

# Evaluating the Usefulness of Content Addressable Storage for High-Performance Data Intensive Applications

Partho Nath<sup>\*</sup>  
Cisco Systems, Inc.  
170 West Tasman Drive  
San Jose, CA 95134  
nath@cisco.com

Bhuvan Urgaonkar Anand Sivasubramaniam  
Department of Computer Science & Engineering  
The Pennsylvania State University  
University Park, PA 16802  
{bhuvan,anand}@cse.psu.edu

## ABSTRACT

Content Addressable Storage (CAS) is a data representation technique that operates by partitioning a given data-set into non-intersecting units called chunks and then employing techniques to efficiently recognize chunks occurring multiple times. This allows CAS to eliminate duplicate instances of such chunks, resulting in reduced storage space compared to conventional representations of data. CAS is an attractive technique for reducing the storage and network bandwidth needs of performance-sensitive, data-intensive applications in a variety of domains. These include enterprise applications, Web-based e-commerce or entertainment services and highly parallel scientific/engineering applications and simulations, to name a few.

In this paper, we conduct an empirical evaluation of the benefits offered by CAS to a variety of real-world data-intensive applications. The savings offered by CAS depend crucially on (i) the nature of the data-set itself and (ii) the chunk-size that CAS employs. We investigate the impact of both these factors on disk space savings, savings in network bandwidth, and error resilience of data. We find that a chunk-size of 1 KB can provide up to 84% savings in disk space and even higher savings in network bandwidth whilst trading off error resilience and incurring 14% CAS related overheads. Drawing upon lessons learned from our study, we provide insights on (i) the choice of the chunk-size for effective space savings and (ii) the use of selective data replication to counter the loss of error resilience caused by CAS.

## Categories and Subject Descriptors

E.2 [Data Storage Representations]: Object representation

## General Terms

Performance, measurement, management, reliability

---

<sup>\*</sup>Most of this work was done while author was a PhD student at the Pennsylvania State University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HPDC'08, June 23–27, 2008, Boston, Massachusetts, USA.  
Copyright 2008 ACM 978-1-59593-997-5/08/06 ...\$5.00.

## Keywords

Content Addressable Storage, Compression, Caching

## 1. INTRODUCTION

High-performance distributed applications that must store, retrieve, manipulate and transfer large amounts of data abound. Data-sets managed by such applications have increased steadily to the order of terabytes [5, 6, 34]. TeraShake [34], a grid application that periodically checkpoints its data for recovery or visualization purposes generates close to 50 TB of data from one run. Such applications impose tremendous strain on the infrastructure in several ways. Continuing with TeraShake as an illustrative application, at least three requirements from the storage/networking infrastructure emerge. First, such applications desire the ability to quickly transfer such large volumes of data from scratch volumes or in-memory buffers on the local node to remote storage. If the network bandwidth is insufficient, researchers are forced to minimize data generation and/or collection to ensure that the buffer or storage capacity of the local node is not overwhelmed. Second, the storage infrastructure must provide enough disk bandwidth at the data repository site to keep up with the flood of incoming data. Finally, there must be enough spare storage capacity to house all experimental data, perhaps even across multiple runs.

A data management technique that can reduce the above needs can potentially change a task (like generation of “on-the-fly” graphics for TeraShake) from the realm of infeasible to feasible. Similar issues exist in the context of data from enterprise-class and Web-based Internet-scale applications, that must often deal with terabyte or even petabytes [20, 26, 35, 36, 37]. In all these systems, reducing the amount of data that must be transferred, stored, and managed is a crucial requirement with implications on application performance, storage costs, and administrative overheads.

Two complementary and well-researched approaches for reducing the amount of on-disk data and volume of network traffic are caching and compression. Caches range from those within operating systems and applications to those within dedicated proxy nodes and are an extremely well-studied topic. Compression techniques, besides their direct benefit of reducing the size of the data-set, also offer the indirect benefit of increasing the fraction of data that can be effectively cached. Many data-intensive storage products already employ various forms of data compression [16, 19, 48]. Content Addressable Storage (CAS) has emerged as an increasingly popular alternative to traditional compression techniques in recent systems literature [27, 22, 47, 24, 43, 32].

CAS operates by partitioning a data set into non-intersecting blocks called *chunks* and then building efficient meta-data to recognize chunks occurring multiple times. This allows it to eliminate

duplicate instances of such chunks, resulting in more efficient use of disk space. The same meta-data allows CAS to compare data residing at two repositories – a local cache and a remote store. This enables CAS to make efficient use of the network by holding and then transmitting from the local cache only the data that did not previously exist on a backing store. It is important to note that a CAS-based cache is unique in that it can optimize away not just reads, but also writes for content that already exists on the backing store. Since CAS does not modify the data content itself, a CAS compressed dataset can be further subjected to standard compression techniques like gzip [49], if greater savings are desired.

Thus CAS is a technique to enable *deduplication* of data. The amount of savings achieved by CAS crucially depends on the chunk size - smaller chunk sizes provide more opportunities for identifying duplicates at the cost of increased meta-data storage and processing overheads. The choice of a small chunksize is not enough to guarantee a certain amount of space savings. Data deduplication using CAS relies on the existence of significant duplicate chunks in data - an intrinsic property of the data-set itself. Therefore, the benefits that CAS has to offer are intimately dependent on the data-set. A side effect of removing duplicate data chunks is the magnification of corruption or loss of a single data chunk in a CAS repository, making reliability a concern.

Unfortunately, minimal investigation of these three issues has been undertaken, especially in the context of two very important classes of applications – enterprise applications and scientific or grid applications. Available literature has mostly explored these issues for web-pages, home-directories and source-code files [41, 38, 30, 9, 44]. We believe that CAS can significantly impact the two most important components of any enterprise and scientific/grid-computing application, namely the network and the backing store. In this paper, we attempt to quantify the impact of CAS on *real-world data* obtained from enterprise applications and scientific applications.

We note that there is a fairly large entry-barrier to undertaking such a study. First, we observe that the benefits of CAS depend on the data in question, forcing the use of real application data, as opposed to synthetically generated data traces. Unfortunately, most readily available file-system traces generated from the execution of real applications lack requisite information to identify the actual content of the data. Thus for a CAS-based study, we attempt to first find and then run real world applications on a more amenable file-system, in order to generate traces with adequate information.

We used a publicly available file-system – CAPFS, the Content-Addressable Parallel File-System [47] to carry out our investigations. A unique consequence of this choice is that we are in a position to comment on CAS related run-time performance overheads obtained on an actual content-addressable file system. This is another significant contribution of this paper.

