Guidelines for Using Compare-by-hash

Val Henson
IBM, Inc.
vhenson@us.ibm.com

Richard Henderson
Red Hat, Inc.
rth@redhat.com

Abstract

Recently, a new technique called *compare-by-hash* has become popular. Compare-by-hash is a method of content-based addressing in which data is identified only by the cryptographic hash of its contents. Hash collisions are ignored, with the justification that they occur less often than many kinds of hardware errors. Compare-by-hash is a powerful, versatile tool in the software architect's bag of tricks, but it is also poorly understood and frequently misused. The consequences of misuse range from significant performance degradation to permanent, unrecoverable data corruption or loss. The proper use of compare-by-hash is a subject of debate[10, 29], but recent results in the field of cryptographic hash function analysis, including the breaking of MD5[28] and SHA-0[12] and the weakening of SHA-1[3], have clarified when compare-by-hash is appropriate. In short, compare-by-hash is appropriate when it provides some benefit (performance, code simplicity, etc.), when the system can survive intentionally generated hash collisions, and when hashes can be thrown away and regenerated at any time. In this paper, we propose and explain some simple guidelines to help software architects decide when to use compare-by-hash.

1 Introduction

Compare-by-hash takes advantage of the fact that applications frequently read or write data that is identical to already existing data. For example, editing a document usually involves saving the same file with slight differences many times. Rather than read or write the identical part of the data a second time to the disk or network, we instead refer to the instance of the data that we already have. Using a collision-resistant cryptographic hash, we can quickly determine with high probability whether two blocks are identical by comparing only their hashes and not their contents. By some estimations, the chance of a hash collision is much lower than the chance of many kinds of hardware errors, and so many feel comfortable ignoring the possibility of a hash collision.

Compare-by-hash made its first appearance in 1996 in `rsync`[26], a tool for synchronizing files over the network, and has since been used in a number of projects, including LBFS[14], a distributed file system; Stanford's virtual computer migration project[19]; Pastiche[7], an automated backup system; OpenCM[21], a configuration management system; and CASPER[24], a block cache for distributed file systems. Venti, a block archival storage system for Plan 9, is described as using compare-by-hash[18] but as implemented it checks for collisions on writes[9]. Compare-by-hash is also used in a wide variety of document retrieval systems, content-caching proxies, and many other commercial software products.

Unfortunately, many uses of compare-by-hash do not take into account two important but lesser-known properties of cryptographic hash functions: they are computationally expensive, and they are relatively short-lived compared to the software applications that use them. "Computationally expensive" means that, in many applications, compare-by-hash has *worse* performance than traditional techniques. "Short-lived" means that the most important property of a cryptographic hash — collision resistance — is defeated by advances in cryptanalysis within a few years of the hash's introduction, typically 2-10 years (see Table 1).

We wrote this paper in response to a common and urgent question from our colleagues: When should I use compare-by-hash? Many want a comprehensive and practical review on the subject to give to colleagues and graduate students. Others want a second opinion on the design of a system using compare-by-hash. In this paper, we propose and explain a practical set of guidelines to help system architects decide when to use compare-by-hash. The guidelines can be summarized by the following questions:

1. Will compare-by-hash provide some benefit — save time, bandwidth, etc.?

Name	Introduced	Weakened	Broken	Lifetime	Replaced by
MD4	1990	1991	1995	1–5 yrs	MD5
Snefru	1990	—	1993	3 yrs	MD5
MD5	1992	1994	2004	2–10 yrs	SHA-1
MD2	1992	1995	abandoned	3 yrs	SHA-1
RIPEMD	1992	1997	2004	5–12 yrs	RIPEMD-160
HAVAL-128	1992	—	2004	12 yrs	SHA-1
SHA-0	1993	1998	2004	5–11 yrs	SHA-1
SHA-1	1995	2004	—	9+ yrs	SHA-256 (?)
RIPEMD-160	1996	—	—	9+ yrs	—

Table 1: Lifetimes for some cryptographic hash functions[20, 6, 28, 3].

2. Is the system usable if hash collisions can be generated at will?

3. Can the hashes be regenerated with a different algorithm at any time?

If the answer to all of these questions is yes, then compare-by-hash is probably a reasonable choice.

We begin with some definitions and examples in Section 2 to provide motivation and some background for the rest of the paper. We then describe our proposed guidelines in more detail in Section 3. We go into more depth on how to estimate the performance of compare-by-hash relative to other techniques in Section 4. We discuss alternatives to compare-by-hash in Section 5, and conclude with a summary of our recommendations.

2 Background

2.1 Motivation

We wrote this paper in response to requests from many of our colleagues. Understanding the caveats and implementation issues involved in using compare-by-hash is not trivial. For example, we found that many people view cryptographic hash functions as magic boxes that spit out unique identifiers, rather than as human-designed mathematical functions with real-world limits and costs (and even political considerations). Those who have successfully implemented systems using compare-by-hash find that others following their lead missed some of the more subtle points of their arguments, and wanted a comprehensive paper explaining the issues. For some computer scientists, the implications of compare-by-hash are obvious and need not be spelled out; for others, this paper will fill a small but important gap in the computer science literature.

2.2 Cryptographic hash functions

A hash function takes a variable length input and produces a fixed length output, the input's *hash value*. TCP checksums are an example of a simple hash function: XOR the TCP header and data, 16 bits at time. A cryptographic hash function is a hash function for which it is computationally easy to calculate the hash value of an input, but difficult to find an input that has a particular hash value, or to find two inputs that hash to the same hash value[20]. Hash functions with this property are often called *collision-resistant* (the term *collision-free* is misleading since a hash function must have collisions as long as it has more inputs than outputs).

Unfortunately, analysis of cryptographic hash functions and therefore the ability to generate hash collisions trails the creation of cryptographic hash functions by only a few years. Cryptographic hash functions thought impregnable when introduced quickly become trivial to break within a few years (see Table 1). So far, the longest-lived cryptographic hash is HAVAL-128, with 12 years between its introduction and the complete break of the algorithm. The most popular unbroken cryptographic hash left, SHA-1, is now 9 years old and has been significantly weakened, with successful attacks up to at least round 40 of the algorithm, out of a total of 80 rounds of computation[3]. Cryptographic hash functions usually operate on a part of a message for a certain number of rounds of computation, then use the output from the previous part of the message as input to the next set of calculations on the next part of the message. More rounds usually makes the function harder to break, but more expensive to compute.

