Efficient and Safe Data Backup with Arrow

Technical Report UCSC-SSRC-08-02
June 2008

Casey Marshall
csm@soe.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
`http://www.ssrc.ucsc.edu/`

This paper was filed as an M. S. project report in June, 2008. Casey's M. S. advisor was Prof. Ethan Miller; Prof. Darrell Long was the second reader for the report.

Efficient and safe data backup with Arrow

Casey Marshall
University of California, Santa Cruz
csm@soe.ucsc.edu

June 12, 2008

Abstract

We describe Arrow, an efficient, safe data backup system for computer networks. Arrow employs techniques of delta compression (or deduplication) to achieve efficient storage and bandwidth utilization, and collision-resistant hashing and error-correction coding to protect against and correct storage errors.

keywords: content-addressable storage; error-correcting storage systems; data backup; deduplication.

1 Introduction

Content-addressable storage, error detection and correction, and deduplication are all interesting topics in the field of archival data storage. Particularly in the case where files are being archived over time, where snapshots of the file tree are taken, and where there is significant data in common between snapshots.

Arrow implements a data backup system that combines collision-resistant hash functions, rolling checksums, and error-correction codes to provide a deduplicating, versioned, error-recoverable archival storage system. Arrow stores files as lists of checksums, and performs a fast checksum search algorithm for determining what parts of a file have changed, achieving both a speedup in time to store a version, and a savings in the amount of physical storage used. There checksums are also used to identify and verify the integrity of data stored in the system, and error-correction codes are present to allow correction of small storage errors.

2 Related Work

Rsync is a popular free software program for synchronizing similar files on computers connected on a network, using a novel checksum search algorithm to reduce the amount of data that needs to be transmitted [11, 12]. Arrow borrows heavily from rsync, using a similar algorithm to search for duplicate chunks in files to be backed up, and uses the same rolling checksum function. The delta-compression ideas behind rsync have inspired other data backup solutions, implemented simply as thin layers on the rsync program itself [8], or as new implementations of the idea [2].

Error-correction codes have been widely used in digital storage and transmission, prominently in media like compact disks and DVDs, since the media is exposed to more physical abuse. The sharp increases in hard drive density and speed, and the relative stability of hard disk failure rates, have meant that errors in disk storage systems have become much more frequent. Hard disks also offer error-correction codes for the data stored to disk, but these error codes are not portable, are vendor-specific, and are often un-verifiable by high-level software. A very common error-correction code is the Reed-Solomon code, which is used today in many digital storage platforms. With the increasing rates in hard disk failure, focus has shifted towards fault-tolerant and repair-driven systems, e.g. chunkfs [3].

File snapshotting and content-addressable storage appear in many systems for file versioning, storage and network optimization, intelligent caching, and so on, and the design of Arrow follows these systems. Elephant [9] replaces single file inodes with a log of inodes to keep each change to files, and so it keeps all versions of files through modifications. Venti [7] references parts of files primarily by a strong hash code of short blocks of data, avoiding duplication of data written to disk. Ivy [5] uses logs of changes and hash code references to blocks to implement an efficient peer-to-peer network file system. The Shark file system [1] optimizes network file transfer with hashes of data chunks, and intelligent file chunking. The Git distributed revision control system uses a mutable index and a write-only object store to implement revision control [13].

3 Overview

Arrow stores files in three separate layers, each managing some part of the source file tree. The *file tree* layer mirrors the source file tree, with a same-named directory per source directory, and a symbolic link per source file. The target of the symbolic link is a *version file,* which describes a list of chunks that comprise the file, and contains a reference to the previous version of the file. The *chunks* are stored in an object storage system where the key is derived from the chunk's hash code. The overall goal of Arrow is to use chunk hashes to reduce the amount of data stored and transferred over the network, and to strongly protect stored data from corruption by providing enough information to detect and correct errors.

Figure 1 illustrates the overall layout of Arrow; the following sections explain how each layer is implemented.