Our empirical evaluation provides a number of valuable insights, both positive and negative, into the applicability and utility of CAS for real-world applications. We find that a chunksize of 1 KB or 2 KB provides best results for optimizing storage requirements. More interestingly, we observe that at times, even though storage space reduction is not possible (dataset contains mostly unique chunks), network bandwidth reductions to the order of 5% or more, even when a standard caching policy would require transmission of all data to the remote backing store. We also find that CAS makes the data highly vulnerable to loss, but a selective replication using 5% of additional storage space can offset the loss probability while keeping the storage space gains. We choose to ignore the issue of hash-based collisions by agreeing with the view that they may not pose as significant a threat [28, 8] as perceived earlier [21].

## 2. CAS OVERVIEW

Content addressable storage (CAS) is a data management technique that can potentially improve storage and network efficiency. CAS operates on data by dividing it into non-intersecting blocks or chunks, and eliminating the common chunks as described next. First, a cryptographic hash is generated to uniquely identify/name each chunk. The use of a cryptographic hash function, typically SHA1, ensures that two chunks will hash to the same name if and only if their contents are identical. By storing only unique hashes (and their corresponding data chunks) in the CAS repository, duplicate chunks in the data are eliminated. As a trivial example, two chunks, both comprising of zeroes would hash to the same name. Hence by storing just one instance of the above chunk, savings in storage space and network bandwidth is achieved.

The compression ratio thus achieved is highly dependent on the data being stored. A dataset with high *commonality* will have some chunks occurring a large number of times. On the other hand, a dataset with hardly any commonality would be comprised almost wholly of unique data chunks. Thus the commonality in data directly affects the savings achievable by CAS. The sources of commonality could be numerous. For example, backup or checkpoint storage systems like Internet Suspend/Resume([32]) can have significant time invariant data being stored multiple times. Another example could be a data-store housing trace data from multiple runs of an application.

In addition to the commonality inherent to the data, the ability of CAS to exploit it depends on the choice of chunksize. In the previous example, consider the case when two runs of the same application generating a single, *nearly* identical data chunk, differing by a single bit. In this case, since the chunks are not identical, CAS cannot obtain any savings. However, by using a smaller chunksize, the data common to the two chunks could have been eliminated. The choice of a chunksize is a tunable parameter that we explore in this paper.

Use of CAS also gives another advantage – *global naming*. A user can present *any* CAS repository with a hash and check if the data block is present in that repository. Addressing a block by its content-hash thus provides a way to globally name a block – independent of the namespace where the block may be housed, independent of geographical or server IP address locations etc. By removing these constraints on the *naming* of the block, issues such as retrieval of data and location based caching are simplified.

A file-system using a CAS-based store or back-end needs to translate requests for a file offset into requests for data corresponding to a chunk name (hash). For this purpose, an additional meta-data structure called a *recipe* must be maintained by the file-system. A recipe is a sequential list of chunk names (hashes), corresponding one-to-one with the file data (chunks) that make up the data file. For example, if on a file-system with chunksize 4 KB, there exists a 12 KB file, then it would need to be stored as three chunks. If the hashes for these chunks are  $h_1, h_2$  and  $h_3$  respectively, then the recipe for the file would be  $\{h_1, h_2, h_3\}$ . In this example, a request to read bytes 4097 to 5097 would generate a request to read the data for the chunk with hash  $h_2$ . The use of a recipe can provide for bandwidth optimizations or consistency needs [12, 43, 22, 32] and for signing/verifying data [31]. On the downside, additional meta-data (recipe) needs to be maintained for every file. In the above example, even if the file were entirely composed of zeros, and hence  $h_1, h_2$  and  $h_3$  were identical, the recipe would still need to store a hash for all three chunks. At small chunksizes, this can be a significant overhead that eats away at the savings obtained by the use of CAS.

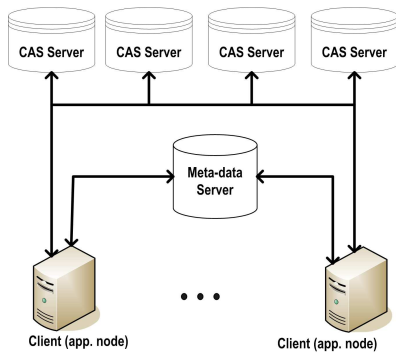


Figure 1: Simplified CAPFS system architecture.

### 3. METHODOLOGY

Currently available public domain file-system traces are inadequate to undertake a CAS based analysis. Most traces file-system, or disk level traces record the timestamp, block number or {file-name,offset} tuple requested and type of operation (a *read* or *write*). To undertake a CAS based analysis of the data, knowledge of the block number or offset being requested is of no use – what is (minimally) required is a *trace* of the same read/write operations and the content of the read/write operation or its equivalent (the chunk names/hash values). Note that for an accurate analysis, this trace needs to be collected *live*, while the application is running. This is because due to over-writes, the chunk-names cannot be generated by post-processing the final file data, at the end of the application execution.

Unfortunately, such a trace, collected at a particular chunksize, does not permit a CAS based analysis at any other chunksize. To synthesize a trace for a smaller chunksize, it would be impossible to generate the sequence of hashes that would have been requested, since the data/content is not available. One might tediously be able to generate a trace for a larger chunksize by correlating file offsets or block numbers with the hashes being requested. However, even then this would not be accurate since if the application was run natively with the larger chunksize, then the sequence of reads/writes observed would have been different from what was observed at a smaller chunksize size owing to reads/writes straddling chunk boundaries.

Hence, we need an observational infrastructure that can, (i) capture hashes for read/write operations *live*, and (ii) can be configured to collect the trace at different chunksizes. To accomplish this, we developed CAPFS, the Content Addressable Parallel File-System [47] and use it here for our empirical study.

#### 3.1 CAPFS: A Content Addressable Parallel File-System

As explained above, we need a mechanism to run the application(s), calculate and record the SHA1 hashes of the reads and writes. For this reason we chose to use CAPFS, a Content Addressable Parallel File System designed and implemented by us [47]. Figure 1 shows the simplified CAPFS system architecture. The client nodes mount the file-system exported by the meta-data server. The CAS servers form the back-end storage and house the file-system data. The blocks are striped in parallel across the CAS servers for performance. Each CAS server exports two primitives: i) get(hash) and ii) put(hash,data). To implement these primitives, the CAS server runs a simple database that mirrors the above get and put operations. When the application is run on the client nodes, meta-data accesses are synchronized at the meta-data server and all

Name	Type	Description	Size
gene-dbase	Static	Contains genes from GeneBank	103 MB
btio	Live	Part of NAS parallel benchmark	400 MB
bssn	Live	The BSSN PUGH benchmark	1.6 GB
heat-solver	Live	Generic heat-solver	106 MB
dbt2	Live	OSDL Database Test 2	306 MB

Table 1: Benchmarks used.

data read/write requests are sent directly to the CAS servers.

