The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

CONTENT ADDRESSABLE DATA MANAGEMENT

A Thesis in

Computer Science and Engineering

by

Partho Nath

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2007

The thesis of Partho Nath has been reviewed and approved* by the following:

Anand Sivasubramaniam
Professor of Computer Science and Engineering
Thesis AdviserChair of Committee

Bhuvan Urgaonkar
Assistant Professor of Computer Science and Engineering

Padma Raghavan
Professor of Computer Science and Engineering

Qian Wang
Assistant Professor of Mechanical and Nuclear Engineering

Michael A. Kozuch
Computer Scientist at Intel Research Pittsburgh
Special Member

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

A direct implication of both the industry and academia proclaiming the Age of Tera- (even the Peta)-scale computing, is that applications have become more *data intensive* than ever. The increased data volume from applications tackling larger and larger problems has fueled the need for efficient management of this data. In this thesis, we evaluate a technique called *Content Addressable Storage* or CAS, for managing large volumes of data. This evaluation focuses on the benefits and demerits of using CAS for, i) improved application performance via lockless and lightweight synchronization of accesses to shared storage data; ii) improved cache performance; iii) increase in storage capacity; and, iv) increased network bandwidth. We present the design of a CAS-based file store that significantly improves the storage performance providing lightweight and lock-less user-defined consistency semantics. As a result, our file-system shows a 28% increase in read-bandwidth and a 13% increase in write bandwidth, over a popular file-system in common use. We use the same experimental file-system to analyze CAS on data from real world application benchmarks. We also estimate the potential benefits of using CAS for a virtual machine based user mobility application, that was in active use at a public deployment for over a period of seven months.

Table of Contents

List of Figures viii

Acknowledgments xi

Chapter 1. Introduction 1

Chapter 2. Background on CAS 6

Chapter 3. CAPFS : Design of a Content Addressable Parallel File System 13

3.1 Introduction 13

3.2 Design Issues 16

3.2.1 Tunable Consistency 16

3.2.2 Lightweight Synchronization 18

3.2.2.1 To Lock or Not to Lock? 19

3.2.2.2 Invalidates or Updates? 21

3.2.2.3 Content Addressability 22

3.2.3 Background & Definitions 25

3.3 System Architecture 28

3.3.1 PVFS Architecture 29

3.3.2 CAPFS: Servers 30

3.3.3 CAPFS: Clients 31

3.3.4 System Calls 31

3.3.4.1 Steps for the `open` and `close` System Call 32
3.3.4.2 Steps for the `read` System Call 32
3.3.4.3 Steps for the `write` System Call 33
3.3.4.4 Commit Step 34
3.3.5 Conflict Resolution 35
3.3.6 Client-side Plug-in Architecture 41
3.4 Experimental Results 43
3.4.1 Network Performance 44
3.4.2 Aggregate Bandwidth Tests 45
3.4.3 Tiled I/O Benchmark 51
3.4.4 NAS BTIO Benchmark 55
3.5 Related Work 57
3.6 Chapter Summary 61

Chapter 4. Content Addressable Storage : Pros and Cons 62
4.1 Introduction 62
4.2 Methodology for Evaluating the Efficacy of CAS 64
4.2.1 Applications used as Benchmarks 65
4.3 CAS: Pros 67
4.3.1 Savings in Storage Space 67
4.3.1.1 Impact of chunksize 68
4.3.1.2 Applications benefiting from CAS 69
4.3.1.3 Commonality Profile 71

4.3.2 Content Addressable Caching: Savings in Network Bandwidth . . . 73
4.4 CAS: Cons . . . 78
4.4.1 Meta-Data Overheads . . . 79
4.4.2 Decreased Error Resilience . . . 81
4.4.3 CAS Performance Overheads . . . 84
4.5 Discussion . . . 87
4.6 Related Work . . . 88
4.7 Chapter Summary . . . 90

Chapter 5. Case Study: Internet Suspend/Resume . . . 92
5.1 Introduction . . . 92
5.2 Background . . . 95
5.2.1 Internet Suspend/Resume . . . 95
5.2.2 Content Addressable Storage . . . 97
5.3 Methodology . . . 97
5.3.1 Pilot Deployment . . . 98
5.3.2 Data Collection . . . 99
5.3.3 Analysis . . . 100
5.4 Results: CAS & Storage . . . 103
5.4.1 Effect of Privacy Policy on Storage . . . 103
5.4.2 Effect of Chunksize on Storage . . . 106
5.5 Results: CAS & Networking . . . 111
5.5.1 Effect of Privacy Policy on Networking . . . 112

5.5.2 Effect of Chunksize on Networking 118

5.5.2.1 Effect on Upload Size 118

5.5.2.2 Effect on Download Size 118

5.6 Related Work 125

5.7 Chapter Summary 126

Chapter 6. Conclusions 128

References 129

List of Figures

2.1 CAS based naming 6

2.2 CAS based storage 7

2.3 Effect of chunksize on commonality 8

2.4 SHA-1 hash collision probability using Birthday Paradox 11

3.1 Design of the CAPFS parallel file system 17

3.2 Read-write sharing for parallel applications. 22

3.3 System architectures: CAPFS design incorporates two client-side caches that are absent in PVFS. 28

3.4 Action sequence: multiple-readers single-writer 36

3.5 Action sequence: multiple-readers multiple-writers 38

3.6 The client-side plug-in API and the CAPFS client-daemon core API. On receiving a system call, the CAPFS client-daemon calls the corresponding user-defined pre- and post- functions, respectively, before servicing the system call. . 40

3.7 Myrinet (a) Point-to-point latency in μsec (b) Bisection bandwidth in MB/s 44

3.8 Design space constituting a sample set of consistency policies: SEQ-1, SEQ-2 implement sequential consistency; FOR-1, FOR-2 implement a slightly relaxed mechanism where commits are forced; REL-1 implements an even more relaxed mechanism. The X in rows 1 and 3 denotes a don't care for the variable's value. 46

3.9 Aggregate Bandwidth in MB/s with varying block sizes: CAPFS vs. PVFS for (a) read-8-clients, (b) read-16-clients, (c) write-8-clients, (d) write-16-clients. 48

3.10 Tile reader file access pattern: Each processor reads data from a display file onto local display (also known as a tile). 52

3.11 Tile I/O benchmark bandwidth in MB/s: (a) non-collective read, (b) non-collective write, (c) collective read, (d) collective write. 53

3.12 Execution time for the NAS BTIO benchmark. 56

4.1 Benchmarks used 66

4.2 Savings in storage space as a function of chunksize 68

4.3 Identifying commonality in data due to iterative behavior 70

4.4 Commonality profile. 72

4.5 Percentage savings in network I/O when using a content addressable cache . . . 75

4.6 Percentage savings in network I/O with a CAS based cache on a multi-node experiment 77

4.7 Savings in storage space as a function of chunksize, including meta-data overheads 79

4.8 Storage profile for 128-byte chunks 81

4.9 Error resilience of data store for different chunksizes: worst-case scenario. . . . 83

4.10 Effect of replication on error resilience. 84

4.11 Time required to write 200 MB of data to a CAS store 86

5.1 An ISR system. 95

5.2 Summary of ISR pilot deployment. 98

5.3 Observed parcel checkin frequency 99

5.4 Storage policy encryption technique summary. 100

5.5 Growth of storage needs for Delta, IP, and ALL. 104

5.6 Storage space growth for various chunksizes without meta-data overhead (y-axis scale varies). 107

5.7 Server space required, after 201 deployment days. 109

5.8 Meta-data overhead expressed as a percentage of user data. 110

5.9 CDF of upload sizes for different policies, without and with the use of compression. 113

5.10 Search space for identifying redundant blocks during data synchronization operations. Note that for download, the system inspects the most recent version available at the client (which may be older than $N - 1$). 114

5.11 Upload sizes for different chunksizes. 116

5.12 Download size when fetching memory image of latest version. 120

5.13 Download size when fetching memory *and disk* of latest version. 124

Acknowledgments

This point in time marks the close of yet another chapter in my life, that of a graduate student at Penn State. This period of almost six years has given me the opportunity to experiment, learn and grow along the many directions that life has to offer. I've been lucky to have the company of a few good friends and a great human being as my advisor, who have helped me pull through trying times. This thesis has seen fruition thanks to their collective support.

I am deeply indebted to my thesis advisor Dr. Anand Sivasubramaniam, who has been a guiding rail for not only my technical endeavors, but also for the murkier non-technical issues that one encounters as a doctoral candidate. It is due to his very patient handling in early times, and constant attention throughout, that I've been able to complete my doctoral studies. I would like to thank Murali Vilayannur, now a researcher at VMware, for being a mentor, a good friend in times of need and otherwise, and a great sounding board for my research ideas.

I would like to thank Intel Research Pittsburgh for giving me an opportunity to intern at their lab-let. A significant portion of this thesis has been a result of this close interaction, lasting for over a period of three years. I would also like to thank Dr. Michael Kozuch for mentoring and guiding me on numerous technical issues during my time as an intern at Intel Research Pittsburgh. I cannot thank him enough for spending so much time and energy on me, helping me grow professionally and personally. I would also like to thank Prof. Mahadev Satyanarayanan and Dr. David O'Hallaron from Carnegie Mellon University for allowing me a chance to work on the Internet Suspend/Resume project. I would like to thank NSF, the funding agency that has funded me for the bulk of my graduate program.