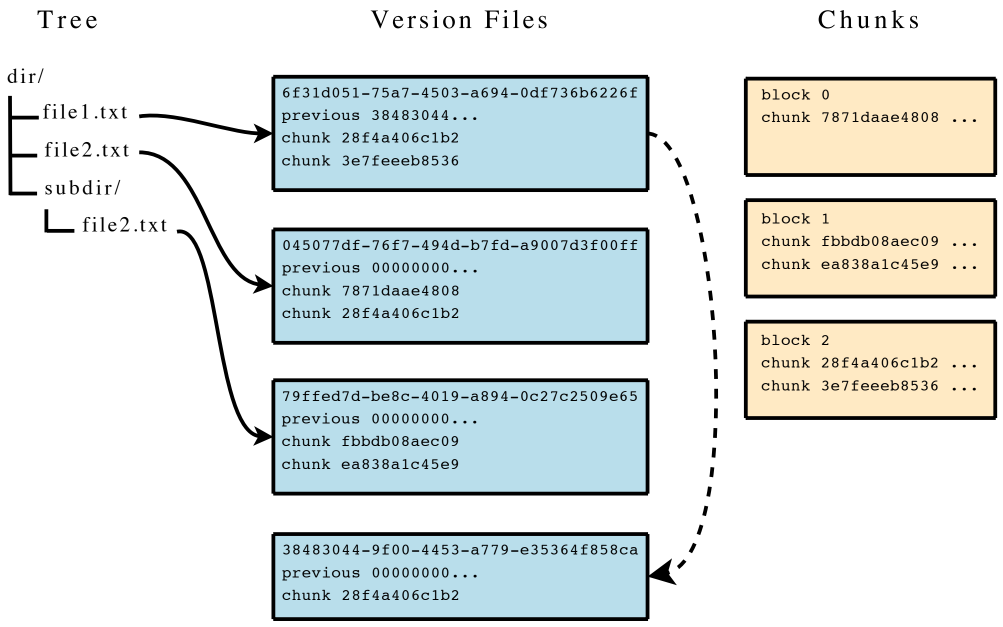

Figure 1: Overview of how Arrow stores files

4 Chunk Storage

Arrow stores files as *chunks,* which are small runs of data from files (between 700 and 16000 bytes), and chunks are identified by a pair of hash codes over the data. The hash pair consists of a four byte simple checksum, similar to Adler-32, which has a *rolling* property: given a checksum over values $[m, n]$, it is simple to compute the sum over values $[m + 1, n + 1]$, if we have bytes m and $n + 1$. The other hash is the 16-byte MD5 hash of the data. The concatenation of these two values is the chunk identifier.

To store chunks, Arrow uses a simple file format shown in figure 2. The store begins with a simple header; followed by a fixed number of block keys, along with the offset and length of each chunk, and the size of the key space is chosen once at store creation time; then the chunk values; the file ends with parity bytes computed over the rest of the file, using a Reed-Solomon error correction code.

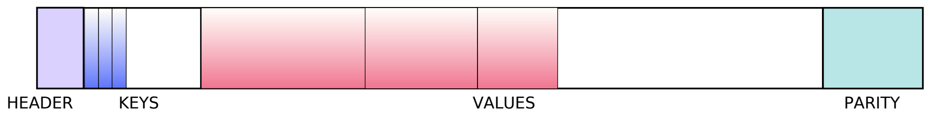

Figure 2: Storage layout in arrow.

Each of these chunk files — called *blocks* — is relatively small; large enough to store up to about 200 chunks, assuming that chunks average between 700 and 16000 bytes. The key space in a block is fixed-size; if needed, the data space will grow if it isn't of sufficient size. We would quickly fill up a single block, so we need to store chunks across multiple blocks and have a method for managing them. The technique used by Arrow is *linear hashing,* first proposed by Witold Litwin in [4]. Core to linear hashing is the function that maps chunk keys to block identifiers, given in figure 3.

```
LINEAR-HASH(K)
1  $x \leftarrow K \bmod 2^i$
2  if $x < n$
3     then
4           $x \leftarrow K \bmod 2^{i+1}$
5  return $x$
```

Figure 3: The linear hash function

i and n are initially zero. If we fill up a block beyond a “load factor” — in Arrow, if 70% of the slots in the block are filled — we add a new block $n + 2^i$, rehash each key in block n, which will move some keys from block n to block $n + 2^i$, and increment n. Once $n = 2^i$, we increment i and set $n \leftarrow 0$. The keys are effectively random, given the nature of MD5, so rehashing the keys of a block should move approximately half the entries into the new block.

This implementation allows the chunk storage to grow one block at a time, while maintaining an $O(1)$ lookup time, and achieves $O(1)$ insertions, with an increased cost at regular intervals when a block needs splitting. the rationale for this being that looking up a chunk involves a single execution of LINEAR-HASH, followed by a linear search through a list of at most 200 chunk keys. This intuitive analysis does not consider the costs of doing the lookup considering the underlying file system, however, which can add significant cost to the operation.