In our experimental setup CAPFS was configured to use a single CAS server with POSIX consistency semantics. The CAS server was configured to house a Berkeley DB [3] stored in a btree format, indexed by 20 byte SHA1 hashes corresponding to data blocks. Using a single CAS server made it easy to log the get/put requests for hashes as they arrived. Similar logging was enabled at all the client nodes running the benchmark. The trace thus generated at the client and server end, helped us reconstruct the execution of the benchmark. CAPFS was run with various chunksizes and the traces thus obtained were analyzed. By instrumenting the CAPFS client and server daemons we also gained valuable insight into other performance issues including SHA1 computation overhead and lookup overheads at the CAS servers.

The experiments were run on an IBM pSeries 20-node cluster. Each each node has a dual hyper-threaded Xeon clocked at 2.8 GHz, equipped with 1.5 GB of RAM and a 36 GB SCSI disk. The nodes run Redhat 9.0 with Linux 2.4.20-8 kernel compiled for SMP use and are connected by a gigabit ethernet network.

#### 3.2 Benchmarks

A synthetically generated dataset may either contain too little commonality (a dataset made from random data) or might have too many regular patterns (a dataset made by repeating same data). With this in mind, we use the five *real-world* datasets listed in Table 1 as benchmarks for studying the performance of CAS. Except for the *gene-dbase* benchmark, all other traces contain the sequence of read/write operations as carried out during the application run, and are hence labeled as *live* datasets. For these datasets, we can analyze the impact of CAS on network bandwidth and effectiveness of caches.

The *gene-dbase* dataset contains a few randomly chosen genomes from the NCBI GenBank database [33]. This dataset was statically analyzed as-is for inherent commonality and hence is not the result of a live application, as indicated in Table 1. The BTIO benchmark [2] is based on a computational fluid dynamics (CFD) code that uses an implicit algorithm to solve the 3D compressible Navier-Stokes equations. We ran the A class version of the benchmark using the full-mpio version. The *bssn* benchmark is a numerical relativity code using finite differences on a uniform grid [1]. The *heat-solver* is a standard heat solver written in Fortran using MPI. The *btio*, *bssn* and *heat-solver* benchmarks run in iterations, create data periodically and update either the same data or generate new data during the application lifetime. These three scientific applications can be configured to run on a single node or multiple nodes. The *dbt2* benchmark is the OSDL Database Test 2 benchmark [17]. It is a TPCC-like benchmark that loads tables into a mysql server at startup and runs queries on the tables for a specified time. We configured the benchmark to run with three warehouses (with default values) and ran queries on all three warehouses for five minutes. The mysql server was configured to store its innodb tablespace on the file-system exported by CAPFS. All other tables that were loaded by mysql were also stored on this file-system. The doublewrite buffer was stored on a local scratch file-system. We trace

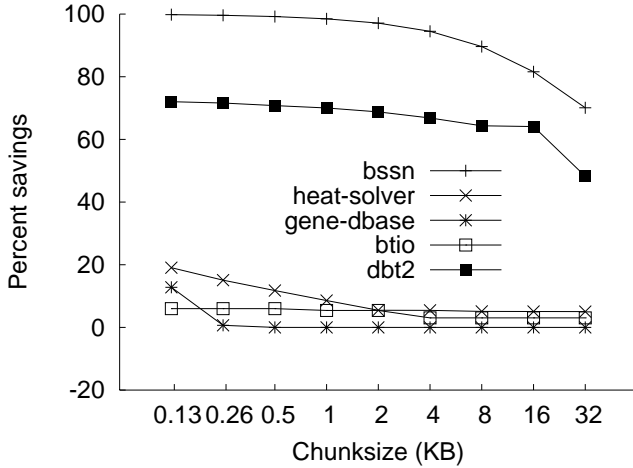


Figure 2: Savings in storage space as a function of chunk size.

the execution of all the above *live* applications by instrumenting the CAPFS file-system. The trace logs thus obtained help us to do post-mortem analysis on the data. The above five datasets cover data from different sources and of varied size, hopefully helping us make informed conclusions.

## 4. CAS: PROS

In this section we look at the advantages of using CAS - namely the savings in storage space and the savings in network bandwidth from using a CAS-based cache on a client.

### 4.1 Savings in Storage Space

CAS has a direct impact on the amount of space required to store data. We compare the savings obtained by the use of CAS against a default non-CAS case where the data is stored on a regular file-system with no in-place writes.

#### 4.1.1 Impact of chunksize

The chunksize in use for a CAS-based store is a tunable parameter that affects its performance. The curves in Figure 2 study the effect of chunksize on storage space required. The common observable trend for all the datasets is that savings in storage space decrease with increase in chunksize. The reasoning behind this is intuitive. Imagine two blocks differing only at the first bit. At the current chunksize, both chunks will have a different hash, forcing the CAS store to house two chunks, thus providing no space savings. On halving the chunksize, we get four chunks, two of which are identical, yielding a 25% savings in space. Hence, smaller chunksizes can identify more commonality (when it exists at all), and thus yield higher savings.

Figure 2 shows that all the live applications benefit from CAS. The *bssn* benchmark at a 128-byte chunksize achieves 99% disk-space savings. As the chunksize increases to 1 KB and 2 KB, the savings fall only marginally to 98% and 97% respectively. The savings for the *heat-solver* data decrease from 19% at a 128-byte chunksize to a steady value of 5% at a 2 KB chunksize and beyond. The use of CAS saves 6% at a 128 byte chunksize for *btio*, and about 3% at chunksizes greater than 4 KB.

The *dbt2* benchmark also benefits tremendously from the use of CAS, with the savings declining marginally from 72% at a 128-byte chunksize, to 64% at a 16 KB chunksize and declining further from there on. The reason behind the gentle decline in savings till the 16

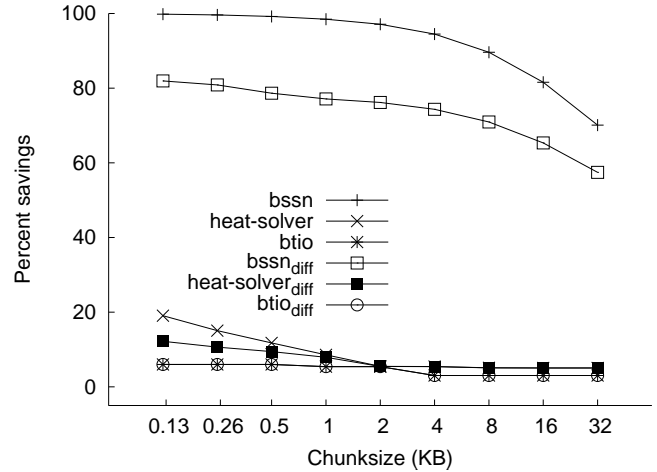


Figure 3: Identifying commonality in data due to iterative behavior.

KB chunksize lies in the fact that the innodb tablespace housed on the CAS store uses an internal page size of 16 KB. As a result, all chunksizes less than or equal to this value extract the same amount of commonality from the tablespace data. This also explains the sharp drop in savings beyond the 16 KB chunksize. The *gene-dbase* data has some exploitable commonality (12%), only at the smallest chunksize of 128-bytes.