I also wish to thank my thesis committee members and referees: Dr. Bhuvan Urgaonkar, Dr. Padma Ragahavan and Dr. Qian Wang for taking time off their busy schedules, writing recommendation letters for me and being a part of my thesis committee. Their strong support and valuable comments have made my job search much easier and helped refine my dissertation.

I am grateful to Sri Sri Ravi Shankar, my spiritual guru, for helping me find the strength and confidence to successfully navigate through whatever life has thrown my way. I am also thankful to Birjoo Vaishnav for introducing me to the Art of Living, which has freed me from so many self-imposed boundaries and limitations.

I have been very lucky to have a group of excellent colleagues and lab-mates, who were also my best friends at Penn State: Murali Vilayannur, Chun Liu, Gokul Kandiraju, Sudhanva Gurumurthi, Amitayu Das, Angshuman Parashar, Shiva Chaitanya, Jianyong Zhang, Sriram Govindan, Youngjae Kim, Niranjan Kumar Soundararajan, Dharani Sankar Vijayakumar, Arjun R. Nath, Byung Chul Tak and Bo Zhao. I am thankful to all my friends and acquaintances at Penn State for making life a pleasure.

None of the work described in this thesis would have been possible without the prompt attention of the lab support team at Penn State. I am deeply indebted to Eric Prescott, Nate Coraor, John Domico, David Heidrich and Barbara Einfalt for their dedication and prompt attention to administering all the machines in the department. All the secretaries in the office extended so much warmth and help that have really made my stay in the department pleasurable, especially Vicki Keller who manages to do so much work and yet remain cheerful all the time. Without her help, I cannot imagine how things would have turned out for me.

I am grateful to my parents and my brother. Their love and care have always been with me during these years. Their trust, encouragement and continual support made it possible for me to step away from home, and successfully complete this thesis.

Chapter 1

Introduction

The landscape of computer systems is becoming more *data intensive*. Scientists are faced with mountains of data that stem from four trends, i) the flood of data from new scientific instruments driven by Moore's Law, increasing their computational capacity at an alarming rate; (ii) the flood of data from larger and more complex simulations; (iii) the ability to economically store huge amounts of data; and (iv) the Internet and Internet driven applications that makes data accessible to anyone anywhere, allowing the replication, creation, and recreation of more content [48]. Precedents for petabyte-scale systems already exist at data-centers for Google, Yahoo!, and MSN Search [45]. Such systems have tens of thousands of processing nodes and have close to 100,000 locally attached disks to deliver the requisite bandwidth. Similar data-centric behavior exists in applications in other areas — medical imaging, data analysis and mining, video processing, global climate modeling, computational physics and chemistry. These applications often manipulate data sets ranging from several megabytes to terabytes [33, 36, 89, 30]. The computer systems required to support contemporary applications in everyday use in many data-centers and computing laboratories, are commonly architected as a network of workstations, commonly known as *clusters*. Managing the storage and movement of data on such systems, from storage nodes (nodes housing the requisite data), to the processing nodes, poses substantial software challenges.

The major challenges in efficiently managing large amounts of data are at least four-fold. First, the problem of storing so much data, is only the tip of the iceberg. Second is the issue of efficiently delivering the data over the network. For large data, this may not be trivial – data for even a single file could potentially be distributed across the disks at multiple nodes (for reasons of reliability or parallel access). The third issue is that of handling concurrent accesses to shared data. Analyses are often performed simultaneously by a collaborating set of nodes requiring arbitration of accesses to shared data. With the shift to very large, scaled-out cluster architectures, this is a very critical performance related issue that *must* be dealt with. The fourth issue, closely related to the third, is that of efficient and smart caching techniques. Data needs to be re-processed each time a new algorithm is developed, or each time the application is run with different parameters.This generates even more I/O. If the data must be moved, it makes sense to store a copy at the destination for later reuse. Thus caching or re-use of locally available data is the fourth challenge. To extract maximal performance, data needs to be managed efficiently along these dimensions. In this thesis, we evaluate a data management technique called Content Addressable Storage or *CAS* to handle these four challenges.

We observe that for three of the challenges outlined above, namely i) storage efficiency, ii) network bandwidth and iii) caching efficiency, the use compression-like techniques can yield benefits. A reduction in the dataset size via data compression techniques, would reduce the space required to store not just the data itself, but also any replicas, thus increasing the scalability of the system. The network interconnect that ships the data from a storage node to the compute node and vice-versa would see a corresponding increase in throughput, owing to the reduced data volume. Data caches, if used on either the storage node or the compute node, also stand to gain by

being able to accommodate more data, hence increasing the effectiveness of the cache. Traditional lossless compression techniques like gzip [145] suffer from at least two problems. First, due to I/O and memory overheads, the operation is too slow, and second, the operation transforms data from one representation to another, requiring a reverse transform (de-compression) before any use. On the other hand, specialized compression techniques [140] promise larger savings, but would require customized code for every application, and hence do not scale to a system-wide solution.

An alternative technique that has gained significant popularity in recent literature for its potential to reduce the size of a given dataset is Content Addressable Storage or *CAS* [80, 93, 94, 37, 130]. CAS operates on data by breaking it into small contiguous *chunks*, and storing only the identical chunks in the data, discarding any duplicates. Since the data representation itself is not modified, no de-compression is required. Since CAS can be applied on the raw data chunks itself, other compression techniques like gzip can be applied over and on top of CAS, if desired. Similar to gzip-like compression, however, the performance of CAS depends on the workload. A dataset wherein most of the chunks occur just once will not provide much space savings with CAS. The choice of the *chunksize* or the granularity of the data chunks is another significant factor that affects the performance of CAS. For example, the Farsite file-system [37] indicates a storage space savings of up to 46% when applying CAS on a file-level granularity, while other data ([80, 114, 120]) promise much larger savings when applying CAS at a sub-file granularity.

In addition to reducing data volume via compression, CAS can further help reduce network traffic in a cluster environment by exploiting *commonality* of data between different files. When transferring a file between two nodes, CAS recognizes chunks of data the recipient already

has available locally in other files and avoids sending those chunks over the network. Such commonality could arise out of an older version of the file being present or incidental commonality between two files. In spite of its growing popularity, details on performance of CAS on real world data are relatively scant in modern literature. It is the goal of this thesis to evaluate the pros and cons of using CAS for data emanating from real world applications.

On the other hand, the first and foremost concern of most scientists is performance. One of the most critical factors affecting performance of applications running in a cluster environment is that of handling accesses to shared data and while keeping caches up-to-date. The recent explosion in the scale of clusters, coupled with the emphasis on fault tolerance, has made traditional locking less suitable for cluster environments. In a cluster file system ([15, 109]), arbitration for shared accesses usually involves a process acquiring a lock from a central lock manager on a file before proceeding with the write/read operation. As the number of processes writing to the same file increases, performance degrades rapidly from lock contention. On the other hand, fine-grained file locking schemes, such as byte-range locking, allow multiple processes to simultaneously write to different regions of a shared file. However, they also restrict scalability because of the overhead associated with maintaining state for a large number of locks, eventually leading to performance degradation. Furthermore, any networked locking system introduces a bottleneck for data access – the lock server.

In this thesis, we make innovative use of CAS to manage concurrent accesses to shared data. We present the design of a Content Addressable Parallel File System (*CAPFS*) that makes use of CAS to provide *lightweight, optimistic* concurrency in a lockless manner while allowing for client-side caching of data and meta-data. By avoiding the use of distributed locking, CAPFS

enhances the scalability and fault-tolerance of the system while optimistic concurrency enhances file system throughput.

We next describe the assumptions made in this thesis, the background and the terms used in Chapter 2. Chapter 3 describes the design of CAPFS that makes innovative use of CAS to enhance application performance. We use the experimental CAPFS platform to evaluate the benefits and challenges in the use of CAS on data from real world application benchmarks. The results from this analysis are detailed in Chapter 4. This analysis, however does not completely capture the essence of a data management tool like CAS. For example, a real person often runs the same application multiple times, not just once, generating the same data multiple times. To incorporate such usage behavior in the real world, Chapter 5 details a case study that evaluates CAS on usage data collected over a seven month period for a virtual machine based user mobility application. We believe that this application is representative of a growing set of applications [55, 76, 1, 106], all of which would benefit from the use of CAS. Chapter 6 summarizes the conclusions of this thesis.

Chapter 2

Background on CAS

Content Addressable Storage or *CAS* is based on the principle of addressing or naming a data object in such a manner that the name is representative of its content. Typically this is accomplished by running a cryptographic hash function on the data, and using the thus generated hash as the name of the object (the *CAS name*), as shown in Figure 2.1. In this thesis, we use the SHA1 cryptographic hash function [87] as the basis for all our discussions and experimental results. The resulting CAS names (SHA1 hashes) are of fixed-length (20 bytes each).