We have a fairly rough strategy for splitting blocks, too: block n is the one that gets split, even if some other block overflowed its load factor. In practice this shouldn't be a problem, since block n is necessarily the one that was split least recently, and since our keys have a uniformly random distribution, block n will be the one expected to have the highest load.

Since arrow stores the chunk's hash along with the chunk, storage errors can be detected easily by iterating through each chunk key, recomputing the hash on the chunk, and comparing the values. If there is a mismatch, we know that there is a storage error either in the key or in the chunk, so we can attempt to repair storage errors in either location, using the parity information. Arrow computes $(253, 255)$ Reed-Solomon codes over 253-byte subblocks, producing 2 parity bytes, and on an error it will attempt to correct the error in the key or in the value, by repeatedly attempting to fix the subblocks that overlap the apparently-corrupt key or value, confirming the fix by checking the hashes again. Arrow assumes few errors; it can detect large errors, but won't be able to correct errors spanning more than a few bytes.

5 File Storage

Files in arrow are stored primarily as lists of chunk references, that is, the hash pair of the chunk, and will store very small runs of data directly. The layout of a file is as follows:

- The name of the underlying file.
- The MD5 hash of the entire file.
- The identifier of the previous version of this file, if any.
- The file size, file mode bitset, and modification, status change, and creation times.
- The chunk size used to chunkify the file. This is inherited from previous file versions.
- A list of chunks. Each chunk is either a reference to a chunk stored in the chunk storage layer, or (if the chunk is smaller than a chunk identifier) the chunk bytes stored directly.

Files are identified by a random 16-byte UUID, and files that have no previous versions are stored with the null UUID (sixteen zero bytes) as the previous-version UUID. Each of these files is stored as simple data files in a flat namespace, with the file's UUID as its file name.

File hierarchies are stored to match the underlying file hierarchy, where directories are stored as-is, with the same name, and files are stored as symbolic links to the head revision of the file.

This scheme was chosen for its simplicity; there are possible refinements we could make to storing file and version hierarchies, such as storing versioned directories as well as files. For example, when a directory is backed up, not only are files that changed in that directory committed as new versions, but also the directory as a whole — which may include new, modified, or deleted files — is committed as a new version of the directory after all changed files are committed. A scheme like this would better reflect the needs of a system that stored historical versions, since files in the same directory are often related, and rolling back to previous versions for groups of files may make more sense than rolling back single files. This file tree scheme also ignores deleted files at the moment, and cannot handle replacing a file with a directory of the same name (or vice-versa), but none of these are fundamental issues — they are just left unimplemented here.

6 File Synchronization

When Arrow has a new version of a file to be backed up, it first fetches the version file of the previous version of the file, which will be the *basis* file. Each checksum pair $(weak, strong)$ that is the underlying chunk size long — through the backup process, chunks shorter than the base chunk size may be stored, but these are ignored in the synchronization, which only uses the underlying chunk size — is inserted into a hash table with 2^{14} entries, by taking the weak sum modulo the table size. A running rolling checksum is taken over the new file version, one byte at a time, and each time the hash table is probed for that weak value. If a possible weak checksum is found, an MD5 is computed over the current chunk, and if *that* matches, we have found a duplicate chunk reference, and can emit that reference in the new file.

The pseudocode, with some details omitted, for this process is given in figure 4. The procedures ROLLSUM-COMPUTE and ROLLSUM-ROTATE are the rolling checksum algorithm from `librsync` [6]. HASH-PROBE searches the hashtable for the weak hash only, and HASH-CONTAINS tells if the hashtable contains the weak and strong hash combination.

The consequence of this procedure is that any chunks in both $basis$ and the new file to be backed-up will be stored only once. If any chunks that exist in the new file that were found in previous versions of any file, then again, that chunk will only be stored once — though, it is very unlikely that we will find duplicate chunks by chance, unless they have specific signatures, such as a chunk that consists of all zero bytes.

