Master's thesis

Venti analysis and memventi implementation

Designing a trace-based simulator and implementing a venti with in-memory index

Mechiel Lukkien
mechiel@xs4all.nl

August 8, 2007

Committee:
prof. dr. Sape J. Mullender
ir. J. Scholten
ir. P.G. Jansen

Faculty of EEMCS
DIES, Distributed and Embedded Systems
University of Twente
Enschede, The Netherlands

Abstract

[The next page has a Dutch summary]

Venti is a write-once content-addressed archival storage system, storing its data on magnetic disks: each data block is addressed by its 20-byte SHA-1 hash (called *score*).

This project initially aimed to design and implement a trace-based simulator matching Venti behaviour closely enough to be able to use it to determine good configuration parameters (such as cache sizes), and for testing new optimisations. A simplistic simulator has been implemented, but it does not model Venti behaviour accurately enough for its intended goal, nor is it polished enough for use. Modelled behaviour is inaccurate because the advanced optimisations of Venti have not been implemented in the simulator. However, implementation suggestions for these optimisations are presented.

In the process of designing the simulator, the Venti source code has been investigated, the optimisations have been documented, and disk and Venti performance have been measured. This allowed for recommendations about performance, even without a simulator.

Beside magnetic disks, also flash memory and the upcoming MEMS-based storage devices have been investigated for use with Venti; they may be usable in the near future, but require explicit support.

The focus of this project has shifted towards designing and implementing memventi, an alternative implementation of the venti protocol. Memventi keeps a small part of the scores in main memory for lookups (whereas Venti keeps them on separate index disks). Memventi therefore does not scale to high storage capacity, but is much easier to set up, much simpler in design, and has good performance.

Abstract

[The remainder of this report is written in English.]

Venti is een programma dat blokken data opslaat op magnetische harde schijven. Een eenmaal geschreven blok data kan nooit meer verwijderd of overschreven worden; het kan uitsluitend worden geadresseerd aan de hand van de 20-byte SHA-1 hash (genaamd de *score*) van de data.

In eerste instantie was het doel van dit project het ontwerpen en implementeren van een *trace-based* simulator die Venti's gedrag nauwkeurig genoeg simuleert om gebruikt te kunnen worden voor het bepalen van configuratieparameters zoals groottes van de diverse *caches*, maar ook voor het testen van nieuwe optimalisaties. De geïmplementeerde simulator is echter niet nauwkeurig genoeg om hiervoor te kunnen worden gebruikt, en is ook niet genoeg doorontwikkeld voor normaal gebruik. Het gemodelleerde gedrag wijkt te veel af voornamelijk doordat de geavanceerde optimalisaties die in Venti aanwezig zijn niet in de simulator zijn geïmplementeerd. Wel worden ontwerpen voor, en opmerkingen over mogelijke simulatorimplementaties van deze optimalisaties vermeld.

Tijdens het ontwerpen van de simulator is de broncode van Venti gelezen, zijn de aanwezige optimalisaties gedocumenteerd, en zijn benchmarks van harde schrijven en van Venti uitgevoerd. Met het inzicht dat hiermee verkregen is, kunnen ook zonder simulator aanbevelingen worden gedaan over goed presenterende Venti configuraties.

Behalve magnetische harde schijven zijn ook flash geheugen en de nog in ontwikkeling zijnde MEMS-schijven onderzocht voor gebruik met Venti. Ze zullen in de toekomst te gebruiken zijn, maar hebben dan expliciete ondersteuning nodig in Venti.

De focus van dit project is verschoven naar het ontwerpen en implementeren van memventi, een alternatieve implementatie van het venti protocol. Memventi houdt van elke aanwezige *score* een klein deel van de 20 bytes in het geheugen, precies genoeg om de kans op dubbelen hierin klein genoeg te houden (Venti zelf bewaart alle scores op index schijven). Memventi gebruikt relatief weinig geheugen, maar schaalt niet naar de hoge opslagcapaciteit waar Venti wel naar schaalt. Memventi is echter veel eenvoudiger in ontwerp en om in gebruik te nemen, en levert goede prestaties.

Preface

I would like to thank everyone who has helped me with my master's project, both on technical and non-technical level. This master's thesis has taken me longer to finish than I (and others) had hoped, but I am content with the results. I am very thankful to Sape Mullender for supervising this project, helping with technical issues and being patient and helpful from start to finish. The same goes for Axel Belinfante, our regular progress talks have been very useful on the technical level and have helped keeping this thesis on track.

I hope you enjoy reading this thesis. I am continuing development of tools related to Venti, so please direct further questions or discussion to my e-mail address, mechiel@xs4all.nl.

Mechiel Lukkien
Enschede, July 11th 2007

Contents

1 Introduction **13**
1.1 Protocol . . . 15
1.2 Hash trees . . . 15
1.3 Performance requirements . . . 18
1.4 Report . . . 20

2 Venti design **21**
2.1 Index sections, Arenas . . . 21
2.2 Terminology . . . 22
2.3 Optimisations . . . 25
2.3.1 Disk block cache, dcache . . . 25
2.3.2 Index entry cache, icache . . . 26
2.3.3 Lump cache, lcache . . . 26
2.3.4 Bloom filter . . . 27
2.3.5 Queued writes . . . 27
2.3.6 Lump compression . . . 27
2.3.7 Disk block read-ahead . . . 28
2.3.8 Lump read-ahead . . . 28
2.3.9 Prefetch index entries . . . 28
2.3.10 Sequential writes of index entries . . . 28
2.3.11 Sequential writes of disk blocks . . . 28
2.3.12 Opportunistic hash collision checking . . . 29
2.3.13 Scheduler . . . 29
2.3.14 Zero-copy packets . . . 29

3 Venti clients **31**
3.1 Vac . . . 31
3.2 Fossil . . . 34
3.3 Conclusion . . . 35

4 Disks & performance **37**
4.1 Disk internals . . . 37
4.1.1 Implications for Venti . . . 39
4.2 Operating system disk handling . . . 40
4.3 Testing hard disks . . . 40
4.4 SCSI disk results . . . 42
4.5 IDE disk results . . . 44
4.6 Conclusion . . . 45

4.7 Alternatives to magnetic disks . . . 46
4.7.1 Compact-flash memory . . . 46
4.7.2 MEMS-based storage . . . 50

5 Venti performance **53**
5.1 Basic Venti performance . . . 53
5.1.1 Analysis . . . 54
5.2 SHA-1 performance . . . 55
5.3 Whack performance . . . 55

6 Venti simulator **57**
6.1 Design . . . 58
6.2 Trace files . . . 60
6.3 Vsim and vtrace . . . 62
6.4 Future work . . . 63
6.4.1 Bloom filter . . . 63
6.4.2 Index entry prefetch from arena directory . . . 63
6.4.3 Better disk model and multiple disks . . . 64
6.5 Conclusions . . . 64

7 Memventi design & implementation **67**
7.1 Storing only a part of the score . . . 68
7.2 Implementation . . . 70
7.2.1 Features . . . 71
7.2.2 Data integrity . . . 73
7.2.3 Differences with design . . . 74
7.2.4 Performance . . . 75
7.2.5 Problems . . . 79
7.2.6 Future work . . . 79
7.2.7 Conclusion . . . 82

8 Conclusions **83**

9 Future work **87**

Bibliography **93**

A Test setup **95**

B Disk tests **97**
B.1 SCSI disk . . . 97
B.2 IDE disk . . . 97

C Basic Venti tests **103**
C.1 sametest . . . 103

D Memventi tests **105**

E Compact flash tests **107**
E.1 IDE to flash adaptor . . . 107
E.2 USB flash memory card reader . . . 107

F Technical documentation **111**
F.1 Pseudo code for handling venti operations 111

G Tools **115**
G.1 ptio.c 115
G.2 test-sha1.c 120
G.3 test-whack.c 121

Chapter 1

Introduction

Venti [1, 2, 3] is a write-once content-addressed archival block storage server. It is a user-space daemon communicating with software clients over TCP connections. The blocks stored can be up to 56KB in size. Once stored, blocks can never be removed. The address of the data is the 160-bits SHA-1 hash (called a *score*) of that data. SHA-1 is a cryptographic secure hash, the implications of which are explained later in this section. Data blocks are stored on magnetic disk allowing for fast random access.

In essence, Venti provides the following two functions:

1. *write*(*data*, *type*) → *score*
2. *read*(*score*, *type*) → *data*

Venti only stores data, and retrieves it later on request. It does not interpret the data, nor does it need to. The *type* parameter to the read and write functions is part of the address, and used mostly for convenience, such as easy debugging. It is not an essential part of Venti or Venti clients. Venti is a building block for storage applications, and therefore it is not very useful all by itself. For example, the default file system in Plan 9[4], Fossil [5, 6], uses Venti as a permanent backing store. The backup utility Vac [7] uses Venti as backing store. Vacfs [8] transparently presents the backed-up file tree as a file system. Vbackup [9] writes a disk image to Venti, and vnfs [9] serves the files in the file system of that disk image back using the NFSv3 protocol.

Venti only stores and retrieves blocks of data; once stored, blocks can never be removed. The use of the score of the data as an address results in some interesting storage properties. If the same block of data is written to Venti multiple times, it is stored only once. Duplicate writes are easily detected: the address has already been written to. This is called write-coalescing. When a write-request comes in, Venti calculates the score, i.e. the 'address' and looks it up in an index. If the score is already present, Venti can safely assume the data is already present. This follows from the fact that SHA-1 is a secure cryptographic hash: it is collision-resistant and preimage resistant. Collision-resistant means that it is computationally infeasible to calculate differing data m_1 and m_2 that have the same SHA-1 hash. Preimage resistant means that it is computationally infeasible to either calculate data m_1 to go with a given score, or to calculate data m_2 differing from given data m_1 that has the same score. Finding a collision

(known as a *birthday attack*) takes $2^{n/2}$ operations, with $n = 160$ (the number of bits in the hash) for SHA-1, under the assumption that it has not been broken. A preimage attack takes 2^n operations. The vast amount of CPU cycles needed for these operations are not within reach of modern and immediately future computers. The probability that different data stored in Venti have the same SHA-1 hash is very low likewise. The example from the original Venti paper[1] states that a Venti storing an exabyte of data (10^{18} bytes) in 8KB blocks (10^{14} blocks) has a probability of less than 10^{-20} of a collision. A score can therefore safely be used as a unique identifier. Thus, after the first write of data, subsequent writes of that same data can safely be 'ignored' and marked as a successful write. This can be very useful in backups, where a complete copy of data may be written every day. Only modified data will result in new data being stored. Thus, disk storage is used efficiently and transparently to clients. Traditional file systems have no way to coalesce occurrences of identical data. Fossil, and any Venti client for that matter, gets this feature for free. Files stored multiple times in Fossil will be written to Venti multiple times, but Venti only stores it once.

Since the address is determined by the data, one cannot request Venti to write different data on an address of choice, or an address that already references data. Writing something at an address already present in Venti automatically means the data is identical to the data already present at that address. Stored blocks cannot be removed, there is no way of telling Venti to remove a block of data. At first, this might feel wrong with respect to security and privacy. Anyone who can connect to a Venti server can request any block of data. However, one can only retrieve data one has an address for. Requesting random addresses (160 bits long), yields an extremely low probability of actually retrieving data. Only when an address falls into the wrong hands, security or privacy may be an issue. This policy of never removing data is not new: an earlier Plan 9 file system, the Cached WORM file system[10], already took permanent snapshots of an active file system.

The write-once nature of Venti simplifies Venti's design. First, once data is written, it is sealed by Venti. This means Venti will not write into that part of its storage ever again. This prevents software bugs to accidentally destroy data. Second, disk storage is used append-only, meaning there is no fragmentation as in normal file systems. This allows for efficient disk usage. The write-once, append-only scheme also gives opportunities for compression, to use even less disk space.

Another important consequence of content-addressing is that the integrity of data can be checked. After having retrieved the data with a given score, a client can calculate the score of the retrieved data. If it does not match, the data is bad and Venti might have a bug or broken disks. If the scores do match, the client can be sure the data retrieved is the same as the data originally stored. Note that normal magnetic disks as used by file systems have no such feature. The data retrieved is assumed to be the same as that stored. Such assumptions are not without risk, as became clear during this project: one disk used for testing would return random data for a single 512 byte sector, without reporting read or write errors. Venti caught this.

As mentioned, access control is not an issue in Venti. An attacker can only 'steal' data if he has the address of it: the address is also the key to the data. This makes Venti protection a capability-based protection mechanism as used in

Amoeba[11]. The address can only be calculated from the data. This scheme is only suspect when the ‘secret’ score is stolen. Thus, users of Venti should keep their scores as secure as they normally would their data. The only problem Venti currently has, is that it allows anyone who can connect to write data. A malicious user can fill up the disk storage of a Venti server by writing random data. Of course, this is true for other data stores as well, such as magnetic disks.

The next section describes the venti protocol, which should help to form an idea of how Venti provides its functionality.

1.1 Protocol

Venti exports it functions over TCP, using a protocol that is fully documented[2]. It is a rather trivial protocol specifying the packing and unpacking of the half dozen remote procedure calls. Each request is packed into a single message, and each elicits a single response, always in a single protocol message. Requests (and thus messages) are always initiated by the client. Each message is accompanied by a tag, a one-byte number used to match responses to requests (responses can be sent out of order). The following request (`T`) and response (`R`) messages are documented:

Thello, Rhello Handshake of connection. The client sends `Thello` along with a *version*, *uid*, *crypto strength* and *crypto* (last two are not used). The server responds with `Rhello`. This is always the first exchange of messages on a new connection.

Tping, Rping A client may send a `Tping` to which the server responds with an `Rping`. This message does not have parameters and is not used often.

Tread, Rread To read a given score, a `Tread` is sent along with the *score*, *type* and *count* (maximum size). The server responds with the *data* or an `Rerror` when the score/type-tuple is not present.

Twrite, Rwrite To store *data*, a `Twrite` is sent along with a *type* and the *data*. The server responds with the *score* of the data.

Tsync, Rsync A `Tsync` message tells the server to flush all cached blocks to disk. When the client receives the `Rsync`, all data previously written can assumed to be flushed to disk and survive forever.

Rerror When an error occurs (e.g., a `Tread` of a score that is not present), the server responds with `Rerror`, which contains an *error string*.

Tgoodbye To close the connection, the client sends a `Tgoodbye` and closes the connection. The server closes the connection when it reads a `Tgoodbye`.

1.2 Hash trees

As mentioned before, Venti does not interpret the data it stores. The semantics of the data is entirely determined by the programs that use Venti as a data store. Applications using Venti do have similar methods of storing common occurrences of data, such as files, directories and entire file hierarchies. Programs such as

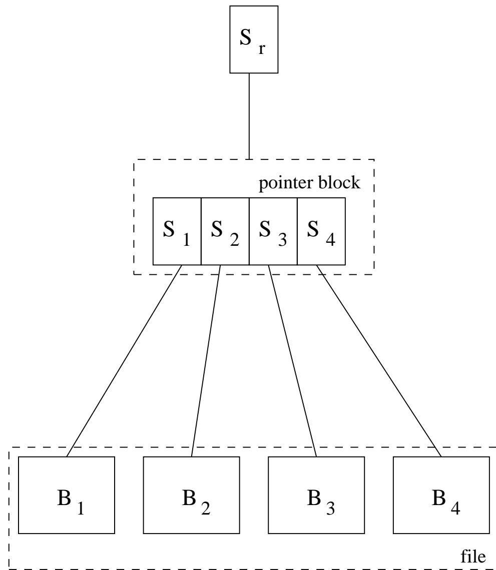

Figure 1.1: Hash tree, B_n represents block n, S_n represents score n belonging to block n, S_r is the root score.

Vac and Fossil make use of hash trees to store these types of data. A simplified view of a hash tree is given in Figure 1.1.

First, let's consider the case when a client wants to store a large file in Venti. Blocks can be up to 56KB in size. Therefore, the file must be split into smaller blocks. Clients typically use 8KB blocks for this purpose (larger blocks would decrease the likelyhood write-coalescing applies). A 10MB file will be split into 1280 blocks, all written to Venti and each resulting in a score. Each score is 160 bits large, or 20 bytes. The client then groups (concatenates) scores to make up 8KB blocks, called *pointer blocks*. Such a pointer block can contain $8192/20 = 409$ scores (rounded to whole scores). For a 10MB file, four such pointer blocks are needed. Each pointer block is written to Venti, resulting in four scores at the first level (depth) of the hash tree. The same process is repeated for the four scores: They are concatenated in a new pointer block at the second level, which is stored in Venti. The resulting score represents the entire file.

Now consider the following three cases of file modification. First, data in the middle of the file is changed (the length of the file remains the same). Each modified block will have a new score. All pointer blocks that contained the old scores need to be updated, resulting in the pointer blocks having new scores as well, up to the root score. A second typical use case of files, is data being appended to a file. This may result in a modified last data block (at the lowest level of the hash tree), and/or new blocks in the hash tree. This is shown in Figure 1.2. Third, consider the case where a part of the file has been shifted. Either by inserting or removing some data. This changes the block in which the modification occurs, but also all following blocks, making write-coalescing not applicable.

In a similar way, top scores of files can be combined (concatenated and

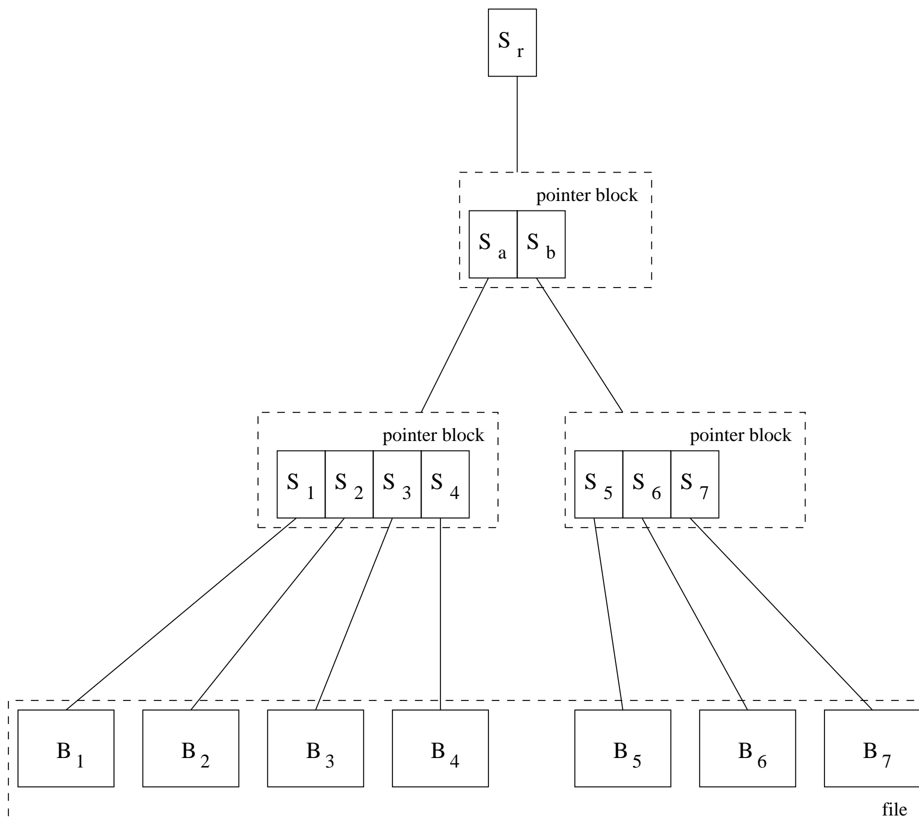

Figure 1.2: Hash tree, data appended relative to Figure 1.1, an additional level of pointer blocks is introduced.

stored as a hash tree) into directory blocks (including file system access control information). This allows entire file systems to be stored.

The data blocks in the hash tree are written after *zero truncating* them. This means all trailing zeros (null bytes) are removed, preventing storage of data blocks with different amounts of trailing zeros. Since the blocks stored are of fixed size (typically 8KB), the original block can be reconstructed easily after having been retrieved from Venti, this is called *zero extending*.

Vac is a small tool that writes file trees to Venti. Vacfs is a Plan 9 file server that serves the file trees over 9P [12]. Vac takes a path to store as argument. It stores all directories and files depth-first. Each file is split into fixed-sized blocks (8KB by default) and a hash tree written to Venti. The resulting scores are each wrapped in a data structure called an *entry* which is then stored in Venti. Directories are represented as data consisting of a concatenation of entries. That data is treated identically to a normal file with data: it is stored using a hash tree and itself wrapped in an entry data structure. File meta-data, such as file name, ownership, access rights and modification time, are stored in a separate file, again as a hash tree, where each meta-data structure points to an entry in the directory file.

Fossil is a file server serving files over 9P. A disk partition is used for the active file system and non-permanent snapshots. Snapshots are archived by flushing the blocks that have changed since the previous snapshot to Venti. A Fossil can be initialised by reading a previously written snapshot—referenced by its score—from Venti. Snapshots are typically accessible at */n/snap/yyyy/mmdd/hhmm* at the appropriate year, month, day, hour and minute. Archived trees are accessible at */n/dump/yyyy/mmdds* at appropriate year, month, day and sequence.

Vbackup is a new tool written for Plan 9 from User Space. First, it can write disk images (file systems) to Venti. Second, it can serve back these disk images read-only from Venti over NFSv3.

1.3 Performance requirements

Since Venti is used as backing store for Fossil, a typical file system (typical in the sense that it is used for reading/writing files as done by other file systems; it was designed to work well with Venti's features and its design deviates from that of a typical file system), it needs to perform well. Or rather, as close to more traditional file systems as possible, or faster. This would be in the range of tens of megabytes per second of data transfer.

To explain the performance problem that using Venti as backing store introduces, consider the following, simplified, description of how a traditional file system on a magnetic disk works. A magnetic disk is divided into logical blocks, which are addressed by number. Block 1 resides at the start of the disk, block 1000 somewhat further down the disk. A file system uses these block numbers to get at data. In the first 10 blocks, there may be a table mapping file names to blocks on the disk. If a file needs to be stored, the file system can reserve series of blocks, say 15-20 and 30-40, to store the file in. The file system stores this mapping in some file system-specific format in blocks reserved for use by the file system. For reading the file, the file system only needs to request blocks 15-20 and 30-40 from the disk in order to handle a read request. The blocks do not all have to be sequential. The point it that the file system can look at

a table that is on a few disk blocks, know which blocks it needs, and request them from the disk. The disk will do the hard work of returning the data.

Now consider the case with Venti. Say one would want to store the same file, consisting of 5 + 10 blocks, on a Venti-backed file system. One cannot ask Venti to reserve 15 blocks. One can only ask it to store a certain block of data. The 'address', i.e. the score, is determined inherently by the data to store. So, when storing the equivalent 15 blocks in Venti, 15 scores are returned. A file system would again concatenate these scores into a new single block and store it in Venti. The file is now addressable by a single score. Later, the file has to be read. The table block is requested using the single score. The block contains another 15 scores which Venti retrieves from its disks. So far, it seems the case of storing on a magnetic disk or in a Venti seems similar. The difference however, is that each score is 20 bytes of random data, the score. To read a file, Venti needs to look up where the data of each score resides on the disk. For a traditional file system, this step is not necessary, the table simply states the block numbers that can be passed to the disk. The lookup is quite expensive, an example shows the scope of the problem.

Assume we have a 1TB Venti system. Not all that much storage capacity, higher capacity systems are in use already. Blocks are typically 8KB in size. This leaves us with

$$1\ TB\ /\ 8\ KB\ =\ 134,217,728$$

addressable blocks. A score is 20 bytes in size, and approximately 20 more bytes are necessary for accounting and being able to find the block on the disk. This sums up to

$$134,217,728\ blocks\ *\ 40\ bytes\ per\ block\ =\ 5,368,709,120\ bytes\ \approx\ 5120\ MB$$

of memory. And that for only a moderately sized Venti server that also needs memory for running Venti including a disk cache and running the operating system. Putting the entire index into memory does not scale to large Venti servers. Startup considerations are also important. Startup time of a Venti server could easily exceed minutes when loading the entire index into main memory at startup.

The solution the designers of Venti have chosen, is to use magnetic disks for an index to map scores to a disk location. As per the example calculation of an index in memory, an index could fit on a single disk, though using multiple disks is beneficial to performance. The index they implemented is just a huge hash-table. Each block of 8KB forms a hash bucket. The disk is divided into n buckets of 8KB. In order to look up a score, the score is hashed and the right bucket determined. This is done just by looking at the score, no disk access is needed thus far. Since all buckets are 8KB, knowing the bucket means one can calculate at which offset on the disk the bucket resides. A single disk seek to this location, and reading of 8KB of data is enough to read the bucket. The score can now be compared to the scores in the bucket. Either the score is not present, or the entry in the bucket spells out where on the data disks the data can be found. For each lookup of a score, Venti will have to turn to the index disks and determine whether the score is present and where on the disks the data resides. When sequentially reading a file consisting of multiple blocks (stored in a hash tree, each block referenced by a random score) many random index disk reads

are necessary, followed by reads from the data disks. This turns practically all reads (even ‘sequential’ reads) into random disk reads.

To make Venti performance acceptable, various read-ahead and caching techniques are have been implemented. These are discussed in the next chapter.

1.4 Report

In this chapter an introduction to Venti has been given. Chapter 2 has a more in-depth analysis of the implementation of Venti, including the optimisations in place that ensure high performance. Chapter 3 discusses *Fossil* and *Vac*, programs that use Venti for storage, and how their use of Venti influences Venti performance. Next, in Chapter 4, magnetic disks are analysed, and future developments in storage devices are reviewed for relevance for Venti. In Chapter 5 performance measurements of Venti are presented, along with measurements of the SHA-1 hashing and whack compression algorithms. Chapter 6 explains the approach to designing and implementing a Venti simulator, and the results of that effort. Memventi, an alternative implementation of Venti, is presented in Chapter 7. The original idea and design are discussed, followed by a report about the implementation. More details about memventi are presented: features, problems encountered during the implementation and how they were overcome, performance measurements, suggestions for improvements and conclusions. Finally, in Chapter 8 a conclusion is drawn about the simulator, memventi and Venti in general, and the report concludes with ideas for future work (some of which are already being developed, inspired heavily by this project) in Chapter 9. The appendices contain elaborate results of the performance measurements.

Chapter 2

Venti design

This chapter describes the details of the Venti distributed with Plan 9 from User Space.

Venti was originally designed and implemented for Plan 9 by Sean Quinlan and Sean Dorward, at Bell Labs. The implementation was in C using the Plan 9 C libraries. Later on, it was partly rewritten by Russ Cox with more optimisations and fitted into Plan 9 From User Space [13] (a port of Plan 9 libraries and tools to UNIX). As a result, Venti now can not only be run on Plan 9, but also on Linux, BSD's and other UNIX-like operating systems. As per typical Plan 9 program design, Venti is heavily multithreaded.

2.1 Index sections, Arenas

This section introduces the Venti implementation and basic terminology. The next section contains a listing of definitions used by Venti or necessary for describing Venti.