Such a naming scheme has several advantages over using a human generated name for a data object. The first and most important advantage from the use of a cryptographic hash function is that, a) two identical data objects will have the same name; and b) barring collisions, two different data objects will have different CAS names. A second property that results from the use of this scheme is *global naming*. Addressing a data object by its content-hash provides a way to globally name a block – independent of the file, server or any originating domain. By

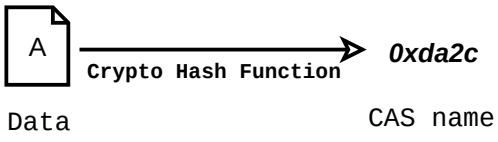


Fig. 2.1. CAS based naming

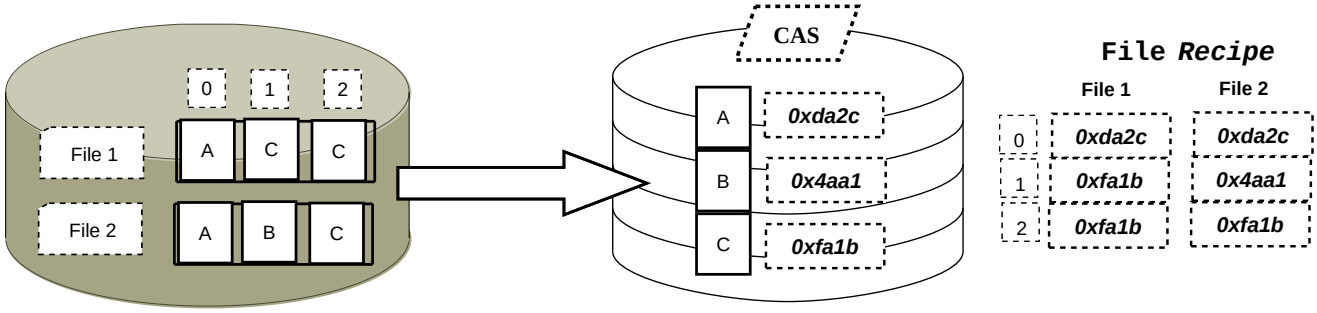


Fig. 2.2. CAS based storage

removing these constraints on the *naming* of the data object, a CAS based data repository has the ability to compare data objects across filename and other namespace boundaries.

The above two properties together form the basis for the space savings obtained by the use of CAS, as described next. CAS operates on a data entity like a file, a raw disk partition or even a data stream by dividing it into fixed-size *chunks* (we discuss variable sized chunks later). The CAS name for the chunk is then generated. The CAS data repository, which houses the data chunks, indexed by their CAS names, is then searched for the presence of the chunk being processed. If not found, the chunk is added to the repository. But in case the chunk already exists in the repository, it is not stored a second time, thereby saving storage space that would have been required for this chunk. This process is shown in Figure 2.2. In this figure, a traditional file store uses the space required to store six chunks from two files. However, the CAS repository weeds out the duplicate content, thus storing only three chunks, resulting in a 50% space savings for the data.

The file *recipe* **:** The savings achieved by CAS come at the expense of additional meta-data. For example, in Figure 2.2, to read chunk 1 of *file 2* from a CAS repository, one would need

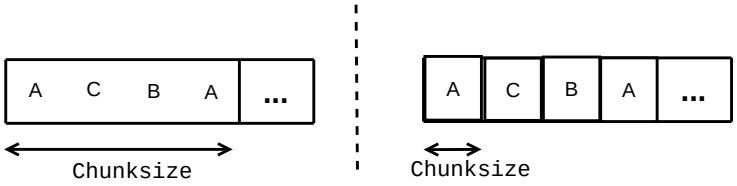


Fig. 2.3. Effect of chunksize on commonality

to figure out the CAS name for chunk 2 (without having access to its contents). To solve this problem, we maintain an additional table called the file *recipe*, that maps a file chunk-number to its CAS name. In our experiments, we implement the recipe as a flat list of the CAS names of each chunk of a file, listed in proper sequence. Since the CAS names are of fixed size (20 bytes), it is easy to navigate the recipe. A very important point to note here is that the size of the meta-data (recipe) depends on the *number of chunks in the original data*, and not on the chunks in the CAS repository. As we shall see in later chapters, this can be a significant performance drag on CAS.

Savings in network bandwidth: In addition to reducing data volume by detecting redundancy within a file, CAS can further reduce bandwidth requirements by exploiting cross-file similarities. CAS can take advantage of the fact that the same chunks of data often appear in multiple files or multiple versions of the same file. For example, to transfer a file between two nodes, the recipient node determines the chunks of data it already has other files and avoids transfer of these chunks over the network. If the recipient node is the CAS repository, then such cross-file *commonality* is detected by simply querying the CAS repository. If the recipient is a compute node, then the node can query it's local content addressable cache which could contain data from other files, or simply an older, out of date version of the same file.

Commonality : CAS based savings are achieved by exploiting data commonality within a single file or across different files. The presence or absence of commonality is highly dependent on the data being stored. Commonality in a dataset might be incidental (e.g. across unrelated files) or due to relationships in the data (e.g. storing a newer version of the same file in a CAS repository). The commonality of a chunk refers to the number of times it occurs in the original non-CAS data. The commonality of a dataset also depends on the *chunksize*, or the granularity at which the data is broken up for application of CAS.

Chunksize : The exploitable commonality in a dataset also depends on the chunksize. For example, on reducing the chunksize to one-fourth it's original value in Figure 2.3, CAS can detect more commonality in the data leading to greater savings. The downside to using a chunksize that is four times smaller, is that the meta-data overhead (recipe) size goes up four times, irrespective of how much commonality exists in the data. For datasets with not too much commonality, use of a smaller chunksize with larger recipes may outweigh any savings obtained by the use of CAS.

Chunksize : fixed or variable ? Uptil now, the discussion has assumed that CAS is applied on data that is partitioned into fixed sized chunks. For some workloads, this may not be the best data chunking policy to use. For example, consider a user editing a text document over multiple sessions. Such a workload will have insertion and deletion of data in the middle of the file, but the content mostly stays the same across versions. As a result of the insert and delete operations, the chunk boundaries of the new version may not align perfectly with the same chunk boundaries in the previous version. Hence, when the data for the latest version must be uploaded to the server (CAS repository), the fixed sized chunking scheme may not detect that most of the content already lies on the CAS repository.

The use of a *variable sized* chunking scheme can avoid the sensitivity to shifting file offsets. LBFS [80] uses Rabin fingerprints (hashes) to determine chunk boundaries in a file [96]. When the low-order 13 bits of the fingerprint equal a particular constant value, the bytes producing that fingerprint constitute the end of a chunk. Since this scheme bases chunk boundaries on file contents (i.e. the particular byte that causes the specific pre-determined fingerprint), insertions and deletions only affect the surrounding chunks. As a result only a few chunks will change, thus saving network bandwidth where a fixed-sized chunking scheme would have failed.

Note that the variable sized chunking mechanism has a higher cost - chunk boundaries need to be calculated in addition to running a SHA1 like function on the chunks themselves. The benefits of a variable sized chunking scheme occurs in workloads with a significant number of insert and delete operations. Additionally, such 'inserts' and 'deletes' may not be visible if CAS is applied at a disk-block level. Owing to file-system fragmentation and non-contiguous file allocation, the contents of a single file may be spread all over the disk. Hence the insert/delete operations may translate to new writes at free disk blocks placed between data blocks of any arbitrary file. In this thesis, neither of these conditions hold. Our analysis of CAS on the CAPFS platform in Chapters 3 and 4 evaluate applications that have well co-ordinated over-writes in the middle of the file and appends at the end-of-file, with no insert/delete operations in the middle of the file. The analysis of real-world data in Chapter 5 has been undertaken at the disk-block level. Hence the simpler fixed-size chunking mechanism has been used for all our experiments. Another argument favoring fixed chunks is that with variable sized chunks, the recipe file is larger and parsing it is a more complex (expensive) operation.

Use of cryptographic hashes : On running a cryptographic hash function, a digest or a hash is produced. This hash is used as the CAS name of the data. A key property of cryptographic

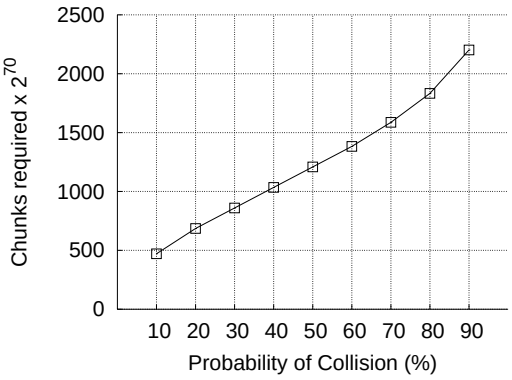


Fig. 2.4. SHA-1 hash collision probability using Birthday Paradox

hash functions like SHA1 that is exploited here is that the hash function *h* is *collision resistant*, i.e. it must be computationally intractable to find a tuple *(a,b)* such that h(a) = h(b). Of course, such *a* and *b* must exist, given the infinite domain and finite range of *h*, but finding such a pair should be very hard. Functions like SHA1, RIPEMD, MD5 are designed to withstand differential cryptanalysis, i.e. they will map correlated inputs to uncorrelated outputs. In the absence of an adversary, a collision will occur due to just plain bad luck. We estimate using the 'birthday paradox' that the probability of an an undetectable TCP bit-flip [118] is greater than that of a SHA-1 hash collision, for a CAS repository housing less than 4.5×2^{60} chunks. At a small chunksize of 128 bytes, this implies that we need to worry more about undetectable network transmission errors as long as the CAS store houses less than 576 petabytes of data. Figure 2.4 indicates the probability of a SHA-1 collision (estimated using the birthday paradox) in terms of the number of chunks being stored in the CAS repository.

