent b-trees

Johan Brinch <zerrez@gmail.com>

Haskell BTrees

10 messages

Johan Brinch <zerrez@diku.dk> **Sun, Dec 18, 2011 at 20:57**
To: drtomc@gmail.com

Hello Thomas Conway,

I have just discovered your mails on Haskell-cafe from 2007 about
btrees in Haskell:
http://haskell.org/pipermail/haskell-cafe/2007-February/022296.html
http://haskell.org/pipermail/haskell-cafe/2007-August/030800.html

I am interested in learning what approach you ended up taking and
whether it was successful performance wise?
Did you do a write up on the process or is the code available somewhere?

Thanks in advance,

--
Johan Brinch,
Dept. of Computer Science,
University of Copenhagen

Thomas Conway <drtomc@gmail.com> **Sun, Dec 18, 2011 at 22:25**
To: Johan Brinch <zerrez@diku.dk>

Hi Thanks for your interest.

Shortly after I wrote those posts, my work took a very different path
(out of information retrieval and into bioinformatics), and I've
barely considered the problem since, unfortunately.

To the extent that I got it all working, the performance was about
what you'd expect from a Haskell implementation - slow compared to
carefully written C++, but with much higher confidence of correctness.
:-)

I have not written up the process further. I can hunt for the code -
it might have been on the machine of my then employer who went bust.

However, if you're interested, I can have a go at reviving the
approach - perhaps collaboratively. I haven't done much Haskell in the
mean time.

T.

[Quoted text hidden]

--
Thomas Conway
drtomc@gmail.com
My friends, love is better than anger. Hope is better than fear.
Optimism is better than despair. So let us be loving, hopeful and
optimistic. And we'll change the world. - Jack Layton

Johan Brinch <zerrez@diku.dk> **Mon, Dec 19, 2011 at 12:42**
To: Thomas Conway <drtomc@gmail.com>

> On Sun, Dec 18, 2011 at 22:25, Thomas Conway <drtomc@gmail.com> wrote:
> > I have not written up the process further. I can hunt for the code -
> > it might have been on the machine of my then employer who went bust.
> >
> > However, if you're interested, I can have a go at reviving the
> > approach - perhaps collaboratively. I haven't done much Haskell in the
> > mean time.
>
> This could be interesting.
> Did you go for immutable nodes and update the path to the root each
> time, or did you reuse node ids?
>
> And did you do to allow concurrent operations? How much locking did you use?
> Did you use any STM code, or just MVar's?
>
> What are your experiences with caching?
> [Quoted text hidden]

Thomas Conway <drtomc@gmail.com> **Tue, Dec 20, 2011 at 04:37**
To: Johan Brinch <zerrez@diku.dk>

> On Mon, Dec 19, 2011 at 10:42 PM, Johan Brinch <zerrez@diku.dk> wrote:
>
> > Did you go for immutable nodes and update the path to the root each
> > time, or did you reuse node ids?
>
> > And did you do to allow concurrent operations? How much locking did you use?
> > Did you use any STM code, or just MVar's?
>
> > What are your experiences with caching?

Let me answer these by outlining the architecture, then visiting some of the details.

What I wanted was a highly concurrent external BTree/B+Tree, modified "in-place".

The first version was an in-memory concurrent B+Tree that used STM - each Node has TVar "pointers" to children.

```
data Node = Internal [(String,TVar Node)] | External [(String,String)]

type BTree = TVar Node
```

Now the problem with making it highly concurrent in the traditional sense is that you have to lock all the nodes from the root to the leaf if making an update. To avoid this, I leveraged the very nice work by your compatriot Kim S Larsen on "relaxed balance". Basically, you allow one transaction to yield a slightly unbalanced tree, and then use a later transaction to repair the balance. The nice property is that each transaction only needs to lock a small number of nodes. In the STM paradigm, this is equivalent to needing to write to only a small number of TVars. Have a look at Kim's papers - they're well written and nicely done.