The designers of Venti have chosen to use magnetic disks as storage for the data blocks as well as for the index mapping scores to the disk locations of the data blocks. Using magnetic disks is feasible because of ever-decreasing costs of those disks, helped by coalescing duplicate of writes, and the disks are necessary in order to provide decent random access performance.

A Venti configuration has at least one disk for storing data blocks. Preferably, the entire disk is reserved for this purpose for performance reasons. Venti divides the disk into fixed-sized *arenas*. There is always only one active arena: the arena where newly written data blocks are stored. Newly written blocks are always appended to the arena and are immutable: they will never be moved, modified or deleted. This design has been chosen to minimise complexity, and therefore minimise the chance of introducing bugs (a sensible design-choice for a permanent backup storage server). It also eliminates fragmentation. When an arena is full, Venti seals it by calculating the SHA-1 score of the entire arena, and making the next unused arena the active arena. The arena size, 500MB by default, is chosen to make it easy to backup to other media, such as a CD-ROM. Note that it is recommended practice to use a RAID configuration for the disks on which the arenas are stored. A nice feature is that when all arenas on a disk have filled up, additional disks can be added and new arenas created on them.

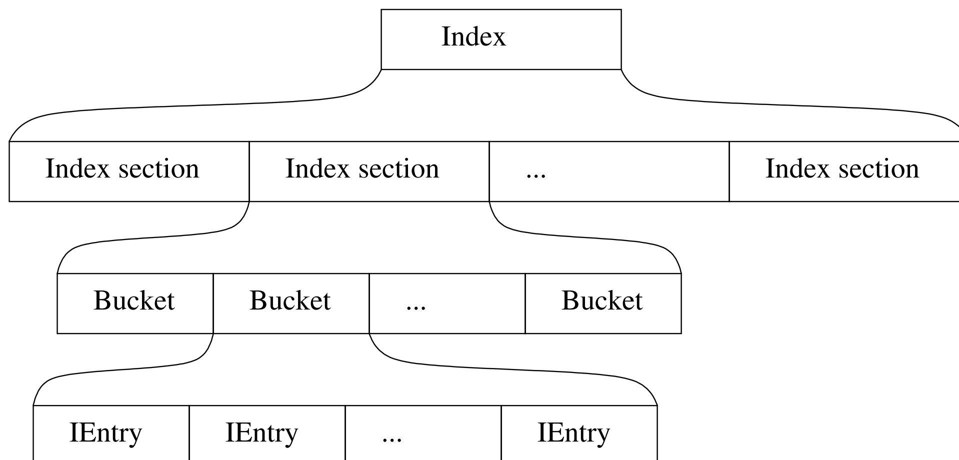

Figure 2.1: Overview of components of an index.

When reading a score from Venti, Venti has to find the right arena and right offset in the arena. Recall that scores are random and the arenas are append-only. This means the data belonging to any given score can be anywhere in any arena. Obviously, it cannot read through the arenas until it finds the data. An index residing on a magnetic disk is implemented as a big hash-table, made up of 8KB hash-buckets. Each entry in the hash-bucket maps a score to an arena and an offset in that arena. This allows Venti to determine in a single disk-seek and single disk-read (i.e. seeking to and reading the bucket) whether the score is present, and if so, where and in which arena it resides. Due to the randomness of the scores, buckets statistically have a low probability to overflow. An index typically consists of multiple disks. This way, lookups (bucket seeks and reads) can be done in parallel.

Since using the index is still expensive, a bloom filter[14] is used to reduce the number of accesses necessary. The bloom filter is read from disk at startup, and remains entirely in memory during operation, and updated when performing writes. Periodically and at shutdown, it is written to disk. The bloom filter is used to determine whether a given score is definitely not present. This is most useful for writing data: the index does not have to be consulted to find out a score is not present in Venti. More optimisations are used to reduce the number of disk accesses, see Section 2.3 for more information.

2.2 Terminology

Venti uses an interesting collection of names for the various components and principles of the system. The following list describes the names used. Figures 2.1 and 2.2 give an overview of the terminology used for the on-disk index and the on-disk arenas. Details such as certain headers in the structures have been left out for clarity.

index The index is essentially a big on-disk hash table. The table is split up in *index buckets*, each responsible for a range of scores. The assignment

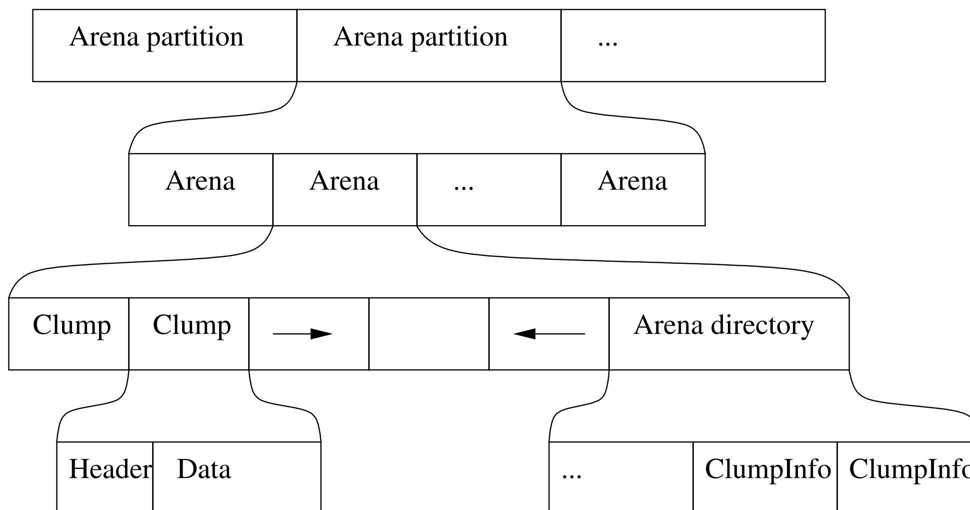

Figure 2.2: Overview of components of the arena partitions.

of score ranges is done during initialisation, by spreading the score range equally over the buckets. An index is made of one or more *index sections*, which are usually entire disks. For improved performance, multiple index sections on multiple disks are recommended to allow concurrent index lookups and thus higher performance.

index section The Venti configuration file specifies on which index section the ranges of buckets can be found. Each range resides in an index section, which is typically a whole disk, but can also be a partition.

arena All data is stored in *arenas*. A Venti installation usually has many arenas. They are all filled up one after the other, so there is always a single arena in use for writing, the others are either filled or empty. All arenas in a Venti installation have the same size, set during initialisation. By default, arenas are 500MB, making them easily copyable to other media, such as to CD-ROMs, for backups. The data block to store is wrapped in a header and written right after the previously written data block. The combination of header and data is called a *clump*. At the end of each arena the *arena directory* is stored. It grows toward the beginning of the arena, towards the clumps. When the clumps and directory reach each other, the arena is full and write activity continues in the next arena. The filled arena is sealed: a checksum is calculated over the entire arena.

arena partition An *arena partition* contains one or more arenas. It is usually an entire disk, not a disk partition in the more common meaning, though it can be. Additional arena partitions (with its arenas) can be added to a Venti to increase its storage capacity. Clumps are addressed by a global address that spans all arenas, even from multiple arena partitions. When a clump is read, the arena is determined from the address and an arena address is calculated.

arena directory Resides near the end of an arena and grows towards the start

of the arena. It contains a *clumpinfo* structure for each clump, which can be used to rebuild the index, or to read the equivalent of all *index entries* in an arena without having to read all data in the arena as well.

lump Represents a *(score, type, size, data)*-tuple in memory. Lumps are kept in the lump cache to satisfy reads without having to go to disk. They also remain in the cache until written when the *queued writes* optimisation is enabled.

clump A lump on disk, in an arena. A header precedes the data. The data may be compressed before being stored on disk, reducing disk space consumption. The validity of a clump on disk is verified using a magic value that is unique for each arena.

clumpinfo block, ciblock On-disk structure, containing part of the information of a clump header. Contains *(score, type, size, size on disk)*-tuple. It is present in each clump header, and stored in the arena directory.

index bucket, ibucket A hash bucket for the index hash table. A bucket has a fixed size and can thus store only a fixed number of *index entries*. The size of the bucket is chosen during initialisation and such that it is unlikely to overflow, considering that scores are evenly distributed. When a bucket does overflow, the index needs to be reconfigured and rebuild.

index entry, ientry An entry in the index table. Contains the score and *index address*—specifying the location of the clump in the arenas—along with other, non-essential information.

index address, iaddr Contains the address of a clump, uncompressed size and data type. The address is a global address, the address space spans all arenas.

bloom filter A bloom filter is used to test membership of a set. This bloom filter sits in front of the index: it tests whether scores are present in the index. Bloom filters have false positives, but do not have false negatives. For reading and writing scores that are not yet present, the bloom filter eliminates the need for an index access.

disk block, dblock Disk blocks are all 8KB in size by default. Practically all reads and writes in Venti go through the disk block cache.

lump cache, lcache Caches lumps. At startup, a fixed amount of memory is reserved. Lookups are done by score and type, the lumps are kept in a hash table. For reclaiming blocks, a heap is used. Lumps are reused with an LRU-strategy. When the *queued writes* optimisation is enabled, Venti does not immediately write lumps to disk, but puts the lump in a queue and returns success immediately.

index entry cache, icache Caches index entries. At startup, a fixed amount of memory is reserved for the index entries. A lookup is done by score and type, the index entries are kept in a hash table. Index entries can be dirty, they are flushed periodically or when a threshold is reached. For reclaiming index entries, the first non-dirty entry in the next (since previous reclaim) bucket is taken.

disk block cache, dcache Caches disk blocks. Keeps read-only as well as writable/dirty disk blocks. A fixed amount of memory is used, specified at startup. The blocks are kept in a heap, lookups are done by disk address. Dirty blocks are flushed periodically in batches (grouped by which type of data they contain to guarantee consistency) or when a threshold of dirty blocks has been reached. The least recently used blocks are reclaimed first.

2.3 Optimisations

This section explains the optimisations currently in use in Venti.

2.3.1 Disk block cache, dcache

When anything is read from a disk by Venti, it is done with 8KB at a time. The 8KB is actually configurable during formatting of the arenas and index sections, but it is the default and recommended size. Arena partition writes also go through the disk block cache. At startup of Venti, a fixed amount of memory is allocated for the disk block cache, which remains in use as long as Venti is running.

An 8KB data block is called a *dblock*. Dblocks are stored in the cache in both a hash table and a heap. The hash table is used to locate the block based on address in a partition (and checked whether it is on the right partition). The heap is used to implement an LRU eviction scheme. Unused blocks are kept on a free list.

Blocks can be marked dirty; they will be flushed to disk periodically, or when a threshold (2/3 of all blocks dirty) has been passed.

Due to locality of reference, the disk block cache can be useful. Consider writing in the arena directory. The clumpinfo structures are only 25 bytes large. Writes to the arena directory are always sequential (writes start at the end of the block and grow towards start of the block). As long as the disk block remains in memory, writing does not result in a disk access. Multiple blocks can remain in the cache and written to disk sequentially. In the same way, writes of data to the arena can be cached: consecutive writes may partially end up in the same blocks. Reads with high locality may also benefit from the disk cache, but the lump cache—which caches scores and their data, in more detail described later on—will often nullify the use of the dcache for this purpose. For the index sections, only reads go through the disk cache. Writes of new entries are handled specially by doing sequential reads and writes, described later.

Dirty blocks have a tag associated with them. It indicates whether a dirty block is an arena data block (containing clumps), an arena directory block (containing clumpinfo blocks), or an arena meta-data block (trailer). When dirty blocks are flushed, the blocks to flush are sorted. First by tag (arena blocks first, arena trailer blocks last), then by partition the blocks are to be written to, then by address. This ensures that writes occur in parallel as much as possible, and in an order that ensures consistency when the write is interrupted, e.g. due to a power failure.

2.3.2 Index entry cache, icache

The *index entry cache* caches index entries, which map score and type to an arena address. These may either have been read from disk for a lookup, or may be entries of new data that have just been written.

Entries are flushed to disk periodically (every two hours) or when the threshold of half of all index entries dirty is reached.

The index entries are kept in a hash table, lookups are done by score and type. When a new entry needs to be cached and all are in use (the free list is empty), the next (since previous eviction) bucket is searched and the first non-dirty element is taken.

Each index section has a write process that handles the periodical flushing of dirty index entries to disk. When the signal to flush comes, it gathers all dirty index entries for its index section from the cache and sorts them. Then it reads from the index section (not going through the disk block cache), 8MB at a time, inserts the index entries and writes back (again not going through the disk cache). If an index section disk block happens to be in cache (it will always be read-only), the block in the cache is updated. After having flushed the entries to disk, the arena is marked as having the clumps up to that point flushed to the index. This is used the next time Venti is started (e.g., after a crash), to allow Venti to add only the clumps to Venti after that point.

An additional optimisation is executed when four consecutive index entries have been read from disk of which the data is stored in a single arena. In this case, the entire directory arena is read and all clumpinfo contained therein are inserted into the cache.

2.3.3 Lump cache, lcache

A lump is a *(score, type, data, meta-data)*-tuple. The on-disk version of this tuple is called a clump. The meta-data consists of an *encoding* (whether and how the data is compressed), *creator* (the first client who wrote this data) and the *time* of the first write of the data.

Lumps are used to store data that have been read from disk, or are to be written to disk. Typically, the blocks to be written to disk are written immediately after allocation of the lump, but when write queueing is enabled, they are placed in a write queue which is flushed regularly. The blocks are 'dirty' while they have not yet been flushed, though the lumps do not know this themselves explicitly, they are just kept referenced while in the write queue. Writing a block means putting it in the lump cache, it is treated the same as a read block.

The data in a lump is always kept uncompressed. Thus, for each hit, data need not be uncompressed again. This is the same for lumps that will be written to disk.

Lumps are kept in a hash table and are addressed by their score. Free blocks are kept on a separate chain. The lumps are also kept in a heap to allow reclaiming blocks using an LRU eviction strategy.

At initialisation, the size of the cache is set. Lump structures are allocated at this point, but not the memory for the data blocks (since it is unknown how large the data blocks will be). Memory for the lumps is allocated either when reading from disk, or when reading a write request from the network. The

current memory use is accounted and the cache makes sure it does not allow more memory being used.

2.3.4 Bloom filter

Bloom filters [14] are data structures that can probabilistically determine membership of a set using little space. Venti uses an in-memory bloom filter to avoid having to go to index section disks to determine whether a score is present. This is most important for writing new blocks to Venti: new blocks will not be found in the index, it would be nice not to have to read the index, which takes a full disk seek. A bloom filter can give false positives in the membership test, but not false negatives. Therefore, when the bloom filter determines the score is absent, it really is absent. When the bloom filter determines the score is present, it may be absent after all (this will be realised after having read the index).

A bloom filter is sized according to address space and number of elements in the set. The more elements in a bloom filter, the higher the probability of false positives will be. A balance has to be found, an explanation for doing this can be found in [3].

2.3.5 Queued writes

Venti protocol writes are normally written to disk block cache immediately. This may slow down burst of writes, since index lookups may have to be performed. To remedy this, writes may be kept in a write queue temporarily.

This optimisation is implemented by having one write queue for each index section. Each write queue has one process consuming from it. When a protocol write comes in, the lump is wrapped in a structure called *wlump* and placed in the appropriate queue. The process then picks it up. If the queue is full, the operation blocks. When the queue is non-full again, the operation is fulfilled.

This optimisation is off by default. It is mostly effective when handling duplicate writes. The bloom filter allows to quickly determine that a block should be written. If so, the new blocks will end up in the disk block cache. Buffering by queued writes helps only very little in this case. When a score is already present, it has to be verified, perhaps by reading from the index disk. In this case, queued writes ensure that both duplicate and pristine writes (possibly mixed) both quickly return success. Only when many duplicate writes come in, the writing is actually slowed down due to slow lookup to the index sections. This also explains why each index section has its own write queue and process: The processes perform lookups concurrently and so fully use the disks random read bandwidth.

2.3.6 Lump compression

To save disk space, lumps on disk (clumps) can have their data payload compressed using a compression algorithm called *whack*. For a more information about whack and its performance, see Section 5.3. This only optimizes for reduced disk space, not for fast servicing: compression may even be a bottleneck in performance.

2.3.7 Disk block read-ahead

At the moment, the code path to disk block read-ahead is never taken so this feature is essentially disabled. It is described here for completeness.

Disk blocks can be read before being requested. When a disk block is requested by some subsystem of Venti, the call indicates whether disk blocks should be read-ahead. Read-ahead is only for reads (writes may also need a read from disk, e.g. when only half a block is being written). It is implemented by keeping a read-ahead buffer, currently hard-coded to 128 disk blocks. Every time a request for a disk read is done (that is not in the cache), the read-ahead buffer is checked to see if it can fulfil the request. If so, no disk access is needed. Otherwise, enough of the disk is read (starting at the requested offset) to fill the read-ahead buffer.

2.3.8 Lump read-ahead

Lumps may be put in the lump cache before being requested. Venti does this by looking at the use of index entries. When an index entry is inserted in the cache (to satisfy a network read of a score not present in the icache), the subsequent 20 (hard-coded) lumps are read-ahead, their index entries inserted in the cache without doing more read-ahead for those lumps.

2.3.9 Prefetch index entries

When Venti looks up a score from the index sections, it keeps a short history of the arenas in which the scores are stored. The history is currently hard-coded to the last four scores. If all last four scores are from the same arena, the arena directory of that arena is read and all entries inserted in the index entry cache (unless this arena directory was the one previously inserted into the cache). This is useful for both reading and writing existing data sequentially: The lookups do not need expensive index disk seeks, but are all satisfied by a quick bulk read from the arena section.

2.3.10 Sequential writes of index entries

Writes of index entries are handled specially. Since the scores to store are random, random buckets (thus random disk blocks) need to be read, modified and written. However, over time, many dirty index entries can be in the index cache. Therefore, when writing the index entries, Venti sorts them and reads index sections in bulk (8MB chunks at the moment), modifies the necessary buckets in a chunk, and writes them all back. This is efficient when many buckets within an 8MB chunk need to be modified.

2.3.11 Sequential writes of disk blocks

Disk blocks that are modified are marked as dirty. When disk blocks are flushed, they are first sorted by partition and offset on that partition. Since blocks are always written sequentially, this ensures that blocks are written in a single pass over the disk.

2.3.12 Opportunistic hash collision checking

SHA-1 is a secure hash—for now. As computers get more powerful, and cryptanalysis makes more progress, it is possible SHA-1 may become too weak to be used in an application such as Venti. There are already attacks[15] on SHA-1 that have weakened it somewhat. However, Venti does not normally check for hash collisions. If it did, it would slow down performance considerably: for every write of already existing data, not only the (relatively fast) index section has to be consulted, but also the slower arena. Instead, Venti only checks for hash collisions when the data happens to be present in the lump cache.

2.3.13 Scheduler

Venti does some simple scheduling to avoid using disks and CPU for non-urgent tasks while urgent tasks are being executed as well. When the disk block cache accesses the disks, this last access is noted in the scheduler, index cache flushes are logged in the same way. When an index flush is in progress, possible arena checksums are put on hold, this is a non-urgent but disk and CPU intensive task. Also, when the disk block cache is active, either due to reading or writing blocks, flushing index cache entries may be put on hold (when there is no urgent need to flush) or slowed down a bit to make it just fast enough to keep up with the expected number of scores that will be written.

2.3.14 Zero-copy packets

Venti reduces buffer copying by placing data in structs called *packets*. Packets consist of one or more buffers called *fragments*, which contain a piece of memory (multiple fragments may reference the same memory, i.e. when a packet has been duplicated). The library interface is documented[16]. Data buffers are still being copied though. The most important buffer copies are the large copies, i.e. for write requests and read responses.

A read response is handled as follows. First, the data must be retrieved. The data may be in the lump cache (stored as a packet). If so, the packet is duplicated (no memory copies), the next step is creating the protocol message. Otherwise, the lump has to be read from disk. If the block is not in the cache, it has to be read into it. Once in the cache, the necessary block(s) will be copied into a buffer from which the lump data structure is unpacked. Since data may be compressed, it has to be decompressed in another buffer. The same destination buffer is used when no decompression is necessary. Finally, a packet is allocated and the content of buffer is copied into it. Next, the venti protocol message is created. The packet containing the protocol data and the data packet are concatenated, not using any data copies. Finally, all fragments are written on the file descriptor.

The write request is read from the network, directly into a packet. The packet starts with the protocol header, followed by the data to be stored. The message needs to be unpacked. This is done by consuming the protocol header and making a new packet with data, no copies are performed. The data is now isolated in a packet and can be written. The score is calculated from the packet. If the score is in the lump cache the data is compared. If it is different, there may be a hash collision. If the score is not present, the packet is copied into

a buffer. Compression is attempted, the compressed data is written to a new buffer. If the compressed data is not smaller than the decompressed version, the uncompressed data is copied into the new buffer (which also contains space for the on-disk clump header). Finally, disk blocks are retrieved, the packet data copied into them and the disk blocks are dirtied, to be written soon.

The packet library prevents copying when reading the venti protocol messages; but inside Venti more buffer copies occur than strictly necessary.

Chapter 3

Venti clients

Venti is used as a data store by various programs. Fossil, the Plan 9 file system, is the most popular. Another program, Vac, can be used for backups. These programs use Venti in a similar but distinctive way. This results in Venti exhibiting different performance characteristics. To properly optimise Venti, a more detailed view of how the clients interact with Venti is needed. More clients, such as vbackup, are available. The next sections describe Vac and Fossil, and how they make use of Venti. Vbackup is not analysed in detail, but a short description of what it does is appropriate: *Vbackup* writes an entire partition (e.g. *fat32*, *ffs*, *ext2*) to Venti. It treats the partition as a single file. Zero truncation/extending reduces storage consumption, since unused blocks will contain all zeros. The accompanying programs help read back the data written in the disk. *Vcat* simply writes the partition to standard out (which can be redirected to a disk partition). *Vftp* is a client program that provides an ftp-like interactive interface, mostly for debugging. More interesting are *vnfs*, which exports the partition over NFS, and *vmount*, which mounts the file system exported by vnfs. Note that the file systems served by vnfs are read-only.

3.1 Vac

Vac writes a file tree to Venti. It starts at the top of the tree (the root) and adds all files and directories, depth first. It has to do it this way, since the score of the 'deepest' file has to propagate all the way up to the score of the entire archive. The data of a regular file is written as a hash tree described in Section 1.2, with 8KB blocks. The resulting score is wrapped in a 40-byte structure containing the score, size, data type, etc., called an *entry*. Thus, an entry is always the starting point of a hash tree. This chapter will refer to the data blocks, pointer blocks and the entry representing the hash tree, as a *vac file* from now on. A directory is also stored as a vac file; the data in this case is a concatenation of entry structures, each representing a file. Note that the meta-data (file name, permissions) are not present in the entry. Instead, the meta-data of files in a directory (both regular files and directories) are stored in a separate vac file containing structures with file name, permissions, etc. as content. The layout of the structures used by vac are described in the Fossil article[5], from which Figure 3.1 (which represents a file tree stored with vac) has been borrowed.

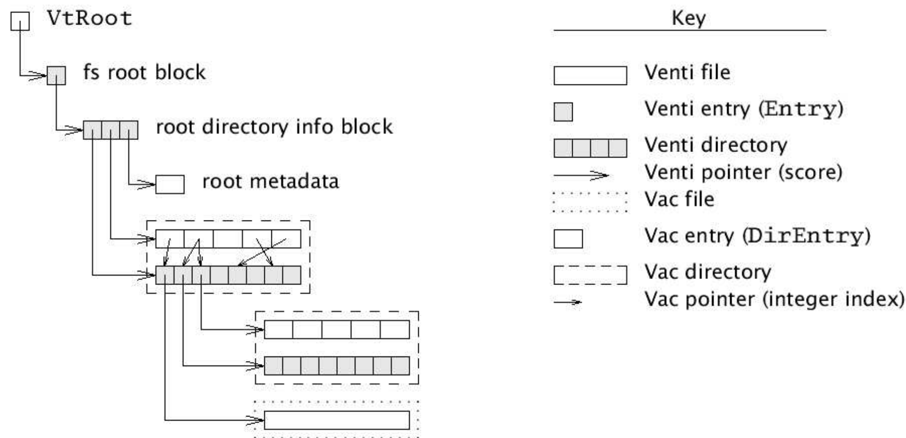

Figure 3.1: Vac hierarchy, image from [5].

For optimisation purposes, the order in which the various types of blocks are written, and how they are referenced, may be relevant. Code inspection reveals that vac writes data to Venti as soon as it is ready. When writing a regular data file, data blocks are written to Venti as soon as the block to write has been read from the source file. As soon as a first-level pointer block of the hash tree is filled up, it is flushed to Venti. When a second-level pointer block is filled, it is written to Venti, etc. After the file has been written, the entry structure is made and added to a vac directory. In short, the pointer blocks are not kept in memory by vac until the entire file has been written, but always flushed as soon as possible. Directories are handled in the same way: as soon as a block with entries or meta-data has been filled, it is written to Venti. Note that this implies that vac writes file trees depth first.

Consider a directory with only regular files, no directories. After each file (its data and hash tree) has been written, its entry structure is added to the vac file containing the file entries; its meta-data is added to the vac file containing the meta-data for the directory. When a block of file entries is filled, it is flushed to Venti. The vac file containing the meta-data is treated identically: flushed to Venti as soon as a block of meta-data has been filled. This means that the blocks making up a vac directory are not always stored close to each other in Venti. The same happens with the meta-data. All source file data and hash tree blocks are written before writing the next vac directory and vac meta-directory blocks. Next consider a directory that contains other directories as well as regular files. Due to vac's depth-first writing, directory and meta entry blocks can be even further apart in Venti. And now even data and hash tree blocks of files in a directory can be stored far apart as well, i.e. when interleaved with a subdirectory that has a wide hash tree (wide in the sense that requires many Venti blocks to be stored). The result is that listing the contents of a directory, or reading many files in a directory does not necessary result in only sequential or nearby reads in the Venti arenas.

On the other hand, reading a file sequentially does result in a lot of sequential accesses (not taking the effects of duplicate writes into account). First, the

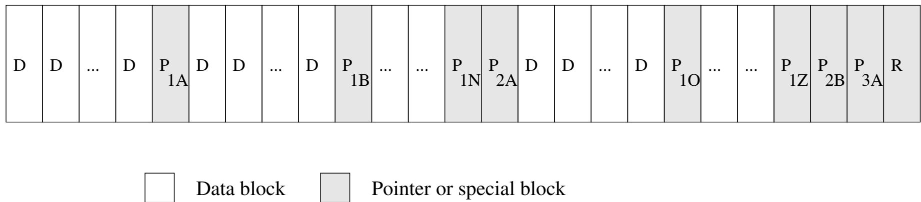

Figure 3.2: Order of blocks in Venti that are written by Vac.

top pointer block is read, which has been stored right after the last data block in Venti. Then, additional lower-level pointer blocks may be read, each written to Venti before its parent pointer block. This leads to the first data block. So far no sequential (but still often nearby) accesses have been made. However, all data blocks referenced in the most recently read pointer block can be read 'sequentially' from disk, because they have been written sequentially. The second lowest-level pointer block to be read again resides in the arena after the data referenced in that block. When it has been read, the data blocks in them can then read consecutively. The blocks and the order they have been written in is represented in Figure 3.2. Data blocks are represented by a D, the first and thus lowest-level pointer block written is P_{1A}, followed by data blocks and pointer blocks at the same level (P_{1B}, P_{1C}, etc.). After the first block of scores has been filled, the first second-level pointer block is written: P_{2A}. Lastly, the top-level pointer block is written, P_{3A} followed by a root block, R. When reading sequentially, the pointer or special blocks (grey in the image) are always read in the opposite direction of how they were written, the data blocks in the same direction. This allows for optimisations such as read-ahead to be effective.