In case of an adversary being present, one would theoretically need to perform 2^{60} operations. Recently however, Wang et. al. [136] have *broken* SHA-1, by reporting that this can

be achieved in 2^{60} operations. However, such an attack is still infeasible due to the immense computational power required. For example, Grembowski et. al. [50] indicate that even using specialized state-of-the-art hardware running at 33 Mhz would take millions of years to find a single collision. A 4 Ghz imaginary hardware could accomplish this in about 170,000 years. It has hence, been argued that CAS based techniques based on SHA-1 are still safe [10]. Details of properties of cryptographic hash functions can be found in [102] or briefly in [10]. We assume for this thesis that SHA-1 is sufficiently safe to use or if not, then we advocate the use of a stronger digest like SHA-256. Further discussion of this topic is unfortunately beyond the scope of this thesis.

Chapter 3

CAPFS : Design of a Content Addressable Parallel File System

3.1 Introduction

High-bandwidth I/O continues to play a critical role in the performance of numerous scientific applications that manipulate large data sets. Parallelism in disks and servers provides cost-effective solutions at the hardware level for enhancing I/O bandwidth. However, several components in the system software stack, particularly in the file system layer, fail to meet the demands of applications. This is primarily due to tradeoffs that parallel file system designers need to make between performance and scalability goals at one end, and transparency and ease-of-use goals at the other.

Compared to network file systems (such as NFS [104], AFS [53], and Coda [61]), which despite allowing multiple file servers still allocate all portions of a file to a server, parallel file systems (such as PVFS [25], GPFS [110], and Lustre [15]) distribute portions of a file across different servers. With the files typically being quite large and different processes of the same application sharing a file, such striping can amplify the overall bandwidth. With multiple clients reading and writing a file, coordination between the activities becomes essential to enforce a consistent view of the file system state.

The level of sharing when viewed at a file granularity in parallel computing environments is much higher than that observed in network file systems [8, 88], making consistency more important. Enforcement of such consistency can, however, conflict with performance and

scalability goals. Contemporary parallel file system design lacks a consensus on which path to take. For instance, PVFS provides high-bandwidth access to I/O servers without enforcing overlapping-write atomicity, leaving it entirely to the applications or run-time libraries (such as MPI-I/O [41]) to handle such consistency requirements. On the other hand, GPFS and Lustre enforce byte-range POSIX [115] consistency. Locking is used to enforce serialization, which in turn may reduce performance and scalability (more scalable strategies are used in GPFS for fine-grained sharing, but the architecture is fundamentally based on distributed locking).

Serialization is not an evil but a necessity for certain applications. Instead of avoiding consistency issues and using an external mechanism (e.g., DLM [54]) to deal with serialization when required, incorporating consistency enforcement in the design might reduce the overheads. Hence the skill lies in being able to make an informed decision regarding the consistency needs of an application. A key insight here is that applications, not the system, know best to deal with their concurrency needs. In fact, partial attempts at such optimizations already exist — many parallel applications partition the data space to minimize read-write and write-write sharing. Since different applications can have different sharing behavior, designing for performance *and* consistency would force the design to cater to *all* their needs — simultaneously! Provisioning a single (and strict) consistency mechanism may not only make such fine-grained customization hard but may also constrain the suitability of running diverse sets of applications on the same parallel file system.

Addressing some of these deficiencies, this chapter presents the design and implementation of a novel parallel file system called CAPFS that provides the following notable features:

- To the best of our knowledge, CAPFS is the first file system to provide a tunable consistency framework that can be customized for an application. A set of plug-in libraries is provided

with clearly defined entry points, to implement different consistency models, including POSIX, Session, and Immutable-files. Though a user could build a model for each application, we envision a set of predefined libraries that an application can pick before execution for each file and/or file system.

- The data store in CAPFS is content-addressable. Consequently, blocks are not modified in place, allowing more concurrency in certain situations. In addition, content addressability can make *write propagation* (which is needed to enforce coherence) more efficient. For instance, update-based coherence mechanisms are usually avoided because of the large volume of data that needs to be sent. In our system however, we allow update messages that are just a sequence of (cryptographic) hashes of the new content being generated. Further, content addressability can exploit commonality of content within and across files, thereby lowering caching and network bandwidth requirements.
- Rather than locking when enforcing serialization for read-write sharing or write-write sharing (write atomicity), CAPFS uses optimistic concurrency control mechanisms [68, 79] with the presumption that these are rare events. Avoidance of distributed locking enhances the scalability and fault-tolerance of the system.

The rest of this chapter is organized as follows. The next section outlines the design issues guiding our system architecture, following which the system architecture and the operational details of our system are presented in Section 3.3. An experimental evaluation of our system is presented in Section 3.4 on a concurrent read/write workload and on a parallel tiled visualization code and for the BTIO benchmark. Section 3.5 summarizes related work and Section 3.6 concludes with contributions and discusses directions for further improvements.

3.2 Design Issues

The guiding rails of the CAPFS design is based on the following goals: 1) user should be able to define the consistency policy at a chosen granularity, and 2) implementation of consistency policies should be as lightweight and concurrent as possible. The CAPFS design explores these directions simultaneously — providing easily expressible, tunable, robust, lightweight and scalable consistency without losing focus of the primary goal of providing high bandwidth.

3.2.1 Tunable Consistency

If performance is a criterion, consistency requirements for applications might be best decided by applications themselves. Forcing an application that has little or no sharing to use a strong or strict consistency model may lead to unnecessarily reduced I/O performance. Traditional techniques to provide strong file system consistency guarantees for both meta-data and data use variants of locking techniques. In this chapter, weprovide tunable semantic guarantees for *file data alone*.

The choice of a system wide consistency policy may not be easy. NFS [104] offers poorly defined consistency guarantees that are not suitable for parallel workloads. On the other hand, Sprite [85] requires the central server to keep track of all concurrent sessions and disable caching at clients when write-sharing is detected. Such an approach forces *all* write-traffic to be network bound from thereon until one or more processes close the shared file. Although such a policy enforces correctness, it penalizes performance of applications when writers update spatially disjoint portions of the same file which is quite common in parallel workloads. For example, an application may choose to have a few temporary files (store locally, no consistency),

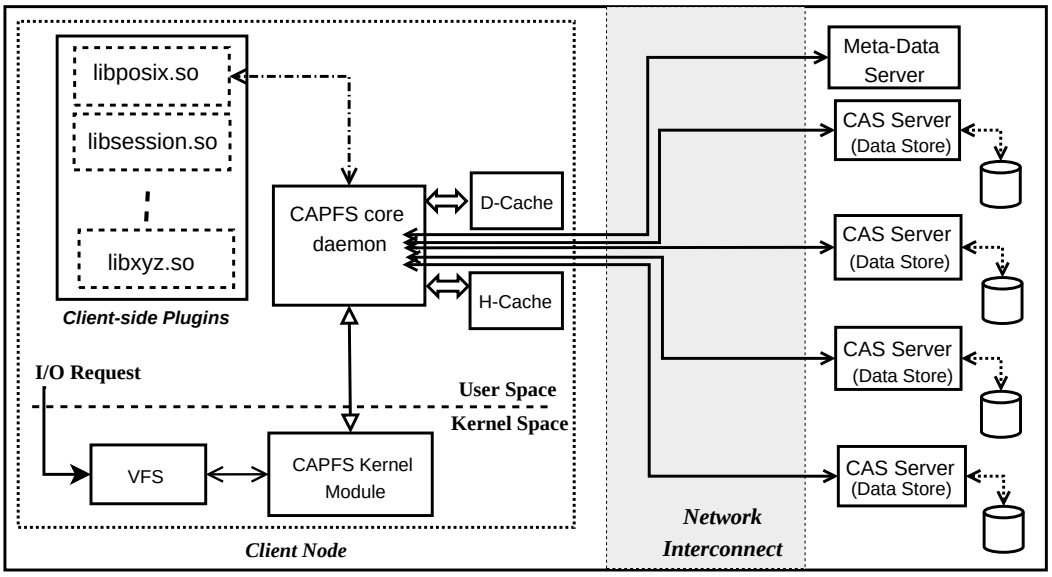


Fig. 3.1. Design of the CAPFS parallel file system

a few files that it knows no one else will be using (no consistency), a few files that will be extensively shared (strong consistency), and a few files that might have sharing in the rare case (weaker user-defined consistency). A single consistency policy for a cluster-based file system cannot cater to the performance of different workloads such as those described above.

As shown in Figure 3.1, CAPFS provides a client-side plug-in architecture to enable users to define their own consistency policies. The users write plug-ins that define what actions should be taken before and after the client-side daemon services the corresponding system call. (The details of the above mechanism are deferred to Section 3.3.6).

The choice of a plug-in architecture to implement this functionality has several benefits. Using this architecture, a user can define not just standard consistency policies like POSIX, session and NFS, but also custom policies, at a chosen granularity (sub-file, file, partition-wide).

First and foremost, the client keeps track of its files; servers do not need to manage copy-sets unless explicitly requested by client. Furthermore, a client can be using several different consistency policies for different files or even changing the consistency policy for a given file *at run-time*, without having to recompile or restart the file system or even the client-side daemon (Figure 3.1). All that is needed is that a desired policy be compiled as a plug-in and be installed in a special directory, after which the daemon is sent a signal to indicate the availability of a new policy. Leaving the choice of the consistency policy *and* allowing the user to change it at run-time enable tuning performance at a very fine granularity. However, one major underlying assumption in our system design is that we anticipate that the file system administrator sets the same policy on all the nodes of the cluster that accesses the file system. Handling conflicting consistency policies for the same file system or files could lead to incorrect execution of applications.