#### 4.1.2 Applications benefiting from CAS

We can view commonality in data as arising from, i) incidental commonality between data chunks (generated in the same iteration) and ii) commonality due to data chunks that are not modified across iterations. The three scientific application benchmarks (*bssn*, *heat-solver*, *btio*) have clear, well-defined iterative behavior in data generation. The iterative nature of these benchmarks contributes to similarity in the data remaining constant across iterations (*iterative commonality*). The *dbt2* and *gene-dbase* benchmarks do not have such iterative data generation pattern. For these two benchmarks, gains from the use of CAS are realized by any incidental commonality in the data itself. It is important to note here that CAS based schemes can exploit *both* types of commonality, while non-CAS based schemes may be able to identify only iterative commonality.

In order to quantify the commonality from the iterative nature of an application, we applied a *diff* like filter to the data generated across iterations. This was done by examining the list of hashes of stored data for one iteration and comparing it with the list of hashes of stored data for the next iteration. The savings thus obtained from this diff-like operation indicates the savings obtained from unchanging data across iterations. This is shown in Figure 3. The curves labeled as *bssn*, *heat-solver*, and *btio* are the same as the ones from Figure 2, while the curves obtained by applying a *diff* appear with a *diff* subscript.

We observe that iterative commonality has a significant role to play. For example, the two curves for *btio* and *btio<sub>diff</sub>* are identical indicating that all the savings for the *btio* benchmark are due to its iterative nature. Similarly, a very large fraction of the savings for the other two benchmarks (*bssn* and *btio*) comes from the iteration based behavior. The amount of incidental commonality within data chunks of an iteration can be seen as the difference in values between a curve and its *diff*-based version in Figure 3. This value varies not just with the specific application itself, but also with chunksize. For *bssn* this value is as high as 21% at a 1 KB

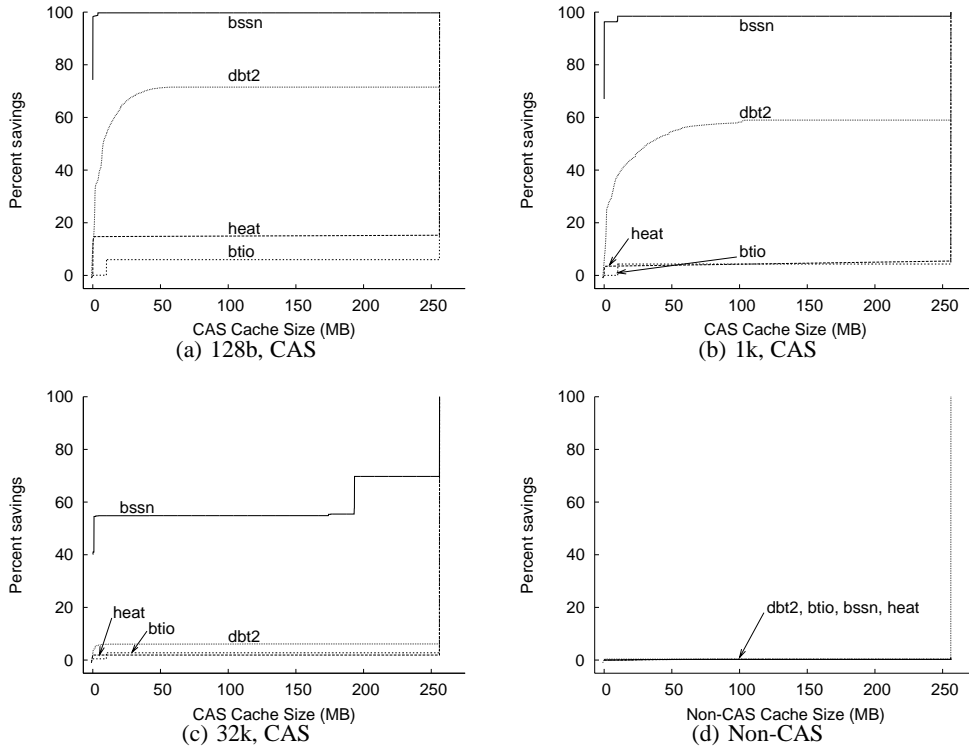


Figure 4: Percentage savings in network I/O when using a content addressable cache

chunksize. In the case of *heat-solver*, this difference is as high as 7% at a 128-byte chunksize, but tapers down to less than a percent at a 1 KB chunksize, while *btio* hardly has any commonality between chunks of the same iteration. This brings us to the following two conclusions,

- Commonality from older iterations provides significant savings. These savings may be realized by the use of a diff-based mechanism.
- Applications also have commonality from data chunks within the same iteration. These savings can only be exploited by CAS based schemes. The extent varies from application to application, and also with chunksize.

We further investigated the cause of commonality in application data. For example, for the *dbt2* application, we find that the most common chunk occurs over ten thousand times and the second most common chunk occurs just ten times. We inspected the data to find that the most popular chunk was a chunk composed entirely of zeroes). This is due to disk space allocation behavior of innodb. An important lesson here is that applications that use sparse matrices or datasets (perhaps arising out of internal allocation policies) can have portions of their data that benefit from CAS compression. This is especially true if an uncompressed representation of sparse datasets is used.

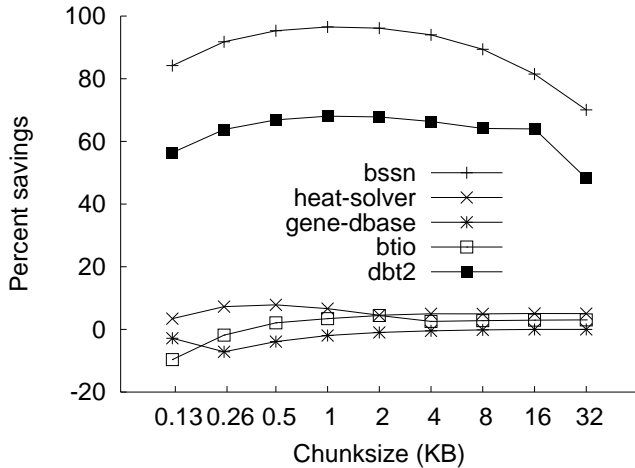
## 4.2 Content Addressable Caching: Savings in Network Bandwidth

In this section we wish to examine the savings in network bandwidth by use of CAS. We note that this section analyzes *live* data requests made on the storage subsystem, as opposed to an analysis of a static snapshot of the storage undertaken in Section 4.

The use of CAS directly impacts the amount of data that needs to be sent over the network in two ways. First, CAS can be used like a compression mechanism. By internally chunking a large read or write into smaller parts, CAS can remove the repeated chunks and send only the unique chunks over the network. Second, CAS can be viewed as a caching technique that can eliminate reads as well as writes (as opposed to traditional caching techniques which do not optimize writes). For example, a write may not require the actual data to be written out if the chunk to be created already exists in the CAS repository. Similarly, a read need not be sent to the repository if it may be satisfied by another chunk with the same content, available from a locally cached pool of chunks fetched earlier. These advantages of CAS extend beyond the case of networked I/O to the case of I/O on a local machine. In case of local I/O such caching is anyway done by the *buffer cache*. CAS can be used to design a content addressable buffer cache to manage objects, tagged by their SHA1 hashes. Thus the results in this section indicate the benefits of using CAS not only for network data, but also for say a system level buffer cache.