Cryptographic hash functions are very expensive to compute; SHA-1, for example, requires 80 rounds of mathematical computations on each 16-byte section of the input[15]. The computation can't be paral-

lelized; the output of the previous 16-byte section is used as one of the inputs in the calculation for the next 16-byte section. Most non-cryptographic checksum functions are much faster to compute than cryptographic hashes, and often have been designed specifically for speed. No matter how fast modern hardware can compute cryptographic hash functions (perhaps with the aid of built-in hardware), computing a simpler checksum, not to mention doing a direct memory comparison, is orders of magnitude faster than computing a cryptographic hash on the same data. We recommend reading through the specification for SHA-1[15] to get a feel for the amount of computation involved.

Cryptographic hash functions are so expensive to compute, in fact, that they have formed the basis of at least one anti-spam measure, called *hashcash*[1]. In hashcash, each email must contain a proof-of-work from the sender in the form of a string containing the date, the recipient's email address, and a string of random bits. When this string is hashed using SHA-1, it must form a partial hash-collision with a specific hash value (the hash value of all zeroes, specifically). In order to find the right random bits to produce a partial hash-collision, the sender must repeatedly hash the string with many different random values until it finds a sufficiently long partial collision. This takes a significant amount of CPU time — approximately half a second in the current implementation — and so guarantees that the sender is severely rate-limited in the amount of email it can send, and is therefore unlikely to be a spammer.

2.3 Compare-by-hash

In compare-by-hash, blocks of data are identified by their cryptographic hashes; blocks with identical hashes are assumed to be identical. This is different from the common use of hashes, in which blocks with different hash values are known to be different, and blocks with the same hash are probably (but not certainly) identical.

Compare-by-hash can be an advantage in many cases, where it is cheaper to compute and store or send a cryptographic hash than to directly compare the blocks. One use of compare-by-hash is to reduce bandwidth usage. Before sending a block, the sender first transmits the hash of the block to the receiver. The receiver checks to see if it has a local block with the same hash value. If it does, it assumes that it is the same block as the sender's. In the case of a 4096-byte block and a 160-bit hash value, this system can reduce network traffic from 4096 bytes to 20 bytes, or about a 99.5% savings in bandwidth.

Input 1
d131dd02c5e6eec4693d9a0698aff95c
2fcab5**87**12467eab4004583eb8fb7f89
55ad340609f4b30283e4888325**71**415a
085125e8f7cdc99fd91dbd**f2**80373c5b
960b1dd1dc417b9ce4d897f45a6555d5
35739a**c7**f0ebfd0c3029f166d109b18f
75277f7930d55ceb22e8adba79**cc**155c
ed74cbdd5fc5d36db19b0a**d8**35cca7e3
Input 2
d131dd02c5e6eec4693d9a0698aff95c
2fcab5**07**12467eab4004583eb8fb7f89
55ad340609f4b30283e4888325**f1**415a
085125e8f7cdc99fd91dbd**72**80373c5b
960b1dd1dc417b9ce4d897f45a6555d5
35739a**47**f0ebfd0c3029f166d109b18f
75277f7930d55ceb22e8adba794**c**155c
ed74cbdd5fc5d36db19b0a**58**35cca7e3
MD5 hash of input 1
a4c0d35c95a63a805915367dcfe6b751
MD5 hash of input 2
a4c0d35c95a63a805915367dcfe6b751

Table 2: Two inputs with colliding MD5 hash values, in hexadecimal format with differing bytes in bold.

This is an incredible savings, but the cost is the risk of a hash collision. Using a standard approximation, we estimate that to have a 50% chance of finding a randomly occurring collision in a hash with n bits of output, we need about $2^{\frac{n}{2}}$ inputs. (Note that this is an upper bound only[2].) For a 160-bit output, we need about $2^{\frac{160}{2}}$ or 2^{80} inputs to have a 50% chance of a collision. Put another way, we expect with about 48 nines ("0." followed by 48 nines, approximately $1 - 2^{-160}$) of certainty that any two randomly chosen inputs will not collide, whereas empirical measurements tell us we have only about 8 or 9 nines of certainty that we will not encounter an undetected TCP error when we send the block over the network[22]. In the face of much larger sources of potential error, the error added by compare-by-hash appears to be negligible. However, this calculation neglects the very real fact that cryptographic hash functions are under constant attack. Now that a collision has been found in MD5[28], if you are given a choice of two inputs, you can make the chance of a hash collision in MD5 equal to 1, not 2^{-128} — see Table 2 for two inputs that collide.

2.4 Example applications

We will use several example applications to illustrate our guidelines. `rsync`[26] is an application for synchronizing files, either locally or over the network. When operating in the default mode and comparing files remotely, it uses compare-by-hash to detect parts of a file that have not changed. LBFS[14] is a network file system that uses compare-by-hash to identify chunks of data shared among all files stored on the server. BitKeeper[4] is a source control system that does not use compare-by-hash. It instead maintains a complete history of changes made; when synchronizing two source trees, it sends only the changes that are not present in the target tree.

3 Guidelines for using compare-by-hash

In the introduction, we gave the following rules for deciding when to use compare-by-hash:

1. Will compare-by-hash provide some benefit — save time, bandwidth, etc.?
2. Is the system usable if hash collisions can be generated at will?
3. Can the hashes be regenerated with a different algorithm at any time?

If the answer to each of these questions is yes, then compare-by-hash is probably a good technique. In this section, we'll discuss each of these questions in more detail.

3.1 Does compare-by-hash provide a benefit?

The first question the software designer should ask herself is whether using compare-by-hash results in an overall improvement of some sort — for example, saving storage space, using less bandwidth, or reducing implementation complexity. Understanding when compare-by-hash is a benefit is more difficult than it appears at first glance. Without a clear understanding of the computational cost of computing and comparing cryptographic hashes, usage patterns, and alternative approaches, we can easily end up with a slower, less efficient, and less correct system than we began with. For example, a frequent proposal is to use compare-by-hash to compare blocks of data located on a single system to eliminate duplicates, e.g., to increase the effective size of a disk cache by keeping only copy of identical blocks. In this case, it is far more efficient to compute a simple, fast checksum to find potential matches, and then directly compare the data, byte by byte, to confirm duplicates. VMWare's ESX server uses this technique to coalesce identical memory pages[27]. This section will give an intuitive overview of the issues involved; for a more rigorous analytical approach, see Section 4.