3.2.2 Lightweight Synchronization

Any distributed file system needs to provide a consistency protocol to arbitrate accesses to data and meta-data blocks. The consistency protocol needs to expose primitives both for atomic read/modify/write operations and for notification of updates to regions that are being managed. The former primitive is necessary to ensure that the state of the system is consistent in the presence of multiple updates, while the latter is necessary to incorporate client caching and prevent stale data from being read. Traditional approaches use locking to address both these issues.

3.2.2.1 To Lock or Not to Lock?

Some parallel cluster file systems (such as Lustre [15] and GPFS [110]) enforce data consistency by using file locks to prevent simultaneous file access from multiple clients. In a networked file system, this strategy usually involves acquiring a lock from a central lock manager on a file before proceeding with the write/read operation. Such a coarse-grained file locks-based approach ensures that only one process at a time can write data to a file. As the number of processes writing to the same file increases, performance (from lock contention) degrades rapidly. On the other hand, fine-grained file-locking schemes, such as byte-range locking, allow multiple processes to simultaneously write to different regions of a shared file. However, they also restrict scalability because of the overhead associated with maintaining state for a large number of locks, eventually leading to performance degradation. Furthermore, any networked locking system introduces a bottleneck for data access: the lock server.

The recent explosion in the scale of clusters, coupled with the emphasis on fault tolerance, has made traditional locking less suitable. GPFS [110], for instance, uses a variant of a distributed lock manager algorithm that essentially runs at two levels: one at a central server and the other on every client node. For efficiency reasons, clients can cache lock tokens on their files until they are explicitly revoked.

Such optimizations usually have hidden costs. For example, in order to handle situations where clients terminate while holding locks, complex lock recovery/release mechanisms are used. Typically, these involve some combination of a distributed crash recovery algorithm or a lease system [49]. Timeouts guarantee that lost locks can be reclaimed within a bounded time. Any lease-based system that wishes to guarantee a sequentially consistent execution must handle

a race condition, where clients must finish their operation after acquiring the lock before the lease terminates. Additionally, the choice of the lease timeout is a tradeoff between performance and reliability concerns and further exacerbates the problem of reliably implementing such a system.

The pitfalls of using locks to solve the consistency problems in parallel file systems motivated us to investigate different approaches to providing the same functionality. We use a lockless approach for providing atomic file system data accesses. The approach to providing lockless, sequentially consistent data in the presence of concurrent conflicting accesses presented here has roots in three other transactional systems: store conditional operations in modern microprocessors [75], optimistic concurrency algorithms in databases [68], and optimistic concurrency approach in the Amoeba distributed file service [79].

Herlihy [52] proposed a methodology for constructing lock-free and wait-free implementations for highly concurrent objects using the load-linked and store-conditional instructions. Our lockless approach, similar in spirit, does not imply the absence of any synchronization primitives (such as barriers) but, rather, implies the *absence of a distributed byte-range file locking service*. By taking an optimistic approach to consistency, we hope to gain on concurrency and scalability, while pinning our bets on the fact that conflicting updates (write-sharing) will be rare [8, 31, 88]. In general, it is well understood that optimistic concurrency control works best when updates are small or when the probability of simultaneous updates to the same item is small [79]. Consequently, we expect our approach to be ideal for parallel scientific applications. Parallel applications are likely to have each process write to distinct regions in a single shared file. For these types of applications, there is no need for locking, and we would like for all writes to proceed in parallel without the delay introduced by such an approach.

3.2.2.2 Invalidates or Updates?

Given that client-side caching is a proven technique with apparent benefits for a distributed file system, a natural question that arises in the context of parallel file systems is whether the cost of keeping the caches coherent outweighs the benefits of caching. However, as outlined earlier, we believe that deciding to use caches and whether to keep them coherent should be the prerogative of the consistency policy and should not be imposed by the system. Thus, only those applications that require strict policies and cache coherence are penalized, instead of the whole file system. A natural consequence of opting to cache is the mechanism used to synchronize stale caches; that is, should consistency mechanisms for keeping caches coherent be based on expensive update-based protocols or on cheaper invalidation-based protocols or hybrid protocols?

Although update-based protocols reduce lookup latencies, they are not considered a suitable choice for workloads that exhibit a high degree of read-write sharing [6]. Furthermore, an update-based protocol is inefficient in its use of network bandwidth for keeping file system caches coherent, thus leading to a common adoption of invalidation-based protocols.

As stated before, parallel workloads do not exhibit much block-level sharing [31] . Even when sharing does occur, the number of consumers that actually read the modified data blocks is typically low. In Figure 3.2 we compute the number of consumers that read a block between two successive writes to the same block (we assume a block size of 4 KB). Upon normalizing against the number of times sharing occurs, we get the values plotted in Figure 3.2. This figure was computed from the traces of four parallel applications that were obtained from [132]. In other words, Figure 3.2 attempts to convey the amount of read-write sharing exhibited by typical

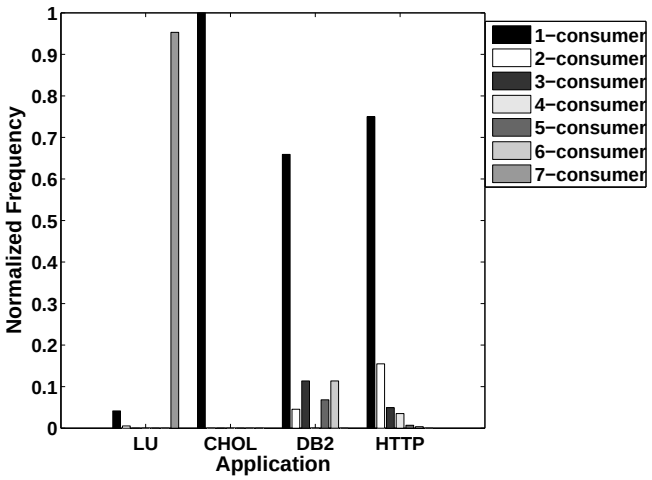


Fig. 3.2. Read-write sharing for parallel applications.

parallel applications. It indicates that the number of consumers of a newly written block is very small (with the exception of LU, where a newly written block is read by all the remaining processes before the next write to the same block). Thus, an update-based protocol may be viable as long as the update mechanism does not consume too much network bandwidth. This result motivated us to consider content-addressable cryptographic hashes (such as SHA-1 [87]) for maintaining consistency because they allow for a bandwidth-efficient update-based protocol by transferring just the hash in place of the actual data. We defer the description of the actual mechanism to Section 3.3.5.

3.2.2.3 Content Addressability

Content addressability provides an elegant way to summarize the contents of a file. It provides the following advantages:

- The contents of a file can be listed as a concatenation of the hashes of its blocks. Such a representation was referred to as *recipes* in a previous study [127]. This approach provides a lightweight method of updating or invalidating sections of a file and so forth.

- It increases system concurrency, by not requiring synchronization at the content-addressable data servers (Figure 3.1). In comparison to versioning file systems that require a central version/time-stamp servers [79] or a distributed protocol for obtaining unique timestamps [46], a content-addressable system provides an independent, autonomous technique for clients to generate new version numbers for a block. Since newly written blocks will have new cryptographic checksums (assuming no hash collisions), a content-addressable data server also achieves the "no-overwrite" property that is essential for guaranteeing any sort of consistency.

- Using cryptographic hashes also allows for a bandwidth-efficient update-based protocol for maintaining cache coherence. This forms the basis for adopting a content-addressable storage server design in place of a traditional versioning mechanism. Additionally, it is foreseeable that the content-addressable nature of data may lead to easy replication schemes.

- Depending on the workload, content addressability might be able to reduce network traffic and storage demands. Blocks with the same content, if in the cache (because of commonality of data across files or within a file) do not need to be fetched or written. Only a single instance of the common block needs to be stored, leading to space savings.

As shown in Figure 3.1, the client employs two caches for performance. The H-Cache, or hash cache, stores all or a portion of a file's *recipe* [127]. A file in the CAPFS file system is

composed of content-addressable chunks. Thus, a chunk is the unit of computation of cryptographic hashes and is also the smallest unit of accessibility from the CAS servers. The chunk size is crucial because it can impact the performance of the applications. Choosing a very small value of chunk size increases the CPU computation costs on the clients and the overheads associated with maintaining a large recipe file, while a very large value of chunk size may increase the chances of false sharing and hence coherence traffic. Thus, we leave this as a tunable knob that can be set by the plug-ins at the time of creation of a file and is a part of the file's meta-data. For our experiments, unless otherwise mentioned, we chose a default chunk size of 16 KB. The recipe holds the mapping between the chunk number and the hash value of the chunk holding that data. Using the H-Cache provides a lightweight method of providing updates when sharing occurs. An update to the hashes of a file ensures that the next request for that chunk will fetch the new content.

The D-Cache, or the data cache, is a content addressable cache. The basic object stored in the D-Cache is a chunk of data addressed by its SHA1-hash value. One can think of a D-cache as being a local replica of the CAS server's data store. When a section of a file is requested by the client, the corresponding data chunks are brought into the D-Cache. Alternatively, when the client creates new content, it is also cached locally in the D-Cache. The D-Cache serves as a simple cache with *no consistency requirements*. Since the H-caches are kept coherent (whenever the policy dictates), there is no need to keep the D-caches coherent. Additionally, given a suitable workload, it could also exploit commonality across data chunks and possibly across temporal runs of the same benchmark/application, thus potentially reducing latency and network traffic.

3.2.3 Background & Definitions

We now define a few terms that would help provide a background for the later discussions.