So once you sort that part out there remains the problem of making them external. The idea here is that you maintain a cache of "paged in" nodes (i.e. BTree pages).

```
type PageAddress = Integer

type Cache = Map PageAddress (TVar Node)
type CachePtr = TVar Cache
```

```
data NodeRef = NodeRef PageAddres

date Node = Internal [(String, NodeRef)] | External [(String,String)]

The operations then take a CachePtr. If a desired PageAddress is not
in the cache, then the transaction can throw an exception which
contains an IO () which when executed loads the desired page and
inserts it into the cache. The driver code can then execute that IO
operation and retry the transaction.

Now this architecture is flawed because there will be a bottleneck
updating the cache. The improved design is to change NodeRef:

type NodePtr = TVar Node
data NodeRef = NodeRef PageAddress (TVar (Maybe Node))

That way the IO () to load the page can be something simple like:

loadPage :: PageAddress (TVar (Maybe Node)) -> ... -> IO ()

loadPage addr dest ... = do
  nd <- readNode addr ....
  atomic $ writeTVar dest (Just nd)

which localises the locking on the cache.

Now we have the problem of flushing dirty nodes, and having the number
of cached pages exceeding the allowable space. This is the part that I
never completely implemented, but here is the design....

Firstly, to handle dirty pages, when a page/node is updated, the tvar
"pointing" to the page is added to a TChan of dirty pages, and the
page is marked as dirty (by adding a bool to the node). A separate
thread[s] can then remove items from the queue, flush the page and
update it to mark it as clean.

The next part - the prevent an oversized cache is a bit complicated.
The naive and almost-good-enough version is to use weak refs so that
the loadPage function can evict pages if necessary. The problem is
that this might cause a page required by this or another operation to
be evicted. To avoid this, there needs to be a register of pages
required by current transactions. I never settled on a good concrete
design for this.

Well, that went on for longer than I expected. I hope it helps. Come
back with questions and we can begin to mock up part of the code if
you like.

T.
ps The existing code that I wrote is probably copyright the company I
used to work for, so I think we should avoid it and produce new code.
--
Dr Thomas Conway
[Quoted text hidden]
```

Johan Brinch <zerrez@diku.dk> **Thu, Dec 22, 2011 at 13:10**
To: Thomas Conway <drtomc@gmail.com>

```
Blank post!

First, I must warn you - I'm going to Sweden for the holidays and I
may not have Internet access. This starts tomorrows, the 23rd and goes
through the 30th.
```

I also think it's time for some context. I'm doing my master thesis in cooperation with a friend, and we've set out to write a fully functional backup system in Haskell, including deduplicity and encryption. We already have a very simple implementation of a b-tree that we use for storing hashes and file paths, but it's rather slow and uses a lot of memory. However, this explains my prior experience with b-tree's and caches ;)

Now to the good part.

I've now read Kim Larsen's paper on relaxed b-tree and I like the idea. Particularly, that the rebalancing logic is decoupled from the update operations. I think this will make the code much cleaner. No conflicts between updates and rebalancing; you just need to ensure that none of the changes breaks the wait-less lookup.

I've taken a look at TCache, a cache with a STM interface:
http://hackage.haskell.org/packages/archive/TCache/0.9/doc/html/Data-TCache.html

However, it has it's drawbacks. Firstly, it's written as a global cache, which means different trees can't use there own cache. Secondly, it's much more complicated than what we need. But it's a good start, and the code seems quite clever.

For a simple cache, we could do a fixed size hash table, i.e. a fixed array of buckets of type [TVar (Ref a)] where (Ref a) is reference to value:

```
data State = Clean | Dirty
data Ref a = Ref State (TVar (Maybe a))
```

Something like that. We may be able to replace Maybe with a weak reference, but I don't fully grasp the semantics of when they're freed yet, so I don't know for sure :)

The cache API could be:

```
class Cache ref (Ref a) where
 getRef :: ref -> Ref a
 put :: ref -> a -> STM ()
 fetch :: ref -> STM a
 delete :: ref -> STM ()
```

This way, the locking logic becomes very simple.

I believe the operations of the BTree needs to be inside the IO monad, since we don't want to run the complete operation as a single transaction (hence locking root). But I must admit, I'm no STM expert - I just read SPJ's paper ;)

The BTree could be something like:

```
type NodeId = Word64
data Node k v = Leaf [(k,v)] | Node [k] [NodeId]

data BTree k v = TVar NodeID -- reference to root
```

Of course, the final structure may also include a channel for dirty nodes (which i think is a great idea!) and the Ref structure may include read/write statistics for the flushing mechanism.