Vacfs works on a similar basis as Vac. Blocks are only requested when needed. Pointer blocks for a file are not read-ahead, neither are file entries. Blocks are cached however, currently 1000 blocks by default.

Examining protocol transcripts of a Venti session on which Vac had been writing data and Vacfs had been reading data have helped to verify that Vac and Vacfs indeed function as explained.

This investigation leads to the conclusion that Venti typically benefits most – if not only – from read-ahead in Venti arena's for data blocks; pointer blocks and meta-data blocks related to a file are often not close to each other. If pointer blocks were to be stored separately from the data blocks, read-ahead of the data blocks would perform slightly better. For the pointer and meta-blocks a 'reverse read-ahead' could be beneficial to performance. Vac and Vacfs do not try to read-ahead data by themselves. They always wait for a request to Venti to finish before posting another read or write. Data blocks to write can easily be queued, and even writes of pointer blocks do not have to finish in order to continue writing. For reading, pointer blocks could be read-ahead and cached, the same for data blocks, but only for sequentially read or often-read files. Finally, it should be noted that caching of blocks inside Venti is made less effective because caching takes place in Vacfs.

3.2 Fossil

Fossil is a file system that can use Venti as a backing store for (permanent) archival snapshots. Fossil can function without Venti, but would only have non-permanent snapshots to distinguish itself from other file systems. Fossil keeps its active file system on a disk, which is split into blocks. These blocks are used for storing data, meta-data and pointer blocks; any block can store any data type. When a snapshot is made, these blocks are implicitly marked copy-on-write, making a snapshot a very fast operation. Only when modifying a block, there is a cost for having a snapshot. A snapshot can be — but does not have to be — archived to Venti, after which the blocks are marked as being in Venti. Blocks on disk are addressed by 4 bytes, and archived blocks by 20 bytes (their score). Internally, Fossil uses 20 bytes for all block addresses, even those of 4 bytes wide. The first 16 bytes are set to zero to denote it is a local block, not a Venti block.

For Venti it is important how Fossil writes blocks to Venti and how it reads them. The Fossil code and the article explaining how it functions have been studied and an analysis made. Fossils use of Venti resembles that of Vac in details, and even shares some code with Vac. The code uses at least partially the same data structures.

First a discussion of Fossil writes. As just mentioned, Fossil makes snapshots by marking blocks in the active file system copy-on-write. When a file system block is modified, e.g. written to, a copy of the block is made. For a Fossil file or directory, the pointer blocks and meta-data blocks can be copied as well. Not all snapshots are written to Venti, but the ones that are, consist of two types of blocks: blocks that have already been stored in Venti (this implies they have not been modified since the previous archival snapshot was taken), and blocks that are not yet stored in Venti (these are blocks for new or modified files or directories). Fossil concisely accounts which blocks have been changed, and it is thus able to write only the minimum amount of blocks for an archive. The only duplicate blocks written (blocks with scores that are already present in Venti), are blocks that have been 'modified' but with identical data, or the blocks of the incidental duplicate file on the file system. Thus, Fossil is sparing with block writes.

Then, the order in which blocks are written is important. This is exactly the same as with Vac. The file hierarchy is written in exactly the same format as Vac, and the directory, file and hash trees are traversed in the same manner. Thus, Figure 3.2 and the explanation from the previous section applies to Fossil as well. There is one awkward aspect of how Fossil writes data to Venti: after each write, a sync-request is sent. This is done to ensure Venti will not lose the data in case of unforeseen problems such as power outages. Loosing data could make the Fossil file system become inconsistent.

Fossil performs reads in the same way Vac does. Blocks are requested from Venti only when it directly needs it to fulfil a file system operation. Blocks are never read-ahead. There are some differences between Vac and Fossil however. First, Fossil stores blocks on disk. Blocks that have been flushed to Venti can still be part of the active file system and thus remain on disk. So, only data that is not in the active file system is read from Venti. This may include unmodified blocks from a partially modified file. Second, Fossil has an in-memory cache of blocks. When a block has been read from Venti, it remains in Fossils memory

until the memory it occupies is needed for another block. The cache uses an LRU eviction policy. A block is only read from Venti again after many other blocks have been read. This has implications for the caches in Venti.

3.3 Conclusion

To conclude, Vac and Fossil use Venti in a very similar way. The cache in Fossil reduces the usefulness of the lump cache in Venti. Only few duplicate writes will reach Venti. Blocks are written in the most memory efficient way to write a hash tree: flush as soon as a block is filled. Fossil always waits for an operation to finish before starting another one. Queueing writes on the Fossil side would better utilize Venti and could thus improve Fossil performance, especially when latency to Venti is noticeable. This would complicate the design a bit, mostly with repsect to guaranteeing file system integrity, but the first chunk of performance gain is relatively easy to achieve. It could be implemented as a separate program sitting between Fossil and Venti, but with a few caveats with respect to Venti protocol semantics. Integrating it into Fossil is also an option.

Fossil also does not currently do any form of read-ahead. Performance gains of such a scheme would be uncertain, but could be implemented as a layer between Fossil and Venti as well. More about these ideas can be read in Chapter 9.

Chapter 4

Disks & performance

The service Venti delivers is similar to that of a hard drive: storing and retrieving data. The differences are in addressing and the fact that Venti cannot overwrite data. Similarities are in usage patterns: sequential reads, sequential writes, random reads. Random writes are different: the idea does not exist in Venti. More sophisticated usage patterns such as bursts of reads and writes, sustained writing and nearby-reads can be similar as well. Performance of a normal disk can be compared to performance of a Venti installation for these similar operations.

In this chapter, disk properties and disk performance are analysed. This information is relevant for determining whether optimisation ideas might be viable. It is also useful to get an indication of the overhead Venti incurs.

This chapter continues with an overview of current disk technology, followed by a specification of the disks used in the experiments, the methodology and the results of the performance measurements of the disks. Properties and performance measurements of alternative storage devices (flash memory) are presented. The chapter concludes with an analsis of MEMS-based storage devices and their applicability to Venti.

4.1 Disk internals

First, a brief introduction to disk internals [17, 18, 19]. A magnetic hard disk consists of one or more rotating *platters*. Each platter has two magnetic surfaces on which data is stored. The surface is divided into rings, called *tracks*. A track is divided into *sectors*, typically 512 bytes. Tracks that lie above each other on the surfaces are called a *cylinder*. The platters rotate with a fixed speed. Typical consumer-grade IDE and SATA drives rotate at 7,200 *rotations per minute*, or *rpm*, higher-end SCSI drives at 10,000 rpm and up to 15,000 rpm. Data is read off and written to the platters by the *heads* (one per surface) which can be moved to the desired cylinder (but all heads are moved at the same time). The heads remains at a fixed position above the track, the rotation of the platter causes the right sector to move under the head. These concepts are depicted in Figure 4.1.

Cylinders are grouped into *zones*. Different zones have different bit densities: Zones located at the outer side of a platter have higher bit densities than zones

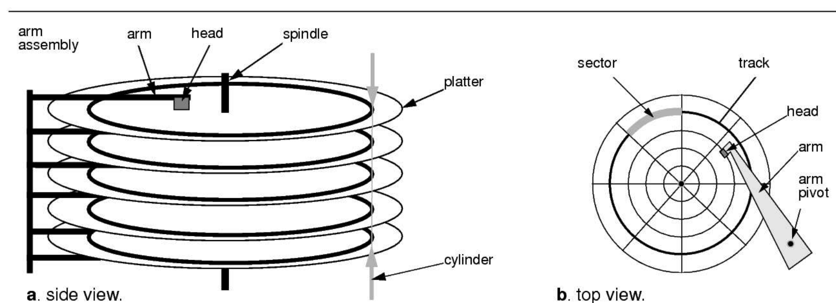

Figure 1: the mechanical components of a disk drive.

Figure 4.1: Disk model, from [17].

on the inner side. More surface moves under the heads at the outer zones than at the inner zones, thus data transfer is higher at the outside than at the inside.

Switching to the next track requires movement of the heads which takes a little time. Because the platters keep moving, the first sector on that next track is not next to the first sector of the current track. It is located there where the head can be *settled* again after the switch. This positioning of tracks is called *track skewing*, it ensures high sustained data transfers.

Since sectors can become damaged after usage, or during the manufacturing process, the drive (tracks) include spare sectors. Damaged sectors can be remapped to these spare sectors. This occurs transparently to the user of the disk. In theory, disk models might be inaccurate due to this mechanism.

Each disk has a seek time property: the time it takes for the head to move to another cylinder. This is an important performance property of the disk. The seek time is usually specified by a minimum, maximum and average, e.g. minimum 4 ms, maximum 10 ms and average 8 ms. Unfortunately, seek times are hard to determine in practice. Seeking to a nearby cylinder costs far less than seeking to the other end of the platter. Also, seeks are more than just arm movements. There is a head acceleration phase, a fixed velocity phase for long seeks, a slowdown of the head and a settle of the heads. Very short seeks have almost no movement of the head, just a settle. All these factors make disks hard to model in detail, only few hard drives have been modelled [20]. Software has been developed to simulate a hard drive and related system components such as buses and adaptors, e.g. disksim [19].

The second important factor of disk performance is rotation speed. Say a head has moved to the right track. It has to wait for the right sector to move underneath the heads. In the worst case, the drive has to make nearly an entire rotation before this occurs. One rotation takes 8.3 ms for a 7,200 rpm drive, or 6 ms for a 10,000 rpm drive. Assuming there is no relation between requests and current rotational location, an average operation takes half the value on average, i.e. 4.2 ms and 3 ms respectively. Since rotation speed is not exactly fixed, rotational location cannot be used for optimisations outside the disk, e.g. in an operating system scheduler. A disks access time is the time it takes to read data from disk. It depends not only on the seek time (move the head to the right cylinder), but also on the rotation speed.

Some disks, mostly SCSI disks and newer SATA disks accept more than one command at a time[21]. The commands are queued in a *command queue* and the disk firmware may reorder these commands. It may take advantage of current rotational location or head location to schedule 'nearby' operations first. For SCSI, this is called *tagged command queueing*, for SATA this has been dubbed *native command queuing* (typically 32 commands can be queued). Disk scheduling has been an area of extensive research for decades[22, 23]. Many scheduling algorithms have been developed with different properties of average I/O latency, maximum latency, starvation and details needed about the disk to schedule operations. Common algorithms are SCAN and LOOK that move from one end of the disk to the other, and back, servicing requests on their way. C-SCAN and C-LOOK only move in one direction, when at the far end of the disk, they 'skip back' to the start of the disk, lowering maximum service time. FCFS (first come first serve) is the simplest algorithm, but has suboptimal performance. SSTF (shortest seek time first) fulfils the request that has smallest seek distance based on current rotational and head location. Several improvements to this have been made to prevent starvation, either by grouping operations and always finishing a group before starting on another, or by taking operation age into account while scheduling. A slight improvement over SSTF is SPTF (shortest positioning time first) and an aged variant, they need more information about the disk which is only available in disk firmware. Fine-grained scheduling in general is left to the disk firmware, since the operating system driving the disk does not know details of the disk such as current rotational location or even current head location: the disk controller does not provide that information to the operating system.

A disk has a cache in volatile memory. New SCSI or SATA disks have 8MB or 16MB of cache. This cache functions as a buffer, to match the bus transfer rate to the disk transfer rate. It also caches data of previous reads and writes, which can useful when the same data is requested from the disk. However, in practice, the operating system disk block cache will usually receive the cache hits. The cache is also useful to store *read-ahead* data. When a block is requested (typically a few kilobytes at a time by the operating system), the firmware reads more data from disk into its cache to be able to quickly fulfil subsequent requests for the 'next data'. This greatly helps sequential reads. Normally, the cache is in write-through mode, writing data to disk immediately. However, it may be configured as write-back, letting the disk decide when it writes data to disk. This may cause data-loss on power failures, and is therefore not a default on disks, nor is it recommended.

4.1.1 Implications for Venti

Note that for Venti index sections, the on-disk caches are useless for recurring reads since reads are entirely random: cache hit rates will be very low. A write-through cache is equally unhelpful: blocks just written are no more likely to be read soon than any other block. Sequential writes, when flushing the index sections, may benefit from a write-back cache: integrity is not ultimately necessary (the index can be rebuilt) and writes are batched in quantities that fit in the cache (thus the disk will quickly return success for the data and allow more data to be flushed). Large read-ahead is not useful during index lookups, reads are small and random. It *is* useful when flushing index entries to disk:

many blocks are read sequentially.

Then for the arena partitions. Writes are mostly sequential, except for the arena trailers (which reside at a fixed location) and the arena directory (written sequential, but in reverse direction). Reads can be random in principle, but many reads are assumed to have a high locality of reference, just like with normal hard disk usage patterns. Write-coalescing makes this less true. Disk read-ahead may be useful to quickly fulfil the read-ahead requests from Venti. The on-disk read cache is not useful: Venti itself has a much larger disk block cache. A write-through cache is not useful for the same reasons. A write-back cache is very dangerous since it may cause the arena trailer or arena directory to become out of sync with the data and is therefore not recommended.

Another optimisation that has been used to improve response time[24] is making sure the head is in a location that has average low latency to a random location. For example, placing the head in the centre of the disk. For our use, this is not a useful optimisation since we are not really interested in individual response times, but in continuous sustained random accesses.

4.2 Operating system disk handling

Operating systems using a disk usually have some optimisations related to disks that should be taken into account when analysing these disks. For example, the Linux kernel (version 2.6.8) has a disk block cache which can use all unused memory and thus can be quite large. This disk block cache is many times larger than the disks cache, thus typically reducing the use of the on-disk read cache. But note that Venti and the test program used circumvents the kernel's disk block cache by opening the disk file with flag O_DIRECT. The Linux kernel can also do read-ahead when it thinks it is useful, it can do this with much larger buffers than the on-disk cache. Not surprisingly, the kernel also has a disk command queue, also larger than the on-disk command queue. However, an on-disk command queue may deliver better performance because it is closer to hardware and has more information available to schedule operations (such as rotational location and head position).

4.3 Testing hard disks

To test the performance of disks, a small utility has been written, called *ptio*, available in Appendix G.1. It reads a list of byte offsets to operate at from a file (generated by either *randoffsets* or *seqoffsets*) and is started with parameters that indicate whether it should read or write, etc. Many test runs have been conducted, each with different values for the system-wide configuration parameters, such as the kernel disk command queue. The parameters and possible values are listed in Table 4.1. In principle, it should be interesting to see the impact of device command queueing, however, initial tests with the SCSI drive showed that turning it on or off did not influence performance, perhaps the tested disk did not support queueing correctly, or did never really disable it. The results are presented and discussed in the next sections.

Ptio is a pthreads program running on Linux. It starts one or more threads

Name	Values
Bus	IDE, SCSI
Write cache	on, off
Number of processes	1, 4, 32
Device read-ahead	on, off
Kernel command queue depth	0, 16, 128
Kernel read-ahead	0, 128
Blocksize	1KB, 8KB, 128KB
Type of offsets	random, sequential
Operation	read, write

Table 4.1: Parameters of disk performance tests.

that each read one offset at a time from the offsets list that has been read from a file. Once the offset is retrieved, the operation is executed by a call to *pread* or *pwrite*. The file to operate on (which should be the entire disk, e.g. /dev/hda on Linux), is opened for reading and writing and with the O_DIRECT option, bypassing the kernel disk block cache and transferring the data from disk directly to the user space buffer, just like Venti does.

First some more information about the measured properties and expected results. *Throughput* will be quite low for random access (both in read and write), much lower than for sequential access. Each operation takes an access time. Assuming the access time is 6ms, only 166 operations per second can be fulfilled, resulting in a throughput of 1.33 MB/s when using 8KB blocks. The *average access time* is determined by dividing the duration of the test by the number of operations. For random access, this value approximates the random disk access time. For sequential access, this value will be much lower but does not say much about access time. The *mean waiting time* is the time it takes for the operating system to return success on the read or write system call. When using many processes in *ptio* and thus many concurrent operations, each thread has to wait for its turn and thus mean waiting time will be higher with more processes. When the kernel is allowed to queue (and reorder) disk operations, completion of the operation will take longer than a disk access. The *standard deviation* of the mean waiting time indicates how variable the wait time is, e.g. due to reordering: some operations may be handled promptly, others delayed. This measures properties of the disk firmware and operating system disk scheduler.

For the random operations, 5120 operations (offsets to operate at) were generated using *randoffsets*, independent of the blocksize. For sequential operations, as many offsets were generated using *seqoffsets* to read or write 1GB of data, dependent on the block size. Only the parameters that might give useful information have been varied. For example, the read tests have only be performed with the write cache off. The write operations have only been performed with device and kernel read-ahead off. These settings have no effect on the performance for these operations.

The following sections contain an analysis of the results. The raw results can be found in Appendix B.

Model	Compaq BD009222BB
Interface	SCSI U2W, 80 MB/sec
Cache size	Unknown
Rpm	10,000
Disk size	9,100 MB
Cache	Write-through
Other	Tagged command queuing, depth 16

Table 4.2: Disk properties, SCSI Compaq BD009222BB

4.4 SCSI disk results

The SCSI disk that has been tested is a *Compaq U2W* of 9100MB, rotating with 10,000 rpm. All known details are in Table 4.2. Unfortunately, specifications for the disk were unfindable, so the cache size remains unknown.

The characteristics are split first and foremost by type of operation: random or sequential. A shortened version of the results are given in Table 4.3 and Table 4.4. The command queue depth is not shown in this shortened version because the full results show that having a large kernel command queue is always beneficial. It is also 128 (the largest value tested) by default, and thus the results for a smaller command queue are not interesting and have either been discarded (when they did have an impact, i.e. with more processes than kernel command queue slots) or used for averages. Also, the results showed that the device command queue, device read-ahead, write cache and kernel read-ahead did not influence performance. Therefore, the results presented are the averages of the runs when disregarding these settings.

A typical use case for random operations is a single process continuously reading from random disk locations. The results show that for reading, the average operation time is 7.80 ms, 8.14 ms and 13.51 ms for blocks of 1KB, 8KB and 128KB respectively. The 7.80 ms is almost the lower bound of the access time since the minimum addressable unit on a disk is 512 bytes. These results show that each operation takes around 7.75ms and almost 0.05 ms per KB (this value approximates the sequential read speed).

The number of processes used is also very important. With 32 processes, the throughput is much higher, e.g. for 8KB blocks, 1.575 MB/s versus 0.960 MB/s. This is because the kernel and disk firmware performs scheduling on the blocks. Thus, nearby blocks are handled first, the blocks further away later. When many offsets to operate at are posted to the system, it has more blocks to choose from and can therefore schedule more efficiently. In practice however, it may be quite rare to need 32 operations to be executed concurrently. There is also the problem with response time, as can be seen in the full results with standard deviation of mean wait times. They go up considerably with many processes, just as the mean time itself. However, the high standard deviation is particularly worrisome: some operations are completed only after a very long waiting time.

Random write performance (as well as sequential) is somewhat slower than read performance. However, the same trends can be seen with respect to behaviour with varying block sizes and processes. It shows that the 8KB block size used by Venti for random access is quite sensible: only little extra time above

Blocksize	Procs	Read Throughput	Average	Write Throughput	Average
1KB	1	0.125	7.80	0.118	8.29
1KB	32	0.179	5.87	0.189	5.17
8KB	1	0.960	8.14	0.901	8.68
8KB	32	1.575	4.96	1.409	5.55
128KB	1	9.250	13.51	8.423	14.84
128KB	32	12.423	10.06	10.693	11.69

Table 4.3: SCSI disk, random operations. Throughput in MB/s, average in ms.

Blocksize	Procs	Read Throughput	Average	Write Throughput	Average
1KB	1	8.931	0.11	0.162	6.03
1KB	4	0.597	1.64	0.943	1.04
1KB	32	3.690	0.27	3.729	0.26
8KB	1	26.101	0.30	1.235	6.33
8KB	4	4.596	1.70	4.491	1.74
8KB	32	21.801	0.36	19.400	0.40
128KB	1	26.184	4.77	10.901	11.47
128KB	4	21.869	5.72	22.631	5.52
128KB	32	21.707	5.76	22.607	5.53

Table 4.4: SCSI disk, sequential operations. Throughput in MB/s, average in ms.

the minimum access time is spent on the actual reading and writing.

The results for sequential operations from Table 4.3 are surprising. The behaviour for reads and writes are very different. So, first the reads.

Reading is fastest with a single process. The maximum read throughput of 26.1 MB/s is reached already with 8KB blocks, apparently the operating system and disk cache are able to schedule these efficiently. This is definitely not the case for 1KB blocks. Also, using many processes is detrimental to performance: a fifth of throughput is lost at 8KB blocks. A symptom that remains unexplained is the extreme drop in throughput when using four processes; perhaps the reads are not performed sequentially anymore as an artifact of the scheduler. These results show that a single process should to the reading in case of sequential operations.

For writes, the more processes writing, the better. This is especially important for the 8KB blocks: four processes manage 4.49 MB/s throughput while 32 processes reach 19.4 MB/s, approaching maximum throughput. The low throughputs for a single process are peculiar. Maximum throughput is reached with 128KB blocks, even with four processes, at 22.6 MB/s. This may be a scheduler artifact as well, or due to operating with O_DIRECT and combined with timing problems.

To conclude, maximum sequential read throughput is around 26.1 MB/s, write throughput is around 22.6 MB/s. Random read throughput for a single process and 8KB blocks is 0.96 MB/s, for 8KB blocks and 32 processes 1.58 MB/s, over 50% more. For random writes, the throughputs are 0.90 MB/s and 1.41 MB/s.

Model	Western Digital WDC WD800JB
Interface	UDMA, 100 MB/second
Cache size	8192 KB
Rpm	7,200
Disk size	80,026 MB
Average Latency	4.20 ms (nominal)
Read Seek Time	8.9 ms
Write Seek Time	10.9 ms (average)
Track-To-Track Seek Time	2.0 ms (average)
Full Stroke Seek	21.0 ms (average)
Cache	Write-through

Table 4.5: Disk properties, IDE Western Digital, from [25].

4.5 IDE disk results

The IDE disk tested is a *Western Digital WDC WD800JB*, a 80GB disk. Table 4.5 lists all known disk properties. The same tests have been conducted as with the SCSI disk.

A shortened version of the full results can be found in Appendix B.2. The most important and interesting properties have been summarised in Tables 4.6 and 4.7. As with the SCSI results, the kernel command queue value is not displayed in the tables. The default queue depth of 128 gives the best results although only marginally (except when using 32 processes where it is significant). The same goes for the devices write cache: it had no influence on the performance and thus has been left out. The values shown in the tables are averages that include results for both the write cache on and off. Similarly, the kernel read-ahead also had no influence, probably due to use of O_DIRECT for opening the device, and has been left out of the results.

The device read-ahead did have a clear influence on performance. It has only been enabled for reading. Random reads slow down because of device read-ahead. This is because the disk keeps reading but the data read beyond 8KB is never requested. Skipping to another operation when it comes in is apparently a bit more time consuming than being idle when the request comes in. The performance degradation ranges from 8% to 12%. Device read-ahead is crucial for sequential reading. 8KB blocks can be read with over 38 MB/s with read-ahead, and only with 0.922 MB/s without (when using a single process for reading). With 128KB blocks 49.0 MB/s is reached, but only with device read-ahead.

The results show that using more processors does not improve random throughput, both read and write. It actually degrades performance. This is somewhat surprising, apparently the kernel does not efficiently schedule operations when they come in concurrently. An 8KB block can be read in 8.56ms when a single process is used. A 1KB blocks is read in 8.42ms and thus almost the lower bound of random disk reads with 512 bytes as smallest addressable block. Random writes are slightly slower than random reads, as expected.

A bigger difference is seen with sequential write throughput, which does not exceed 21.4 MB/s and that only with 128KB blocks. This does not show up in the table because it is only seen with a kernel command queue depth of 16. This

Blocksize	Procs	DRA	Random Throughput	Average	Sequential Throughput
1KB	1	off	0.116	8.42	0.117
1KB	1	on	0.104	9.36	7.926
1KB	32	off	0.107	9.16	1.500
1KB	32	on	0.094	10.43	3.683
8KB	1	off	0.913	8.56	0.922
8KB	1	on	0.821	9.52	38.168
8KB	32	off	0.852	9.18	6.843
8KB	32	on	0.744	10.50	10.480
128KB	1	off	11.297	11.07	11.503
128KB	1	on	10.476	11.93	48.999
128KB	32	off	10.584	11.81	11.967
128KB	32	on	9.575	13.06	43.859

Table 4.6: IDE disk, read operations. Throughput in MB/s, average in ms.

Blocksize	Procs	Random Throughput	Average	Sequential Throughput
1KB	1	0.103	9.45	0.117
1KB	32	0.104	9.36	1.809
8KB	1	0.817	9.57	0.921
8KB	32	0.823	9.50	10.303
128KB	1	9.223	13.55	11.488
128KB	32	9.227	13.55	11.490

Table 4.7: IDE disk, write operations. Throughput in MB/s, average in ms.

is certainly unexpected and suboptimal behaviour. Write throughput should reach far higher values, closer to the maximum read throughput. This is worth investigating further.

The raw data also shows some instable behaviour when reading or writing 8KB blocks with 32 processes. The throughput reaches about 10 MB/s, but individual runs have values deviating by a whole MB/s. I have of no satisfactory explanation for this. It may be an artifact of how *ptio* interacts with the kernel.

To conclude, these tests show a random throughput at 8KB of about 0.91 MB/s for reading and 0.82 MB/s for writing. Sequential reads reach 38.2 MB/s but writes are stuck at 10.3 MB/s (or 21.4 MB/s with 128KB blocks, but that is still slow). Interesting is that issuing many random operations concurrently is not beneficial. This behaviour is very different from that of the SCSI disk, which behaves more as expected.

4.6 Conclusion

The throughputs measured for these disks indicate the upper bound of performance attainable by Venti. For writing, any data written to Venti has to be stored on the disk so Venti is limited by the disks performance. For reading, the same applies. Considering random accesses, any optimisation technique used in Venti could be used on top of a raw disk as well for fair comparison. When

using multiple disks, the performance is limited by the combined performance of the disks.