- A file *session* is a series of file system calls that a process executes beginning with the `open`ing of a file and ending with the `close` of the file. All file system calls (`read`, `write`, `truncate` etc) between the `open` and `close` for a particular file are designated to be a part of the file's session.

- *Concurrent write-sharing [85]* occurs when 2 or more file sessions accesses the same file in conflicting modes, where at-least one of them opens the file for writing.

- Any concurrent execution of a set of actions is said to be *serializable* if it is equivalent to any serial execution of the same set of actions. Each action is therefore the granularity at which serializability is guaranteed also known as the serializable unit. Typically, most file system implementations define this unit to be a single file system call/operation, while relational databases offer it at the granularity of a transaction.

- The question of how accurately an execution reflects the actual serialization order was addressed in [111]. An execution is *real-time consistent* if for any two conflicting operations *op1* and *op2*, *op1* precedes *op2* in execution if and only if *op1* occurred in real time before *op2* [111]. An example of a non real-time consistent serialized execution is as follows, if a stale copy of a file that was last updated by a process P1, is cached by a process P2, then although P2 accesses the file later in real time than P1, P2 is serialized before P1 since it does not see P1's update.

One of the important issues that needs to be addressed when designing a file system is its behavior in the presence of concurrent conflicting requests which is commonly referred to as the file system semantics issue. There are four commonly used types of file semantics namely,

- *POSIX/UNIX semantics* uses the basic file system calls as the units of serializability as mentioned briefly previously. UNIX semantics requires that these operations also be real-time consistent, i.e every write operation's updates should be immediately visible to all read operations that follow. Although, guaranteeing such strict semantics does not inhibit performance for disk-based file systems, this can greatly impact the performance and efficiency of parallel/network file systems due to the associated overheads in making sure that updates get propagated or appropriate invalidate messages are sent to every copy of the file in the entire cluster and in applying the updates in the same order. These semantics are essentially identical to the *sequential consistency* semantics that was formalized by Lamport in [69] in the context of multi-processor architectures. Sprite [85] is an example of a file system that implements UNIX semantics.

- *Session Semantics* requires that the beginning of a file session reflects the updates from the previously closed file session. It does not guarantee that updates from concurrently open sessions will be visible to the newly opened session. Consequently, all read/write operations for any newly opened file session is performed locally on the cached copy. At the end of a file session, the cached copy from the current file session is made visible to subsequent sessions. The implications of such semantics is that updates from concurrently executing sessions could potentially be completely lost. In some sense, a session serves as synchronization points at which consistency of is guaranteed that is similar to the Release

Consistency model [44] for multi-processor architectures. AFS [53] is an example of a file system that implements Session Semantics.

- *Immutable Files*: In order to overcome the disadvantages stated above that are associated with session semantics, immutable file semantics proposed that the end of an update session would create a new version of the file and thus any old version of the file would still be retained and be accessible. In many ways this is similar to *versioning file systems*(like [113, 105]) with the only difference being the granularity at which the file system creates new versions, i.e., the former (immutable files) creates new versions at the end of an update session, whereas the latter (versioning file systems) creates new versions based on a user-specified granularity (which could potentially be after every update).

- *Transactional Semantics* requires that file sessions be serializable, or in other words the serializable unit is guaranteed to be a session. Thus, any execution of a file session would be appear to be an atomic action. Each file session could thus be compared to a transaction in a database system.

Traditionally, most distributed file systems allow clients to cache data, which in turn introduces a potential level of file inconsistency. In the past, researchers have attempted to solve this problem by, placing the burden on servers to call-back and inform the caches of updates, or disallow client caching during specific periods of times, or by placing the burden on clients to check the validity of their cached copies before allowing access. Information about the changed state of the file system is disseminated by sending appropriate invalidation messages to the client caches. Unlike in distributed shared memory systems, update-based protocols are not typically

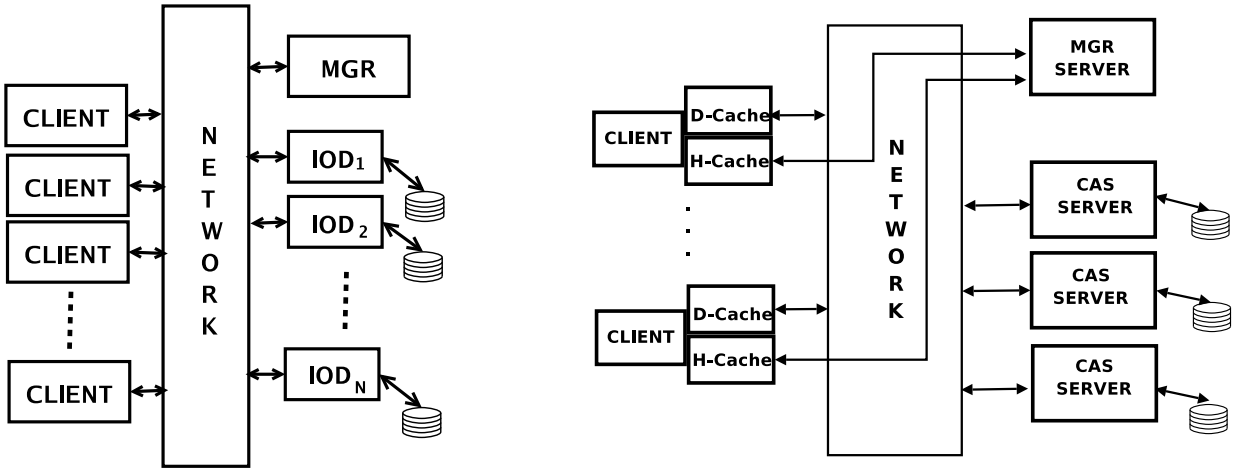


Fig. 3.3. System architectures: CAPFS design incorporates two client-side caches that are absent in PVFS.

used due to the excessive amount of data that would have to be sent on the network, and the need to ensure that updates are seen in the same order by all clients.

3.3 System Architecture

The goal of our system is to provide a robust parallel file system with good concurrency, high throughput and tunable consistency. The design of CAPFS resembles that of PVFS [25] in many aspects — central meta-data server, multiple data servers, RAID-0-style striping of data across the I/O servers, and so forth . The RAID-0 striping scheme also enables a client to easily calculate which data server has which data blocks of a file. In this section, we first take a quick look at the PVFS architecture and its limitations from the perspective of consistency semantics and then detail our system's design. Figure 3.3 depicts a simplified diagram of the PVFS and CAPFS system architectures.

3.3.1 PVFS Architecture

The primary goal of PVFS as a parallel file system is to provide high-speed access to file data for parallel applications. PVFS is designed as a client-server system, as shown in Figure 3.3 (a).

PVFS uses two server components, both of which run as user-level daemons on one or more nodes of the cluster. One of these is a meta-data server (called MGR) to which requests for meta-data management (access rights, directories, file attributes, and physical distribution of file data) are sent. In addition, there are several instances of a data server daemon (called IOD), one on each node of the cluster whose disk is being used to store data as part of the PVFS name space. There are well-defined protocol structures for exchanging information between the clients and the servers. For instance, when a client wishes to open a file, it communicates with the MGR daemon, which provides it the necessary meta-data information (such as the location of IOD servers for this file, or stripe information) to do subsequent operations on the file. Subsequent reads and writes to this file do not interact with the MGR daemon and are handled directly by the IOD servers.

This strategy is key to achieving scalable performance under concurrent read and write requests from many clients and has been adopted by more recent parallel file system efforts. However, a flip-side to this strategy is that the file system does not guarantee any data consistency semantics in the face of conflicting operations or sessions. Fundamental problems that need to be addressed to offer sequential/ POSIX [115] style semantics are the *write atomicity* and *write propagation* requirements. Since file data is striped across different nodes and since the

data is always overwritten, the I/O servers cannot guarantee write atomicity, and hence reads issued by clients could contain mixed data that is disallowed by POSIX semantics. Therefore, any application that requires sequential semantics must rely on external tools or higher-level locking solutions to enforce access restrictions. For instance, any application that relies on UNIX/POSIX semantics needs to use a distributed cluster-wide lock manager such as the DLM [54] infrastructure, so that all `read/write` accesses acquire the appropriate file/byte-range locks before proceeding.

3.3.2 CAPFS: Servers

The underlying foundation for our system is the content-addressable storage model, wherein file blocks are *addressed and located* based on the cryptographic hashes of their contents. A file is logically split into fixed-size data chunks, and the hashes for these chunks are stored in the *hash server daemon*. The hash server daemon, analogous to the meta-data server (MGR) daemon of the PVFS system design, is responsible for mapping and storing the hashes of file blocks (termed recipes [127]) for all files. In essence, this daemon translates the logical block-based addressing mode to the content addressable scheme, that is, given a logical block *i* of a particular file *F*, the daemon returns the hashes for that particular block. Even though in the current implementation there is a central server, work is under way to use multiple meta-data servers to serve a file's hashes for load-balancing purposes. Throughout the rest of this chapter, we will use the term MGR server synonymously with hash server or meta-data server to refer to this daemon.

Analogous to the PVFS I/O server daemon is a content-addressable server (CAS) daemon, which supports a simple *get/put* interface to retrieve/store data blocks based on their cryptographic hashes. However, this differs significantly both in terms of functionality and exposed interfaces from the I/O servers of PVFS. Throughout the rest of this chapter, we will use the term CAS server synonymously with data server to refer to this daemon.