For this study, a trace of all the read/write requests was generated by running all the live benchmarks. This trace was then used to evaluate the performance of a local buffer, with LRU being the eviction policy. The percentage savings obtained with a buffer of a size  $S$  in Figure 4 indicates, i) savings when using a CAS based cache to absorb reads/writes, and ii) savings in network bandwidth when outgoing messages are buffered by upto  $S$  bytes. The baseline case (the 100% mark) is the amount of data to be sent when no caching used.

The three scientific application benchmarks perform very poorly when using a traditional LRU-based cache (Figure 4(d)) and provide less than 0.5% savings even in the best case (32 KB chunksize). The underlying reason is that these benchmarks are mostly



**Figure 5: Savings in storage space as a function of chunksize, including meta-data overheads**

sequential write workloads, hence the only gains from caching arise out of spatial locality. The most important point here is that even while all the benchmarks perform miserably under LRU, the CAS based schemes perform better. In fact, we can reason that a CAS based LRU policy will always outperform an LRU based cache. In cases where there is absolutely no locality of reference, and hence no exploitable commonality, a CAS based cache will perform as well as the non-CAS LRU cache.

We observe that the use of CAS tremendously benefits *bssn* (Figure 4), in spite of this being a sequential, mostly write-only workload. This is an interesting result because traditionally a cache cannot reduce the amount of write data when temporal locality is absent (no in-cache over-writes). However, a CAS based cache can detect that some or more of the  $N$  dirty pages have the same content, and hence need to be written out just once, reducing the number of writes to less than  $N$ . As a result, the *bssn* benchmark, which has a large commonality in its data can reduce the amount of data to be written by almost 100% at a 1 KB chunksize, using a very small CAS based cache.

The use of CAS brings tangible results for all the three scientific application benchmarks. Intuitively, these three benchmarks being *iterative* benchmarks, as long as the CAS based cache is large enough to hold data from an entire iteration, the next iteration can exploit commonality across iterations. This is corroborated in Figure 4(b), where *btio* (which generates 10 MB of data per iteration) performs better when using a cache at least 10 MB large. Similarly in the case of *heat-solver*, which generates 2 MB files per iteration, a 2 MB CAS based cache is enough to exploit commonality. For both applications, savings of the order of 4% is achieved. On the other hand, for *bssn* which generates about 260 MB of data per iteration, the savings occur with a very small cache size. This indicates that it is not the size of the data in the whole iteration that impacts the minimal cache size. Rather, it is the size of the unique data generated per iteration. Recall from Figure 2 that about 99% of the *bssn* data can be eliminated via CAS. This leaves about 1% of the data as data belonging to unique chunks, which fits in well with our hypothesis regarding the cache size.

The *dbt2* benchmark also benefits from the use of a CAS based cache, indicating that eliminating redundant data brings very significant gains (Figure 4(b)). The performance is severely penalized for using a larger chunksize of 32 KB than the internal page size of

16 KB, but the CAS based cache still brings some savings.

Another interesting observation is that a CAS based cache has the distinct advantage of being able to look at *all* the data encountered, even across file-names. As a result, for the *heat-solver* benchmark, which creates new data in new files every iteration, the CAS cache has a reasonable hit-rate.

We conducted the same experiment with a CAS based cache when running the scientific application benchmarks on multiple client nodes. With each node managing a specific subset of the whole data, we found that CAS finds significantly more commonality on every node. For example, on increasing the number of nodes from 1 to 4 to 9 for *btio*, the savings obtained by the use of a very small cache increase from about 4% to 18% to almost 40% respectively. As the number of nodes increase, the data generated per node, per iteration decreases, hence a progressively smaller cache size is required to realize the benefits. On the other hand, a less than ideal data-partitioning across nodes leads to minimal or negative increase in savings, with increase in number of client nodes. Poor data partitioning could occur due to a poor algorithm or due to the algorithm data mapping poorly onto the number of nodes.

In summary, the use of a content addressable caches provide significant benefits over a non-CAS based cache. For the CAS based cache to be effective for an iterative application, the size allocated should be larger than the size of the unique data generated per iteration.

## 5. CAS: CONS

In this section we look at the challenges brought up by the use of CAS. We look at the problem of added overheads in terms of maintaining extra meta-data, concerns of decreased error resilience at the CAS store and performance related issues.

### 5.1 Meta-Data Overheads

Figure 2 from Section 4.1 indicates that a CAS based store has the potential for significant space savings. These savings have an added cost – that of having to maintain an additional mapping from file offset to the hash (the name) of the chunk. This meta-data called the *recipe* (described in Section 2) stores a hash value for each  $N$  bytes of the file ( $N$  being the chunksize of the file-system). On deducting the cost of storing these hashes (20 bytes per hash for SHA1), the net savings obtained from the use of CAS are shown in Figure 5.

For the *bssn* data, after including the overheads, the savings peak at 96.5% at a 1 KB chunksize, and then decrease monotonically to 70% at a 32 KB chunksize. This interesting curve is a result of the tension between two opposing trends : commonality in data and meta-data overhead. Smaller chunksizes have the potential to expose more commonality, and hence can save more disk space. However, smaller chunksizes lead to more chunks per file, hence larger *recipes*, leading to more meta-data overhead. The large overhead at a 128 byte chunksize reduces the CAS savings from almost 99% in Figure 2 to 84%. As the chunksize increases to 1 KB and 2 KB, the commonality decreases marginally (less than 1% in Figure 2), while the meta-data overheads drop 8-fold and 16-fold respectively. This remarkable drop in the meta-data overhead for almost no drop in the savings leads to 1 KB being the most optimal chunksize for storing *bssn* data. Beyond a 2 KB chunksize, the commonality itself drops appreciatively, leading to a significant drop in the savings. The nature of the *dbt2* curve is almost identical, peaking at a 1 KB chunksize.