3.1.1 Trading computation for bandwidth

Compare-by-hash allows you to trade computation for bandwidth, allowing you to send only the hash of a block of data, rather than the block itself. The most famous example of this tradeoff is `rsync`, an application for synchronizing files. When comparing two files in remote locations, `rsync` compares MD4 hashes of blocks likely to be identical; if they match, it assumes they are identical and does not send that block over the network. This is an advantage when it is faster to compute a block's MD4 checksum than it is to send it over the network. However, in many cases (such as most local area networks or high-speed Internet connections), it is faster to send the data than compute its MD4 checksum.

3.1.2 Trading computation time for storage

You can also trade computation time for storage space; for example, you can record whether you have seen a particular block before by storing only the hash and not the block itself. Here also, storage space must be tight enough to warrant the performance hit, since a cheap checksum combined with direct compare will perform better. Note that a recent survey showed most desktop disks are half full[8], and a recent mathematical analysis showed that peer-to-peer data-sharing networks are limited by cross-network bandwidth rather than disk space, a situation that will worsen with current hardware trends[5]. Be certain that the savings in disk space is worth the performance hit.

3.1.3 Trading computation time for implementation complexity

For many programmers, the greatest attraction of compare-by-hash is that it greatly simplifies management of state information. Many programmers respond to criticism of the correctness of compare-by-hash by pointing out that a more complex implementation which does not use compare-by-hash is more likely to have a bug, and therefore be in-

correct more often than compare-by-hash. After all, `rsync` was originally developed not because using `diff` and `patch` (two common UNIX source management tools) was too slow or used too much bandwidth, but because patch management using `diff` and `patch` was too hard — a statement that is difficult to disagree with. Andrew Tridgell, describing the source of his inspiration for `rsync`, wrote: "It was possible to transfer just the changes by keeping original files around then using a differencing utility like diff and sending the diff files to the other end but I found that very inconvenient in practice and very error prone[25]."

This is a valid point, but in many cases the comparison of error probability is not so straightforward — see Sections 3.2 and 3.3 for more detail. In addition, in many cases the tradeoff includes slower performance. `rsync` is simpler to implement than BitKeeper, a source control system which does not use compare-by-hash, but BitKeeper sends less data over the network than `rsync` in many cases because it keeps state about which files have changed since the last synchronization.

3.1.4 Cryptographically strong checksums needed for other reasons

Finally, an application may have some need to generate cryptographically strong checksums on all data in any case — e.g., to detect malicious tampering — in which case cryptographic hash values are "free" and no performance tradeoff exists. However, in most applications the authors have encountered, compare-by-hash comes first and detection of malicious tampering is what the user gets "for free," and is not really a requirement of the system.

3.2 Is the system usable if hash collisions can be generated at will?

The next question to ask is if the system is still usable when hash collisions occur. Another way to ask this question is, "Can MD4 be used as the hash function?" Some readers might be surprised to hear that the most effective and successful implementations of compare-by-hash use MD4 as the hash algorithm, despite the fact that collisions in MD4 can be trivially generated — Xiaoyun Wang recently claimed that they can be calculated "by hand"[28]. The answer to this question takes into account many different factors.

3.2.1 Why worry about hash collisions?

For any given system using compare-by-hash, hash collisions are inevitable. As we stated in the introduction, and showed in Table 1, it is only a matter of time between a cryptographic hash function's introduction and its eventual demise at the hands of cryptanalytical research. Currently, any system using MD5 and compare-by-hash can, at the whim of those who provide its inputs, suffer a hash collision (see Figure 2 for two different messages with the same MD5 hash value). MD5 was the state-of-the-art in hash functions as recently as 1995; in 2004 no one can state that an MD5 collision is less likely than a hardware error. How long until the same is true of SHA-1?

3.2.2 Is the hash value large enough to avoid accidental collisions?

This may seem obvious, but the hash value must have enough bits to avoid accidental collisions. This is slightly tricky, both because accidental collisions occur with a much smaller number of inputs than expected, and because data set size grows more quickly than expected. Intuitively, one might expect a 50% chance of a hash collision after about half the number of possible outputs are produced; for an n-bit hash value, this would require 2^{n-1} inputs. In reality, it takes only about $2^{\frac{n}{2}}$ inputs to have a 50% chance of collision. This is exemplified by the "birthday paradox" — how many people do you need in a room to have a 50% or greater chance that two of them have the same birthday? The answer is 23 people (assuming that birthdays are uniformly distributed and neglecting leap years). This is easier to understand if you realize that there are $23 \times (22/2) = 253$ different *pairs* of people in the room. Second, data sets grow at an exponential rate. Disk capacity is currently doubling about every 9 months[23]; while actual data stored lags disk capacity, we can reasonably estimate data set size as doubling every year. Given the birthday paradox, this chews up about two bits of hash output per year. If, for example, 16 bits of hash output is insufficient now, then 32 bits of hash output will be insufficient in 16 years or less.

3.2.3 Can malicious users cause damage by deliberately causing hash collisions?

Today, a malicious user can generate a hash collision in a system using compare-by-hash and MD5 — or

MD4, HAVAL-128, MD2, or 128-bit RIPEMD[28]. However, this isn't a concern for many systems using compare-by-hash because deliberate hash collisions can only harm the user generating the collisions. `rsync` still uses MD4 as its hash algorithm because the "address space" created by the person using `rsync` is not shared with any other users. It isn't even shared across files — the hash values are only compared with the hash values in the second copy of the same file. If someone deliberately creates a file containing blocks with identical MD4 checksums and uses `rsync` to transfer it, that user will end up with a corrupted file — one they could create themselves without the bother of generating MD4 hash collisions.