Let me know what you think!
I may start building a simple cache over the holidays :)

I may start building a simple cache over the holidays :)

These mails are getting long, are there any better means of communication?
Perhaps a wiki to summarise the ideas?

[Quoted text hidden]

Johan Brinch <zerrez@diku.dk> **Fri, Jan 6, 2012 at 11:51**
To: Thomas Conway <drtomc@gmail.com>

Hi Thomas,

I've now implemented the following:
0) A transactional hashtable in STM
1) A transactional cache in STM using 0)
2) A B+ tree using 1)

The tree still lacks a few things, like the rebalancing logic for delete, but insert/lookup works.
Rebalancing works in a separate process and lookup is wait-free but with occasional need for retry (due to rebalancing).

I've also written a flush process for the cache, that flushes by the ~least recently used~ strategy.

I currently see about 20k inserts per sec and 50k lookup per sec on my X301 laptop.

The code resides in a private github repository for the time being, but i see no problem in sharing it, if you want to take a look.

There are several things I'd like to improve, but it works, and since our thesis is due in April, we'll focus on the things we need for the project for now. I would however like to see this as an open source B+ tree in hackage at some point (perhaps May).

Not much is needed for a release and I even have a pretty sweet quick check suite ready, to verify it automatically. We could even to several QC checks concurrently.

[Quoted text hidden]

Thomas Conway <drtomc@gmail.com> **Mon, Jan 9, 2012 at 01:23**
To: Johan Brinch <zerrez@diku.dk>

That's great. Sorry for the longish silence - everything shuts down round here from Christmas till after New Year. I'm back, but we've been busy getting a paper together for the ISMB deadline.

I composed a long email which I only got half way through just before christmas which looks to be redundant now. :-)

WRT delete, one of the Larsen papers does go in to detail. It turns out that for the same reason that insert is simpler with relaxed balance, delete is also relatively simple. Perhaps you knew this already. :-)

Have you tested things to see how they scale as you increase concurrency? To me that is the most interesting part.

[Quoted text hidden]

--

[Quoted text hidden]

Johan Brinch <zerrez@diku.dk> **Tue, Feb 7, 2012 at 16:50**
To: Thomas Conway <drtomc@gmail.com>

> On Mon, Jan 9, 2012 at 01:23, Thomas Conway <drtomc@gmail.com> wrote:
> > WRT delete, one of the Larsen papers does go in to detail. It turns
> > out that for the same reason that insert is simpler with relaxed
> > balance, delete is also relatively simple. Perhaps you knew this
> > already. :-)

I'm aware of this, but it's going to be after the master project.
We've freezed the code until the report is ready.

> > Have you tested things to see how they scale as you increase
> > concurrency? To me that is the most interesting part.

It doesn't seem to scale well beyond 2 cores. I don't know why. The
tree should allow for multiple updates without too many STM conflicts.
It does however gain some from the threaded runtime - even on a single
core. I guess this is due to threads working while others are waiting
for disk. We've tried running it on 8 cores, and even though it does
use tons of CPU (500% or so) it doesn't use less wall time. Could seem
like STM is conflicting somewhere.

Btw, i've implemented to more features:

1) A generation concept that allow a thread to freeze the current
version of the tree and query it while other workers are still
updating. One such version can be maintained until no one is using it
anymore. This is implemented directly in the cache on a page level.

2) One can hint the underlying cash about dependencies: This page must
be written before that page. We use this to hint that tree nodes must
be written before the parent who's referencing it (bottom-up). Nodes
without hints (e.g. deleted nodes) are written after nodes with hints
(updated nodes). This way we can flush the tree while maintaining a
consistent external state at all times. And by 1) while other workers
are still updating it.

[Quoted text hidden]

Johan Brinch <zerrez@diku.dk> **Thu, Mar 8, 2012 at 10:05**
To: Thomas Conway <drtomc@gmail.com>

Hey Thomas,

I need your acceptance for us to include our email correspondance as
an appendix of our master thesis. Of course, you are mentioned in the
thesis itself for your guidance ;)

Also, I'm going to work more on the tree after the project, and see if
I can improve performance further.
I have some ideas, but most of them require the tree to be locked to
bytestring keys and bytestring values. Do you think this would be too
strict?