3.3.3 CAPFS: Clients

The design of the VFS glue in CAPFS is akin to the upcall/downcall mechanism that was initially prototyped in the Coda [61] file system (and later adapted in many other file systems including PVFS). In this design, file system requests obtained from the VFS are queued in a device file and serviced by a user-level daemon. If an error is generated or if the operation completes successfully, the response is queued back into the device file, and the kernel signals the process that was waiting for completion of the operation. The client-side code intercepts these upcalls and funnels meta-data operations to the meta-data server. The data operations are striped to the appropriate CAS servers. Prototype implementations of the VFS glue are available at [133] for both Linux 2.4 and 2.6 kernels.

3.3.4 System Calls

The CAPFS system uses optimistic concurrency mechanisms to handle write atomicity on a central meta-data server, while striping writes in parallel over multiple content-addressable servers (CAS servers). The system has a lockless design: the only form of locking used is mutual-exclusion locks on the meta-data server to serialize the multiple threads (whenever necessary), as opposed to distributed locking schemes (such as DLM [54]).

3.3.4.1 Steps for the `open` and `close` System Call

- When a client wishes to open a file, a request is sent to the hash-server to query the hashes for the file if any.
- The server returns the list of hashes for the file (if the file is small). Hashes can also be obtained on demand from the server subsequently. The server also adds H-cache callbacks to this node for this file if requested.
- After the hashes are obtained, the client caches them locally (if specified by the policy) in the H-cache to minimize server load. H-cache coherence is achieved by having the server keep track of when commits are successful, and issuing callbacks to clients that may have cached the hashes. This step is described in greater detail in the subsequent discussions.
- On the last close of the file, all the entries in the H-cache for this file are invalidated for subsequent opens to reacquire, and if necessary the server is notified to terminate any callbacks for this node.

3.3.4.2 Steps for the `read` System Call

- The client tries to obtain the appropriate hashes for the relevant blocks either from the H-cache or from the hash server. An implicit agreement here is that the server promises to keep the client's H-cache coherent. This goal may be achieved by using either an update-based mechanism or an invalidation-based mechanism depending on the number of sharers. Note that the update callbacks contain merely the hashes and not the actual data.

- Using these hashes, it tries to locate the blocks in the D-cache. Note that keeping the H-cache coherent is enough to guarantee sequential consistency; nothing needs to be done for the D-cache because it is content addressable.

- If the D-cache has the requested blocks, the read returns and the process continues. On a miss, the client issues a *get* request to the appropriate CAS servers, which is cached subsequently. Consequently, reads in our system do not suffer any slowdowns and should be able to exploit the available bandwidth to the CAS servers by accessing data in parallel.

3.3.4.3 Steps for the `write` System Call

Writes from clients need to be handled a little differently because consistency guarantees may have to be met (depending on the policy). Since writes change the contents of the block, the cryptographic hashes for the block changes, and hence this is a new block in the system altogether. We emphasize that we need mechanisms to ensure write atomicity not only across blocks but also across copies that may be cached on the different nodes. On a write to a block, the client does the following sequence of steps,

- Hashes for all the relevant blocks are obtained either from the H-cache or from the hash server.

- If the write spans an entire block, then the new hash can be computed locally by the client. Otherwise, it must read the block and compute the new hash based on the block's locally modified contents.

- After the old and new hashes for all relevant blocks are fetched or computed, the client does an *optimistic put* of the new blocks to the CAS servers, which store the new blocks.

Note that by virtue of using content-addressable storage, the servers do not overwrite older blocks. This is an example of an optimistic update, because we assume that the majority of writes will be race-free and uncontested.

- If the policy requires that the writer's updates be made immediately visible, the next step is the *commit* operation. Depending on the policy, the client informs the server whether the commit should be forced or whether it can fail. Upon a successful commit, the return values are propagated back.

- A failed commit raises the possibility of *orphaned* blocks that have been stored in the I/O servers but are not part of any file. Consequently, we need a distributed cleaner process that is invoked when necessary to remove blocks that do not belong to any file. We refer readers to [133] for a detailed description of the cleaner protocol.

3.3.4.4 Commit Step

- In the commit step, the client contacts the hash server with the list of blocks that have been updated, the set of old hashes, and the set of new hashes. In the next section, we illustrate the need for sending the old hashes, but in short they are used for detecting concurrent write-sharing scenarios similar to store-conditional operations [75].

- The meta-data server atomically compares the set of old hashes that it maintains with the set of old hashes provided by the client. In the uncontested case, all these hashes would match, and hence the commit is deemed race free and successful. The hash server can now update its recipe list with the new hashes. In the rare case of a concurrent conflicting updates, the server detects a mismatch in the old hashes reported for one or more of the

client's commits and asks them to retry the entire operation. However, clients can override this by requesting the server to force the commit despite conflicts.

- Although such a mechanism has guaranteed write-atomicity across blocks, we still need to provide mechanisms to ensure that client's caches are also updated or invalidated to guarantee write atomicity across all copies of blocks that may be required by the consistency policy (sequential consistency/UNIX semantics require this). Since the server keeps track of clients that may have cached file hashes, a successful commit also entails updating or invalidating any client's H-cache with the latest hashes.

- Our system guarantees that updates to all locations are made visible in the same order to all clients (this mechanism is not exposed to the policies yet). Therefore, care must be exercised in the previous step to ensure that updates to all clients' H-caches are atomic. In other words, if multiple clients may have cached the hashes for a particular chunk and if the hash-server decides to update the hashes for the same chunk, the update-based protocol must use a two-phase commit protocol (such as those used in relational databases), so that all clients see the updates in the same order. This is not needed in an invalidation-based protocol however. Hence, we use an invalidation-based protocol in the cases of multiple readers/writers and an update-based protocol for single reader/writer scenarios.

3.3.5 Conflict Resolution

Figure 3.4 depicts a possible sequence of actions and messages that are exchanged in the case of multiple-readers and a single-writer client to the same file. We do not show the steps involved in opening the file and caching the hashes. In step 1, the writer optimistically writes

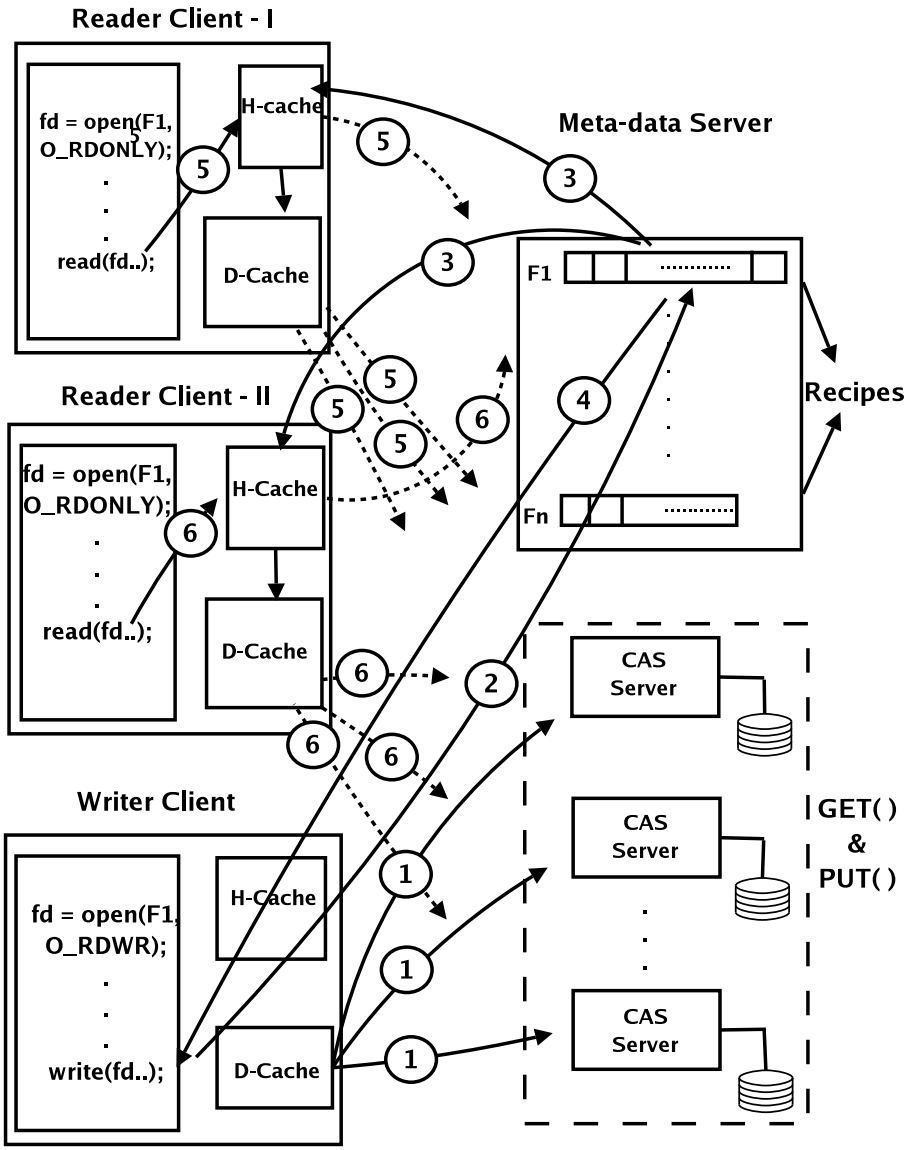


Fig. 3.4. Action sequence: multiple-readers single-writer

to the CAS servers after computing the hashes locally. Step 2 is the request for committing the write sent to the hash server. Step 3 is an example of the invalidation-based protocol that is used in the multiple reader scenario from the point of view of correctness as well as performance. Our system resorts to an update-based protocol in the single sharer case. Sequential consistency requires that any update-based protocol has to be two-phased for ensuring the write-ordering requirements, and hence we opted to dynamically switch to using invalidation-based protocol in this scenario to alleviate performance concerns. Steps 5 and 6 depict the case where the readers look up the hashes and the local cache. Since the hashes could be invalidated by the writer, this step may also incur an additional network transaction to fetch the latest hashes for the appropriate blocks. After the hashes are fetched, the reader looks up its local data cache or sends requests to the appropriate data servers to fetch the data blocks. Steps 5 and 6 are shown in dotted lines to indicate the possibility that a network transaction may not be necessary if the requested hash and data are cached locally (which happens if both the `read`'s occurred before the `write` in the total ordering).