A similar behavior is also observed for the *heat-solver* and *btio* benchmarks. The *heat-solver* benchmark peaks at a 512-byte chunksize saving 7.8% and falls marginally to 6.6% at 1 KB. For small

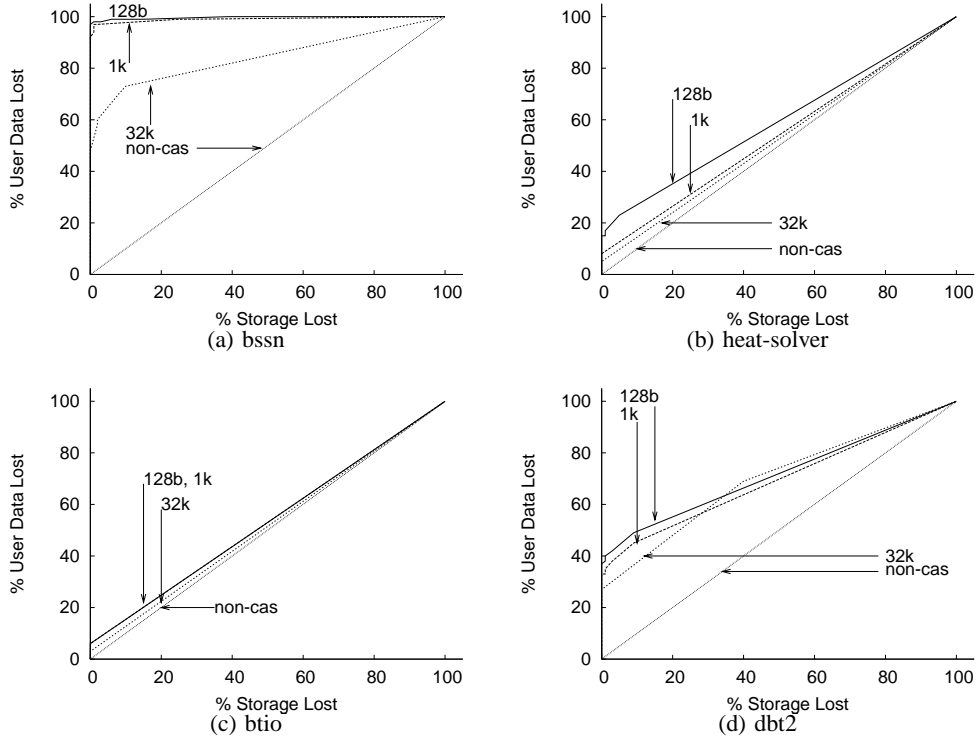


Figure 7: Error resilience of data store for different chunk sizes.

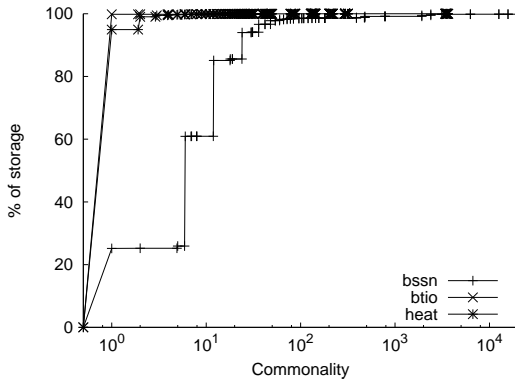


Figure 6: Storage profile for 128-byte chunks

chunk sizes, the savings for the *btio* benchmark starts in negative territory when including the meta-data overhead cost. At a 128-byte chunk size the CAS store requires 9.6% more storage than a conventional data store. With increase in chunk size from 128-byte to 2 KB, commonality stays relatively constant at around the 5% mark (Figure 2), while the meta-data overhead drops 16-fold. As a result, the curve has its highest savings of 4.4% at 2 KB chunk size. The *gene-dbase* data does not exhibit any commonality (Figure 2) and the added meta-data overhead makes the CAS store inefficient.

The above analysis indicates that a) small chunk sizes of around 1 KB are best, and b) applications with iterative data generation behavior have better chance of profiting from CAS.

## 5.2 Decreased Error Resilience

By storing duplicate data chunks just once, a CAS based store

achieves space savings or *compression* at the cost of error resilience. If the value of a chunk were measured in the amount of file data (user data) that would be lost on losing a single chunk, then in a traditional data store each chunk would be equally valuable. While in a CAS based store, a chunk with a higher commonality would be more valuable. Using the data from Figure 6, we can estimate the amount of user data rendered unusable on losing a certain amount of (CAS based) storage. Specifically, we would like to find out the fraction of the user data lost on losing a certain fraction of the storage space. We observe that the amount of user data lost when a chunk  $i$  in the storage system is destroyed, is given by

$$loss_i = commonality_i * chunksize \quad (1)$$

In the worst case scenario for a CAS store, we would lose the most valuable chunks (chunks with highest  $loss_i$  values) first. Arranging all the chunks in non increasing order of their  $loss_i$  values as the sequence in which the storage data is lost gives us Figure 7. In line with our expectations, higher commonality in data leads to poorer error resilience. From Figure 7 we observe the corresponding trend — the smaller the chunk size (more commonality), the farther the curve from the non CAS case, and hence higher damage on loss of single chunk. The *bssn* and *dbt2* data which have a large amount of commonality, also have the worst error resilience. At a 1 KB chunk size, losing a few percentage of the storage space can destroy close to 100% of the user data for *bssn*. Increasing the chunk size to 32 KB reduces this probability to about 60%. In the *heat-solver* and *btio* data, we notice that the use of a 1 KB chunk size brings the loss probability close to the 32 KB chunk size loss probability. The above observations indicate that the use of a 1 KB chunk size is a much better choice than a 128-byte chunk size for safety reasons. We note that the  $y = x$  line shows the baseline case

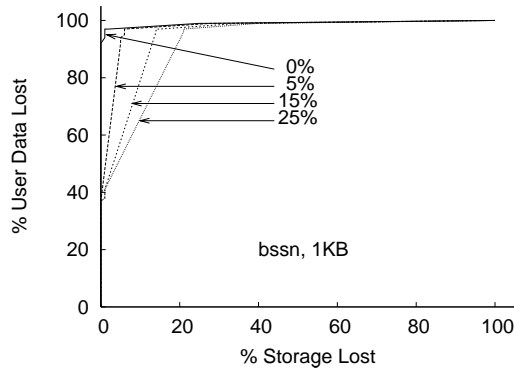


Figure 8: Effect of replication on error resilience.

– the non CAS store.

On the other hand, CAS exposes enough information about the chunks to make informed replication choices. One might choose a suitable replication policy, perhaps based on commonality of a chunk, or its access popularity, or its age, or even a combination of the above. Using such a policy, one might significantly improve the availability of the *right* chunks, or the dataset as a whole, without losing too much space. Figure 8 shows the effect of one such replication policy on error resilience of the *bsn* dataset for a 1 KB chunksize. In this policy, the chunks are replicated in a greedy manner depending on their commonality. The percentage number indicated in the graph indicates what percent of the un-replicated CAS data store was additionally allocated for replicated chunks. Even a small amount of replication (5%) makes significant difference to the error resilience of the dataset as a whole.