A few conclusions can be drawn from the tests. For random accesses, it helps to post many operations (reads are of most interest for Venti) concurrently: the operating system scheduler or disk firmware will efficiently schedule the writes and substantially improve throughput. A thing to look out for is that the on-disk scheduler may interfere with the operating system scheduler. Or more likely, the other way around. For sequential accesses, 128KB blocks are ideal, which is expected since it involves fewer seeks than smaller blocks. However, the 8KB blocks used by Venti often result in very decent performance as well. How Venti uses the hard disk on Linux, opening the disk with the O_DIRECT flag, could have bad consequences. It could be the cause of the unexplained anomalies—such as the seemingly bad performance of the IDE disk—and should be investigated further. Also, looking at more control over the scheduler could help Venti. At least the Linux disk scheduler can be fine tuned.

The basic measured properties of the disks can be found in Table A.3 in Appendix A.

4.7 Alternatives to magnetic disks

Venti currently stores its index section on magnetic disks. Since the initial design of Venti, alternative non-volatile storage systems have become available. These may be used for the index instead of magnetic disks. The following sections explore compact-flash memory and (the not yet available) MEMS-based storage.

4.7.1 Compact-flash memory

At the moment of writing, 16GB compact flash modules are commonly available. Compact flash should allow for fast random access and the newer models also have fast sequential throughput. Could they be used for the index in place of magnetic disks? Given the recommended size of the index disk of 5% of the arena disks, a 4TB installation would need just over 200GB of index disks. At the moment, that is much more expensive than the equivalent in magnetic disks. The index section layout now in use was not optimised for small size however. If it would—together with tighter bounds with respect to bucket fullness—the size can probably be reduced by a factor of 2 to 4. This could be just enough to make using compact flash memory for the index sections viable. Also, large solid state disks are becoming more and more common: they could also benefit from a more storage-efficient index section layout.

The compact flash tests were performed on a 1GB compact flash card, a *SanDisk Extreme III 1.0*GB. The only performance characteristic documented is a maximum sequential read/write speed of 20 MB/s. Random access performance is not mentioned in the specifications. The compact flash card has been tested using a no-name compact flash to IDE converter that is able to hold two compact flash cards. Unfortunately, the adaptor does not support DMA, only the IDE PIO modes and thus will not have a high (sequential) throughput. Therefore, it has only been used to test random accesses. A USB 2.0 compact flash reader, a *SanDisk ImageMate*, has been used to perform additional tests, both random and sequential access. The tests are mostly the same as the tests on

Op	Blocksize	Procs	KCQ	Throughput	Per op	Mean	Stddev
read	512	1	1,16,64	1.171	0.417	0.416	0.047
read	512	32	64	1.159	0.421	13.472	199.511
read	8192	1	1,16,64	2.665	2.932	2.930	0.063
read	8192	32	64	2.662	2.935	93.874	500.859
write	512	1	1,16,64	0.019	25.513	25.512	3.8197
write	512	32	64	0.020	24.419	781.171	429.440
write	8192	1	1,16,64	0.311	25.131	25.130	5.334
write	8192	32	64	0.321	24.318	777.470	428.912

Table 4.8: Shortened results of the compact flash card tests with IDE adaptor; throughput in MB/s, latency in ms.

the magnetic disks: many runs of ptio were conducted and the results analysed. The parameters were the *operation* (read or write), *blocksize* (512 or 8K bytes), *number of processes* (1, 4, 32), *kernel command queue length* (1, 16, 64), *type of I/O* (random, sequential) and *kernel queue read-ahead* (0KB or 128KB). For the random access tests, the *kernel queue read-ahead* was set to 0KB. The kernel I/O scheduler was the default CFQ, complete fair queueing, which supposedly guarantees that all processes receive a fair share of I/O operations. The test machine is described in detail in Appendix A. The following sections present the results of the tests followed by an explanation and possible implications for Venti usage. Appendix E has more detailed results of the tests.

Compact flash to IDE adaptor

Shortened results from the test with the no-name IDE adapter can be found in Table 4.8, the full results can be found in Table E.1 in Appendix E. The most important observation to be made is that read latency is very low: a 512 byte block can be read in 0.42 ms. A 8KB block — 16 times larger — takes 2.93 ms (7 times slower than for a 512 byte blocks). This last number is only mildly impressing when comparing to high rpm SCSI disks. The write latency however is very high: around 25 ms for both 512 byte and 8KB blocks. Thus, writing the large blocks takes as much overhead or waiting as writing small blocks.

These latency translate into a read throughput for 512 byte blocks of 1.2 MB/s and 2.7 MB/s for 8KB blocks; for writing they result in 0.020 MB/s and 0.321 MB/s respectively. Writes are slow and render using this adaptor and compact flash card unusable for Venti index disks.

The other factors — number of processes and kernel command queue length — have no surprising consequences for performance. More processes or longer kernel command queues do not improve performance. This was to be expected: flash memory does not benefit from scheduling of its accessing, or at least not in the same way as magnetic hard drives, for which the kernel drivers were written. The results do show that it is detrimental to performance to have more processes than slots in the kernel command queue. It is not clear why this is the case, perhaps due to queue data structure overhead or process scheduling. However, the default kernel command queue length is 128, so this will not pose a problem in any default configuration.

Mean waiting time increases linearly with the number of processes. This was

expected, again, because queueing and scheduling of operations do not improve performance for flash memory. Thus, when four processes continuously issue a read concurrently, on average each of them will have to wait for four operations to finish to have their read fulfilled. Interesting to see, the standard deviation increases a lot with multiple processes. This is likely due to the kernel scheduling operations for fast access as it would for magnetic disks. For flash memory, this scheduling should be disabled entirely.

In short: write performance makes this adaptor unusable for random writes, but the low read latency is very attractive, but mostly for the really small blocks.

Compact flash USB 2.0 adaptor

The results of the *SanDisk ImageMate* adaptor, given in Table 4.9 or the full results in Table E.2, are similar to those of the IDE adaptor for random writes: 512 byte blocks reach a throughput of 0.021 MB/s and 8KB blocks 0.330 MB/s, only slightly higher than the IDE adaptor. Again, but a bit more noticeable, the more processes and the longer the kernel command queue, the higher the performance, but only very marginally: it is only noticeable for 8KB blocks: 0.325 MB/s versus 0.330 MB/s. For random reads, the 512 byte blocks are slower: 0.63 ms latency with 0.78 MB/s throughput. This is probably due to USB overhead and perhaps a slower adaptor. The random reads of 8KB blocks are much faster than for the IDE adaptor: 1.31 ms latency resulting in 5.99 MB/s throughput. This is quite an improvement over random 8KB block access on magnetic disks. For reads, a single process and long kernel command queue are best, but only slightly. As with the IDE-adaptor, it seems crucial that the kernel command queue has more slots than processes.

Random writes are still slow: latency ranges from 23.1 and 23.6 ms for 512 byte blocks and from 23.7 ms to 24.1 ms for 8KB blocks. The same recommendations for processes and kernel command queue apply. Also, the same standard deviation behaviour is observer: the more processes, the higher the standard deviation.

Sequential operations are faster than random operations. For 512 byte blocks, throughput ranges from 0.27 MB/s to 1.78 MB/s: the highest throughput is achieved with 32 processes, then a single process, and the lowest with four processes. This is curious, perhaps the high throughput is achieved due to a form of read coalescing, where multiple small reads are transformed into a single larger read. Or perhaps the kernel has a small cache and a minimum block size that is larger than 512 bytes. The results for 8KB blocks ranges from 3.19 MB/s (for four processes) to 7.77 MB/s (for 32 processes).

Behaviour of sequential writes is much more interesting. For 512 byte blocks, a maximum throughput of 5.84 MB/s can be reached (with 32 processes), but only 0.94 MB/s with a single process, and 1.23 MB/s for four processes. The difference is enormous, it may be due to a form of write coalescing in the kernel, USB adaptor or memory card and is worth investigating further. The same occurs with 8KB blocks, albeit less dramatically: throughput ranges from 6.67 MB/s to 11.11 MB/s. The more processes, the better, and a longer kernel command queue helps only a little bit. Also, for all sequential access, more processes than kernel command queue slots are not bad (in contrast with random access).

The results show that kernel read-ahead has no influence, this seems peculiar and an explanation has not been found. The kernel command queue length has

Type	Op	Blocksize	Procs	KRA	KCQ	Throughput	Per op	Mean	Stddev
rand	read	512	1	0	1,16,64	0.777	0.629	0.627	0.043
rand	read	512	32	0	64	0.771	0.634	20.256	248.895
rand	read	8k	1	0	1,16,64	5.987	1.305	1.303	0.067
rand	read	8k	32	0	64	5.978	1.307	41.789	354.345
rand	write	512	1	0	1,16,64	0.021	23.580	23.580	3.566
rand	write	512	32	0	64	0.021	23.140	740.031	394.015
rand	write	8k	1	0	1,16,64	0.325	24.053	24.051	5.206
rand	write	8k	32	0	64	0.330	23.700	757.940	402.586
seq	read	512	1	0	1,16,64	0.977	0.500	0.498	0.004
seq	read	512	1	128	1,16,64	0.977	0.500	0.498	0.004
seq	read	512	32	128	64	1.780	0.280	8.933	1.982
seq	read	8k	1	128	1,16,64	6.257	1.249	1.247	0.016
seq	read	8k	32	0	1	7.744	1.009	32.272	11.885
seq	read	8k	32	128	64	7.758	1.007	32.217	11.641
seq	write	512	1	128	1,16,64	0.935	0.522	0.520	0.265
seq	write	512	32	128	1,16,64	5.842	0.084	2.662	0.429
seq	write	8k	1	128	1,16,64	6.669	1.171	1.170	0.397
seq	write	8k	32	0	1	10.932	0.715	22.814	5.244
seq	write	8k	32	128	64	11.108	0.703	22.461	0.869

Table 4.9: Shortened results of the compact flash card tests with USB card reader; throughput in MB/s, latency in ms.

very little influence. A long queue is ideal (especially for random access), has no negative consequences and is the default (probably for those reasons). Lots of processes for random access shows that the kernel reorders requests: a high standard deviation is observed. The fact that lots of processes are needed to write 512 byte blocks sequentially at high throughput makes makes it unlikely it will be implemented. Fortunately for Venti, there seems little need to write 512 byte block sequentially. Finally, a quick and informal test showed that larger blocksizes (16KB and 32KB) did not improve sequential write throughput.

Conclusions

The latency of random access on the IDE adaptor indicate that reads can be served in under a millisecond. Larger blocks read with higher latency on that adaptor possibly due to missing DMA support. The USB adaptor probably suffered from USB and adaptor overhead. With a better adaptor, 8KB blocks could probably be served in under a millisecond and with a high throughput. This would make them very usable for score lookups, i.e. when used as index sections. Unfortunately, random writes are extremely slow, a considerable obstacle. However, because of the small individual capacity of the flash chips, multiple memory card may be combined and written to concurrently. When using five times more compact flash chips than magnetic drives, performance will be similar in this regard. On the other hand, sequential performance is quite high on an individual memory card, therefore the current Venti optimisation of reading and writing 8MB of buckets of index section would already make a single memory card usable as index section. In any case, low capacity implies many memory

cards are needed anyway.

For all methods of using flash memory as index sections, the kernel should be made aware that operation scheduling is not beneficial. Or at least not in the same way as for magnetic drives. A quick and informal test showed that the standard deviation can be reduced enormously by using the no-op I/O scheduler.

Flash NAND memory can be used in a far more efficient way than over IDE or USB. When used directly on a system bus, it offers far better performance. Data sheets from Intel[26] and Micron[27], large NAND flash memory suppliers, claim a memory module can fulfil a random page read request—pages are typically around 2KB or 4KB—in 25 microseconds, and sequential page reads in 25 or 30 nanoseconds. A page can be written in 250 microseconds. However, a page write can only change bits from 1 to 0. Changing bits from 0 to 1 must be done a block at a time. Blocks are typically around 128KB. Erasing a block can be in 1.5 or 2.0 milliseconds, relatively slow compared to other operations. The bandwidth the memory can sustain will further depend on the system bus. If flash memory can be added to a fast system bus in amounts large enough to use them for score index, it can provide very good performance. The slow block erases can probably be avoided altogether: entry structures consisting of all ones can be taken to be invalid and thus unused. Since scores are never removed from index buckets (and do not necessarily have to be sorted), using only fast page writes will suffice.

At the moment of writing, solid state drives based on NAND flash memory with 128GB capacity are being released. A device just released by *Mtron* [28] has a SATA interface, claims 0.1 ms read random access and 80 MB/s and 100 MB/s sequential read and write throughput respectively, probably to be taken with a little grain of salt. The random write performance seems relatively low, with only 119 operations per second, i.e. comparable to a normal drive. According to their own measurements with 8KB blocks, sequential read throughput reaches 40.3 MB/s, sequential write 52.7 MB/s, random read throughput 47.1 MB/s and random write 0.92 MB/s. The write throughput is relatively slow, but still comparable to that of magnetic disk drives. Also, this may be because the device design has not matured, other or newer devices may have much higher random write throughput. The 'disk' contains many separate flash chips that are mapped to allow these fast accesses. The chips can be remapped to prevent them from being worn down by many local writes. Such a device would instantly provide a major performance boost for even the larger Venti configurations.

To conclude, standard compact flash cards combined with a high-quality adaptor are a candidate for use as Venti index sections, to increase Venti performance. However, for large installations, with large index sections, too many large and thus expensive memory cards are needed.

4.7.2 MEMS-based storage

Much research is being done in the area of MEMS-based storage. These are devices with many read/write *tips* (comparable to the heads of magnetic disks) arranged in a square, with a (magnetic or other) *media sled* to read from or write to, also arranged in a square. Figure 4.2 depicts a MEMS storage device. Currently, no such storage devices are available, but prototypes have been build. In the mean time lots of research has been done in how to use these new devices to make faster disks. A nice introduction to MEMS-based devices is given in [29].

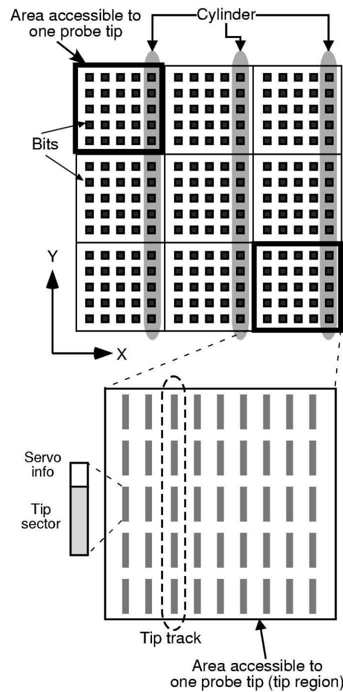

Figure 4.2: Representation of MEMS-storage device, from [30].

This paper is from 2000, in an early stage of development of these devices. Lots of predictions are done and probable characteristics of MEMS-storage devices are presented. The first generation of devices are expected to have a seek time under 1 ms, a sequential throughput of 25.6 MB/s and a capacity of a single media sled of 2.5GB. In second generation devices seek times would be halved (to 0.5 ms), throughput more than tripled and capacity upgraded to 4GB per media sled. They even predict third generation device properties. In short, these devices look quite similar to disk drives, but have much lower seek times and due to the many heads potentially higher throughput in the future.

The main point of interest is, of course, the fast random access to blocks. If the devices are really capable of that, it would allow for a very fast Venti index stored on non-volatile memory. When these devices become available, it is likely that they can be used as drop-in replacement for traditional magnetic disk drives. The principles of tracks, cylinders and sectors of disks do not differ substantially from the principles of MEMS disks, and as such current disk protocols such as over SCSI can still be used [31]. However, given the many read/write tips, higher parallelism can be achieved in theory. For example, certain ranges of blocks can be accessed simultaneously. This can be used to read an entire drive 'in the background', i.e. piggybacking on foreground operations[32]. For Venti, it could be used to flush parts of the index concurrently. This would need explicit support however. Also, scheduling algorithms have been designed specifically for MEMS-based storage devices[33], exploiting their physical properties.

Given the relatively small size of a single MEMS-storage media sled, other uses than magnetic disk replacement have been investigated. Typically, MEMS-storage devices are used as a cache sitting right in front of the magnetic disk[30, 34]. Due to relatively small work set size, many operations are handled quickly, resulting in low average response time. For index sections for Venti, this scheme would not provide any benefit since the accesses are entirely random and the cache hit ratio would be low.

To conclude, MEMS-based storage devices look promising: they have very low access time, and high potential for parallelism and throughput. Unfortunately, they are not available on the market yet. When they do become available, their actual properties (parallelism, access time patterns, scheduling algorithms) will have to be analysed and probably specific support in Venti is needed to get decent performance with them. It is questionable whether the first incarnations of the devices will have enough storage capacity to serve as index sections.

Chapter 5

Venti performance

This chapter investigates basic Venti performance. It starts with an analysis of simplistic Venti usage: sequential and random reading, writing, and random reading (there is no such thing as random writing in Venti). The chapter ends with performance tests of SHA-1 calculation and whack compression.

5.1 Basic Venti performance

The tests were conducted using the machine specified in Appendix A. The entire 9.1GB SCSI drive was configured as index disk. The entire 80GB IDE drive as an arena partition, filled with arenas. A bloom filter of 64MB was chosen by *fmtbloom*. Of the 512MB of main memory, 256MB was given to the index cache, 32MB to the disk block cache and 16MB to the lump cache. The tests have been performed using *randtest*, distributed with Venti. Venti protocol overhead has been tested by running *sametest*, a tool described in Appendix C.1. More details about the tests and results can be found in Appendix C. The results are listed in Table 5.1. For the tests, 2GB of data has been read and written, multiple times. Pristine writes are the writes of new data, i.e. data not yet present in Venti. Duplicate write performance has been tested as well. Duplicate sequential writes are duplicate writes written in the same order as the original. Duplicate permuted writes are writes written in permuted order relative to the first write of the data. Sequential and permuted reads are reads posted in the same or permuted order as the writes of that data. Sametest should measure only the overhead of memory lookups and buffer manipulation. The 68GB write test was done with 128MB of disk block cache, as was the 28GB read test. The 28GB read test was intended to read 68GB but stopped prematurely due to a read error. Further testing revealed the disk had a bad sector, but would not indicate this in a write to or read from the disk. Without Venti, a disk error of this kind, and the resulting data corruption might never have been discovered. Luckily, the disk was still in warranty.

Before an analysis of these results, a few caveats about these tests must be explained. First, not all advanced features have been enabled for this test. The bloom filter has been enabled, but queued writes have not. This follows the idea of using mostly defaults: bloom filters are encouraged by default, while queued writes are not on by default. Second, performance on larger systems

Type of test	Average throughput
randtest write pristine 2g	6.50 MB/s
randtest write duplicate sequential 2g	18.68 MB/s
randtest write duplicate permuted 2g	18.04 MB/s
randtest read sequential 2g	8.60 MB/s
randtest read permuted 2g	0.88 MB/s
sametest write 2g	18.93 MB/s
sametest read 2g	20.67 MB/s
randtest write sequential 68g	2.99 MB/s
randtest read sequential 28g	8.40 MB/s

Table 5.1: Basic performance results for Venti, in MB/s.

with multiple disks for the index sections and arenas is likely to be higher. Also, the test were conducted with only a single concurrent operation, multiple operations are likely to increase performance.

The throughput for *sametest* shows that there is quite some overhead in Venti. The program reads or writes the same block over and over. This means that for writes, the block very quickly is detected as being already present, with only in-memory lookups. For reads, the data is retrieved from the lump cache, also requiring only few memory accesses. Probably throughput could be increased by using more processes in the test program, more than with the other tests at least. The performance for duplicate writes is very close that of writes using *sametest*. This is not surprising since all index entries for the 2GB of data fit in main memory. Thus, checking whether blocks are already present needs only memory accesses. However, most current Venti clients do not send new requests before the previous finishes, so the chosen behavior best resembles real behavior.

5.1.1 Analysis

Permuted reads are obviously limited by random accesses of disks and show performance close to the maximum random access throughput of disks when posting one operation at a time. Using multiple disks for the arenas will increase performance only when requests are issued concurrently and the data is spread over the disks. Sequential reads have stable performance: reading 2GB or reading 28GB results in similar throughput. Though not close to sequential read performance of the underlying disk, see Table 4.6. This leaves room for improvement. The reason the 28GB reads with this throughput, is that index entries for arenas currently read a lot (as is the case with sequential access) are loaded into the index cache. This happens once for every arena, which is 500MB large. This optimisation is described in section 2.3.9.

The write throughput in the 68GB write test is disappointing. Pristine writes of 68GB are limited by writes to the index sections: writes in chunks of 8MB. Given the little amount of memory for the index entries, relatively few entries are written in each chunk, and therefore the dirty percentage of the index cache does not drop fast enough to sustain high throughputs. For the 2GB pristine writes, the index cache did not need flushing at all since the maximum percentage of dirty index entries was not reached. Thus, the 6.50 MB/s is sequential

write throughput without having to take index writes into account. Beside sequentially writing the data to the arenas, the arena meta directory must also be written, reducing write performance. But even with this in mind, write throughput is substantially lower than what the underlying disk is capable of. Queued writes are likely to increase performance.

The most important conclusion to be drawn from these results is that sequential reads and writes are slow compared to the throughput the underlying disks can achieve. A detailed trace of Venti's disk operations might give insight to where improvements can be made, especially when combined with the results of the previous chapter. For writes, queued writes probably would improve throughput; for reads, clients should issue multiple requests to improve throughput. Also, it seems that writing the index entries in 8MB chunks does not necessarily improved performance.

5.2 SHA-1 performance

SHA-1 performance has been tested with a simple tool called *test-sha1*, which can be found in Appendix G.2. It repeatedly performs SHA-1 on 8KB random data many times, 10,000 by default. Memory alignment can be specified, up to 32 bytes.

Results (each run done 5 times) show that 4-byte aligned SHA-1 is only slightly faster, not worth optimising for. On my 2.4 GHz *Celeron* processor, throughput is approximately 1500 MB/s for 4-byte aligned data, 52 microseconds per operation. For 2-byte or 1-byte aligned data, the throughput is approximately 1340 MB/s, 58 microseconds per operation.

Clearly, SHA-1 is not a bottleneck in Venti performance.

5.3 Whack performance

Whack is the compression algorithm used by Venti to store blocks in the arenas. It is an LZ77-like algorithm, meaning repeated data in a stream is replaced by a reference to the original occurrence. It does not need an explicit dictionary, so there is no fixed size-overhead. Highly random data will not find repeating patterns large enough to replace it with a reference, thereby achieving no compression. If there is no progress during compression, the operation is aborted. In this case Venti simply stores the data uncompressed.

For whack, the compression algorithm, a similar throughput test has been devised as for SHA-1. The program, *test-whack* can be found in Appendix G.3. It whacks blocks of 8KB in size, by default 10,000 times for each invocation. The data is either random (generated using *prng()* from the Plan 9 C library), from a binary (*/usr/bin/gs-gpl*, the largest binary in /usr/bin on my Linux system, over 2MB), or from a text file (*/usr/share/perl/5.8.4/unicore/UnicodeData.txt*). Only whacked blocked are unwhacked as well. Again, the *Celeron* 2.4 GHz has been used for the tests and resulted in the averages over five runs, shown in Table 5.2.

This shows that write throughput can be quite low for some types of data. It may even become a bottleneck when writing sequentially at full disk speed. It may seem surprising that compressing random data is fast, but it is not:

Type of data	Whack	Unwhack	Compressed size
Random	25.6 MB/s	-	100.0%
Executable	14.0 MB/s	69.8 MB/s	45.8%
Text	17.0 MB/s	112.9 MB/s	20.2%

Table 5.2: Whack performance.

the algorithm detects early on that compression is not useful, and aborts, not wasting any more CPU cycles. This also explains why no random data has been unwhacked. Unwhacking is never a bottleneck, this is typical for compression algorithms.

Chapter 6

Venti simulator

This chapter describes the results of designing and implementing a simulator for Venti.

Initially, the main goal of this master's project was to make a simulator. Mostly to be able to play around with the various Venti configuration parameters. For example the index, lump and the disk block cache sizes. But also optimisations such as queued writes, the bloom filter and its size, and prefetch heuristics. And of course, to measure the effects of new optimisations. All this could, of course, be tested using a real Venti, but that would take too much time. A test of an acceptable scale involves reading tens or hundreds gigabytes of data into Venti, and even with todays fast disks this simply takes too long. A simulator would allow much faster analysis of how Venti would handle a given workload.

In order for the simulator to be representive of actual behaviour, and to allow simulation of new algorithms easily, the simulator has been designed to be integrated into the Venti source code. This avoids having to write algorithms both for Venti and for the simulator, and ensures the simulator has comparable overheads for the optimisations. Basically, all—or at least most—disk accesses have to be avoided, thus simulated since that is where Venti spends the bulk of its running time. The idea of using traces of Venti operations (with enough information in them for Venti to walk through the logic paths, for example for determining which caches to read from, or simulate disk accesses) survived from almost the start of the project, and indeed seems to be the most straightforward way to test real loads, and in a reproducible way. Thus the initial goal: making such a simulator. The design for the simulator was determined through discussion with Russ Cox, who also (re)wrote the Venti implementation the simulator should simulate.

The next sections describe the evolution of the trace generator *venti-trace* and simulator *venti-sim*: the design and how another Venti simulator released during the project influenced it, the trace file format and how to make trace files, suggestions for making a full featured simulator in the future and finally conclusions about the simulator effort.

6.1 Design

The simulator was to read a trace file, and either execute or simulate actions on the caches, disks, process communications, etc. to handle a Venti operation. Section F.1 gives a pseudo-code overview (mixed with natural language) of how read and write requests are handled in the Venti code, which functions are called and which branching decisions are made. This results in the following actions for four combinations of read/write, score present/absent.

read, score not present

1. Lookup in lump cache, returns negative.
2. Lookup in index cache, returns negative (Venti does not cache negative results).
3. Lookup in bloom filter. If absent, done: no disk activity at all.
4. Read bucket from index, proves score to be absent.

read, score present

1. Lump cache may contain data. If so, done: no disk activity at all.
2. Look in index cache, may contain index entry. If so, no disk activity on index disks. Otherwises, read bucket from index section.
3. Read data from arena.

write, score not present

1. Lookup in lump cache, returns negative.
2. Lookup in index cache, returns negative.
3. Lookup in bloom filter. If absent, no disk activity on index disks. Otherwise, read bucket from index section, proves score to be absent.
4. Write data to arena.
5. Write clumpinfo to arena directory.
6. Write bucket to index disk.

write, score present

1. Lump cache may contain data. If so, done: no disk activity at all.
2. Look in index cache, may contain index entry. If so, done: no disk activity at all.
3. Read bucket from index section, proves score to be present.

Note that the arena directory is only used for writing not yet present blocks, it is never read in direct response to a Venti read or write operation. The bloom filter resides in memory, thus accessing it never results in disk activity. Only the periodical writing of the bloom data results in activity, but this can be neglected for a first version of the simulator. Besides, the first version of the simulator does not simulate the bloom filter at all.

The format of the trace file decided upon is described in the next section, the code for reading the trace file is not interesting and therefore not discussed further.