[Quoted text hidden]

Thomas Conway <drtomc@gmail.com> **Thu, Mar 8, 2012 at 10:16**
To: Johan Brinch <zerrez@diku.dk>

That will be fine. Sorry I haven't been more communicative! Life gets busy as you know, I am sure.

[Quoted text hidden]

[Quoted text hidden]

Bibliography

[1] Bitcasa. URL http://goo.gl/yVq0f.

[2] brackup: Flexible backup tool. Slices, dices, encrypts, and sprays across the net. URL http://goo.gl/zF3BS.

[3] Dropbox. URL http://goo.gl/BGjRd.

[4] Git: the fast version control system. URL http://git-scm.com.

[5] Apple Time Machine. URL http://goo.gl/OEHld.

[6] Yesod Web Framework. URL http://goo.gl/KS8oD.

[7] Oracle key-compressed index, 2008. URL http://goo.gl/nfriC.

[8] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet: a survey of distributed garbage collection. ACM Comput. Surv., 30(3):330–373, September 1998. ISSN 0360-0300.

[9] AES. Advanced encryption standard. URL http://goo.gl/NdHfh.

[10] Jesper Louis Andersen. A concurrent bittorrent client. URL http://goo.gl/jnBXP.

[11] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. Sha-3 proposal blake. Technical report, December 2010.

[12] Henry G. Baker. Infant mortality and generational garbage collection. SIGPLAN Not., 28(4):55–57, April 1993. ISSN 0362-1340.

[13] Elaine Barker and Allen Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. SP-800-131a, U.S. DoC/National Institute of Standards and Technology, 2011. See http://goo.gl/Ti2Eo/.

[14] Fernando "Brujo" Benavides. *Very* basic Erlang-like process support for Haskell. URL http://goo.gl/Qcnpr.

[15] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005. URL http://goo.gl/iFSBz.

[16] Daniel J. Bernstein. The poly1305-aes message-authentication code. 03 2005. URL http://goo.gl/ffVga.

[17] Daniel J. Bernstein. Extending the Salsa20 nonce. 11 2008. URL http://goo.gl/ZRa7K.

[18] Daniel J. Bernstein. Cryptography in NaCl. 03 2009. URL http://goo.gl/lqQ7K.

[19] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. 01 2011. URL http://goo.gl/E5B5C.

[20] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak specifications. Technical report, October 2008.

[21] David I Bevan. An efficient reference counting solution to the distributed garbage collection problem. Parallel Computing, 9(2):179 – 192, 1989. ISSN 0167-8191.

[22] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In Proceedings of the 35th SIGMOD international conference on Management of data, SIGMOD '09, pages 283–296, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2.

[23] Christopher M. Bishop. Pattern Recognition and Machine Learning (Information Science and Statistics). Springer, 1st ed. 2006. corr. 2nd printing edition, October 2007. ISBN 0387310738.

[24] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422–426, July 1970. ISSN 0001-0782.

[25] Hans-J. Boehm. Dynamic memory allocation and garbage collection. Comput. Phys., 9(3):297–303, May 1995. ISSN 0894-1866.

[26] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. Softw. Pract. Exper., 18(9):807–820, September 1988. ISSN 0038-0644.

[27] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In ACM International Conference on Management of Data (SIGMOD 1995), 1995.

[28] Andrei Z. Broder. On the resemblance and containment of documents. In In Compression and Complexity of Sequences (SEQUENCES'97, pages 21–29. IEEE Computer Society, 1997.

[29] George Candea and Armando Fox. Crash-only software. In Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9, HOTOS'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[30] David Chase. GC FAQ – draft. URL `http://goo.gl/YkPKL`.

[31] Jonathan D. Cohen. Recursive hashing functions for n-grams. ACM Trans. Inf. Syst., 15(3):291–320, July 1997. ISSN 1046-8188.

[32] Douglas Comer. Ubiquitous b-tree. ACM Comput. Surv., 11:121–137, June 1979. ISSN 0360-0300.

[33] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction To Algorithms. Mit Press, 2001. ISBN 9780262032933.

[34] On P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: making backup cheap and easy. In OSDI: Symposium on Operating Systems Design and Implementation, pages 285–298, 2002.