Systems where the hash value "address space" is shared are vulnerable to deliberately generated hash collisions and are not good candidates for compare-by-hash. LBFS[14] and Pastiche[7] are two systems using compare-by-hash where many users share the same address space. A hash collision in LBFS may result in either a corrupted file for some other user or the disclosure of another user's data. The malicious user can control what data is written to the LBFS server by creating data whose value collides with data it predicts another user is going to write.

3.2.4 Can the system detect and recover from hash collisions?

In many systems, compare-by-hash is used only when caching or distributing data; if a collision occurs, the good data still exists somewhere and can be recovered. In `rsync`, collisions in the MD4 checksum on individual blocks are checked for by comparing the MD5 checksum for the entire file (not foolproof, but catches most errors); if the whole file MD5 checksum does not match, the algorithm runs again with a different seed value for MD4. If the user suspects that there has also been a collision in the MD5 checksum, she can detect collisions by doing a full compare of the two files. In the end, the correct copy of the file still exists and is not altered by any hash collisions. In LBFS, on the other hand, a hash collision is not detectable, and results in the permanent loss of the colliding data, absent any outside effort to copy or save the data on the client machine. Finally, any application of compare-by-hash must have some fallback mode in which the user can choose not to use compare-by-hash at all, in case their data does have hash collisions.

3.3 Can the hashes be regenerated with a different hash function at any time?

Systems in which hash values can only be generated once and are stored indefinitely will fall victim to short cryptographic hash function lifetimes and the growth of data sets, and will be unable to recover from hash collisions when they occur. A cryptographic hash value is only an ephemeral and temporary "unique" signature; systems which depend on the collision-resistance of a particular hash function for more than a handful of years are doomed to obsolescence. To combat this, the system must be designed such that the cryptographic hash function can be changed at will, and the hash values of the data being compared can be regenerated when necessary. `rsync` generates hashes anew every time it is run; hash values are used only to address blocks in the cache in LBFS and can also be regenerated as needed.

3.4 Example of correct use of compare-by-hash: `rsync`

`rsync`, for example, is a perfectly reasonable use of compare-by-hash. `rsync` regenerates hashes on each invocation, and automatically uses different seed constants when the whole-file checksums do not match (indicating a collision has occurred in the per-block checksums). The "address space" of hashes is limited only to the blocks in the files the user has requested to be synchronized, keeping collision probabilities much lower than in other systems. Its use is optional, so users who do wish to transfer files full of blocks with the same MD4 checksum can choose a different transport mechanism. The most telling point in favor of `rsync` is that it uses MD4 (for which collisions can be trivially generated) as its cryptographic hash, but it is still usable and useful today because malicious collisions can only be generated by the owner of the files being synchronized. `rsync` does not claim to be faster than diff-and-compress all or even most of the time, it only claims to be simpler, more convenient, and faster for a important subset of file synchronization workloads.

4 Estimating the performance of compare-by-hash

To help estimate whether compare-by-hash provides a performance benefit compared to traditional techniques, we will give an in-depth analysis of its effect on total elapsed time under various conditions. The three comparisons we make are compare-by-

hash vs. whole file, compare-by-hash plus compression vs. whole file compression, and compare-by-hash plus compression vs. diff-and-compress (we will explain these terms further in the following sections). Depending on the relative bandwidth of the hashing function and of the network, the amount of data changed, and the compressibility of the data, compare-by-hash can be either slower or faster than traditional techniques.

4.1 Sample implementation benchmarks

This section gives a few representative real-world measurements of the various hashing, differencing, and compression algorithms to use later on in our analysis. This section is not intended to (and does not) prove the superiority of one technique over another. We examine `rsync`, which uses compare-by-hash when operating on remote files, `diff`, a conventional differencing program optimized for source code, and `xdelta`, a conventional differencing program that works on binary files as well as text.

The benchmarks were run on a 2.8 Ghz Pentium 4 with one hyperthreaded CPU, 1.25 GB of memory, and a 512 KB L2 cache, running Red Hat 9.0 with Linux kernel version 2.4.20-20.9smp. Elapsed time and CPU time were measured using `/usr/bin/time`. Each benchmark was run three times and the time from the third run used; usually the difference between the second and third runs was less than a tenth of second. The system had enough memory that the files stayed in cache after the initial run. The two data sets we compared were the Linux 2.4.21 and 2.4.22 source trees. The bandwidth of the difference generating programs was measured relative to the size of the newest version of the files alone, rather than both the old version and the new version. For example, if we ran a difference generator on two files, where the old file is 2 MB and the new file is 3 MB, and it took one second to run, the bandwidth of that difference algorithm is 3 MB/s. The *hit rate* is the percentage of data that is unchanged; we approximate it by subtracting the patch size from the total size of the new version. For some comparisons, we concatenated the files together into one large file, sorted by file name. For the case where a file tree was differenced recursively, the total size of the target files is that as measured by `du -s`, which takes into account the directory metadata associated with the files. All output was sent to `/dev/null` when measuring the bandwidth of differencing algorithms to help reduce file system performance as a factor.

The bandwidth and CPU time for the various differencing algorithms are in Table 3, and various statistics related to the generated differences are in Table 4. `diff` and `rsync` were run in recursive mode, comparing the 2.4.21 and 2.4.22 versions of the Linux kernel source file by file. The `rsync` runs used the `rsync` daemon and its native network protocol. We ran `rsync` on source trees with different timestamps on unchanged files, because `rsync` normally assumes that files with identical timestamps and sizes are the same and does not compare the contents. While timestamp comparison is a very efficient optimization for `rsync`'s normal workloads, it does not help us compare the efficiency of compare-by-hash because it is doing compare-by-timestamp for unchanged files, not compare-by-hash. We used the log messages from `rsyncd` to estimate patch size. Since `xdelta` has no support for recursively generating differences, we sorted the files by pathname and concatenated them into one large file for each version and ran `xdelta` and `rsync` on the concatenated files. The "`-n`" option to `xdelta` turns off generation of MD5 checksums on the source and target file (this checksum is a sanity check and is not required for correct operation). All command line options used during the runs are shown in Table 3.

Table 5 shows the bandwidth and compression factor of several common compression programs, as measured on the patch from Linux 2.4.21 to 2.4.22 as generated by `diff -ruN`. Table 6 shows the amount of hash data transferred by `rsync` relative to total data size.