The approach for the simulator was to modify the logic in the Venti code, such as the if-else branches. For example, index lookups are not actually performed: where the Venti code would perform an index disk read to determine the arena disk location of a block, the simulator would only count the disk access, skipping the actual read from the index disk, and take the disk location from the trace file, continuing the program flow as normal. This was quite easy to implement.

However, after having integrated this into the Venti code, a few important problems had to be solved. First, how would one simulate disk behaviour? A disk model seemed necessary to determine whether a read was sequential or random: very important for determining how much time it would take on a real disk. At that time not all disk properties were understood, which is one of the reasons for analysing disks, the results of which are described in Chapter 4. A detailed model of a disk was deemed necessary, mostly to keep track of head and rotational position, so disk operations were intercepted at a level (*read* and *write* system calls on the disks) that allowed for that. In hindsight, has was too fine-grained for an initial version of the simulator.

Another problem was how to schedule processes. Venti consists of many processes communicating with each other. Especially operations that take long to finish, such as disk accesses, are often sent to dedicated processes. These processes then return the result when they are finished. This seemed to complicate time-keeping a lot. It would have been nice to be able to simply add some milliseconds to a global clock for each disk access. But with multiple readers from multiple disks, this would not be possible. A solution to this problem came from Russ Cox and Shaun Foley, in the form of their Venti trace generator and simulator *vtrace* and *vsim*. The same solution has been used in venti-sim from then on as well. The details of their solutions, and more about their simulator, are described in Section 6.3. Summarised, it amounts to changing all processes to cooperative threads running inside a single process, each reporting the amount of time their current operation will take them to finish to the scheduler. The scheduler keeps track of all finishing times, waking up each process in the order of being unblocked after their time-consuming operation, and keeps a simulated clock. Before this method, venti-sim was still designed to start all processes and have them communicate with each other. But that would make some Venti features hard to implement, such as queued writes: how should accounting be done for postponed disk accesses? It must be noted that the simulated clock using cooperative threads is a nice and simple approach, but may be too simplistic when using multiple disks. Section 4.4 showed that random multiple operations posted concurrently can quite dramatically increase throughput. However, this model makes it hard to model concurrent disk operations: the threads have to decide up front how long their disk operation will take, which is influenced by operations posted after that decision in reality. Also, multiple cooperative threads may use the same disk and erroneously assume they are the only users of the disk, causing wrong finishing times for the operations to be calcuated.

While modifying the Venti code, lots of error handling code paths were encountered. Some were for assertions and thus left mostly intact. Others, handling more serious errors, are assumed to never happen, to keep the simulator from growing too complex. All reads and writes to disk will succeed, and no error conditions are assumed to occur at all. Note that some error conditions really can occur in a Venti installation, such as filled buckets in the index hash

table. But they should be rare, require a restart of Venti and generally indicates a misconfiguration, so are assumed to be irrelevant.

Next is the question of what output should be emitted by the simulator. Of course, the more statistics the better, and Venti already generates a lot of statistics while running. These include disk reads and writes to each type of disk, cache hits/misses, bloom filter fullness, venti protocol messages by type, etc. Actually more statistics than useful for the purpose of the simulator. The most important statistic is the total running time of the simulated clock, the time it took to execute all trace operations. But also, mostly for sanity checking: the time spent accessing disks, the number of disk (index and arena disks split) accesses, and the number of cache operations (inserts, hits, misses). The running time is the most obvious statistic of how Venti performs, however it has the downside that using the total running time does not reflect 'local' performance during a trace. To clarify, consider a trace consisting partially of random reads and partially of sequential reads. The total running time does not give any indication whatsoever about how well each of the types of operation performs. This is not easily solved, the best way to avoid it is to simulate random reads and sequential reads (or other statistic-blending operations) separately. The downside of which in its turn is that that probably is not a real workload, but rather an artificial, synthetic one, and thus might not be realistic.

The code for venti-trace and venti-sim can be found at

http://www.xs4all.nl/~mechiel/files/venti-simulator.tgz

Note however, that it comes without instructions on how to use venti-sim and will need changes before it can do simulation runs. Trying vtrace and vsim (described in Section 6.3) first is recommended.

6.2 Trace files

One of the first tasks for the simulator was generating trace files. Without trace files, there is no good way of simulating traces. Implementing this proved again to be relatively easy: the Venti source code was modified, print statements inserted at the right locations in the Venti code for each read/write/sync operation, all to collect enough information to mimic the operation in the simulator without needing to go to disk. The trace file was designed to be human-readable for easy handling (composing, editing, interpreting), and contained information such as score, block type, operation type (read, write, sync), whether the score was present in Venti, the disk location (index bucket) of the index entry on the index disks, the location in the arena, etc.. This is an outline of such a line of the trace file:

```
read/write present score type size csize arenapart
      arenapartaddr arenasectorssize arenablockssize
      ciaddr ciblockssize isectpart isectaddr isectblockssize
```

The fields needed in the trace file where determined by interpreting the pseudo-code for handling the read/write requests (see Appendix F.1), and the derived overview from the previous section.

- Score

- Operation (read or write)
- Type of the data
- Whether the block is present on disk
- Size of the data
- Compressed size as stored on disk (in bytes, without header)
- Arena partition the data must be read from or written to
- Arena partition offset to read or write at
- Number of on-disk bytes for data + header, rounded up to disk sectors (512 bytes)
- Number of bytes to read or write for data after rounding disk cache block
- Clumpinfo block address (on same arena partition as data)
- Clumpinfo block data size to write
- Index section partition
- Bucket offset on index partition
- Bucket size
- Venti global address, used for debugging

More information than strictly necessary is included, at least for a first version of the trace file. For example, both the (uncompressed) size of the data, and the compressed size are available. This allows for accounting compression time in the future as well. This should probably have been of later concern. The *arenasectorssize* and *arenablockssize*, or *ciblockssize* and *isectblockssize* are unnecessary as well. They should have been global—or per trace file—parameters.

At first, the need for a 'pause' message had been envisioned, representing idle time. In this time, caches could age (entries get stale) and buffers could get flushed to disk. This has been scrapped at an early stage: the sync message is called by any sane client of Venti (flushing the disk buffers), and ageing caches did not seem to change Venti behaviour a lot—if at all—in cache behaviour.

The procedure for creating a trace file is as follows. Start venti-trace just like a normal Venti is started. It will automatically write a half-finished trace file to the file *semitrace.txt* during operation. Now use any Venti client to read/write/sync to the Venti. For example, use *randtest* to write data and later read it back (though note that this is not really representative of normal client behaviour). When finished, the semi trace file will contain the operations that have been executed by the Venti. The program *tracemake* will read the *venti.conf* the Venti was started with, and the *semitrace.txt*, producing a new file *trace.txt* in the trace file format of the example above.

This has serious drawbacks: the disk was read/written during the normal Venti operations, but had to be consulted in tracemake again to get to the disk locations used for the score. *Vtrace* has a better approach: it records the disk I/O's while they are happening. Venti-trace did not use this approach at first mostly because Venti's design was not yet understood well enough. The concern was whether (concurrent) disk operations could be mapped to a single Venti operation. *Vtrace* solved this problem by using a sequence number passed up to the functions that printed a part of the trace message. The lines of the traces file were later reassembled by matching sequence numbers. The *vtrace* format contains roughly the same information, but of course the syntax was different.

By this time synthetic trace files had been considered. For workloads of multiple terabytes, generating even a trace file takes a lot of time. So, being able to make up a trace file would be a big plus. Tools for generating trace files have not been implemented, but with some knowledge about Venti's internal structures, mostly arenas and index sections, it is not very hard to generate synthetic traces.

6.3 Vsim and vtrace

Vsim and *vtrace* are a Venti simulator and Venti trace generator written by Shaun Foley and Russ Cox. It is described in an (unpublished) article[35] which also includes measurements of the accuracy of the simulator. It is interesting to see the vtrace and vsim approach is very similar to that of venti-trace and venti-sim. The similarities are in how the Venti code has been modified, which logic paths were modified and how. As mentioned in Section 6.1, the vsim method of keeping time was quite elegant. From their article

> Vsim replaces disk operations with calls to alarms that block until some specified time. If Venti used only one disk, these alarm calls could immediately increment the virtual time and return. However, because Venti uses a separate thread for each disks, computing elapsed real time is more complicated. When a disk read or write would have occured, the thread calls a delay function for the estimated time based on the disk model. This sets an alarm causing the thread to be woken up after the desired delay.
>
> Alarms are coordinated via a clock thread that waits until all other threads are blocked. We removed or replaced all blocking system calls, ensuring that each thread only sleeps by setting an alarm or waiting on a lock. At this point, the clock finds which alarm will be the next to fire, skips the virtual time ahead to that time, and wakes up the corresponding thread. The alarms let the simulator run at CPU speed rather than disk speed, while introducing the virtual clock lets the simulator skip past inactive time.

They note that vsim does not implement all optimisations (including the important ones that have a big effect on performance): index entry prefetching, an advanced disk model, lump read-aheads and compression. With this simple model they conducted simulation runs, to verify that simulated behaviour is close to the behaviour of Venti. Of course, to this effect they disabled the advanced features from the reference Venti and ran tests using *randtest* and varying workloads (random/sequential, varying cache size). Some artifacts due to the size of the cache were present in the results (as expected): up to a certain amount of reading and writing the operations could no longer be fulfilled from the cache (because it could hold no more entries) and slow down. The main conclusion of their article is that the simulated time is within 1.41 of a real time, and 1.21 for tests involving more than 64MB of data.

A few sidenotes should be placed. First, they used a single disk that hosted both the index section and the arenas. This is not what a typical Venti installation looks like (it has separate index and arena disks), and may have an

influence on the performance of Venti. A typical disk usage pattern is a seek on the index partition (disk), followed by a seek on the arena partition (disk). Using the same disk for these partition sabotages locality of reference of the disk heads. Second, vsim has an accuracy of being within 1.21 of actual performance. New optimisation strategies are not likely to cause such huge performance gains, meaning it will be very hard to use the simulator to show that a different configuration or new algorithm will improve performance. And third, as their *future work* section mentions, vsim does not simulate enough of Venti to match a real Venti close enough. The same problem the unfinished simulator of this chapter has.

6.4 Future work

The currently available simulators are not accurate or complete enough for performance measurements. They do not simulate all behaviour of the system, but of course, that is what a simulator is about: leaving out some parts and replacing them by a simple surrogate. Still, some mechanisms or optimisations depend on others, the behaviour of one can radically change the behaviour of the other, e.g. filling a cache by reading-ahead could either trash a cache resulting in lots of cache misses, or could make the cache effective resulting in many cache hits. With lots of cache misses, other optimisations can kick in further altering behaviour of the system. For example, Venti has a scheduler that can postpone an I/O intensive background operation when there is a certain I/O load. This is a result of some of the optimisations being based on heuristics. But besides that, even a single optimisation can result in a big performance gain. Therefore, some optimisations that have not been simulated should have been for representative results: the bloom filter, index entry prefetching from the arena, and a better disk model and a disk scheduler understanding multiple disks.

Implementing these optimisations have been looked at, and ideas to that effect are described in the next sections.

6.4.1 Bloom filter

The bloom filter prevents lots of index accesses when writing pristine blocks. So it is needed for that reason. Whether the bloom filter returns 'hit' or 'miss' depends on the size of the bloom filter, and how filled it is. There is a relatively easy way to simulate the bloom filter: While making traces, the probability of a hit or miss could be recorded, or the current filledness (the size would be a per trace file global parameter). During simulation, this information (along with a random function) would be used to determine whether a hit or miss should occur for a particular operation. Relatively minor extensions to the trace file format, trace generator and Venti simulator would implement this.

6.4.2 Index entry prefetch from arena directory

This problem is harder to solve. Venti has heuristics to determine if an arena is currently being read from heavily, and if so, reads all index entries from its

arena directory into the index cache. Leaving this out has big consequences for sequential reads, keeping it in can have consequences for random reads.

There is a problem with simulating this: when the simulator would want to load index entries into the cache, which entries would that be? The entries (scores) cannot be included in the trace, because with terabytes of data, that would not scale: it would have to contain gigabytes of scores. Russ Cox proposed a solution, namely to pretend a range of arena addresses were loaded in the cache. On a subsequent cache request, the range(s) of addresses could be used to determine if the index entry is supposed to be in the cache. However, this would make the simulator complex. How many scores are present in the range? But more important are the caches LRU-eviction decisions. Which of the 'vitual' entries from a range will be evicted? How are hits from a partially evicted range determined? It can be done, but it is easy to inadvertedly change the semantics of the operation to have a negative effect on simulator accuracy. Also, the index cache code would have to be modified, making it diverge from the Venti code.

6.4.3 Better disk model and multiple disks

The approach to disk simulation in venti-sim and *vsim* is to assume a fixed access time and a variable (based on content length) read/write time. The simulation time for a disk operation is the sum of the two. This is a simplistic model, the access time could be made dependant on whether it is a read or write (writes are slower). But a more advanced solution could include *disksim*, a disk simulator. A problem with doing that is that is does not seem to play very nice with the virtual clock approach: you cannot post an operation to it and get the finishing time returned. This is because multiple operations can be posted, influence each other and finish out of order. Also, disksim seems to want control over a global clock. Nevertheless, the disk models it has should be very accurate, and the difficulties explained can probably be overcome.

6.5 Conclusions

A simulator suitable for the purpose envisioned has not been finished. Some basic simulations have been run, but the results were not representative for a real Venti. One of the mistakes was to assume a disk model that was too fine-grained: the disk model available was neither fine-grained nor precise.

Many of the important optimisations have not been implemented. Implementing them seems possible (as described in the previous section), but could become complex and will be hard to get right. They are necessary for simulations that come close enough to the real Venti. However, it is not clear that it will come close enough to actual Venti performance to allow testing and verification of the remaining not-so-low-hanging optimisation fruits. Also, the behaviour Venti exhibits has an impact on the behaviour of the client, making it again harder for a simulator to simulate real workloads.

Some of the original assumptions appear to not always hold true. For example, the idea was that algorithms would only have to be implemented once (in Venti), and would run in the simulator as well. It turns out that this does not always work, e.g. for index entry prefetch (as discussed in the previous section). Another assumption, that it would be easy to keep the simulator integrated with

the Venti code does not appear to hold true either. Keeping the two versions in sync is hard and will make the code a lot less clean and readable.

Finally, of course, the appearance of the idea underlying memventi (described in the next chapter) shifted the focus of this project, but the difficulties just explained helped as well.

So how can one currently determine whether a Venti configuration parameter or new algorithm will improve performance? First, an analysis of all existing optimisations in place helps, considering their implications for the intended workload. The workload is very important for performance, random reads are very different from sequential reads, so are very small and very large reads or writes. Also, a small scale test with a real Venti can be conducted, the results can be interpolated to a larger Venti when being careful. Parameters such as cache sizes may have to be scaled down to keep the test representative in that case.

Chapter 7

Memventi design & implementation

A large Venti configuration needs quite a few disks in the index section to deliver adequate performance. An early installation from 2000 at Bell Labs [36] used eight 10,000 RPM 9GB SCSI disks just for the index. The index is the most obvious candidate for optimisation. Its high on-disk lookup time is expensive. It would be nice to be able to place the index in main memory. This would be much faster and eliminate the need for lots of other optimisation strategies, allowing for a simpler design.

The size of the index is linearly related to the number of blocks it is able to store. The maximum number of blocks stored depends on the size of the arenas and the average block size. Data is chopped up into 8KB chunks by most applications, thus these occur frequently. Smaller sized blocks are typically ends of files, pointer blocks and meta-data blocks. Lets assume that the average compressed block size is 4KB. The question now remains: How many blocks of 4KB can be stored in how much main memory for an in-memory index?

Let's assume we want to store 2TB of data, in 4KB blocks. The total number of blocks would then be 512M blocks. These have 512M different scores of 20 bytes each. The data is addressed by a 8 byte arena address. The block size is stored in 2 bytes, the type takes 1 byte and then there are 4 bytes for data-keeping. This adds up to 35 bytes per index entry. For 512M entries, we would need 18GB of memory just to store the entries. Only using many separate systems loaded with memory could this scheme work in practice.

To make an index-in-memory scheme feasible, the size of the index entries must be reduced. There are a few ways to do this. Lets consider the data fields in the index entry:

Type The type is part of the address, just as the score. It should be noted that not all 8 bits are used, only 5 bits are currently used by Venti clients, though the protocol does allow using all 8 bits. Other information could be piggybacked.

Size The size is not really necessary for a lookup, but is nice to have around when reading the data from disk. Most blocks will be exactly 8KB, otherwise they are likely to be smaller. Since read-ahead is performed on

the arenas for fast sequential transfers reading less than 8KB is not very useful. It would be nice to know when the data is larger (up to 56KB). This information could perhaps be piggybacked in the type field, using a bit that specifies the block is larger. If so, 56KB can be read and the actual size discovered from the on-disk header. But really, the size field can be removed from the index.

Address The address is 64 bits, this allows us to address vastly more data blocks than may ever fit in memory. Thus, reducing the addressable data space does not change the maximum size a venti with in-memory index can be. With 48 bits (6 bytes), 256TB can be addressed.

Data-keeping Index entries in the current Venti are implemented as a linked list. This eases adding of entries to the hash table. Memory consumption for this can be reduced by allocating multiple index entries at once and only linking these groups in a linked list. When grouping 16 entries, 1 byte is necessary for accounting how many entries are in use and 4 bytes are necessary to link to the next group. This takes 5 bytes per 16 entries, totalling to 0.32 byte per entry. Of course, allocating elements before they are used will waste some memory, so care must be taken. The group size can always be reduced to limit memory wasting: grouping only two entries already reduces memory consumption. A combination of different group sizes can be used (e.g. at startup, read the entries into 32-entry groups, use 4-entry groups for the last entries and for new entries). This way already stored scores take little memory and new scores do not take too much memory. Note that the number of entries that fit in the group has to be stored somehow.

Score The score is 20 bytes. This is most of the memory of the index entry. Without reducing its size, the index will never fit in memory. However, it *is* possible to reduce the number of bytes stored. Remember that the scores are distributed randomly and that hash collisions are infeasible to find. The current Venti index entries map the full 160-bit score to a disk location. When a venti read requests comes in, the 160 bits have to match before the data is retrieved from the data section. However, the index entry could also contain fewer than 160 bits and thus compare fewer than 160 bits, say 80 bits. Then in some (still very rare) cases, more than one of these half-scores may match a requested score. The disk locations for these matches have to be read until a full match is found. The data header in the arena will have the full score and the right data can be returned (if any). The following section further investigates this idea.

7.1 Storing only a part of the score

The index entries can be kept in a linked list of which the heads are pointed to from an array. This array could be 2^{24} elements large, each pointer of 4 byte in size, taking 64MB of memory. The index serves as part of the score: e.g., at index i, the scores are stored that start with the three-byte value representing i. The remaining part of the address can be stored in the entries. First, consider using 40 bits of the 160-bits score in the index entries. Of these, 24 bits are

encoded in the lookup table, and only 16 bits are stored in each index entry. With a 6 byte address, a 1 byte type, and 0.3 byte overhead, this leads to 9.3 byte per entry. For a machine with 2GB of memory, 1500MB may be used for the index. With 9.3 byte per entry, 1500MB (with 64MB taken for the lookup table), this results in around 154M entries. With blocks that need 4KB on disk, this would take 616GB of disk space to store the data. For larger venti installations multiple such machines may be used and load-balanced together.

As mentioned before, it can occur (and is quite likely to occur at some point) that scores are stored with the same 40 bit score prefix. This means two identical entries will be stored in the index. Later, when a venti read request comes in, both index entries will be found, and both may have to be read from disk. If this happens too often, it could be a considerable performance penalty. The commonly known birthday paradox dictates this will happen sooner or later for these address ranges, but how often? We start with a single (half-)score in the 154M (less than $n = 2^{28}$) scores (entries). What is the probability a second (half-)score is the same as the first? The second (half-)score, just as the first, has to be one of the $d = 2^{40}$ possibly scores, and each is equally likely. All choices are independent from each other and the probability that a single second score is the same as the first is 2^{-40}. When taking $n + 1$ scores, the probability that the first score is duplicated is:

$$p_{dup} = q(n, d) = 1 - \left(\frac{d-1}{d}\right) \approx 2^{-12} \tag{7.1}$$

Low enough it seems. However, any pair of scores should be considered. For each kth chosen score the probability of a duplicate needs to be taken into account, leading to the total number of duplicates:

$$\sum_{k=1}^{n} q(k-1, d) = n - d + d\left(\frac{d-1}{d}\right)^{n} \tag{7.2}$$

For $n = 2^{28}$ and $d = 2^{40}$, this results in $32K$ duplicate entries. So, $2^{15}/2^{28} = 2^{-13}$ of a filled venti will be duplicates. The data blocks will be in random locations in the arenas, so each needs a random disk access to be found. When a single random disk access takes 10 ms, the average random venti block access would take around 10.001 ms: only in one in every 2^{13} blocks needs an additional disk access. This is for reading only. Writing is potentially more expensive: the same amount of collisions can occur as with reading, however, blocks that are already present (a common scenario) need an additional disk access (but if the full score is found, the block does not have to be stored). Note that the probability that a particular score is already in the index is based only on address space (2^{40}) and fullness: a huge address space or empty venti will have very few collisions. The 10.001ms average block access only occurs when the venti is (almost) full. It should be said that second-order collisions can and will occur in practice as well. These are score-prefixes that occur three times in the index (they can be generalised to n-order collisions). However, these occur even less frequently, having a negligible impact on average block access time.

An optimisation for writing could be to not immediately verify the data is already on disk which needs disk accesses, but to temporarily store the data on a separate disk and wait until the venti is idle before verifying and possibly

storing the data. This would make writing duplicate data very fast but also introduce significant accounting overhead and complexity.

Using the bloom filter the determination whether a block should be written can be made quickly, without going to the on-disk index. This new scheme does not benefit from a bloom filter: if the block is not already present, the index will indicate this (except for once every 2^{-13} scores); if the block *is* present, we have to read from disk to verify anyway.

Some disk space has to be reserved for the index entries, when flushed. The on-disk size will be at least 1500MB, and at most a few hundred MB more. When only flushing when shutting down, a 2GB portion of the arena disks can be used.

Another problem with this scheme may lie in the fact that malicious users are able to insert non-randomly chosen scores and induce lots of duplicates. The current Venti suffers from this same problem with its on-disk index entry buckets.

Another catch might lie in how to flush the index to disk. 1500MB would be too much to flush to disk when a system is being shutdown. A battery-backed machine can probably be in time on power failures though. Keeping track of which index entries are new might also work, but incurs some memory overhead. When flushing the entire index to disk the on-disk index is sorted, which helps implement the scheme mentioned earlier about allocating groups of entries for entries read at startup.

Venti startup time will be slow since 2GB of data might have to be read at most. Current hard drives can easily read at 50 MB/s sequentially, needing just over a minute to start.

As for cost, a 660GB venti would need a minimum of 33GB of index disks. Two 36GB SCSI disks (smaller disks are not available anymore) could do the job, allowing for limited concurrent lookups. These would cost about the same as 1.5GB of main memory.

Some of these ideas may also be usable in the current Venti system. For example, the current index on disk can be kept, but not the entire score has to be kept in memory.

7.2 Implementation

In order to verify the feasibility of this scheme, I implemented the idea of storing only part of the score in main memory for lookups. In general, the design as outlined in the previous section proved to be usable. On minor points, the strategy had to be adapted or has been improved.

The resulting program, called *memventi*, is written in UNIX C and uses the pthread library. It runs as a daemon. To give an idea how it works, this is a possible invocation:

```
memventi 14 19 35
```

The numbers have the following meaning. First is the 'width' (number of bits) used for the heads, 14 bits thus 2^{14} heads in this case. Then for the score-part in each entry, 19 bits. The address is 35 bits long, thus memventi is able to address 2^{35} bytes in the data file, approximately 32GB (in which the data block

headers are also included). The original design assumed the widths were byte-sized, however, there is no reason to demand this, so it has been generalised to bits.

7.2.1 Features

Memventi behaves like a typical daemon: by default it forks into the background, logs errors to syslog, has options to influence verbosity and debug printing and to run it in the foreground.

In most respects it is very simple and does only the minimum a venti server has to do. For example, it does not store or verify in any way the user name the Venti client sent at connection startup. Similarly, there is no way to authenticate a connection. The only method implemented to prevent abuse of a publicly accessible memventi is that connections can be made read-only. At startup, additional host names and port numbers to listen on can be specified as arguments to the *-r* and *-w* options, specifying that connections to the host are read-only or read/write. When nothing is specified, *-w localhost!17034* is implied.

The implementation is reasonably small, around 2000 lines of code. This includes lots of error handling and logging to ensure problems can be fixed when they occur. For example, when a block has been read from file and the score in the header does not match the data, the offset and mismatching scores are logged. Or when the data file address space has been exhausted this is logged. When an error occurs, memventi puts itself in degraded mode, not writing any new blocks and only servicing read requests. This ensures the data file is not made inconsistent any more that it might already be. To get into normal mode again, memventi has to be shut down, the problem fixed manually and restarted again. Also, at startup, the number of bytes read and corrected is logged, along with the startup time.

In the example memventi call above, no disks to store data on have been specified. This is because it is not necessary: by default, files named *data* and *index* are used to store the data blocks and the index entries respectively. The next sections explains the format of these files and why they are reliable to use. The file names can be overridden using the *-d* and *-i* options.

On reception of the signals USR1 or USR2, statistics are printed. For USR1, the statistics are the histogram of the number of entries in each head, but only for the lengths that occur at least once. For USR2, the histogram for the number of matching scores in the index for all read and write requests since startup is printed, but only where the value is greater than zero. Thus, for a freshly started memventi without data, USR1 and USR2 will not print any entries yet. After the first few operations, the matching scores histogram list all operations as requiring at least one disk access, unless duplicate blocks are already written.

The memory layout has changed slightly from the design described earlier. The heads table does not contain pointers to structures containing data. That level of indirection has been removed. The heads table is an array of *Chain*'s, a head is a linked list of chains. Each chain contains a pointer to a data buffer (in which the score prefix, types and addresses are stored), a 1-byte count with the number of entries the data buffer has memory for and a pointer to the next chain-element. The pointer to data is necessary because the data can be of variable length, thus cannot be part of the struct itself. At startup, all chains

are initialised with a count of zero elements, no data is allocated. Only when the first entry is added, memory for at least the minimum number of elements is allocated. At startup, the total number of elements in the index file is calculated and considered evenly spread among all heads. Now, newly allocated buffers are made larger than the default (8 at the moment, up to 255 elements since the count is a single byte), when the head does not yet contain the average number of elements per head. The data buffer is laid out as follows: The first *count* (number of entries) bytes are the types, these take one byte per entry. The subsequent bytes are bit-addressed: an entry has *entryscorewidth* bits for the score followed by *addrwidth* for the address; entries follow each other, without padding. Unused entries have their address bits set to all ones, this is used to determine what the last valid entry is. When the data buffer is full, a new chain is allocated and the *next* field of the current one made to point to it.