### 5.3 CAS Overheads

Data in a CAS based chunk store undergoes more operations than in a traditional file system. For example in the CAPFS file system, the CAPFS kernel module intercepts a write (and all other system calls) and passes it down to a user-space CAPFS daemon. This daemon chunks the data and generates a SHA1 hash (CAS name) for all chunks. It updates the file recipe and sends the chunks to the CAS data-servers. On receiving a chunk, the data-server first looks up a database of hashes to find if the chunk already exists. If it is a new chunk then the database is updated and assigned a disk location. The chunk is then finally written out to disk. The chunksize critically affects the performance of the CAS store. A small chunksize increases the recipe size of the file, causes inefficient network messaging and poor disk throughput at the data server. Unlike a traditional file-system where large contiguous writes are sent to disk, the largest contiguous write that the data-server can request equals the chunksize (before processing the next chunk). The number of chunks to be processed (which depends on the chunksize) affects the processing overheads (hash generation time) as well as the time required to query and update the database at a data-server.

In order to quantify the net effect of the above factors, we observed the wall-clock time to store 200 MB of data into CAPFS. In our experiment we ran CAPFS with the data-server and the meta-data server housed on the client itself. This setup avoids network latencies. Then we generate a 200 MB file from `/dev/urandom` and place it in `/dev/shm`. The time to copy this file to the CAPFS file-system was noted and averaged over multiple runs. Between each run the file-system was un-mounted, cleaned of pre-existing data and re-mounted again.

Figure 9 shows the results of our evaluation. The *lookup* curve

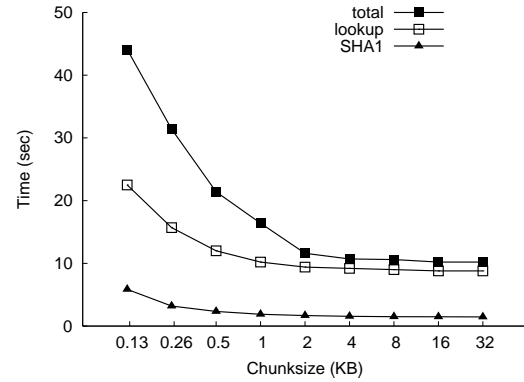


Figure 9: Time required to write 200 MB of data to a CAS store

indicates the time required to do everything but write the chunks to disk on the data-server, while the *total* curve indicates the total time required to complete the operation including disk I/O. The *SHA1* curve indicates the time required to generate the SHA1 hashes.

The SHA1 hash generation cost is larger at small chunksizes. At a 128-byte chunksize it accounts for 5.8 seconds out of a lookup time of 22.5 seconds and drops to 1.9 seconds out of 10.2 at a 1 KB chunksize. At 32 KB this reduces to 1.5 seconds out of 8.8 seconds for the lookup. In general the SHA1 cost is about 14% of the total time. The disk I/O overhead at the data-server shows up as the difference between the *lookup* and the *total* cost curves. It accounts for almost half the total time at small chunksizes and settles at nearly to less than 20% of the total time for chunksizes of 2 KB and more. The lookup overhead comprises of SHA1 hash generation, updating file recipe, time required to lookup each chunk name in the data-server database and other constant miscellaneous overheads. The file recipes are updated in-memory and do not contribute much to the above times. On excluding the SHA1 hash generation cost, the remaining lookup overhead takes 16.6 seconds at a 128-byte chunksize, 8.3 seconds at a 1 KB chunksize, 7.7 seconds at a 2 KB chunksize and finally settling to 7.3 seconds at a 32 KB chunksize. At small chunksizes the database component is significant due to the large number of hashes stored in the in-memory database. The original CAPFS project [47] used an inefficient implementation of the database where this cost was much higher for small chunks.

Removal of unused chunks resulting from over-writes or deletes would require garbage collection at the data-server. This feature was disabled in our tests and the cost has been ignored in this study. We conclude that at 2 KB chunksize or larger yields good performance. We also note that the SHA1 hash generation cost accounts for about 14% of the total time.

## 6. LESSONS LEARNT

*When to use CAS.* As seen in Section 4.1, the amount of commonality varies from application to application. In certain applications commonality it may be obvious. For example consider data from a backup tool like Internet Suspend/Resume [23] that takes a non-incremental snapshot of the entire hard-disk of the user. In this case one can expect significant similarity between one snapshot and another. Similar is the case for applications that periodically checkpoint their results for recovery purposes. Other cases include applications that deal with large sparse matrices, bitmaps or tables. Applications the periodically generate data could also have some

commonality across iterations. When such commonality is not obvious (as might be the case most of the time), space savings should not be the primary motivation behind the use of CAS.

Caching or network bandwidth always gains from the use of CAS – in the worst case of no extractable commonality, a CAS based cache will perform as well as a non-CAS cache with added overheads. This boost in performance comes from the ability of CAS to look not just at different parts of a file, but across files as well (*global naming*). By using this property of CAS to de-link the chunk name from the filename, a system can exploit caches without overheads typically associated with consistency as outlined in [47, 44].

**What chunk-size to use.** Our results indicate that at small chunk-sizes (specifically, smaller than 1 KB), the overheads outweigh any gains from the use of CAS. Use of a chunk-size of 1 KB or even 2 KB provides a good tradeoff between gains from space savings/caching, meta-data overheads, error resilience concerns, and performance.

**Recommendations.** Based on our results and discussions, we recommend that, (i) a CAS-based cache should always be used, (ii) iterative scientific applications should use a CAS based store and (iii) enterprise applications can use a CAS based store if there are sparse databases involved.

## 7. RELATED WORK

The Farsite distributed file-system from Microsoft was perhaps the first study [10] into existing commonality in file-systems. In their investigation however, Bolosky et. al. look at eliminating duplicate data at the file granularity rather than at a file-system block granularity. Muthitachoen et. al. proposed the Low Bandwidth File System [30] that eliminates commonality in the network data stream across variable sized chunks. Rabin [40] introduced the idea of generating variable sized chunks by detecting natural block boundaries. Broder et. al. present applications of this algorithm [11] and Chan et. al. provide an implementation for the same [13].

The Farsite project implements a Single Instance Store (a CAS store) for the Windows 2000 NTFS volume [9, 18]. The Plan 9 project from Bell Labs uses the Fossil file-system [39] to store snapshots of a live system on top of Venti [38], a content addressable backend. CAPFS or the Content Addressable Parallel File System [47] is a cluster file-system that exploits CAS for bandwidth savings. Various content based chunking methods have been used in Pasta[29], Pastiche[14] and REBL[24]. Ajtai et. al. [4] provide a comparison of the above methods.

Tolia et. al. first coined the term *recipes* [45] as a means of using hashes to summarize file content. The use of hashes as a means of detecting similar content has been looked at in [12]. The rsync protocol [46] uses MD5 hashes for comparing files. Cryptographic hashes have also been used to synchronize content across replicated collections [43, 22]. Chord [42] and CFS [15] exploit the global naming property of CAS. Sundr [25] and Ivy [31] uses CAS based hashes to verify the integrity of data.

Nath et. al. provide an analysis of data from another real world application called Internet Suspend/Resume in [23]. That study estimates the benefits of using CAS to house virtual machine snapshots.