2.4.21 vs. 2.4.22, file tree		
Command	Bandwidth	CPU time[a]
diff -ruN	8.65 MB/s	10.7 s
diff -rN	16.9 MB/s	10.4 s
Null update of 2.4.22, file tree		
rsync -r	4.51 MB/s	11.2 s
diff -ruN	132.45 MB/s	1.35 s
2.4.21 vs. 2.4.22, concat. files		
xdelta	21.4 MB/s	8.35 s
xdelta -n	29.2 MB/s	6.18 s
Null update of 2.4.22, concat. files		
rsync	9.20 MB/s	3.32 s
xdelta	95.3 MB/s	1.90 s

Table 3: Performance of differencing utilities on the Linux 2.4.21 and 2.4.22 source trees.

[a]Local CPU time only — for `rsync`, the remote `rsync` process uses significant CPU time.

Command	Patch size	Comp. patch	Comp. factor	Hit rate
diff -rN	9.53 MB	2.10 MB	4.54	0.95
diff -ruN	30.0 MB	6.30 MB	4.75	0.83
rsync -r	24.7 MB	6.35 MB	3.89	0.86
xdelta (concat. file)	8.82 MB	3.55 MB	2.48	0.94

Table 4: Patch size, compressed patch size, compression factor, and percentage of data unchanged for differences generated.

Variable name	Definition	Sample value
B_n	Bandwidth of network	0.100 MB/s
B_h	Bandwidth of hashing algorithm	5 MB/s
B_d	Bandwidth of differencing algorithm	17 MB/s
B_c	Bandwidth of compress algorithm	12 MB/s
B_u	Bandwidth of uncompress algorithm	82 MB/s
B_{cu}	Average bandwidth of compress/uncompress	10 MB/s
S_h	Size of hash	20 bytes
S_b	Size of block	2048 bytes
S_h/S_b	Ratio of hash information to block size	0.008
N_b	Number of blocks	—
C_f	Compression factor	4
H_r	Hit rate (fraction of blocks that are the same)	0.85
M_r	Miss rate (fraction of blocks that have changed)	0.15

Table 7: Variable names, definitions, and sample values.

Program	Bandwidth	Comp. factor
gzip	11.6 MB/s	4.75
gunzip	81.6 MB/s	—
gzip + gunzip	10.2 MB/s	—
zip	11.6 MB/s	4.54
unzip	60.0 MB/s	—
zip + unzip	9.73 MB/s	—
bzip2	2.44 MB/s	5.57
bunzip2	7.81 MB/s	—
bzip2 + bunzip2	1.86 MB/s	—

Table 5: Compression algorithm performance.

Hash, etc. data	Total data	Percent of total
1.49 MB	180.21 MB	0.83%

Table 6: Size of `rsync` hashes and other metadata vs. total data.

4.2 Variable names

For convenience, we define all the variable names used in this section, along with representative values, in Table 7.

4.3 Simplifying assumptions

In order to make the analysis tractable, we make the following simplifying assumptions. Network latency is not a major factor, so we ignore the time for round trips. If network latency dominates network bandwidth, then methods of reducing the amount of data transferred are less interesting. Compare-by-hash is relatively ineffective on compressed files; small changes in compression input often result in widespread changes throughout the file, leaving little commonality in terms of sequences of bytes between the old file and new file. Some techniques have been proposed to reduce this effect[25], but they have not been widely adopted. Startup and tear-down costs and other constant factors are negligible compared to costs proportional to file size. Old and new files are the same size, and the block size (the unit of data over which hashes are generated) is constant. Hash data is incompressible, because the hashes need to have very high entropy (high randomness) to be efficient. Difference application time is dominated by

file system overhead, and can therefore be ignored for purposes of this analysis.

We make somewhat more radical simplifying assumptions about the differencing algorithms, both conventional and compare-by-hash. We assume that the differences generated by compare-by-hash and other differencing algorithms are about the same size, and that the algorithm speed is independent of the amount of differences, even though in reality difference algorithms are sensitive to the amount and type of changes. We should also note that many difference algorithms behave well only in a certain range of file sizes and with a certain type of file data. For example, `diff` is really only efficient for relatively short source code files with insertions and deletions at line break boundaries, while `xdelta` and `rsync` can handle any type of file data with shared sequences of bytes.

4.4 Compare-by-hash vs. whole file

We first examine the total time to synchronize two files using compare-by-hash without compression of the changed blocks versus the time to send the entire uncompressed file. We can use this exercise to define the domain of interest for the ratio of hash function bandwidth to network bandwidth. Obviously, if we can send the whole file faster than we can compute the hash function over it, then compare-by-hash won't be faster.

The time to send the whole file, T_w, is the time to send the data over the network:

$$T_w = \frac{N_b S_b}{B_n}$$

The time to send the file using compare-by-hash, T_h, is the time to compute the hashes, send the hashes, and then send the changed blocks:

$$T_h = \frac{N_b S_b}{B_h} + \frac{N_b S_h}{B_n} + \frac{N_b S_b}{B_n} M_r$$

Note that we are assuming that the hashing of the second copy of the file can be done simultaneously and won't add to the elapsed time. Compare-by-hash is an advantage when:

$$\frac{N_b S_b}{B_h} + \frac{N_b S_h}{B_n} + \frac{N_b S_b}{B_n} M_r \leq \frac{N_b S_b}{B_n}$$

Multiply through by $B_n/N_b S_b$, let $S_h/S_b = 0.008$ (from Table 7), move M_r to the right side and remember that $(1 - M_r) = H_r$, and we get:

$$\frac{B_n}{B_h} + 0.008 \leq Hr$$

In all the cases we examine, S_h/S_b has a negligible effect on the total time, so we ignore it from now on (giving an advantage to compare-by-hash). In this case, it raises the required hit rate by less than 1%. This gives us:

$$\frac{B_n}{B_h} \leq H_r$$

This equation gives us the expected result that compare-by-hash can only be a win when the network bandwidth is lower than the hash algorithm bandwidth, because the maximum hit rate is 1. The lower the network bandwidth relative to the hash bandwidth, the lower the hit rate needs to be for compare-by-hash to be a win.