Since each connection has its own process, the data structures have to be protected against concurrent access. For this purpose, a lock for writing to the data file is used to ensure new blocks are added one-by-one. For the heads table, 256 read/write locks are used, each controlling an equal part of the heads. Threads for handling venti read requests can always access heads concurrently, but venti write requests need exclusive access for allocating and updating the data buffer. With 256 locks, it is unlikely connections have to wait for each other often.

There is one more important thing to discuss: how does one determine reasonable values for the *headscorewidth*, *entryscorewidth* and *addrwidth*? A program called *calc.py* is supplied for this purpose. It prints the best configurations for the parameters supplied on the command line, which are the following:

maxdatafile type: power-2 range
: Maximum data file size, e.g. *32g-128g*.

blocksize type: power-2 range
: Average blocksize, e.g. *4k*.

collisioninterval power-10 range
: Collision interval, e.g. *1000* means one in 1000 scores have a duplicate in the index.

maxinitmem type: single value
: Maximum memory used for an empty memventi.

maxtotalmem type: single value
: Maximum total memory used when filled, e.g. *350m* for a dedicated system with 512M total main memory.

minchainentries type: power-2 range
: Number of entries in a single chain struct. This is set at compile time, so not a run time option.

minmaxblocksperhead type: interval
: Minimum and maximum blocks per head when filled, e.g. *-3000* (meaning 0 to 3000).

All values can be specified with a suffix specifying kilobytes, megabytes, and so forth, i.e. *4k* means *4096* and *32g* means *34,359,738,368*. Ranges can be

specified by *start-end*, e.g. *32g-64g* or *10-1000*. A power-2 range of *32g-64g* generates the values *32g* and *64g*; the power-10 range of *10-1000* generates *10*, *100* and *1000*. Multiple ranges can be specified, each comma-separated. An interval is similar to a range, except that only a single range is allowed, comma's are not; also, *start* may be omitted implying a *0*, e.g. *-4k*, is equivalent to *0-4k*.

For each combination of parameters, a configuration is calculated and sorted: least maximum memory usage first. To illustrate, an example invocation (many configurations are produced, only the first is reproduced here):

```
$ ./calc.py maxdatafile 32g blocksize 4k collisioninterval 1000
{'addrwidth': 35,
  'blocksperhead': '513',
  'chaindatasize': '62',
  'collisions': 4095,
  'config': {'blocksize': 4096,
             'collisioninterval': 1000,
             'maxdatafile': 34359738368,
             'maxinitmem': None,
             'maxtotalmem': None,
             'minchainentries': 8,
             'minmaxblocksperhead': None},
  'entryscorewidth': 19,
  'entrywidth': 62,
  'headcount': '16k',
  'headscorewidth': 14,
  'maxmemused': '72m',
  'memperblock': 9.1005859375,
  'minmemused': '144k',
  'sort': 76341248,
  'totalblocks': '8192k',
  'totalscorewidth': 33,
  'unusedmem': '576k'},
```

The output is a bit unpolished, but all possibly relevant information is present. The *headscorewidth* is 14 bits, resulting in a *headcount* of *16k* entries. The *entryscorewidth* is *19*. The *addrwidth* is *35* bits, the total number of *collisions* is *4095*. The minimum memory used is *144k*, the average *unused-memory* (spent in partially-used data buffers) is *576k*. When filled, the index uses *72m* of internal memory.

Note that the configuration can be changed by shutting down the *memventi* and starting it with different parameters. The memory-efficient layout is only used in memory, the index file contains values wide enough for all hypothetical configurations.

7.2.2 Data integrity

As Venti does, memventi also considers data integrity to be very important. Therefore, it uses a very simple file layout for both the index file and the data file. The files are opened append-only: previously written data can never be altered by memventi, the operating system protects against overwriting. Files

are not split up in arenas as they are in Venti: when backing up a filled up part of the data file, simply read that part of the data file, e.g. by using the UNIX-tool *dd*.

Blocks are written immediately to the file when processing the venti write request. First the header is written, followed immediately by the data itself. The header contains a block marker (a 'magic' value), the full score, the type and the size (and length) of the data. The data is not *sync*-ed immediately, that happens only periodically, once every 10 seconds, after an explicit venti sync request and after closing down a connection. The index file is synced after the data file.

After the data has been written to the data file, the index header is appended to the index file. The index header contains the first 8 bytes of the score followed by the one-byte type, followed by the 6 byte address.

There are several scenarios in which writing data may go wrong. For example, after writing the data header, the disk may be full. Such conditions are logged to syslog.

At startup, memventi checks the last block in the index file. It reads the corresponding block from the data file and checks whether the (partial) scores are consistent. If the data block just read is not last block in the data file some index entries are obviously missing. The data file is therefore read to the end, blocks are verified while being read and added to the index file. When this is done, the index file is complete and consistent again and normal loading of the index file into memory commences. Loading the index file is done by reading it sequentially and adding each entry to memory (possibly allocating new chains) on the way.

Finally, for every block read from disk to satisfy a read request, the score in the header is checked against the actual score of the data freshly read.

7.2.3 Differences with design

Memventi differs at a few points from the design described earlier. Some have been described already, such as the different memory layout (which has been specified completely). At first, the index was supposed to take 2^{24} entries as minimum memory for any decently large memventi. This has been reduced a lot. The design talked about arenas, implying a similar on-disk structure would be used. This was deemed unnecessary however, and normal flat files on a file system are used, simplifying the design considerably (e.g. raw disk blocks have to be written in 512-byte-sized blocks).

Some ideas have not been implemented. Most notably the idea to speed-up writes: *'do not check whether the blocks are already present when the venti write request comes in, just write it immediately to another disk (or file)'*. It may still be a worthwhile optimisation, however a few caveats should be taken notice of. First, the design will be a lot more complex. What if this 'temporary' storage file fills up? It would need immediate flushing. This can happen if a lot of data is written consecutively. How should a read request be handled? First, the block is looked up in the index, that would indicate a match to one or more permanent and temporary storage files. What if the index entry has to be removed? How does one determine whether a venti is 'idle enough' for handling a batch of writes? How to handle crashes and how to recover at startup? Does

the temporary storage also need an index? The complexity of the design will increase substantially with this feature.

The foreseen problem of having to write a 1500MB index file at shutdown turned out to be no problem at all. Initially, there was the idea to write the index sorted, so when loading them at the next start up memory allocation into the heads could be done efficiently without having any gaps. The current approach is to consider the scores to be spread over the heads evenly and just read them in the order they were written. The only remaining issue is the slow startup time, since the entire index has to be read into main memory before it can handle read or write requests.

7.2.4 Performance

Memventi has not been analysed exhaustively for performance. A few simple tests have been performed and the numbers can be found below. The most important performance characteristics are: reading data that is present (randomly and sequentially), writing data that is already present and writing data that is not yet present. Memventi does not cache any data file blocks, neither for reading nor writing. The operating system probably does have a disk block cache, a potentially large one for reading and probably a smaller one for writing (writes have to be flushed soon to guarantee they are permanently stored). Also, the operating system probably employs some type of read-ahead. Thus, reading sequentially should perform reasonable. Reading randomly probably performs less. Writing small amounts of blocks that are not yet present should be fast: no data has to be read from disk, and the writes are buffered in the write cache without having to wait for them to be on disk. Larger amounts of writes have to go to disk almost immediately and will be slower. Writing blocks sequentially that are already present (and were written sequentially before) should be about as fast as reading those blocks sequentially. The same disk reads are necessary. It is nice that the operating system delivers the disk optimisation for free.

The performance is not great in part because for each connection only a single operation is handled at a time. Normally, a venti client can send out up to 256 operations before needing a response. The venti server could in principle handle them at the same time. Currently, handling messages is done serially: a protocol message is read, then parsed, then processed (creating a response message), than unparsed and finally the response is written after which this cycle starts again. Having multiple disk operations pending allows the kernel to schedule the operations for optimal throughput. Perhaps a hand-crafted read-ahead would be of help as well.

When a venti is filled, it may have 3000 entries in a single head. For each lookup, these have to be read. The current code walks through them sequentially. The entries are not sorted, so a binary search is not possible. The amount of memory to search through is relatively small however, especially when compared to a disk access. Lots of bit operations are needed for reading the data buffer, it seems they do cost considerable amounts of CPU time.

Startup will always be slow, since the entire index has to be read. Unfortunately, it is slower than necessary. Reading an entry and parsing its content are done in lockstep. Read-ahead by the operating system may help here, but it seems to be insufficient. This results in a lot of waiting for incoming data. In theory, index reading should be possible at near-maximum sequential read

Type of test	Average throughput
randtest write pristine 2g	11.30 MB/s
randtest write duplicate sequential 2g	8.08 MB/s
randtest write duplicate permuted 2g	0.90 MB/s
randtest read sequential 2g	11.06 MB/s
randtest read permuted 2g	0.88 MB/s
sametest write 2g	17.98 MB/s
sametest read 2g	17.41 MB/s

Table 7.1: Performance results for memventi, in MB/s.

throughput of the disk. However, a simple test shows that reading an index of 3.75MB (for a 2GB data file) took almost 1.5 seconds.

Now for the tests. The index and data file each resided on a separate dedicated disk with an otherwise empty *ext2* file system, the index was on the 9GB SCSI drive, the data file on the 80GB IDE drive. The machine, disks and configuration are described in more detail in Appendix A. *Randtest* from Venti is used to write and read data. A slightly modified version called *sametest* writes and reads the same data block over and over. This shows the maximal throughput when the data is fully cached (after the first access the block remains in the operating systems disk block buffer cache) and is thus an indication of memventi protocol handling overhead. During the sametest runs, CPU usage was 100% indicating that the throughput may be higher on faster system. This is something worth investigating. For randtest, the amount of data written has been set to at least four times the size of available main memory, in order to eliminate influence by the buffer cache when reading data. For sequential access sweeps, this makes sure the buffer cache has overwritten previous sequential accesses. All tests have been executed three times, Table 7.1 presents the averages. Performance already is quite good compared to Venti. The commands that have been run and the results for all tests are given in Appendix D.

These results should be read along with these question and observations:

- Why is writing duplicate blocks slower than reading blocks? The only difference is that memventi reads a big protocol message and writes a small one in the case of duplicate writes, and reads a small protocol message and writes a big one in the case of reading.

- The performance for permuted blocks may seem bad, however it simply reflects disk performance for random accesses. Multiple disks used in parallel would improve performance (when using multiple processes). See Appendix A for the basic disk properties.

- Sequential memventi access is not close enough to sequential raw disk access, it should be possible to get much closer.

- The sametest tests were limited by CPU time available. Also, sametest used a considerable amount of CPU time as well. Ideally, these tests should be done on a multiprocessor machine. I did quickly run a test on a newer, dual core machine, it got to 40 MB/s throughput. The causes may be sought in bit manipulation, buffer copies or score calculating.

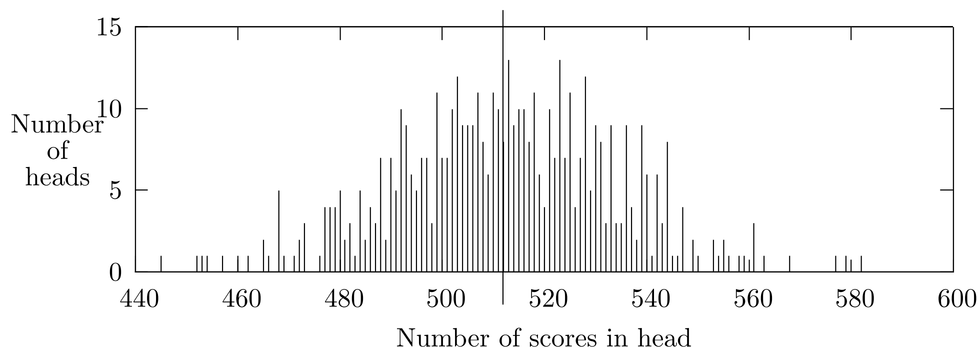

Figure 7.1: Head length histogram for a 2g memventi

- The values say nothing about the separate performance of the index and data disks and how they influence overall performance.

The following is printed by a memventi after having received a signal USR2. The memventi had been started freshly, had 2GB of random blocks in its data file and just served all those blocks for reading once. It shows by far most reads required one disk access, and only a few perhaps an additional one.

```
count   frequency:
      1  262028
      2  116
total memory lookups: 262144
```

The head length histogram has been plotted in Figure 7.1. There are not enough values to positively identify a normal distribution, however the histogram for 68g of 2k blocks does clearly show a normal distribution. It started with a single occurrence of a head with 440 entries and ended with a single occurrence with a head with 582 entries. The mean and median frequencies are both at the heads with 512 entries. These values closely match the behaviour predicted by *calc.py* which calculates an average of 513 blocks per head for a 2GB memventi. The write and read throughputs were 11.39 MB/s and 11.03 MB/s respectively.

A test with a larger memventi has also been done to stress test high main memory usage. 68GB of random blocks of 2k (a small block size to fill main memory) were written and read again. *calc.py* predicted a maximum memory usage of 317MB. The memory usage observed in top after writing and reading the 68GB was 313MB of resident memory. The read/write throughputs were 4.32 MB/s and 3.21 MB/s respectively, significantly lower than for the 2GB tests.

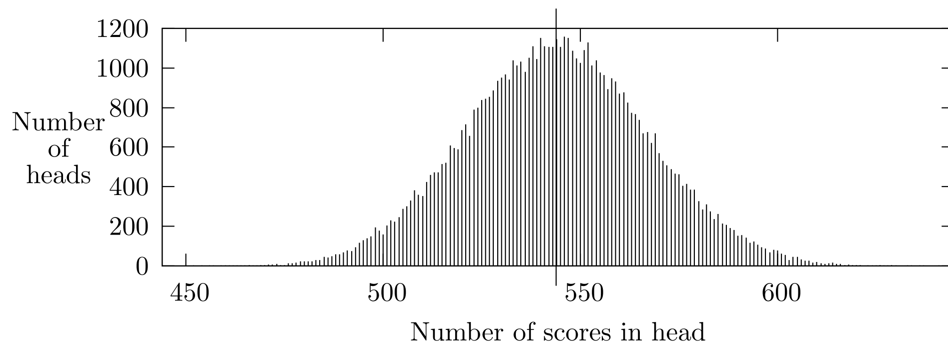

Figure 7.2: Head length histogram for a 68g memventi

This may have to do with the CPU usage of memventi when reading the 68GB of data: it was around 70%, while randtest took around 20% and no idle time left. This is worth investigating further.

The head length histogram can be seen in Figure 7.2, it has the expected normal distribution. The mean and median are both at 544 entries, shown in the figure as a line. The matching scores histogram after writing 68GB:

```
count   frequency:
      1  18093
      2  5
total memory lookups: 35651584
```

These values have the following meaning: 5 is the number of blocks that, when written, already had two matching score prefixes in the index. After adding, this means there are 5 unique score-prefixes that are shared by three scores each. 18093 is the number of blocks that, when written, already had a single matching score prefix in the index. This does not mean there are now 18093 duplicate score prefixes, the 5 triplet score-prefixes are also counted as single duplicate once. Thus, 18088 score-prefixes have two matches.

After having written the 68GB, the same data was read (without restarting memventi). When reading was done, another matching scores histogram was printed, see below. As can be seen, 15 scores had three hits in the index (5 prefixes are unique). The amount of scores with two hits for their score-prefixes are $2 * 18088 = 36176$ (the +5 comes from the histogram for writing, the values were not reset). The only curious thing is the total amount of memory lookups: it is larger than twice the number of blocks. Somehow 465 additional blocks have been looked up (and would have needed a single disk access), I do not know how this happened.

```
count   frequency:
      1  35633951
      2  36181
      3  15
total memory lookups: 71303633
```

These results show memventi is definitely able to store large amounts of data blocks and the current bottlenecks are sequential reads and probably CPU usage, likely due to reading non byte-aligned data containing score-parts and addresses.

7.2.5 Problems

Aside from the changes to the design already mention, a few other problems, some interesting, have been encountered while implementing *memventi*. For example, at first, the memory allocated for the index was not *mlock*-ed. Thus, it could easily be swapped out by the kernel it was running under. This could destroy performance. Thus, the memory was to be locked into physical pages from then on. Unfortunately, after refusing to startup, it turned out that OpenBSD/i386 (e.g. OpenBSD/amd64 does not suffer this deficiency) does not allow non-root users to lock memory, despite *rlimit* settings. Of course, *memventi* should be run as a low-privileged user for security reasons.

Besides needing an additional pointer in the chain structs, the pointers are not 4 bytes but 8 bytes on AMD64 systems. This increases the per-block memory overhead on those systems, but allows more memory than 4GB to be addressed. It might be possible to run memventi in 32-bits compatibility mode, but I did not succeed because the Linux distribution I used (Debian Etch) did not have a 32-bits pthread library. Note that even with 32-bit systems the full systems memory, larger than 2GB, may be used by running multiple memventi's with a load-balancer in front (redirecting based on a few bits of the score).

The next problem was solved quickly. At some early stage, memventi used more memory than it should based on the memory usage calculations. It turned out the chain struct was not 'packed'. The elements were not single-byte aligned, but eight-byte aligned on AMD64, turning the lean 17 byte struct into a 24 byte monster. Adding an inelegant but convenient `__attribute__((__packed__))` solved the problem.

Lastly, a typical case thread stack overflow was encountered. Incoming data (up to 56KB) was stored on the stack. Often, two of these would be allocated on the stack at once, overflowing the stack. The data has been moved to preallocated buffers.

7.2.6 Future work

Most aspects of *memventi* have been presented. A few ideas for improvement have been suggested, other ideas may not be worthwhile to pursue. Next is a short overview the options:

Startup time, the need to read entire index It would be nice to be able to resolve read/write requests without having the index in main memory. It does not have to be as fast as when the index is in memory. I do

not know of a scheme that would allow this other than keeping a sorted (possibly with gaps) index file as well. This would complicate the program considerably.

Binary search in chains For each lookup, all memory in a head is read (and bit-unpacked) sequentially up to the end. If the entries were sorted, a binary search wold help. Entries can be sorted within a chain or within the entire head. Sorting just chains is much easier to implement, but does not reduce CPU time as much.

More threads per connection Using more threads per connection, to handle multiple protocol messages at concurrently, would speed up memventi a lot. In this case, disk accesses are requested from the kernel concurrently and allows it to schedule them effectively. Together with read-ahead of data, these are the most promising optimisations.

Caching of data This is likely to be ineffective or even counter-productive. The kernel will probably do disk block caching at a more system-wide level, thus working better with other programs on the system, replacement strategies are likely to be highly optimised and would be hard to match.

Read-ahead of data In case the operating system does not adequately do read-ahead, this may improve performance slightly. The biggest performance gain can probably be gotten by introducing multiple threads (processes) per connection. A quick test has verified that speed can be improved noticeably, but the scheme used in that case would break badly when doing non-sequential access or multiple clients concurrently.

Verify more data on startup At the moment, only the block last referenced in the index file is checked against the data file. More blocks should be checked.

Tools for checking data and index file There are currently no tools to check the data file for consistency. E.g., verifying the score in the header with the score of the data. The index is not that important, it will be regenerated at startup (slowly though, could be done faster as well, perhaps by a separate program) when removed. For now, the only inconsistency that is assumed to occur is trailing blocks that are invalid, these have to be truncated manually.

Read index file faster By reading large chunks of the index file before they would be requested when reading entries one-by-one, reading of the index can probably approximate raw sequential disk (file system) read throughput. The current performance should be determined more thoroughly first though.

Lower memory usage A better scheme to reduce memory usage may be found. The type of a score is currently 8 bits. The current Venti clients, such as Vac and Fossil, do not use all bits, it seems 6 bits may be enough. Note that due to the way memventi allocates memory at startup, stopping and start memventi after it has written lots of data may lower memory consumption because fewer chains are used to store the entries. Memory can

also be spared by removing the data pointer in the chain struct and inlining the data buffer. Multiple chain structs with varying capacities allow almost equal memory savings as the current variable length buffers in the chains. The type of chain has to be represented in the structures somehow.

Compression of blocks *Memventi* does not compress blocks, it would not be hard to add. The average block size will be smaller when compressing data and more main memory will be needed to fill the same amount of disk space.

Store colliding scores entirely in memory The few scores that cause the collisions could be kept in memory entirely. Only a relatively small amount of scores would have to be kept in memory to prevent needing more than a single disk access. When writing a new score that causes a collision, the colliding score already present has to be read from disk and stored in a separate table in main memory. Thus, writing of a collision is not made any faster (writing of the third colliding is faster though, because their full scores are already in main memory). If this were to be implemented, an additional file would be needed to store this state between a shutdown and restart, causing additional complexity.

Heuristics for trying multiple hits When multiple index entries match a score, they all have to be read from disk in the worse case. In the optimal case, the first score read from disk is the one needed. There might be a good heuristic for reducing the average number of lookups required. Trying the data addresses in random order would give decent average performance. Other possible heuristics could be to try the data address closest the few address last read. Or to try the 'highest' (last written) address first. For now, they are read in the order returned by the lookup routine, which returns them in the order they were written, e.g. the block first written to the data file is checked first.

Raw disks Memventi stores its data and index file on normal disk partitions. This induces a bit of overhead and might destroy the assumption that sequentially written data is stored sequentially on disk and thus fast to read sequentially as well. Also, dependence on a file system disables memventi to be used as part of a root file system. The disadvantage of using raw disk partitions is that data has to be written in 512-byte-sized chunks (for current hardware) but the data is not 512-byte aligned. Also, meta-information such as file size has to be stored somehow. Last but not least, the operating systems buffer cache may be circumvented by this, destroying the high speed of short write bursts and repeated reading of the same data.

When taking these performance numbers and proven memory consumption into account, it seems it should be possible to run a 1TB memventi if it would have block compression. According to calc.py, this would need just under 2500MB of main memory.

The latest version of memventi, version 2 at the moment of writing, can be found at http://www.xs4all.nl/~mechiel/projects/memventi/.

7.2.7 Conclusion

Memventi has quite decent performance, is easy to deploy, simple in design and can therefore store data safely. There are some rough edges, such as slow startup time and relatively high CPU usage, these can be improved upon. The fact that it needs a file system for storing data means it is harder to use as a building block for a root file system than Venti. The other main difference with Venti is that it needs lots of main memory (but very little index disk space) and does not scale to the many terabytes of data storage Venti can handle.

For many current Venti installations, a memventi is an alternative worth considering.

Some of the ideas used in *memventi* are applicable to Venti. Index entries can be made much smaller, meaning many more index entries can be kept in main memory. It may be possible, though not easy, to replace Venti's main memory index cache by the scheme used by memventi. The caveat is that it would need a mechanism to replace entries in the index, requiring additional (memory-consuming) accounting. It would make Venti a bit more complex though, but on the other hand, some of the current optimisation schemes could be made superfluous.

Chapter 8

Conclusions

An introduction to Venti has been given: the venti protocol, how hash trees are used to store data in Venti, and basic Venti performance properties. The design of Venti has been analysed and Venti terminology explained, including the meaning of *scores*, the SHA-1 hash of data stored in Venti; *lump*, a data block and associated meta-data, such as type, score, and size; *arenas*, the on-disk data structure in which lumps are stored; *index*, the on-disk hash table used to look up the arena address of a score.

The description of optimisations present in Venti has resulted in better understanding of Venti and its behaviour on the various workloads: A large index cache is important to reduce the amount of random index disk accesses necessary. A disk block cache is important for holding dirty (unwritten) blocks and for often-accessed parts of the disk. The lump cache, keeping parsed lumps from disk in memory, is useful mostly in combination with lump read-ahead which prevents many expensive index lookups. It is not very useful for as a normal cache block. The disk block cache, and caches on the client side can do a far better job. The bloom filter is very effective in avoiding index lookups for non-duplicate lump writes, but at a relatively high memory cost. Reading the index entries from the arena directory of a often-accessed arena into the index cache is very effective for reducing index lookups for sequential reads, or reads with high locality, but may trash the index cache under bad (and luckily rare) conditions. Keeping dirty disk blocks in the cache and sorting them before writing a batch of them is helpful for increasing write throughput. Flushing index entries in buckets of 8MB may be a good idea for large index entry caches, but writing individual 8KB buckets is a better choice when only few dirty index entries can be stored in each 8MB quantum. This shows again that the various optimisation strategies are entangled and often cannot do without each other.

The analysis of Vac, Vacfs and Fossil, popular Venti clients, have resulted in insight in how Venti is used, and in which order the blocks (both data and pointer) of a hash tree are written to Venti, and stored on disk. This insight could be used in providing better throughput: handling different types of blocks differently, e.g. storing them in different locations and performing different read-ahead or caching strategies on them. But since the types do not have semantics known to Venti, using this information for optimisations is tricky and may be counter-productive when the assumed semantics are violated. It is therefore not recommended to continue in this direction. Another useful result of investigating

Vac and Fossil is that they make no attempt to read-ahead data or post multiple requests to Venti concurrently.

Magnetic disks are used as a building block for Venti. Venti depends on them for delivering performance. Measuring them has resulted in an upper bound of performance Venti can deliver and there is still room for improvement. Furthermore, some interesting disk or operating system scheduler traits have been uncovered: High sequential throughput can easily be achieved by using 128KB blocks, but 8KB blocks can as well but are better for random accesses; posting many random reads concurrently greatly increases throughput. Unfortunately, also some unexplained behaviour has been encountered: low write throughput for the tested IDE disk; very different behaviour of SCSI or IDE disks. The unexplained behaviour might very well be an artifact of the Linux disk scheduler or how it is used by the test program and Venti. The tests also show that it can be beneficial to have multiple processes operating concurrently on a disk. Taking a closer look into the effects of operating system disk handling will likely be worthwhile.

The original goal of the project, building a simulator to be able to test the performance of new optimisations strategies and to test various cache size paritionings, has not been fully achieved. Only a simplistics simulator has been implemented, one that is not accurte enough for testing Venti behavior as intended. There are several reasons for this, discussed below. The simulator build is able to generate trace files. Generating a trace file is done by starting a modified Venti which operates as normal but also writes all Venti operations and the resulting disk operations to the trace file. Thus, these trace files resemble real workloads. The simulator reads a trace files and returns the running time and other statistics as result. The running time is the most important performance characteristic. However, not enough of the Venti optimisations have been implemented to make the simulator accurate. For example, the bloom filter is not simulated, neither is index entry prefetching. Both are required for accurate simulations. Also, the disk model used is not accurate enough. More complications in implementing a Venti simulator have been encountered. The idea was that the code for the simulator could be integrated in the normal Venti code. This is possible—it has been done—but not in a clean enough way to have the two coexist without problems. Thus, the simulator will likely have to be maintained separately, making it harder to keep the two synchronised. Another key idea was that the Venti code could be used for the simulator as well. This is the case for some of them, but not for all, such as the bloom filter and index entry prefetching. Especially for the latter, simulating would involve quite some code that is unrelated to the implementation in Venti. This introduces complexity and further drives the code bases of Venti and the simulator apart. Having to implement optimisations specifically for the simulator also means the simulator has to be verified to be accurate after such a new feature: it may have bugs in its implementation, or it may simply not be an accurate approximation. Finally, a problem fundamental to a Venti simulator is that the behaviour of Venti clients, such as latencies between subsequent requests, have impact on Venti performance. And Venti behaviour has effect on the client behaviour. This could in theory reduce the accuracy of the simulator.