This synchronization procedure is based on the rsync algorithm, though it presents some limitations to the delta compression that this version can achieve. Since the rsync algorithm uses a single block size, and since these block boundaries are fixed in a single file, there are limits to the number of duplicate blocks between the basis file and the new file we can find.

```
FILE-SYNC($basis$, $newfile$, $input$)
 1  $H \leftarrow$ hashtable of $2^{14}$ entries
 2  for each $chunk$ in $basis$
 3      do if $chunk.length = basis.chunk\_size$
 4          then HASH-INSERT($chunk.id$)
 5  $buffer \leftarrow basis.chunk\_size$ bytes from $input$
 6  $weak \leftarrow$ ROLLSUM-COMPUTE($buffer$)
 7  while $\neg$END-OF-FILE($input$)
 8      do if HASH-PROBE($H, weak$)
 9          then $strong \leftarrow$ MD5($buffer$)
10              if HASH-CONTAINS($H, (weak, strong)$)
11                then insert chunks between the last match and the
12                    current match into $newfile$
13                    insert $(weak, strong)$ into $newfile$
14                    $buffer \leftarrow basis.chunk\_size$ bytes from $input$
15                    $weak \leftarrow$ ROLLSUM-COMPUTE($buffer$)
16                    continue
17         $c \leftarrow$ READ($input$)
18         ROLLSUM-ROTATE($weak, buffer[0], c$)
19         add $c$ to $buffer$
20         remove the first element from $buffer$
```

Figure 4: File synchronization procedure

7 Performance Evaluation

This first test used a simple C program that will synchronize single files, and that program was used in a Python program to manage the version files. This test used the Linux kernel source [10], starting with version 2.6.23, and applying each of the 17 patches released for this kernel. Table 1 shows the results of backing up each patched kernel tree with Arrow. Figures 5 and 6 plot the patch size versus Arrow's increase in backup size, and the time the backup took versus the number of files that changed in that patch release.

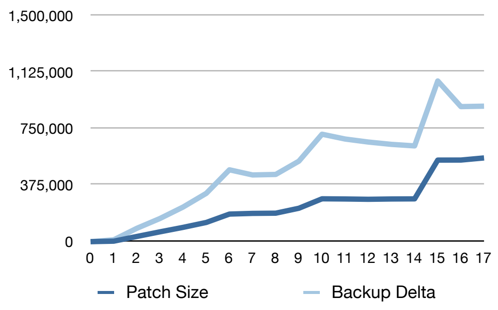

Figure 5: Text patch size and Arrow backup delta

The *patch size* column is the size of the patch file, in bytes, decompressed. *Source* is the size of the kernel tree with the patch applied. *Backup* is the size of the Arrow backup, storing all cumulative version trees, by counting the bytes used by the symlink tree, the version files, and the chunks used in the block store (since many slots of the block

Patch	Patch Size	Source	Backup	Time	Files
0	0	252,065,213	283,107,750	451.08	22,530
1	3,218	252,065,491	283,119,871	91.83	2
2	34,402	252,066,473	283,208,456	89.34	18
3	65,218	252,072,178	283,362,188	102.47	53
4	94,870	252,073,453	283,591,942	143.06	70
5	127,389	252,077,945	283,912,484	181.44	81
6	182,957	252,082,973	284,389,157	193.47	115
7	187,496	252,084,315	284,831,665	220.82	113
8	188,895	252,084,342	285,277,506	229.20	107
9	221,604	252,087,238	285,811,410	238.36	132
10	284,301	252,091,147	286,524,145	240.94	181
11	283,078	252,091,071	287,204,874	254.57	175
12	281,040	252,090,640	287,865,620	256.07	158
13	283,099	252,090,838	288,510,953	255.74	254
14	283,830	252,090,842	289,146,161	257.06	146
15	541,078	252,147,239	290,212,238	265.04	228
16	541,304	252,147,258	291,108,027	267.25	217
17	555,389	252,150,368	292,006,750	277.29	218

Table 1: Backup test of Linux kernel versions

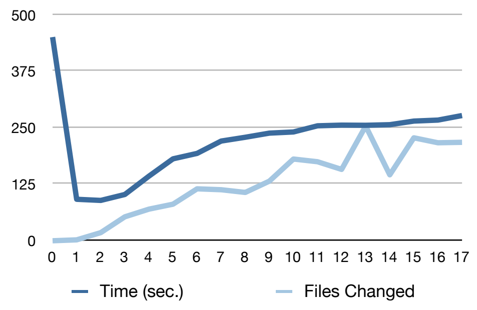

Figure 6: Backup time and number of files changed

store may be unused, the actual disk space used by the backup is significantly larger). *Time* is the time in seconds it took to synchronize that kernel tree. *Files* is the number of changed files in that kernel version.

Table 2 lists the sizes of the various parts of Arrow's storage system, the symlink tree, the version files, and chunk storage, from the same test against the Linux kernel source. Also included is the size difference after backing up that patch version.