Let's apply this equation to a real-world case where compression isn't possible: using compare-by-hash to write out to disk only dirty subpages of a superpage being evicted from memory[16]. Compression can't be used because the disk hardware doesn't understand it, and the memory management unit provides only one dirty bit per superpage, so our only choices are using compare-by-hash to write out the dirty subpages, and writing out the entire superpage. In this case, the "network bandwidth" is the bandwidth to disk, and the hash bandwidth is the bandwidth of SHA-1 on inputs of size 8 KB, the smallest page size on the system studied. On our benchmark system, we measured $B_n = 32$ MB/s and $B_h = 124$ MB/s.

$$\frac{B_n}{B_h} = \frac{32}{124} = 0.26$$

A hit rate of at least 26% is required for compare-by-hash to be faster than writing the plain data; a dirty superpage must contain on average fewer than 74% dirty subpages for compare-by-hash to be a win. This assumes that a superpage is only hashed if it is written to; in this implementation all superpages were hashed on the initial read from disk, which contributed to a degradation in performance of 15–60%[16]. The authors suggest computing the initial hashes of superpages only when the CPU is idle. Even with this optimization, unless the dirty superpage is more than 26% clean, it's still faster just to write out the entire page rather than finding and writing out only the dirty subpages (neglecting the not inconsiderable fixed per-I/O cost of writing out many small pages versus one large page).

4.5 Compare-by-hash plus compression vs. whole file compression

We next compare compare-by-hash and compression of the changed blocks versus compressing and sending the entire file.

The elapsed time for compare-by-hash plus compression, T_{h+c}, is the time to compute the hashes plus the time to compress the changed blocks plus the time to send the compressed changes:

$$T_{h+c} = \frac{N_b S_b}{B_h} + \frac{N_b S_b}{B_{cu}} M_r + \frac{N_b S_b}{B_n C_f} M_r \qquad (1)$$

The elapsed time for compressing and sending the entire file, T_c, is the time to compress the entire file plus the time to send the compressed file:

$$T_c = \frac{N_b S_b}{B_{cu}} + \frac{N_b S_b}{B_n C_f}$$

Compare-by-hash is a win when:

$$\frac{N_b S_b}{B_h} + \frac{N_b S_b}{B_{cu}} M_r + \frac{N_b S_b}{B_n C_f} M_r \leq \frac{N_b S_b}{B_{cu}} + \frac{N_b S_b}{B_n C_f}$$

Rearranging, we get:

$$(\frac{B_n}{B_h})/(\frac{B_n}{B_{cu}} + \frac{1}{C_f}) \leq H_r \qquad (2)$$

Figure 1 is a plot of the minimum hit rate necessary for compare-by-hash to be faster for ratios of network bandwidth to compression and compare-by-hash bandwidth ranging from 0 to 1, and with $C_f = 4$ (see Tables 5 and 7). For example, if the compression factor is 4, the compression bandwidth is 5 times faster than the network ($B_n/B_{cu} = 0.2$), and the compare-by-hash algorithm is 3 times faster ($B_n/B_h = 0.33$), then using Equation 2, the hit rate must be at least:

$$(0.33)/(0.2 + 0.25) = 0.73$$

for compare-by-hash to be faster. In the graph, values to the right of line indicating $H_r = 1$ are unattainable and compare-by-hash is always slower in this case.

Let's set the hit rate to some interesting values and look at the boundaries where the advantage switches

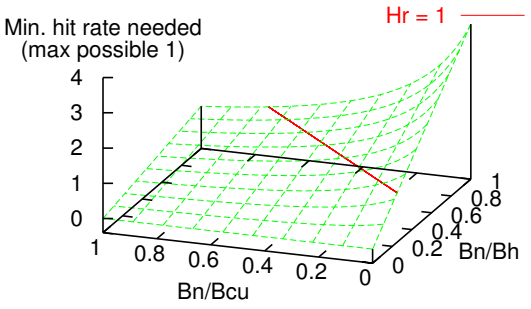

Figure 1: Minimum hit rate for compare-by-hash plus compression to be faster than compression alone, as a function of the ratio between algorithm bandwidths and network bandwidth, with $C_f = 4$.

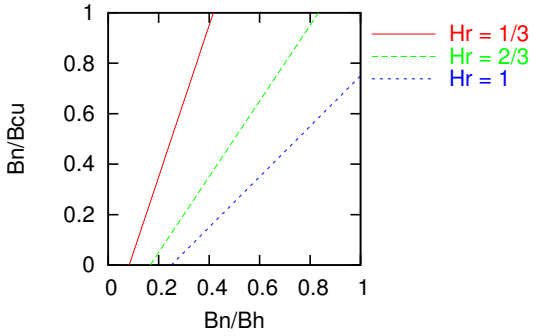

Figure 2: Crossover points for sample hit rates for compare-by-hash plus compression vs. whole file compression.

from compare-by-hash to pure compression. Rearranging Equation 2, we can plug in different hit rates and find out at what relative bandwidths compare-and-hash is faster than diff-and-compress:

$$\frac{B_n}{B_h H_r} - \frac{1}{4} \leq \frac{B_n}{B_{cu}}$$

Figure 2 shows the lines for H_r equal to 1, $\frac{2}{3}$, and $\frac{1}{3}$. Below and to the right of each line, pure compression is a win at any hit rate below that for the relative bandwidths of the compression function, hash function, and network.

Let's look at a real-life example of compare-by-hash versus whole file compression. To isolate the effect of compare-by-hash on LBFS performance, the

designers benchmarked a version of LBFS without compare-by-hash, but with file leases and compression. On a network with an upload bandwidth of 384 Kbps, LBFS was 5–6% faster on the two benchmarks, and more than 5 times as fast as its compression-only cousin on one benchmark (which involved writing many separate copies of the same file). When network bandwidth was raised to 10 Mbps, LBFS and its compression-only cousin were actually slower than traditional file systems which send the entire file data[14].