As there is no simulator, how can one determine the performance of new optimisations, or the optimal partitioning of memory for the caches? First, an analysis of the optimisations is important, which has been done for this report.

Second, a characterisation of the workload provides further insight in how Venti will behave. This expected behaviour can then be verified by running small scale tests. Cache partitioning remains hard and involves educated guessing. It is also very dependant on the workload and as such a single optimal partitioning does not exist. In general, based on an analysis of the optimisations and common workloads, it seems best to ensure the index cache is as big as possible. The lump cache should be small, just big enough to prevent lump read-ahead from trashing the cache. The disk block cache should be big enough to hold enough dirty, unflushed blocks for high write throughput. Caching of data in the disk block and lump caches to fulfil recurring reads is not useful and much better done in the client, which knows much more about usage patterns.

In the process of understanding Venti, its design considerations and implementation, the idea underlying *memventi* formed. It was not planned to be part of the project, but it has nonetheless. Memventi stores the *(score, type, disk address)*-tuple in main memory in a memory-efficient way. It stores only a small part of the score, only as many as needed to prevent too many collisions of the partial scores to occur, a probability of a collision of 0.001 or lower on a filled memventi seems acceptable. The parameters, the number of bits to use for addressing the data file and number of bits for the score, can be calculated using a separate program. The collisions occur more when the memventi data file is filled more, thus memventi performance degrades gracefully (but still good and according to design) when becoming filled. A 1TB memventi can be created using 2500MB of main memory, which is still a much higher storage capacity than most people need. Some workloads on memventi perform different on memventi than on the normal Venti. Pristine (non-duplicate) writes are about as fast on memventi and Venti, but for memventi this does not come at the cost of main memory for a bloom filter. Sequential duplicate writes are fast on Venti, needing only few disk accesses; random writes are slow for both programs. Random reads are fast on memventi, since are no index disks to consult. Sequential reads are fast on both systems, although memventi achieves this with substanstially fewer optimisations and thus less complexity. One of the obvious other differences regarding complexity is that memventi stores data blocks in a file on a normal file system. This allows memventi to make use of the file system (kernel) disk block cache without any cost, it eases administration (backup of the data file), and making the data file append-only protects against data corruption, again provided by the operating system for free. Of course, memventi is also reliable in that it always checks the score of the data read from disk, detecting disk failures. Overal, memventi performs well when compared to Venti and there is still room for improvement. Memventi also has a few drawbacks: it does not scale to high capacity (many terabytes); reading the index file into main memory at startup is slow for large installations; determining parameters with a separate programs could use some improvement; finally, many more minor improvements are waiting to be implemented, such as a binary search on entries in memory, instead of the current linear search.

Finally, enough work is waiting to be done. This master's project has spawned new ideas related to Venti, some already existed in the original Venti paper. For another, some of these ideas have already been implemented, are in progress of being implemented, or planned to be. The next chapter will elaborate on them.

Chapter 9

Future work

Venti, memventi or the venti clients have not been perfected, all of them can be improved upon. This chapter explains ideas for future work in four directions: improving memventi, finishing the Venti simulator, optimising existing Venti clients such as Fossil, and creating new Venti clients and tools.

Before continuing, a sidenote about continued work needs to be placed. During this project I have become very familiar with the venti protocol and implementation, as well as with the Venti clients and how they store data in Venti. This allowed me to spawn quite a few general ideas and specific designs for continued work. This is part of the reason why I took on the Google Summer of Code (GSoC) project to implement a memventi and vac-tools for Inferno [37] (an operating system that is a close relative to Plan 9 and can run both on bare hardware and as a user-space application on most common operating systems), in the programming language Limbo [38, 39] (a comfortable high-level programming language). All ideas I intend to implement for the GSoC project, or have already implemented, where thought up while working on this project. This is obvious for *ventisrv*, the Limbo version of memventi. But also holds for *vcache*, a venti cache and proxy, and for the Limbo version of Vac that splits blocks on content boundaries using rabin fingerprinting.

First, improving memventi, or its successor: *ventisrv*. Section 7.2.6 has a detailed list of improvements for memventi. Many of these suggestions have already been implemented in ventisrv. Only the features that seemed worthwhile have been implemented:

- Binary search in the chains, it seems to reduce CPU usage and improve throughput.
- Multiple threads per connection: ventisrv spawns up to 32 processes to do data lookups, these processes are used both for reading and for writing.
- Writes are queued in a queue for up to 32 entries.
- Compression of blocks. Unlike Venti, ventisrv uses the deflate compression algorithm (also used by gzip, it is the only algorithm available in Inferno). Compression throughput seems lower than whack, but no thorough comparison between deflate and whack has been made. However, there is a difference in implementation: Venti compresses a single block and stores it. Ventisrv takes multiple blocks and compresses it into one payload, a

special header specifying all scores is prepended. The compression ratio is significantly higher when compressing multiple blocks compared to compressing a single block at a time, this was as expected and is due to deflate having a large history buffer (16KB by default).
- Read index file faster. Big chunks are read and queued for a process that unpacks the entries and inserts them into the main memory index.
- More blocks of the index file and data file are verified at startup.

Some ideas have not yet, but probably will be implemented. Some of these were implemented in memventi, but not in a very clean way.

- Loading the index file into memory can still be improved. Ventisrv reads the index file efficiently, but inserting into memory is slow: all entries are inserted sorted, for which lots of reading and copying of memory occurs. It would be better to insert non-sorted, and sort once when some amount of data has been filled.
- Ventisrv has a slightly changed file format, mostly for the data file. First, the compressed blocks are in a new header. Second, the header has been extended with a *connection time*, recording the starting time of a connection, useful for removing all blocks written by a bad connection.
- Statistics about memory usage, hash table usage, collisions, etc. are not available in ventisrv. Exporting a simple file that returns human-readable and machine-parseable statistics may be useful.

Other features have not been implemented and probably never will, such as caching and read-ahead from the data file. It is better to let operating system do this: it also has more memory available to do so and adapts better to high loads. Next, raw disks will probably not be used to store data blocks on. It would require code for caching disk blocks and read-ahead. If a ventisrv really needs to be used as root file system, its data and index files can be served from a simple file system.

Second, finishing the Venti simulator. It may be worth considering a simulator implementation that is not embedded in the normal Venti code. This is a trade-off between having similar real Venti and simulated behaviour and the ease of writing the simulator. When implementing a full simulator, at least the following the following components should be simulated for realistic results:

- Multiple disks with concurrent accesses
- Queueing of writes
- Bloom filter
- Index entry prefetching

Possible designs for these components have been presented in Section 6.4.

Third, optimising existing Venti clients, such as Fossil and Vac. The ideas from this come from the analysis of Fossil and Vac described in Sections 3.2 and 3.1. Fossil never queues writes and always sends a *sync*-request after each write. This is not efficient, and for example interacts badly with compression in ventisrv. For better performance, Fossil should queue multiple write requests. But care must be taken to keep Fossils file system consistent. If a venti write

or sync fails, the data should still be present. A related idea is that Vacfs could queue reads to reduce latency. There is not much to queue except when multiple processes read from the vac file system concurrently, or when performing read-ahead. Read-ahead is useful only on higher-latency connections with reasonable bandwidth (e.g., an ADSL line). Read-ahead can be performed on the following type of blocks:

- The next (or next two or n) pointer block in the hash tree (at the deepest level), or the next pointer block at each depth.
- The next (or multiple) data blocks.
- The next vac directory (containing the 40-byte entries).
- The next vac directory meta information (containing direntries).

These same kinds of read-ahead might be useful for Vac when it writes only the changes relative to a previous archive. In this case, it reads a venti archive while writing a new one. The files to write to the new archive are usually on the local file system and thus fast to access. Using read-ahead can ensure reading the venti archive does not slow down writing the new archive. Reading-ahead can be implemented as a venti proxy. This proxy would speed up multiple readers in Vacfs as well because each read can be satisfied from the data in the read-ahead cache.

Fourth and last, new Venti-related tools or features can be implemented. Three were posted in the original Venti paper. First, a load-balancing proxy could be useful to stretch the storage capacity of memventi and ventisrv. Venti itself scales reasonably well to practically unlimited size, but memventi does not. However, one can simply switch to Venti when outgrowing memventi. The second suggestion was about security; a score acts as a capability, thus knowing a score implies knowing the data, and the other way around. There is currently no way to authenticate connections, and no encryption on them. Secured networks, authenticated and encrypted, can alleviate most of the need for authentication mechanisms. Third is splitting the blocks on content boundaries using rabin fingerprinting instead of at fixed file offsets. This reduces the number of unique blocks to store in Venti when a file is changed other then appending. An initial version of this has been implemented as part of the aforementioned GSoC project, and indeed increases block reuse.

As for the second item, memventi and ventisrv have a simple solution to part of the authentication problem. They can listen on multiple TCP ports, and handle them as either read-only or read/write. The first type disallows all writes and syncs. This solves the problem that without authentication, just anyone who can read from a Venti can also write to it, and fill it up.

For the GSoC project, a venti proxy server called *vcache* has been implemented that acts as a block cache. Vcache fulfils reads, writes and syncs by forwarding them to a remote (authoritative) server. Responses to reads are cached and kept in memory to quickly respond to future requests for that same block. A look-aside venti server can also be specified: vcache will write the data from read responses from the remote server to the look-aside venti for future lookups. Read requests coming in at the proxy are first forwarded to the look-aside venti server, and if it does not have the data, forwarded to the remote server. Writes and syncs always go to the remote server, but may optionally be

sent to the look-aside venti as well.

Other venti proxies can be devised as well. For example, a memventi or ventisrv can be used for the look-aside server in vcache. However, it will never remove data; the only way to clear up the storage used by the look-aside server is currently to start it anew with cleared state. A better way would be to use a server that reclaims storage for unpopular blocks. Or perhaps just round-robin-writes blocks, assuming blocks written long ago are likely to be unpopular now, and can simply be retrieved at little cost if they are still popular.

Another idea is a proxy that acts as a write buffer, buffering data on disk. This would allow programs like Fossil and Vac to continue with a next write quickly, even when the authoritative venti server is on the other side of a slow link. The write buffer proxy would let the blocks trickle to the remote server in the background. If the proxy would have problems writing the blocks (e.g. due to a system crash), the data is still stored on disk and can be flushed later on. A sync could represent a request to flush the write buffer to the proxies local storage, or to the remote server. The first is probably better, syncing to remote potentially takes a very long time. Note that such a proxy does have implications: if a client writes and syncs through a proxy, another program that does not go through the proxy but directly to the authoritative server would be told the data is not present. In short, as long as the write buffer proxy has not flushed the data to the authoritative venti server, all reads of the data must go through the proxy. A write buffer could also be placed in front of a venti server. The venti server behind the write buffer could do compression (which is relatively slow): the write buffer would buffer fast write bursts and flush the blocks to venti in idle time.

More special-purpose proxy servers can be devised.

Bibliography

[1] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey,CA, 2002.

[2] Plan 9 from User Space. *Venti Overview Manual Page*. Venti(7).

[3] Plan 9 from User Space. *Venti Administration Manual Page*. Venti(8).

[4] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[5] Jim McKie Sean Quinlan and Russ Cox. Fossil, an Archival File Server.

[6] Plan 9 from Bell Labs. *Fossil manual page*. Fossil(4).

[7] Plan 9 from Bell Labs. *Vac manual page*. Vac(1).

[8] Plan 9 from Bell Labs. *Vacfs manual page*. Vacfs(4).

[9] Plan 9 from User Space. *Vbackup Administration Manual Page*. Vbackup(8).

[10] Sean Quinlan. A Cached WORM File System. *Software — Practice and Experience*, 21(12):1289–1299, 1991.

[11] S. J. Mullender and A. S. Tanenbaum. Protection and resource control in distributed operating systems. *Computer Networks*, 8(5):421–432, October 1984.

[12] Plan 9 from Bell Labs. *Plan 9 File Protocol manual page*. Intro0(5).

[13] Russ Cox. Plan 9 from user space. `http://swtch.com/plan9port/`.

[14] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[15] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *CRYPTO*, pages 17–36, 2005.

[16] Plan 9 from User Space. *Venti Zero-Copy Packet Library Manual Page*. Venti-packet(3).

[17] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, 1994.

[18] Elizabeth A. M. Shriver, Arif Merchant, and John Wilkes. An Analytic Behavior Model for Disk Drives with Readahead Caches and Request Reordering. In *Measurement and Modeling of Computer Systems*, pages 182–191, 1998.

[19] John S. Bucy, Gregory R. Ganger, and et al. The DiskSim Simulation Environment Version 3.0 Reference Manual.

[20] J. Schindler and G. Ganger. Automated disk drive characterization, 1999.

[21] Intel and Seagate. Serial Ata Native Command Queuing, 2003.

[22] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 241–251, Nashville, TN, USA, 16–20 1994.

[23] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, CA, 1990. USENIX Association.

[24] Richard P. King. Disk arm movement in anticipation of future requests. *ACM Trans. Comput. Syst.*, 8(3):214–229, 1990.

[25] WD Caviar SE 80 EIDE Hard Drives (WD800JB) - Specifications.

[26] Intel. http://www.intel.com/design/flash/nand/datashts/311998.htm, js29f02g08aanb3, js29f04g08banb3, js29f08g08fanb3, 2006.

[27] Micron. NAND Flash Memory, mt29f4g08aaa, mt29f8g08baa, mt29f8g08daa, mt29f16g08faa, 2005.

[28] Mtron. MSD-S Series Product Specification.

[29] Steven W. Schlosser, John Linwood Griffin, David Nagle, and Gregory R. Ganger. Designing computer systems with MEMS-based storage. In *Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2000.

[30] Feng Wang, Bo Hong, Scott A. Brandt, and Darrell D. E. Long. Using MEMS-Based Storage to Boost Disk Performance. In *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 202–209, Washington, DC, USA, 2005. IEEE Computer Society.

[31] S. Schlosser and G. Ganger. MEMS-based storage devices and standard disk interfaces: A square peg in a round hole, 2004.

[32] S. Schlosser, J. Schindler, A. Ailamaki, and G. Ganger. Exposing and exploiting internal parallelism in MEMS-based storage, 2003.

[33] B. Hong, S. Brandt, D. Long, E. Miller, K. Glocer, and Z. Peterson. Zone-based shortest positioning time first scheduling for MEMS-based storage devices, 2003.

[34] M. Uysal, A. Merchant, and G. Alvarez. Using MEMS-based storage in disk arrays, 2003.

[35] Shaun Foley. Venti simulator. 2006.

[36] Sean Quinlan and Sean Dorward. Venti: a new approach to archival data storage, 2002.

[37] et. al. SM Dorward. The Inferno Operating System. *Bell Labs Technical Journal*, 2(1):5–18, Winter 1997.

[38] Dennis M. Ritchie. The Limbo Programming Language, 1995.

[39] Brian W. Kernighan. A Descent into Limbo, 2000.

Appendix A

Test setup

This chapter gives an overview of the machine that was used in the tests performed during this project, along with a few basic performance properties.

All tests have been performed on a single machine. It had 512MB of DDR memory, a 2.4 GHz *Intel Celeron* processor on an *Intel D845GVSR* motherboard and ran the Linux distribution *Debian Etch (stable)*. All unneeded background processes and daemons were disabled for the tests. A *Tekram* DC-390U2W SCSI controller was present in the system. For testing a *Compaq* U2W 9100MB, 10,000 rpm SCSI hard disk was used (Table A.1), as well as a *Western Digital* UDMA 80026MB, 7,200 rpm IDE hard disk (Table A.2).

Table A.3 presents a few basic performance statistics of the disks. These can be used to value Venti performance. The raw performance results come from the tests described in Appendix B.

The file system tests have been done informally. They disks were partitioned with a single *ext3* file system partition on the whole disk. *Ptio* was used, reading from a single file in the root of the partition. The only modification made to ptio for this test was that the file was not opened with O_DIRECT, thus not bypassing the disk block cache. Sequential operation on the file system is very fast. Random reads are slightly slower on the file system, opening the file with O_DIRECT increased performance to close to raw performance. Perhaps the kernels disk block cache is the bottleneck. Writes are slowed down even more, this too is alleviated by using O_DIRECT. Tests using 128KB blocks showed that especially sequential write performance can go up significantly, for the IDE file system to 45.0 MB/s. For Venti, the sequential operations and random reads are most important. There appear to be no discrepancies in performance between

Model	Compaq BD009222BB
Interface	SCSI U2W, 80 MB/sec
Cache size	Unknown
Rpm	10,000
Disk size	9,100 MB
Cache	Write-through
Other	Tagged command queuing, depth 16

Table A.1: Disk properties, *Compaq BD009222BB*.

Model	Western Digital WDC WD800JB
Interface	UDMA, 100 MB/second
Cache size	8192 KB
Rpm	7,200
Disk size	80,026 MB
Average Latency	4.20 ms (nominal)
Read Seek Time	8.9 ms
Write Seek Time	10.9 ms (average)
Track-To-Track Seek Time	2.0 ms (average)
Full Stroke Seek	21.0 ms (average)
Cache	Write-through

Table A.2: Disk properties, *Western Digital*, from [25].

	9,1GB SCSI	80GB IDE
Raw sequential read	26.1 MB/s	38.2 MB/s
Raw sequential write	22.6 MB/s	10.3 MB/s
Raw random read	0.96 MB/s	0.91 MB/s
Raw random write	0.90 MB/s	0.82 MB/s
File system sequential read	26.3 MB/s	52.9 MB/s
File system sequential write	19.1 MB/s	31.5 MB/s
File system random read	0.70 MB/s	0.78 MB/s
File system random write	0.67 MB/s	0.49 MB/s

Table A.3: Basic performance statistics in MB/s, measured using *ptio*, with 8KB blocks and a single process.

using a raw disk or the file system.

Appendix B

Disk tests

The tests that were conducted are described and the results explained in Section 4.3. The results are presented split by drive, one series of tables for the SCSI disk, and another for the IDE disk.

B.1 SCSI disk

The SCSI disk that has been tested is described in more detail in Appendix A. The Tables B.1, B.2, B.3 and B.4 present the results, split up by read/write and random/sequential.

B.2 IDE disk

The IDE disk that has been tested is described in more detail in Appendix A. The Tables B.5, B.6, B.7 and B.8 present the results, split up by read/write and random/sequential.

Type	Op	Blocksize	Procs	KCQ	Throughput	Per op	Mean	Stddev
rand	read	1,024	1	1,16,128	0.125	7.799	7.798	2.648
rand	read	1,024	4	1,16,128	0.122	7.976	31.890	84.353
rand	read	1,024	32	1,16,128	0.179	5.872	187.242	176.731
rand	read	8,192	1	1,16,128	0.960	8.139	8.138	2.636
rand	read	8,192	4	1,16,128	1.098	7.114	28.442	19.099
rand	read	8,192	32	0	1.101	7.099	226.404	203.803
rand	read	8,192	32	16	1.551	5.039	160.646	112.472
rand	read	8,192	32	128	1.575	4.962	158.175	83.459
rand	read	131,072	1	1,16,128	9.250	13.514	13.513	2.607
rand	read	131,072	4	1,16,128	10.175	12.285	49.123	33.910
rand	read	131,072	32	0	10.203	12.251	390.783	337.343
rand	read	131,072	32	16	12.346	10.125	322.892	220.014
rand	read	131,072	32	128	12.423	10.062	320.906	163.663

Table B.1: Results of tests on SCSI drive, random reads.

Type	Op	Blocksize	Procs	KCQ	Throughput	Per op	Mean	Stddev
rand	write	1,024	1	1,16,128	0.118	8.286	8.285	2.629
rand	write	1,024	4	1,16,128	0.132	7.398	29.581	20.855
rand	write	1,024	32	0	0.132	7.393	235.797	209.600
rand	write	1,024	32	16	0.182	5.365	171.080	119.944
rand	write	1,024	32	128	0.189	5.171	164.833	92.247
rand	write	8,192	1	1,16,128	0.901	8.675	8.674	2.680
rand	write	8,192	4	1,16,128	1.012	7.720	30.867	21.164
rand	write	8,192	32	0	1.015	7.696	245.407	215.416
rand	write	8,192	32	16	1.368	5.714	182.186	126.511
rand	write	8,192	32	128	1.409	5.545	176.783	98.755
rand	write	131,072	1	1,16,128	8.423	14.841	14.840	3.509
rand	write	131,072	4	1,16,128	9.031	13.841	55.349	37.337
rand	write	131,072	32	0	9.010	13.875	442.609	382.072
rand	write	131,072	32	16	10.499	11.906	379.739	258.930
rand	write	131,072	32	128	10.693	11.690	372.862	208.438

Table B.2: Results of tests on SCSI drive, random writes.

Type	Op	Blocksize	Procs	KCQ	Throughput	Per op	Mean	Stddev
seq	read	1,024	1	1,16,128	8.931	0.109	0.108	0.010
seq	read	1,024	4	1,16,128	0.597	1.637	6.545	3.302
seq	read	1,024	32	1,16,128	3.690	0.273	8.715	5.874
seq	read	8,192	1	1,16,128	26.101	0.299	0.298	0.180
seq	read	8,192	4	1,16,128	4.596	1.701	6.801	3.503
seq	read	8,192	32	0	21.384	0.365	11.689	2.255
seq	read	8,192	32	16	21.445	0.364	11.658	2.339
seq	read	8,192	32	128	21.801	0.359	11.480	2.805
seq	read	131,072	1	1,16,128	26.184	4.774	4.772	0.460
seq	read	131,072	4	1,16,128	21.869	5.717	22.861	36.173
seq	read	131,072	32	0	16.266	7.688	245.314	469.188
seq	read	131,072	32	16	18.370	6.855	218.796	223.405
seq	read	131,072	32	128	21.707	5.759	183.942	74.799

Table B.3: Results of tests on SCSI drive, sequential reads.

Type	Op	Blocksize	Procs	KCQ	Throughput	Per op	Mean	Stddev
seq	write	1,024	1	1,16,128	0.162	6.030	6.028	0.165
seq	write	1,024	4	1,16,128	0.943	1.036	4.141	1.706
seq	write	1,024	32	0	0.410	2.383	76.245	86.518
seq	write	1,024	32	16	0.382	2.558	81.849	51.759
seq	write	1,024	32	128	3.729	0.262	8.379	1.648
seq	write	8,192	1	1,16,128	1.235	6.328	6.327	0.564
seq	write	8,192	4	1,16,128	4.491	1.740	6.956	1.505
seq	write	8,192	32	0	2.634	2.966	94.911	108.463
seq	write	8,192	32	16	3.709	2.107	67.398	52.188
seq	write	8,192	32	128	19.400	0.403	12.886	3.534
seq	write	131,072	1	1,16,128	10.901	11.467	11.466	2.016
seq	write	131,072	4	1,16,128	22.631	5.523	22.090	3.286
seq	write	131,072	32	0	15.857	7.883	251.783	220.593
seq	write	131,072	32	16	16.446	7.601	242.778	173.348
seq	write	131,072	32	128	22.607	5.529	176.868	37.822

Table B.4: Results of tests on SCSI drive, sequential writes.

Type	Op	Blocksize	Procs	KCQ	DRA	Throughput	Per op	Mean	Stddev
rand	read	1,024	1	128	off	0.116	8.42	8.42	2.95
rand	read	1,024	1	128	on	0.104	9.36	9.35	3.48
rand	read	1,024	4	128	off	0.101	9.68	38.70	93.78
rand	read	1,024	4	128	on	0.093	10.54	42.16	97.42
rand	read	1,024	32	128	off	0.107	9.16	291.69	913.96
rand	read	1,024	32	128	on	0.094	10.43	332.45	973.08
rand	read	8,192	1	128	off	0.913	8.56	8.56	2.99
rand	read	8,192	1	128	on	0.821	9.52	9.52	3.55
rand	read	8,192	4	128	off	0.797	9.80	39.19	94.37
rand	read	8,192	4	128	on	0.736	10.63	42.49	97.78
rand	read	8,192	32	128	off	0.852	9.18	292.35	914.67
rand	read	8,192	32	128	on	0.744	10.50	334.86	976.23
rand	read	131,072	1	128	off	11.297	11.07	11.06	3.11
rand	read	131,072	1	128	on	10.476	11.93	11.93	3.66
rand	read	131,072	4	128	off	9.943	12.57	50.26	105.70
rand	read	131,072	4	128	on	9.392	13.31	53.22	108.47
rand	read	131,072	32	128	off	10.584	11.81	376.45	1,032.55
rand	read	131,072	32	128	on	9.575	13.06	416.27	1,082.32

Table B.5: Results of tests on IDE drive, random reads.

Type	Op	Blocksize	Procs	KCQ	DRA	Throughput	Per op	Mean	Stddev
rand	write	1,024	1	1,16,128	off	0.103	9.45	9.45	2.95
rand	write	1,024	4	1,16,128	off	0.104	9.42	37.67	6.33
rand	write	1,024	32	0	off	0.084	11.65	371.74	320.47
rand	write	1,024	32	16	off	0.090	10.83	345.69	182.91
rand	write	1,024	32	128	off	0.104	9.36	298.54	23.43
rand	write	8,192	1	1,16,128	off	0.817	9.57	9.57	2.98
rand	write	8,192	4	1,16,128	off	0.822	9.50	38.00	6.36
rand	write	8,192	32	0	off	0.662	11.80	376.59	324.72
rand	write	8,192	32	16	off	0.708	11.03	352.06	185.50
rand	write	8,192	32	128	off	0.823	9.50	302.89	23.33
rand	write	131,072	1	1,16,128	off	9.223	13.55	13.55	3.06
rand	write	131,072	4	1,16,128	off	9.226	13.55	54.17	6.91
rand	write	131,072	32	0	off	7.904	15.81	504.52	434.48
rand	write	131,072	32	16	off	7.699	16.24	517.91	274.08
rand	write	131,072	32	128	off	9.227	13.55	432.01	29.27

Table B.6: Results of tests on IDE drive, random writes.

Type	Op	Blocksize	Procs	KCQ	DRA	Throughput	Per op	Mean	Stddev
seq	read	1,024	1	128	off	0.117	8.33	8.32	0.27
seq	read	1,024	1	128	on	7.926	0.12	0.12	0.10
seq	read	1,024	4	128	off	0.235	4.16	16.63	6.43
seq	read	1,024	4	128	on	0.595	1.64	6.57	3.32
seq	read	1,024	32	128	off	1.500	0.65	20.83	4.89
seq	read	1,024	32	128	on	3.683	0.27	8.49	2.29
seq	read	8,192	1	128	off	0.922	8.48	8.48	0.19
seq	read	8,192	1	128	on	38.168	0.21	0.20	0.08
seq	read	8,192	4	128	off	1.830	4.27	17.08	6.55
seq	read	8,192	4	128	on	4.463	1.75	7.00	3.73
seq	read	8,192	32	128	off	6.843	1.14	36.53	13.24
seq	read	8,192	32	128	on	10.48	0.75	23.92	8.83
seq	read	131,072	1	128	off	11.503	10.87	10.87	0.56
seq	read	131,072	1	128	on	48.999	2.55	2.55	0.56
seq	read	131,072	4	128	off	12.169	10.27	41.08	94.54
seq	read	131,072	4	128	on	44.921	2.78	11.12	50.86
seq	read	131,072	32	128	off	11.967	10.45	333.65	986.96
seq	read	131,072	32	128	on	43.859	2.85	90.68	512.60

Table B.7: Results of tests on IDE drive, sequential reads.