This next test implemented network backups by tunneling a simple protocol over an SSH connection. The client side performs the synchronization, retrieving a version file for a file to be synchronized, after comparing the file's MD5 hash (the file is not synchronized if it has clearly not changed); the server side tracks the chunk store, the version files, and the file tree. The same test of Linux kernel versions from the previous test, tunneling over the local loopback device, measuring the time taken and the bytes read and written. Table 3 shows the results of this test.

These results are quite in line with what is expected: there is a constant overhead for transferring the MD5 hashes for all the files in the backup, and the time needed to compute these hashes. As the differences increase, it tends to take longer and more bytes transferred, including the MD5 hashes, the contents of each version file for those files where the MD5 checksum did not match, and the new file chunks. The overhead for the initial backup is about 12% greater

Patch	Tree	Versions	Chunks	Difference
0	6,020,348	18,715,460	258,371,942	0
1	6,020,372	18,721,280	258,378,219	12,121
2	6,020,372	18,763,104	258,424,980	88,585
3	6,020,369	18,832,200	258,509,619	153,732
4	6,020,366	18,939,252	258,632,324	229,754
5	6,020,371	19,108,140	258,783,973	320,542
6	6,020,365	19,365,364	259,003,428	476,673
7	6,020,370	19,622,648	259,188,647	442,508
8	6,020,368	19,883,304	259,373,834	445,841
9	6,020,364	20,184,188	259,606,858	533,904
10	6,020,355	20,577,220	259,926,570	712,735
11	6,020,366	20,971,084	260,213,424	680,729
12	6,020,361	21,348,916	260,496,343	660,746
13	6,020,356	21,725,972	260,764,625	645,333
14	6,020,361	22,099,124	261,026,676	635,208
15	6,020,361	22,613,212	261,578,665	1,066,077
16	6,020,366	23,116,276	261,971,385	895,789
17	6,020,348	23,620,716	262,365,686	898,723

Table 2: Storage breakdown of Linux kernel versions

Patch	Time	Read	Written
0	440.77	2,752,454	276,687,703
1	25.82	816,904	1,259,019
2	17.93	851,860	1,636,856
3	15.48	870,087	1,487,275
4	22.45	903,195	1,776,221
5	79.02	975,071	2,613,638
6	136.64	1,062,382	2,805,430
7	75.39	1,066,618	1,772,322
8	25.66	1,075,352	1,850,436
9	68.07	1,113,524	2,445,167
10	163.88	1,202,211	3,214,490
11	96.83	1,202,757	2,030,031
12	18.58	1,202,751	2,020,530
13	16.68	1,205,075	2,034,749
14	17.91	1,208,353	2,064,046
15	116.60	1,333,601	4,053,726
16	41.00	1,335,040	2,337,040
17	28.50	1,350,156	2,559,890

Table 3: Network backup of Linux kernel versions

than the data, and about 10% more needs to be sent over the network. Compared with backing up the an entire kernel version, Arrow only needed to transfer at most 2% of the total size of the source data (for the biggest incremental delta, version 2.6.23.15), and storing this new version only increased the backup size by 0.4% of the new version size; or, about 11% of the size of the `bzip2` compressed tar archive sent over the wire, and about 2% increase in storage.

All these tests were run on a low-end Ubuntu Linux server, with a two-core 2.8GHz Intel Pentium D and 1GB of RAM, backing up to a single 7200 RPM hard disk.

Kernel	Files	Time (1 disk)	Time (RAID-5)	Size	Delta
2.6.10	16,583	263.82	150.50	219,538,292	0
2.6.11	5,481	50.42	24.16	249,878,382	30,340,090
2.6.12	7,313	89.42	111.70	294,696,204	44,817,822
2.6.13	8,285	162.65	154.36	348,544,181	53,847,977
2.6.14	8,517	428.72	130.97	399,782,781	51,238,600
2.6.15	9,258	436.17	101.68	460,175,258	60,392,477
2.6.16	10,051	236.86	124.06	526,137,368	65,962,110
2.6.17	10,247	194.90	257.49	594,133,084	67,995,716
2.6.18	11,768	278.77	106.34	666,703,249	72,570,165
2.6.19	12,471	967.12	135.16	748,048,042	81,344,793
2.6.20	11,801	807.57	97.71	819,728,218	71,680,176
2.6.21	11,782	853.37	96.82	896,486,021	76,757,803
2.6.22	12,576	944.28	105.83	982,696,828	86,210,807
2.6.23	12,231	973.92	167.14	1,065,769,264	83,072,436
2.6.24	13,413	995.01	182.68	1,162,798,886	97,029,622
2.6.25	13,940	1185.11	111.60	1,266,597,571	103,798,685