Because compression can be done once in advance, it can be amortized over many downloads (as is common in many software distribution server workloads). Let us ignore the cost of compressing the file and consider compare-by-hash plus compression compared to downloading and uncompressing the whole file, which is Equation 2 with B_{cu} replaced by B_u:

$$(\frac{B_n}{B_h})/(\frac{B_n}{B_u} + \frac{1}{C_f}) \leq H_r$$

Plugging in $B_n = 0.100$ MB/s, $B_h = 5$ MB/s, $B_u = 82$ MB/s, and $C_f = 4$, we get:

$$(\frac{0.1}{5})/(\frac{0.1}{82} + \frac{1}{4}) = 0.08$$

Even with compression time factored out, we need only a 8% hit rate for compare-by-hash to be faster than sending and uncompressing the entire compressed file under these conditions. With common ratios of wide area network bandwidth to compare-by-hash bandwidth, compare-by-hash is almost always faster than sending the entire compressed file.

4.6 Compare-by-hash plus compression vs. diff-and-compress

Next we will add state to the equation and compare compare-by-hash and compression versus differencing and compression. This requires that each end has stored an earlier version of the file that it can use to generate and apply the differences.

The time for compare-by-hash plus compression, T_{c+h}, is as in Equation 1. The time for diff-and-compress, T_{c+d}, is the time to create a patch, compress and uncompress it, and send the compressed patch:

$$T_{c+d} = \frac{N_b S_b}{B_d} + \frac{N_b S_b}{B_{cu}} M_r + \frac{N_b S_b}{B_n C_f} M_r \qquad (3)$$

Using Equations 1 and 3, compare-by-hash plus compression versus diff-and-compress is a win when:

$$\frac{N_b S_b}{B_h} + \frac{N_b S_b}{B_{cu}} M_r + \frac{N_b S_b}{B_n C_f} M_r \leq \frac{N_b S_b}{B_d} + \frac{N_b S_b}{B_{cu}} M_r + \frac{N_b S_b}{B_n C_f} M_r$$

Canceling common terms and multiplying by $B_d B_h / N_b S_b$:

$$B_d \leq B_h$$

With conventional differencing bandwidths of 8.65–29.2 MB/s and compare-by-hash bandwidths of 4.51–9.20 MB/s (from Table 3), diff-and-compress clearly beats compare-by-hash.

What this equation tells us is that compare-by-hash is only a win when the bandwidth of the differencing algorithm is lower than the bandwidth of the hash algorithm. In other words, compare-by-hash is only a win if we can calculate differences between two files remotely faster than we can locally — which seems unlikely.

Another way to look at this is to compare the `xdelta` and `rsync` algorithms for finding identical blocks. Both compute a fast but weak "rolling checksum" at every byte boundary in the file, and then compare these weak checksums (the algorithm for `xdelta` was inspired by `rsync`.). At places where the weak checksum matches, `rsync` computes a truncated MD4 hash, but `xdelta` merely has to compare the two blocks of data sitting in memory. `rsync` must keep computing checksums until there is only an infinitesimal chance of collision, but `xdelta` only has to compute until it reduces the number of full block compares it has to do to a reasonable level. As another example, the VMWare ESX Server implements content-based page sharing as a memory usage optimization[27]. It uses a fast (not cryptographically strong) 64-bit checksum to generate addresses for shared pages. On the largest possible memory configuration of 64 GB, the collision rate is approximately 0.01%. If a hash collision does occur, the colliding pages are simply declared ineligible for sharing. Using a cryptographically strong hash to reduce the collision rate would reduce performance in return for less than a 0.01% savings in memory.

Program	Bytes sent	Normalized
diff -e + bzip2	7594	1.00x
diff -e + gzip	7934	1.04x
xdelta	9543	1.26x
rsync -z	122956	16.20x

Table 8: Bytes transferred while updating Debian packages and sources metadata files from December 11–12, 2003.

Command	Time	Data sent
rsync -rz	16 s	7.0 MB
rsync -rz (no timestamps)	41 s	7.0 MB
diff -ruN	37 s	6.3 MB
xdelta (concat. file)	8 s	3.6 MB

Table 9: Total update times from Linux 2.4.21 to Linux 2.4.22.

4.7 Real-world examples where compare-by-hash is slower

Let's look at a real-world example where compare-by-hash is not the optimal solution. The Debian package update system requires downloading a local copy of the package metadata to the client machine. This is currently done by downloading the entire gzipped file if any of the package metadata has changed since the last download of the file. Robert Tiberius Johnson measured the bandwidth necessary to update the package metadata file using `diff` and `bzip2`, `diff` and `gzip`, `xdelta`, and `rsync`. He found that `diff` and `bzip2` used the least bandwidth with `gzip` and `xdelta` close behind, but `rsync` used 12 times the bandwidth of the best solution[11]. Peter Samuelson reran this comparison on both the packages and sources metadata files with `diff -e`, which generates the minimal output necessary to apply a patch, on the December 12, 2003 updates to the packages and sources files; the results are in Table 8. In addition to transferring more than 16 times the minimum necessary data, `rsync` puts far more CPU load on the server than serving pre-generated patches. For this application, `rsync` is not the right choice.

As another real world example, we updated our Linux 2.4.21 source tree to Linux 2.4.22 using `diff`, `gzip`, and `patch` and compared that to using `rsync` and `rsyncd`. We did the update over the loopback interface, since network bandwidth is irrelevant for this comparison. The results are summarized in Table 9. The total time for updating using `diff -ruN`, `gzip`, and `patch` was 37 seconds. If we assume that the patch is generated in advance and downloaded many times, and count only the time to download, uncompress, and apply the patch, it takes 14 seconds to update. The total time for `rsync -rz` to update the same two source trees is 41 seconds when timestamps on identical files do not match, and 16 seconds when timestamps do match.

With any reasonable differencing algorithm, diff-and-compress will always be faster than compare-by-hash, even without accounting for the amortization of the costs of generating differences and compressing over multiple transfers of the data.

4.8 When is compare-by-hash faster?

Both compare-by-hash and diff-and-compress can only provide performance improvement when the network bandwidth is lower than the bandwidth of the hashing or diff-and-compress algorithm.

Compare-by-hash vs. whole file. Without any compression at all, compare-by-hash is faster than sending the whole file when the ratio of network bandwidth to hash algorithm bandwidth is lower than the percentage of unchanged data. At low network bandwidths, compare-by-hash will almost always be a win for this case, but when moving data around on relatively high-speed networks (such as a local area network or a SCSI bus), compare-by-hash is much less competitive.