Type	Op	Blocksize	Procs	KCQ	DRA	Throughput	Per op	Mean	Stddev
seq	write	1,024	1	1,16,128	off	0.117	8.33	8.33	0.26
seq	write	1,024	4	1,16,128	off	0.234	4.18	16.70	0.38
seq	write	1,024	32	1,16,128	off	1.809	0.54	17.28	0.39
seq	write	8,192	1	1,16,128	off	0.921	8.48	8.48	0.25
seq	write	8,192	4	1,16,128	off	1.809	4.32	17.28	0.59
seq	write	8,192	32	128	off	10.303	0.76	24.36	3.74
seq	write	131,072	1	1,16,128	off	11.488	10.88	10.88	0.66
seq	write	131,072	4	1,16,128	off	11.485	10.88	43.53	0.92
seq	write	131,072	32	0	off	17.753	7.04	224.96	210.69
seq	write	131,072	32	16	off	21.422	5.84	186.33	126.03
seq	write	131,072	32	128	off	11.490	10.88	347.47	12.13

Table B.8: Results of tests on IDE drive, sequential writes.

Appendix C

Basic Venti tests

For measuring Venti performance, the following commands have been executed, each three times:

- Writing sequentially, on an empty Venti:
  `./o.randtest -h localhost -b 8k -n 2g -w`
- Reading sequentially:
  `./o.randtest -h localhost -b 8k -n 2g -r`
- Reading randomly:
  `./o.randtest -h localhost -b 8k -n 2g -r -P`
- Writing sequentially, already present, on a Venti filled by the same command:
  `./o.randtest -h localhost -b 8k -n 2g -w`
- Writing randomly, already present, on a filled Venti:
  `./o.randtest -h localhost -b 8k -n 2g -w -P`
- Writing same block over and over, on an empty Venti:
  `./o.sametest -h localhost -b 8k -n 2g -w`
- Reading same block over and over, on an otherwise empty Venti:
  `./o.sametest -h localhost -b 8k -n 2g -r`

Both the test programs and Venti were running on the same machine. The tests were conducted using the machine specified in Appendix A. The entire 9.1GB SCSI drive was used as index disk. The entire 80GB IDE drive was used as arena disk. A bloom filter of 64MB was chosen by *fmtbloom*. Of the 512MB of main memory, 256MB was given to the index cache, 32MB to the disk block cache and 16MB to the lump cache. All runs of the results are shown in Table C.1. The sequential write of 68GB was only done once since it took a lot of time. Sequential read was stopped at 28GB because of a Venti read error.

C.1 sametest

Sametest is a copy of *randtest* that comes with Venti, with a minor modification. Instead of writing blocks with random contents, it writes the same block over and over. The following *diff* creates sametest from randtest:

```
--- randtest.c  2005-07-12 17:23:35.000000000 +0200
```

Type of test	Run 1	Run 2	Run 3	Average
randtest write pristine 2g	6.00	6.28	7.22	6.50
randtest write duplicate sequential 2g	18.56	19.10	18.39	18.68
randtest write duplicate permuted 2g	17.83	18.15	18.14	18.04
randtest read sequential 2g	8.60	8.61	8.60	8.60
randtest read permuted 2g	0.88	0.88	0.88	0.88
sametest write 2g	18.99	18.90	18.90	18.93
sametest read 2g	21.13	20.40	20.47	20.67
randtest write sequential 68g	2.99			2.99
randtest read sequential 28g	8.38	8.42		8.40

Table C.1: Results of basic Venti performance tests, in MB/s.

```
+++ sametest.c  2007-02-13 19:58:09.000000000 +0100
@@ -97,6 +97,18 @@
        packets = totalbytes/blocksize;
        if(maxpackets == 0)
                maxpackets = packets;
+
+       memmove(buf, template, blocksize);
+       for(i=0; i<packets && i<maxpackets; i++){
+               if(c){
+                       sendp(c, buf);
+                       buf = vtmalloc(blocksize);
+               }else
+                       (*fn)(buf, buf2);
+               cur += blocksize;
+       }
+       return;
+
        order = vtmalloc(packets*sizeof order[0]);
        for(i=0; i<packets; i++)
                order[i] = i;
```

Appendix D

Memventi tests

The memventi test is the same as the basic Venti test from the previous chapter: the commands ran for the tests are identical. The tests resulted in the runs and averages in Table D.1, the values are throughput in MB/s as returned by *randtest* and *sametest*.

During the test of writing already present block sequentially, memventi took around 55% CPU time, randtest used about 20%. While reading the blocks permuted, CPU usage was around 6% and 3% respectively, indicating high CPU overhead due to lookups since in the latter lots of time is spent waiting for data to come in. These numbers were determined by watching top and should be taken with a grain of salt.

Test	Run 1	Run 2	Run 3	Average
randtest write pristine 2g	11.34	11.29	11.29	11.30
randtest write duplicate sequential 2g	8.07	8.08	8.08	8.08
randtest write duplicate permuted 2g	0.91	0.91	0.89	0.90
randtest read sequential 2g	11.09	11.08	11.02	11.06
randtest read permuted 2g	0.89	0.88	0.87	0.88
sametest write 2g	18.37	18.01	17.58	17.98
sametest read 2g	17.51	17.56	17.18	17.41

Table D.1: Results of memventi performance tests, in MB/s.

Appendix E

Compact flash tests

Compact flash memory performance has been measurement. The tests were performed on a 1GB compact flash card, a *SanDisk Extreme III 1.0*GB. The only performance characteristic documented is a maximum sequential read/write speed of 20 MB/s. Random access performance is not mentioned in the specifications. The following two sections present the results of tests in different flash memory adaptors.

E.1 IDE to flash adaptor

The compact flash card was tested using a no-name compact flash to IDE converter that is able to hold two compact flash cards. Unfortunately, the adaptor does not support DMA, only the IDE PIO modes and thus will not have a high (sequential) throughput. Therefore, it has only been used to test random accesses, the results of which can be found in Tabe E.1.

E.2 USB flash memory card reader

A USB 2.0 compact flash reader, a *SanDisk ImageMate*, was used to test both random and sequential access. The tests are mostly the same as the tests on the magnetic disks: Many runs of the *ptio* were conducted and the results analysed. The parameters were the *operation* (read or write), *blocksize* (512 or 8K bytes), *number of processes* (1, 4, 32), *kernel command queue length* (1, 16, 64), *type of* I/O (random, sequential) and *kernel queue read-ahead* (0KB or 128KB). For the random access tests, the *kernel queue read-ahead* was set to 0KB. The kernel I/O scheduler was the default CFQ, complete fair queueing, which supposedly guarantees that all processes receive a fair share of I/O operations. The results can be found in Table E.2.

Op	Blocksize	Procs	KCQ	Throughput	Per op	Mean	Stddev
read	512	1	1,16,64	1.171	0.417	0.416	0.047
read	512	4	1,16,64	1.170	0.417	1.669	18.772
read	512	32	1	0.844	0.580	18.547	101.882
read	512	32	16	0.994	0.491	15.711	84.466
read	512	32	64	1.159	0.421	13.472	199.511
read	8192	1	1,16,64	2.665	2.932	2.930	0.063
read	8192	4	1,16,64	2.665	2.932	11.726	49.574
read	8192	32	1	2.223	3.514	112.402	279.999
read	8192	32	16	2.224	3.513	112.368	277.553
read	8192	32	64	2.662	2.935	93.874	500.859
write	512	1	1,16,64	0.019	25.513	25.512	3.8197
write	512	4	1,16,64	0.019	25.253	101.009	29.601
write	512	32	1	0.020	24.968	798.717	687.721
write	512	32	16	0.020	24.738	791.379	438.425
write	512	32	64	0.020	24.419	781.171	429.440
write	8192	1	1,16,64	0.311	25.131	25.130	5.334
write	8192	4	1,16,64	0.314	24.854	98.955	30.247
write	8192	32	1	0.316	24.738	790.887	656.446
write	8192	32	16	0.318	24.602	786.552	425.414
write	8192	32	64	0.321	24.318	777.470	428.912

Table E.1: Results of the compact flash card tests with IDE adaptor; throughput in MB/s, latency in ms.

Type	Op	Blocksize	Procs	KRA	KCQ	Throughput	Per op	Mean	Stddev
rand	read	512	1	0	1,16,64	0.777	0.629	0.627	0.043
rand	read	512	4	0	1,16,64	0.774	0.631	2.523	24.410
rand	read	512	32	0	1	0.594	0.824	24.734	131.442
rand	read	512	32	0	16	0.596	0.821	26.019	134.849
rand	read	512	32	0	64	0.771	0.634	20.256	248.895
rand	read	8k	1	0	1,16,64	5.987	1.305	1.303	0.067
rand	read	8k	4	0	1,16,64	5.985	1.305	5.219	34.828
rand	read	8k	32	0	1	4.750	1.645	51.103	189.100
rand	read	8k	32	0	16	4.715	1.657	52.719	193.416
rand	read	8k	32	0	64	5.978	1.307	41.789	354.345
rand	write	512	1	0	1,16,64	0.021	23.580	23.580	3.566
rand	write	512	4	0	1,16,64	0.021	23.471	93.880	17.711
rand	write	512	32	0	1	0.021	23.445	749.793	644.731
rand	write	512	32	0	16	0.021	23.317	745.710	408.746
rand	write	512	32	0	64	0.021	23.140	740.031	394.015
rand	write	8k	1	0	1,16,64	0.325	24.053	24.051	5.206
rand	write	8k	4	0	1,16,64	0.325	24.017	96.062	19.563
rand	write	8k	32	0	1	0.326	24.006	767.708	660.488
rand	write	8k	32	0	16	0.327	23.910	764.665	417.661
rand	write	8k	32	0	64	0.330	23.700	757.940	402.586
seq	read	512	1	0	1,16,64	0.977	0.500	0.498	0.004
seq	read	512	1	128	1,16,64	0.977	0.500	0.498	0.004
seq	read	512	4	0	1,16,64	0.274	1.783	7.131	3.640
seq	read	512	4	128	1,16,64	0.275	1.778	7.110	3.620
seq	read	512	32	0	1	1.545	0.321	10.272	2.293
seq	read	512	32	0	16	1.758	0.284	9.084	1.983
seq	read	512	32	0	64	1.526	0.324	10.368	2.316
seq	read	512	32	128	1	1.770	0.282	9.004	1.989
seq	read	512	32	128	16	1.512	0.327	10.450	2.334
seq	read	512	32	128	64	1.780	0.280	8.933	1.982
seq	read	8k	1	128	1,16,64	6.257	1.249	1.247	0.016
seq	read	8k	4	128	1,16,64	3.186	2.453	9.809	4.397
seq	read	8k	32	0	1	7.744	1.009	32.272	11.885
seq	read	8k	32	0	16	7.616	1.026	32.818	12.050
seq	read	8k	32	0	64	7.765	1.006	32.185	11.909
seq	read	8k	32	128	1	7.739	1.010	32.297	11.784
seq	read	8k	32	128	16	7.600	1.028	32.886	12.112
seq	read	8k	32	128	64	7.758	1.007	32.217	11.641
seq	write	512	1	128	1,16,64	0.935	0.522	0.520	0.265
seq	write	512	4	128	1,16,64	1.231	0.398	1.590	0.452
seq	write	512	32	128	1,16,64	5.842	0.084	2.662	0.429
seq	write	8k	1	128	1,16,64	6.669	1.171	1.170	0.397
seq	write	8k	4	128	1,16,64	7.527	1.038	4.150	0.652
seq	write	8k	32	0	1	10.932	0.715	22.814	5.244
seq	write	8k	32	0	16	11.092	0.705	22.485	0.931
seq	write	8k	32	0	64	11.095	0.704	22.489	0.955
seq	write	8k	32	128	1	10.653	0.734	23.417	8.176
seq	write	8k	32	128	16	11.104	0.704	22.459	0.859
seq	write	8k	32	128	64	11.108	0.703	22.461	0.869

Table E.2: Results of the compact flash card tests with USB card reader; throughput in MB/s, latency in ms.

Appendix F

Technical documentation

This chapter provides some insight into the inner workings of Venti. This is done by giving an overview of the processes that a Venti instance consists of, along with a description of how they interact. A description of the types used in the code and a description of the files that the Venti server consists of.

F.1 Pseudo code for handling venti operations

Since the heart of Venti is handling read and write requests, a pseudo code/natural language description-hybrid for handling them as used by Venti is given below.

Pseudo code for handling a write request:

```
threadmain() calls ventiserver() calls writelump() which does:
  lookuplump() to fetch the data from cache (see the pseudo-code for read
        on lookuplump() works).
  if data happened to be in cache (thus score is already present):
    if data read is equal to data to be written:
      return success
    else (data differs):
      a hash collision has been found!
      return error
  else (data was not in cache):
    if queuewrite is set:
      queuewrite():
        put block in queue (wait on condition "queue is full"
          when queue is full)
        wake up thread waiting on condition "queue is empty"
        return success (block will be written by queueproc
          using writeqlump())
    else (queuewrite not set, write immediately):
      writeqlump():
        lookupscore() to determine if score is present in venti.
          if score is present:
            assume it's the same (no hash collision check)
            return success
          else (score is not present, we need to write the data):
            storeclump(), writeiclump(), writeaclump():
              getdblock()
              putdblock()
              writeclumpinfo()
            insertscore() (insert score into cache and possibly
              flush when too many entries dirty)
            insertlump() (insert lump into cache)
            return success
```

Pseudo code for handling a read request:

```
threadmain() calls ventiserver() calls readlump() which does:
   lookuplump() to fetch the data from lump cache:
   if score in lump cache:
      return data
   else (score not in lump cache):
      lookupscore():
         look for score in index cache
            if score not in icache:
               loadientry() to read ientry from disk:
                  if score not in inbloomfilter():
                     return failure, no such score in venti
                  loadibucket(), okibucket(), bucklook()
            possibly loadarenaclumps() to read adjacent scores
            we now have the ientry and thus the address
   readilump() to lookup arena and get lump:
      loadclump() read clump from arena and parse it:
         readarena() to read all clump blocks from arena:
            getdblock(), _getdblock():
               if not in cache:
                  rareadpart(), readpart()
               return data
```

Appendix G

Tools

G.1 ptio.c

```c
#define _BSD_SOURCE
#define _XOPEN_SOURCE 500

/* glibc has 32 bit offset by default */
#define _FILE_OFFSET_BITS 64


#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>

#include <ctype.h>
#include <err.h>
#include <fcntl.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <time.h>
#include <unistd.h>

#include <pthread.h>

typedef long long vlong;
typedef unsigned long long uvlong;
typedef unsigned char uchar;
typedef ulong uintptr;

#define nil NULL
#define O_DIRECT 040000 /* glibc doesn't want to give us this define... */


pthread_mutex_t offlock;
vlong *offsets;
int noffsets;
int ioffset;
int *times;
int ntimes;

```

```
int isread;
char *file;
int blocksize;
int nproc;


void *
roundup(void *p, ulong n)
{
        return (void*)(((uintptr)p+n-1) & ~(uintptr)(n-1));                      50
}

void *
emallocround(ulong n)
{
    void *p;
    p = malloc(2*n);
    if(p == nil)
            err(1, "emallocround");
    p = roundup(p, n);                                                           60
    return p;
}


char *
deconv(uvlong v, char *s)
{
    char *suffix[] = {"", "k", "m", "g", "t"};
    int i;
                                                                                 70
    i = 0;
    while(v > 1024 && (v & (1024-1)) == 0) {
        v /= 1024;
        i++;
    }
    sprintf(s, "%llu%s", v, suffix[i]);
    return s;
}

                                                                                 80
uvlong
conv(char *s)
{
    uvlong r;

    r = strtoull(s, &s, 10);
    switch(tolower(*s)){
    case 't':
        r *= 1024;
    case 'g':                                                                    90
        r *= 1024;
    case 'm':
        r *= 1024;
    case 'k':
        r *= 1024;
        s++;
        break;
    case '\0':
        break;
    default:                                                                     100
        return ~0ULL;
    }
```

```
    if(*s != 0)
        return ~0ULL;
    return r;
}


vlong                                                          110
microsec(void)
{
    struct timeval tv;

    gettimeofday(&tv, nil);
    return (vlong)tv.tv_sec*1000000 + (vlong)tv.tv_usec;
}


void                                                           120
readoffsets(void)
{
    FILE *f;
    char line[1024];
    char *l;
    vlong *p;
    int n;
    uvlong v;
    int len;
                                                               130
    f = fdopen(0, "r");
    if(f == nil)
        err(1, "fdopen");

    n = 0;
    p = nil;
    while((l = fgets(line, sizeof line, f))) {
        p = realloc(p, sizeof p[0] * (n+1));
        len = strlen(l);
        if(l[len-1] == '\n')                                   140
            l[--len] = '\0';
        v = conv(l);
        if(v == ~0ULL)
            errx(1, "reading %dth offset", n);
        p[n++] = blocksize * ((vlong)v/blocksize);
    }
    //fclose(f);

    offsets = p;
    noffsets = n;                                              150
    ioffset = 0;
    if(noffsets < 2)
        errx(1, "refusing to read fewer than two blocks");
}


int
diffcmp(const void *p1, const void *p2)
{
    vlong v1, v2;                                              160

    v1 = *(vlong *)p1;
    v2 = *(vlong *)p2;
    if(v1 == v2)
```

```
		return 0;
	if(v1 < v2)
		return -1;
	return 1;
}
                                                                    170

void
usage(void)
{
	fprintf(stderr, "usage: ptio [read | write] file blocksize nproc\n");
	exit(1);
}


void *                                                              180
reader(void *p)
{
	vlong off;
	uchar *mem;
	int n;
	int ioff;
	int fd;
	vlong start, stop;

	mem = emallocround(blocksize);                                      190

	fd = open(file, O_RDWR|O_DIRECT);
	if(fd < 0)
		err(1, "open %s", file);

	start = stop = -1;
	for(;;) {
		off = -1;

		pthread_mutex_lock(&offlock);                                   200
		ioff = ioffset;
		if(ioffset < noffsets)
			off = offsets[ioffset++];
		if(start != -1)
			times[ntimes++] = stop-start;
		pthread_mutex_unlock(&offlock);

		if(off == -1)
			pthread_exit(nil);
                                                                    210
		start = microsec();
		if(isread)
			n = pread(fd, mem, blocksize, (off_t)off);
		else
			n = pwrite(fd, mem, blocksize, (off_t)off);
		stop = microsec();
		if(n != blocksize)
			err(1, "did not read enough data (n=%d)", n);
	}
}                                                                   220


int
main(int argc, char *argv[])
{
	char tmp1[64];
```

```
int i;
vlong total;
pthread_t *threads;
vlong start, stop;                                                   230
char *op;
double mean, stddev;

if(argc != 5)
    usage();

op = argv[1];
if(strcmp(op, "read") == 0)
    isread = 1;
else if (strcmp(op, "write") == 0)                                   240
    isread = 0;
else
    usage();
file = argv[2];
blocksize = conv(argv[3]);
nproc = atoi(argv[4]);

readoffsets();

printf("op: %s\n", isread ? "read" : "write");                       250
printf("blocks: %d\n", noffsets);
printf("blocksize: %s bytes\n", deconv(blocksize, tmp1));
printf("nprocs: %d processes/threads\n", nproc);

threads = malloc(sizeof threads[0] * nproc);
if(threads == nil)
    err(1, "malloc");

times = malloc(sizeof times[0] * noffsets);
if(times == nil)                                                     260
    err(1, "malloc");
ntimes = 0;

pthread_mutex_init(&offlock, nil);

start = microsec();
for(i = 0; i < nproc; i++) {
    pthread_create(&threads[i], nil, reader, nil);
}
                                                                     270
for(i = 0; i < nproc; i++) {
    pthread_join(threads[i], nil);
}
stop = microsec();

total = stop - start;
printf("total time: %llu ms\n", total/1000);
printf("throughput: %.3f MB/sec\n", ((double)noffsets*blocksize*1000000.0/total)/(1024*1024));
printf("average disk seek time: %.3f ms\n", (double)total/1000.0/noffsets);
                                                                     280
total = 0;
for(i = 0; i < ntimes; i++)
    total += times[i];
mean = (double)total / ntimes;
printf("mean wait time per i/o: %.3f ms\n", mean/1000.0);

total = 0;
for(i = 0; i < ntimes; i++)
```

```
        total += pow(((double)times[i] − mean), 2.0);
    stddev = sqrt((double)total/ntimes);                                    290
    printf("standard deviation wait time: %.3f ms\n", stddev/1000.0);

    exit(0);
}
```

G.2 test-sha1.c

```
#include <u.h>
#include <libc.h>
#include <mp.h>
#include <libsec.h>


static int blocksize = 8*1024;
static int n = 10000;
static int align = 32;
                                                                            10

void *
emallocroundalign(int len, int a, void **pp)
{
    void *p;
    p = malloc(len+2*32);
    if(p == nil)
        sysfatal("malloc");
    *pp = p;
    p = (void*)(((uintptr)p+32)&~32);                                        20
    return p+a;
}


static void
usage(void)
{
    fprint(2, "usage: test-sha1 [-b blocksize] [-n count] [-a align]\n");
    exits("usage");
}                                                                           30


void
main(int argc, char *argv[])
{
    vlong start, stop;
    uchar *datap[1024];
    uchar *data;
    uchar score[20];
    int i;                                                                  40
    vlong total;
    void *p;

    ARGBEGIN {
    case 'b':
        blocksize = atoi(EARGF(usage()));
        break;
    case 'n':
        n = atoi(EARGF(usage()));
        break;                                                              50
```

```
	case 'a':
		align = atoi(EARGF(usage()));
		break;
	default:
		usage();
	}ARGEND

	if(argc != 0)
		usage();

	for(i = 0; i < 1024; i++)
		datap[i] = emallocroundalign(blocksize, align, &p);

	total = 0;
	srand(time(nil));
	for(i = 0; i < n; i++) {
		data = datap[i % 1024];
		prng(data, blocksize);
		start = nsec();
		sha1(data, blocksize, score, nil);
		stop = nsec();
		fprint(2, "%llud\n", (stop-start));
		total += stop-start;
	}
	print("total: %llud ms\n", total/1000000ULL);
	print("throughput: %.3f MB/sec\n", (double)n*blocksize*1000000000.0/(total*1024*1024));
	print("per operation: %llud nsec, %llud microsec\n", total/n, (total/n)/1000ULL);
	exits(nil);
}
```

G.3 test-whack.c

```
#include <u.h>
#include <libc.h>
#include <mp.h>
#include <libsec.h>

#include "whack.h"


static int count = 10000;
static int blocksize = 8*1024;
static int datatype = 0;


void
getdata(uchar *buf, int len)
{
	static int fd = -1;
	static int filesize = -1;
	static int offset = -1;
	Dir *d;
	static int i = 0;
	char *filename;

	switch(datatype) {
	case 0:
		prng(buf, len);
		break;
```

```
	case 1:
	case 2:
		if(fd == -1) {                                                          30
			if(datatype == 1)
				filename = "/usr/bin/gs-gpl";
			else
				filename = "/usr/share/perl/5.8.4/unicore/UnicodeData.txt";
			fd = open(filename, O_RDONLY);
			d = dirfstat(fd);
			filesize = d->length;
			offset = i;
		}
		if(offset+len >= filesize)                                              40
			offset = i++;
		if(pread(fd, buf, len, offset) != len)
			sysfatal("read did not suffice");
		offset += len;
		break;
	default:
		sysfatal("invalid option");
	}
}
                                                                                50

static void
usage(void)
{
	fprint(2, "usage: test-whack [-b blocksize] [-n count] [-t num]\n");
	exits("usage");
}


void                                                                            60
main(int argc, char *argv[])
{
	uchar *plain, *whacked;
	int psize, wsize;
	int i, uwcount, smaller, nsmaller;
	vlong start, stop;
	vlong wtotal, uwtotal;
	vlong bytes;
	Unwhack uw;
                                                                                70
	ARGBEGIN{
	case 'b':
		blocksize = atoi(EARGF(usage()));
		break;
	case 'n':
		count = atoi(EARGF(usage()));
		break;
	case 't':
		datatype = atoi(EARGF(usage()));
		if(datatype < 0 || datatype > 2)                                        80
			usage();
		break;
	default:
		usage();
	}ARGEND

	if(argc != 0)
		usage();
```

```
	psize = blocksize;                                                         90
	plain = malloc(psize);
	whacked = malloc(blocksize);
	if(plain == nil || whacked == nil)
		sysfatal("malloc");

	srand(time(0));
	wtotal = 0;
	uwtotal = 0;
	uwcount = 0;
	nsmaller = 0;                                                              100
	bytes = 0;
	for(i = 0; i < count; i++) {
		getdata(plain, psize);

		start = nsec();
		wsize = whackblock(whacked, plain, psize);
		stop = nsec();
		smaller = (wsize < 0 || wsize >= psize) ? 0 : 1;
		if(smaller)
			nsmaller++;                                                        110
		fprint(2, "whack %d %llud\n", smaller, stop-start);
		wtotal += stop-start;

		if(wsize <= 0) {
			bytes += psize;
			continue;
		}
		bytes += wsize;

		start = nsec();                                                        120
		unwhackinit(&uw);
		unwhack(&uw, plain, psize, whacked, wsize);
		stop = nsec();
		fprint(2, "unwhack %llud\n", stop-start);
		uwtotal += stop-start;
		uwcount++;
	}
	print("whack total: %llud ms\n", wtotal/1000000ULL);
	print("whack compressed percentage of blocks: %.02f%%\n", (double)nsmaller*100.0/count);
	print("whack throughput: %.03f MB/sec\n", (double)blocksize*count*1000000000.0/(1024.0*1024*wtotal)); 130
	print("whack per operation: %llud nsec, %llud microsec\n", wtotal/count, wtotal/(count*1000));
	print("whack compressed size: %.02f%%\n", 100.0*bytes/(count*psize));

	if(uwcount > 0) {
		print("unwhack total: %llud ms\n", uwtotal/1000000ULL);
		print("unwhack throughput: %.03f MB/sec\n", (double)blocksize*uwcount*1000000000.0/(1024.0*1024*uwtotal));
		print("unwhack per operation: %llud nsec, %llud microsec\n", uwtotal/uwcount, uwtotal/(uwcount*1000));
	} else {
		print("no blocks unwhacked, no statistics\n");
	}                                                                          140

	exits(nil);
}
```