[35] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In Proceedings of the 12th conference on USENIX Security Symposium - Volume 12, SSYM'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[36] James da Silva and Blair Zajac. AMANDA: Advanced Maryland Automatic Network Disk Archiver. URL `http://goo.gl/wXJ1r`.

[37] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. Commun. ACM, 21(11):966–975, November 1978. ISSN 0001-0782.

[38] Anthony Discolo, Tim Harris, Simon Marlow, Simon Jones, and Satnam Singh. Lock free data structures using stm in haskell. pages 65–80. 2006.

[39] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. SIGMETRICS Perform. Eval. Rev., 27:59–70, May 1999. ISSN 0163-5999.

[40] John R. Douceur, John R. Douceur, Atul Adya, Atul Adya, William J. Bolosky, William J. Bolosky, Dan Simon, Dan Simon, Marvin Theimer, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS, 2002.

[41] Jeff Epstein. Cloud haskell. URL `http://goo.gl/FVUQT`.

[42] Ben Escoto and Kenneth Loafman. Duplicity: Encrypted bandwidth-efficient backup using the rsync algorithm. URL `http://goo.gl/jwzSk`.

[43] Niels Ferguson and Bruce Schneier. Practical Cryptography. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003. ISBN 0471223573.

[44] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family. Technical report, 2009.

[45] Apache Software Foundation. Apache CouchDB. URL `http://goo.gl/h8cPm`.

[46] Larry Freeman. Looking Beyond the Hype: Evaluating Data Deduplication Solutions. 1988.

[47] fuse. FUSE: Filesystem in Userspace. URL `http://goo.gl/dU7mV`.

[48] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a sha-3 candidate. Technical report, March 2011.

[49] Søren Hansen. Surveilr: A different sort of monitoring system. URL `http://goo.gl/Noea0`.

[50] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9.

[51] Ragib Hasan, Suvda Myagmar, Adam J. Lee, and William Yurcik. Toward a threat model for storage systems. In Proceedings of the 2005 ACM workshop on Storage security and survivability, StorageSS '05, pages 94–102, New York, NY, USA, 2005. ACM. ISBN 1-59593-233-X.

[52] Val Henson. An analysis of compare-by-hash. In Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9, HOTOS'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[53] Tim Hickey and Jacques Cohen. Performance analysis of on-the-fly garbage collection. Commun. ACM, 27(11):1143–1154, November 1984. ISSN 0001-0782.

[54] Simon P. Jones. Beautiful Concurrency. O'Reilly Media, Inc., 2007. ISBN 0596510047.

[55] Kabooza. Kabooza global backup survey. Technical report, Kabooza, Stockholm, Sweden, 2009.

[56] Oleg Kiselyov. Iteratees. URL `http://goo.gl/nQAxo`.

[57] Oleg Kiselyov and John W. Lato. Iteratee-based I/O. URL `http://goo.gl/fTSds`.

[58] Alexander Klink and Julian Wälde. Denial of Service through hash table multi-collisions. December 2011.

[59] Kim S. Larsen and Rolf Fagerberg. B-trees with relaxed balance. In Proceedings of the 9th International Parallel Processing Symposium, pages 196–202. IEEE Computer Society Press, 1993.

[60] Nate Lawson. Timing attack in Google Keyzcar, 2009. URL `http://goo.gl/kNY27`.

[61] U. Maheshwari and B.H. Liskov. Fault-tolerant distributed garbage collection in a client-server object-oriented database. In Proceedings of the Third International Conference on Parallel and Distributed Information Systems, pages 239–248. IEEE, 1994.

[62] Udi Manber. Finding similar files in a large file system. In Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference, pages 2–2, Berkeley, CA, USA, 1994. USENIX Association.

[63] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new `ext4` filesystem: current status and future plans. In Proceedings of the Linux Symposium, pages 21–37, Ontario, Canada, June 2007. URL `goo.gl/260wb`.

[64] Ralph C. Merkle. A digital signature based on a conventional encryption function. In A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology, CRYPTO '87, pages 369–378, London, UK, UK, 1988. Springer-Verlag. ISBN 3-540-18796-0.

[65] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In Proceedings of the 9th USENIX conference on File and stroage technologies, FAST'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association. ISBN 978-1-931971-82-9.