Compare-by-hash plus compression vs. whole file compression. When compression can be used, compare-by-hash plus compression of differences beats transferring the entire compressed file at nearly any hit rate higher than the ratio of the network bandwidth to the hash algorithm bandwidth. For a network bandwidth of 100 KB/s, a hash bandwidth of 5 MB/s, a compression bandwidth of 10 MB/s, and a compression factor of 5, we only need a hit rate of about 10% for compare-by-hash to always be faster than sending the whole compressed file.

Compare-by-hash plus compression vs. diff-and-compress. If we can generate the differences between versions locally and send the compressed patch, compare-by-hash plus compression is always slower. This is because we can always generate differences between two local versions of the data faster than we can do it between a local version and a remote version.

Compare-by-hash without compression is sometimes faster than sending the whole file, compare-by-hash plus compression is usually faster than sending the entire compressed file, and compare-by-hash plus

compression is never faster than diff-and-compress.

5 Alternatives to compare-by-hash

More universal difference and compression algorithms are an interesting research topic, now that CPU time is seldom a limiting factor in today's systems. Compare-by-hash has shown how much CPU time we are willing to spend on some applications; for less CPU time we can make substantial improvements in differencing and compressing data more efficiently. Josh MacDonald's `xdelta` delta generating program and the Xdelta File System[13] deserve more attention. A network file system similar to LBFS[14] but using `xdelta` to generate differences against a locally cached copy of the version of the file on the server would have excellent performance over low-bandwidth networks. This is not entirely trivial to implement: it would require file leases, a feature of NFSv4[17] and LBFS, and efficient detection of similarities between different files as well as the same file. Also interesting are more specific difference and compression algorithms optimized for particular data formats, such as line-wrapped ASCII text, common package formats, and ELF binaries.

Better version control management specialized for specific data types is extremely interesting. For example, BitKeeper[4] is difficult to beat when it comes to distributed source control management for software, but it does not excel at similar functions like package distribution or efficiently storing version history of documentation.

One commonly suggested substitute for compare-by-hash is the concatenation of outputs from multiple hash functions: two different cryptographic hash functions, the same hash function with two sets of different initial constants, or the same hash function run both forwards and backwards on the data. The end result is still compare-by-hash, just with a slightly different hash function, and all the same arguments for or against it still apply. As long as the number of bits in the input is smaller than the number of bits in the output, hash collisions are more than possible, they are guaranteed.

6 Conclusions

Compare-by-hash is a useful technique when hash collisions are not fatal, different users don't share the address space, and compare-by-hash provides some benefit because CPU time is cheap, bandwidth is scarce, or statelessness is important. If compare-by-hash is used as a performance optimization, a convincing argument must be made for why conventional differencing techniques do not offer better performance or are not an option.

For many existing uses of compare-by-hash, including software updates over the Internet and low-bandwidth network file systems, diff-and-compress provides shorter total transfer time and lower server load. For example, a Debian package server using `rsync` to update the package metadata file would transfer an order of magnitude more data than if it used `diff` and `gzip` and would use more CPU time on both the client and server[11] (see Table 8). Interesting applications of compare-by-hash in the area of operating systems appear to be scarce, but are more common in applications.

7 Acknowledgments

This paper benefited greatly from discussions and comments from many people, including Jonathan Adams, Matt Ahrens, Jeff Bonwick, Bryan Cantrill, Miguel Castro, Whit Diffie, Marius Eriksen, Barry Hayes, Chris Lahey, Larry McVoy, Bill Moore, Steven Northcutt, Dave Powell, Peter Samuelson, Kelly Shaw, Marianne Shaw, Bart Smaalders, Niraj Tolia, Vernor Vinge, and Cynthia Wong.

References

[1] Adam Back. Hashcash - amortizable publicly auditable cost-functions. http://www.hashcash.org/papers/hashcash.pdf, 2002.

[2] Mihir Bellare and Tadayoshi Kohno. Hash function balance and its impact on birthday attacks. In *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 401–418. Springer-Verlag Heidelberg, 2004.

[3] Eli Biham and Rafi Chen. New results on SHA-0 and SHA-1. http://www.iacr.org/conferences/crypto2004/C04RumpAgenda.pdf. Announced in live webcast.

[4] BitMover, Inc. BitKeeper - the scalable distributed software configuration management system. http://www.bitkeeper.com.

[5] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[6] Florent Chabaud and Antoine Joux. Differential collisions in SHA-0. In *Proceedings of CRYPTO '98, 18th Annual International Cryptology Conference*, pages 56–71, 1998.

[7] Landon P. Cox, Christoper D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[8] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 1999.

[9] Charles Forsyth. [9fans] compare-by-hash. http://bio.cse.psu.edu/~schwartz/9fans/2003-June.txt.

[10] Val Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[11] Robert Tiberius Johnson. Re: Rsyncable GZIP (was Re: Package metadata server). http://lists.debian.org/debian-devel/2002/debian-devel-200204/msg00502.html.

[12] Antoine Joux, Patrick Carribault, and Christophe Lemuet. Collision in SHA-0. Announced in sci.crypt on 12 August 2004.

[13] Joshua P. MacDonald. File system support for delta compression. Master's thesis, University of California at Berkeley, 2000.

[14] Athicha Muthitacharoen, Benjie Chen, and David Maziéres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[15] National Institute of Standards and Technology. *FIPS Publication 180–2: Secure Hash Standard*, 2002.

[16] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[17] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. In *Proceedings of the 2nd Internation System Administration and Networking Conference (SANE2000)*, 2000.

[18] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, 2002.

[19] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[20] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[21] Jonathan S. Shapiro and John Vanderburgh. CPCMS: A configuration management system based on cryptographic names. In *Proceedings of the 2002 USENIX Technical Conference, FREENIX Track*, 2002.

[22] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. In *Proceedings of the 2000 ACM SIGCOMM Conference*, 2000.

[23] Jon William Toigo. Avoiding a data crunch. *Scientific American*, May 2000.

[24] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Adrian Perrig, and Thomas Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003.

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

[26] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR–CS–96–05, Australian National University, 1996.

[27] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[28] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128, and RIPEMD. http://eprint.iacr.org/2004/199.pdf.

[29] Matt Welsh. Conference Reports. *;login:*, 28(4):70–71, August 2003.