Table 4: Native, local backup of major kernel releases

This final test increased the size of blocks from around 200 entries to 5,000, as an effort to reduce I/O load. The test used the local C client to back up every major point release kernel from version 2.6.10 to 2.6.25, representing almost three and a half years of development. This test was run on the same machine as before, and also on a server-class system with two four-core 3.2 GHz Intel Xeon processors, 8 GB of RAM, and backing up to a five-disk RAID 5 array. The difference in performance is startling, illustrated in table 4 and figure 7. The reason for this seems to be that arrow is heavily I/O bound; running a similar test under a `dtrace` session on Mac OS X revealed that most of the program time is spent in `fread`, `open` and `close`, `memcpy` (the blocks are implemented by mapping the block file into memory), and `munmap`. These results do seem to suggest that Arrow requires a certain amount of disk bandwidth to operate quickly, and that it can saturate a low-bandwidth disk after a modest amount of data is written to it. This raises an important issue with a linear hash table: writes that should be essentially sequential are randomized across the disk; we can't help the case where we are referencing an existing chunk, but adjacent chunks in a single file are scattered throughout the storage system, and in certain cases, doing this scattering can be very expensive.

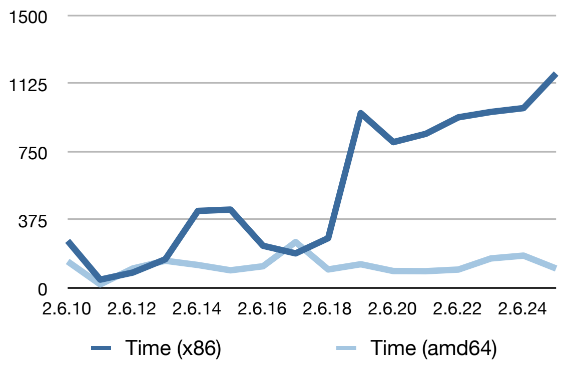

Figure 7: Arrow on commodity (x86) and server (amd64) hardware

8 Conclusions

Arrow is a new application of existing ideas, combining them into a safe, efficient, recoverable data backup system. The current implementation is only a prototype, but proves that the idea can offer a data backup solution that combines storage and transmission efficiency, and strong error detection and correction.

Arrow is still only a beginning. There are many other concerns with archival storage and data backup that are not addressed here. Most especially is distributing the storage across multiple storage systems; for a truly reliable backup system, it can't rely on a single piece of hardware, or a single software system. Arrow's implementation of various pieces introduces some performance bottlenecks, most clearly when the backup size grows very large, which could be addressed separately. To be fully usable as an archival storage system, Arrow still needs support and optimization of the verification and correction processes, and needs a usable method for restoring backed-up data.

Arrow is written in C, is open-source, and is available via a read-only Git repository at `http://git.metastatic.org/readonly/arrow.git`. A web interface is available from `http://git.metastatic.org/`.

References

[1] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: scaling file servers via cooperative caching. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 129–142, Berkeley, CA, USA, 2005. USENIX Association.

[2] Ben Escoto. rdiff-backup. `http://www.nongnu.org/rdiff-backup/`.

[3] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the Second Workshop on Hot Topics in System Dependability (HotDep '06)*, Seattle, WA, November 2006.

[4] Witold Litwin. Linear hashing: a new tool for file and table addressing. In *VLDB '1980: Proceedings of the sixth international conference on Very Large Data Bases*, pages 212–223. VLDB Endowment, 1980.

[5] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[6] Martin Pool and Donovan Baarda. `librsync`. `http://librsync.sourceforge.net/`.

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

[8] Nathan Rosenquist, David Cantrell, et al. rsnapshot. `http://www.rsnapshot.org/`.

[9] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 110–123, New York, NY, USA, 1999. ACM.

[10] Linus Torvalds et al. The Linux Kernel. `http://www.kernel.org/`.

[11] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.

[12] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical report, The Australian National University, 1996.

[13] John Wiegley. Git from the bottom up, May 2008. `http://www.newartisans.com/blog_assets/git.from.bottom.up.pdf`.