[66] John Millikin. Enumerator: Simple, Efficient Incremental IO for Haskell. URL `http://goo.gl/wznyG`.

[67] Michael Mitzenmacher. Compressed bloom filters. IEEE/ACM Trans. Netw., 10(5):604–612, October 2002. ISSN 1063-6692.

[68] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. SIGOPS Oper. Syst. Rev., 35(5):174–187, October 2001. ISSN 0163-5980.

[69] NIST. Cryptographic Hash Algorithm Competition. Technical report, November 2007.

[70] openssl.org. Timing-based attacks on RSA keys. URL `http://goo.gl/znNGf`.

[71] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. An efficient on-the-fly cycle collection. ACM Trans. Program. Lang. Syst., 29(4), August 2007. ISSN 0164-0925.

[72] Avery Pennarun. bup: It backs things up. URL `http://goo.gl/Tjdg6`.

[73] Colin Percival. Tarsnap: Online backups for the truly paranoid. URL `http://goo.gl/CYUsT`.

[74] Colin Percival and Taylor R Campbell. Tarsnap critical security bug. URL `http://goo.gl/Dq3ie`.

[75] pgsql hackers. Postgresql prefix b-tree discussion, 2005. URL `http://goo.gl/mFbtd`.

[76] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from bell labs. In In Proceedings of the Summer 1990 UKUUG Conference, pages 1–9, 1990.

[77] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage, 2002.

[78] M O Rabin. Fingerprinting by random polynomials. Technical Report TR1581 Center for Research in, (TR-15-81):15–18, 1981.

[79] Nikolaus Rath. S3QL: A full-featured file system for online data storage. URL `http://goo.gl/2LHiR`.

[80] Marsh Ray. SSL flaw in renegotiation logic. URL `http://goo.gl/l5LVB`.

[81] Impulse Research. Data backup survey. Technical report, Impulse Research, Los Angeles, California, 2010.

[82] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In USENIX 2008 Annual Technical Conference on Annual Technical Conference, ATC'08, pages 143–156, Berkeley, CA, USA, 2008. USENIX Association.

[83] Nathan Rosenquist, David Cantrell, and David Keegel. rsnapshot: A remote filesystem snapshot utility, based on rsync. URL `http://goo.gl/n4BLa`.

[84] Bruce Schneier. Applied cryptography (2nd ed.): protocols, algorithms, and source code in C. John Wiley & Sons, Inc., New York, NY, USA, 1995. ISBN 0-471-11709-9.

[85] Amazon Web Services. Amazon Simple Storage Service. URL `http://goo.gl/wXJ1r`.

[86] Nir Shavit and Dan Touitou. Software transactional memory. In Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3.

[87] Dilip Simha. The Art of Data Deduplication, September 2011.

[88] Michael Snoyman. Conduit: Streaming data processing library. URL `http://goo.gl/p7Pio`.

[89] Michael Szydlo. Merkle tree traversal in log space and time. In Christian Cachin and Jan Camenisch, editors, Advances in Cryptology - EUROCRYPT 2004, volume 3027 of Lecture Notes in Computer Science, pages 541–554. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-21935-4.

[90] Yujuan Tan, Hong Jiang, Dan Feng, Lei Tian, Zhichao Yan, and Guohui Zhou. Sam: A semantic-aware multi-tiered source de-duplication framework for cloud backup. In Proceedings of the 2010 39th International Conference on Parallel Processing, ICPP '10, pages 614–623, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4156-3.

[91] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical report, Computer Science, Australian National University, Canberra, Australia, 1996.

[92] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In Proceedings of the 16th Annual Symposium on Foundations of Computer Science, SFCS '75, pages 75–84, Washington, DC, USA, 1975. IEEE Computer Society.

[93] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Cumulus: Filesystem backup to the cloud. Trans. Storage, 5:14:1–14:28, December 2009. ISSN 1553-3077.

[94] Paul R. Wilson. Uniprocessor garbage collection techniques. In Proceedings of the International Workshop on Memory Management, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag. ISBN 3-540-55940-X.

[95] Hongjun Wu. The Hash Function JH. Technical report, January 2011.

[96] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In Proceedings of the 6th USENIX Conference on File and Storage Technologies, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.