Bhagwat et. al. have looked at the reliability concerns of CAS based storage in the context of an archive housing web-pages [7]. Compare by hash, the underlying principle of CAS has been criticized in [21] for being prone to collisions (two or more blocks generating the same SHA1 hash). More recently however, the Monotone team [28] and Black [8] have shown that this may not be a large concern.

## 8. CONCLUSION

In this paper we have evaluated the pros and cons of content addressable storage for five real world datasets. We find CAS to be useful for applications that display iterative data generation patterns, or manage sparse tables or datasets. Significant savings in network bandwidth can be achieved by the use of a content addressable cache only a few megabytes in size. We find that a 1 KB or 2 KB chunksize provides the best space savings when accounting for meta-data overhead due to SHA1 hashes. This chunksize also provides good savings in network bandwidth and reasonable error resilience. We note that the overheads of computing the SHA1 hashes in a CAS based store are about 14% and that a chunksize of 1 KB or larger achieves good throughput.

## Acknowledgements

This research was supported in part by NSF grant CNS-0720456 and a gift from Cisco. The authors would also like to thank the anonymous reviewers of the paper for their insightful comments.

## 9. REFERENCES

- [1] BSSN Pugh Benchmark. [http://www.cactuscode.org/Benchmarks/bench\\_bssn\\_pugh](http://www.cactuscode.org/Benchmarks/bench_bssn_pugh).
- [2] NAS PARALLEL BENCHMARKS. <http://www.nas.nasa.gov/Resources/Software/npb.html>.
- [3] Oracle berkeley db. <http://www.oracle.com/database/berkeley-db.html>.
- [4] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *J. ACM*, 49(3):318–367, 2002.
- [5] The babar experiment. <http://www-public.slab.stanford.edu/babar/>.
- [6] Belle. <http://belle.kek.jp/>.
- [7] D. Bhagwat, K. Pollack, D. D. E. Long, T. Schwarz, E. L. Miller, and J.-F. Paris. Providing high reliability in a minimum redundancy archival storage system. In *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, 2006.
- [8] J. R. Black. Compare-by-hash: A reasoned analysis. In *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX'06)*, Boston, MA, June 2006.
- [9] W. J. Bolosky, S. Corbin, D. Goebel, , and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, 2000.
- [10] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. *SIGMETRICS Perform. Eval. Rev.*, 28(1):34–43, 2000.
- [11] A. Broder. Some applications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.
- [12] A. Z. Broder. Identifying and filtering near-duplicate documents. In *COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, 2000.
- [13] C. Chan and H. Lu. Fingerprinting using polynomial (rabin's method). Faculty of Science, University of Alberta, CMPUT690 Term Project, December 2001.
- [14] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *OSDI: Symposium on Operating Systems Design and Implementation*, 2002.

- [15] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM Press.
- [16] Data Domain. <http://www.datadomain.com>.
- [17] OSDL Database Test 2. <http://www.osdl.org/>.
- [18] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS '02: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 617, Washington, DC, USA, 2002. IEEE Computer Society.
- [19] EMC Corp. *EMC Centera Content Addressed Storage System*, 2003. <http://www.emc.com/>.
- [20] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google File System. In *Proc. of the 2003 19th ACM Symposium on Operating System Principles*, October 2003.
- [21] V. Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 13–18, May 2003.
- [22] N. Jain, M. Dahlin, and R. Tewari. Taper: Tiered approach for eliminating redundancy in replicas. In *USENIX Conference on File and Storage Technologies (FAST05)*, 2005.
- [23] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, 2002.
- [24] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey. Redundancy Elimination Within Large Collections of Files. In *USENIX Annual Technical Conference, General Track*, 2004.
- [25] J. Li, M. Krohn, D. Mazierères, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 91–106, December 2004.
- [26] J. McKnight, T. Asaro, and B. Babineau. Digital archiving: End-user survey and market forecast 2006-2010. *The Enterprise Strategy Group*, Jan 2006.
- [27] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, San Francisco, CA, March 2004.
- [28] <http://www.venge.net/monotone/docs/Hash-Integrity.html>.
- [29] T. D. Moreton, I. A. Pratt, and T. L. Harris. Storage, Mutability and Naming in Pasta. In *Revised Papers from the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing*, 2002.
- [30] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, Oct. 2001.
- [31] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):31–44, 2002.
- [32] P. Nath, M. Kozuch, D. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX'06)*, Boston, MA, June 2006.
- [33] NCBI GenBank. <http://www.ncbi.nlm.nih.gov/Genbank/>.
- [34] K. Olsen, J. B. Minster, Y. Cui, A. Chourasia, R. Moore, Y. Hu, J. Zhu, P. Maechling, and T. Jordan. SCEC TeraShake Simulations: High Resolution Simulations of Large Southern San Andreas Earthquakes Using the TeraGrid. In *Proceedings of the TeraGrid 2006 Conference*.
- [35] TeraByte Scale Enterprise databases. [http://members.microsoft.com/customerevidence/Common/FileOpen.aspx?FileName=7405\\_FirstPremier\\_TDM\\_SQL\\_Server\\_Case\\_Study\\_Final.doc](http://members.microsoft.com/customerevidence/Common/FileOpen.aspx?FileName=7405_FirstPremier_TDM_SQL_Server_Case_Study_Final.doc).
- [36] Terabyte scale enterprise databases. [http://www.wintercorp.com/VLDB/2005\\_TopTen\\_Survey/2005TopTenWinners.pdf](http://www.wintercorp.com/VLDB/2005_TopTen_Survey/2005TopTenWinners.pdf).
- [37] Terabyte scale enterprise databases. <http://www.webtechniques.com/archives/1999/02/data/>.
- [38] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, January 2002.
- [39] S. Quinlan, J. McKie, and R. Cox. Fossil, an archival file-server. <http://www.cs.bell-labs.com/sys/doc/fossil.pdf>.
- [40] M. Rabin. Fingerprinting by Random Polynomials. In *Harvard University Center for Research in Computing Technology Technical Report TR-15-81*, 1981.
- [41] S. Rhea, K. Liang, and E. Brewer. Value-Based Web Caching. In *Proceedings of the Twelfth International World Wide Web Conference*, May 2003.
- [42] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001*, San Diego, CA, August 2001.
- [43] T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. *icde*, 00, 2004.
- [44] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating Portable and Distributed Storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004.
- [45] Tolia, N., Kozuch, M., Satyanarayanan, M., Karp, B., Bressoud, T., Perrig, A. Opportunistic Use of Content-Addressable Storage for Distributed File Systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, June 2003.
- [46] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.
- [47] M. Vilayannur, P. Nath, and A. Sivasubramaniam. Providing Tunable Consistency for a Parallel File Store. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST'05)*, 2005.
- [48] L. L. You, K. T. Pollack, and D. D. E. Long. Deep store: An archival storage system architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, 2005.
- [49] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5), 1978.