Figure 3.5 depicts a possible sequence of actions and messages that are exchanged in the case of multiple-readers and multiple-writers to the same file. As before, we do not show the steps involved in opening the file and caching the hashes. In step 1, writer client II optimistically writes to the CAS servers after computing hashes locally. In step 2, writer client I does the same after computing hashes locally. Both these writers have at least one overlapping byte in the file to which they are writing (*true-sharing*) or are updating different portions of the same chunk (*false-sharing*). In other words this is an instance of concurrent-write sharing. Since neither writer is aware of the other's updates, one of them is asked to retry. The hash server acts as a serializing agent. Since it processes requests from client II before client I, the write from client

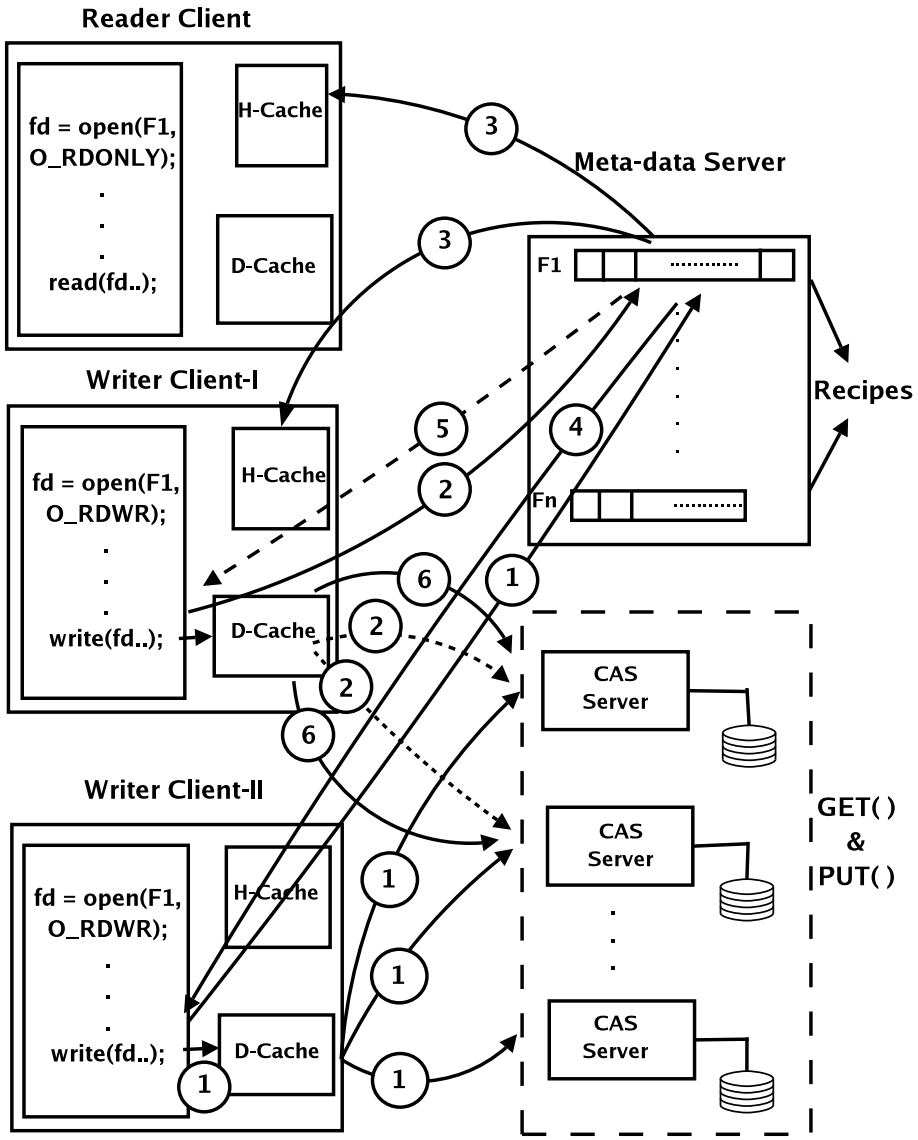


Fig. 3.5. Action sequence: multiple-readers multiple-writers

II is successfully committed, and step 3 shows the invalidation messages sent to the reader and the writer client. Step 4 is the acknowledgment for the successful write commit. Step 5 is shown dashed to indicate that the hash server requests writer client I to retry its operation. The write done by this client in step 2 is shown dotted to indicate that this created orphaned blocks on the data server and needs to be cleaned. After receiving a reply from the hash server that the write needs to be retried, the writer client I obtains the latest hashes or data blocks to recompute its hashes and reissues the write as shown in step 6.

In summary, our system provides mechanisms to achieve serializability that can be used by the consistency policies if they desire. In our system, *read-write serializability* and *write atomicity across copies* are achieved by having the server update or invalidate the client's H-cache when a write successfully commits. *Write-write serializability across blocks* is achieved by having the clients send in the older hash values at the time of the commit to detect concurrent write-sharing and having one or more of the writers to restart or redo the entire operation.

We emphasize here that, since *client state is mostly eliminated*, there is no need for a complicated recovery process or lease-based timeouts that are an inherent part of distributed locking-based approaches. Thus, our proposed scheme is inherently more robust and fault tolerant from this perspective when H-caches are disabled. If H-caches are enabled however, temporary failures such as network disconnects can cause clients to read/write stale data. Further, the centralized meta-data server with no built-in support for replication is still a deterrent from the point of view of fault-tolerance and availability. We hope to address both these issues as future extensions.

```
struct plugin_policy_ops {
handle (*pre_open)(force_commit, use_hcache,
    hcache_coherence, delay_commit, num_hashes);
int (*post_open)(void *handle);
int (*pre_close)(void *handle);
int (*post_close)(void *handle);
int (*pre_read)(void *handle, size, offset);
int (*post_read)(void *handle, size, offset);
int (*pre_write)(void *handle, size, offset,
    int *delay_wc);
int (*post_write)(void *handle, sha1_hashes *old,
    sha_hashes *new);
int (*pre_sync)(const char *);
int (*post_sync)(void *handle);
};
```

Client-Side Plug-in API

```
int hcache_get(void *handle, begin_chunk, nchunks,
      void *buf);
int hcache_put(void *handle, begin_chunk, nchunks,
      const void *buf);
int hcache_clear(void *handle);
int hcache_clear_range(void *handle, begin_chunk,
      nchunks);
void hcache_invalidate(void);

int dcache_get(char *hash, void *buf, size);
int dcache_put(char *hash, const void *buf, size);
int commit(void *handle, sha1_hashes *old_hashes,
      sha1_hashes *new_hashes,
      sha1_hashes *current_hashes);
```

CAPFS Client-Daemon: Core API

Fig. 3.6. The client-side plug-in API and the CAPFS client-daemon core API. On receiving a system call, the CAPFS client-daemon calls the corresponding user-defined pre- and post- functions, respectively, before servicing the system call.

3.3.6 Client-side Plug-in Architecture

The CAPFS design incorporates a client-side plug-in architecture that allows users to specify their own consistency policy to fine tune their application's performance. Figure 3.6 shows the hooks exported by the client-side and what callbacks a plug-in can register with the client-side daemon. Each plug-in is also associated with a "unique" name and identifier. The plug-in policy's name is used as a command-line option to the mount utility to indicate the desired consistency policy. The CAPFS client-side daemon loads default values based on the command-line specified policy name at mount time. The user is free to define any of the callbacks in the plug-ins (setting the remainder to NULL), and hence choosing the best trade-off between throughput and consistency for the application. The plug-in API/callbacks to be defined by the user provide a flexible and extensible way of defining a large range of (possibly non-standard) consistency policies. Additionally, other optimizations such as pre-fetching of data or hashes, delayed commits, periodic commits(e.g., commit after "t" units of time, or commit after every "n" requests), and others can be accommodated by the set of callbacks shown in Figure 3.6). For standard cases, we envision that the callbacks be used as follows.

Setting Parameters at Open: On mounting the CAPFS file system, the client-side daemon loads default values for `force_commit`, `use_hcache`, `hcache_coherence`, `delay_commit`, and `num_hashes` parameters. However, these values can be overridden on a per-file basis as well by providing a non-NULL `pre_open` callback. Section 3.3.4.4 indicates that in a commit operation, a client tells the server what it thinks the old hashes for the data are and then asks the server to replace them with new, locally calculated hashes. Hence a commit operation fails if the old hashes supplied by the client do not match the ones currently on the server (because

of intervening commits by other clients). On setting the `force_commit` parameter, the client forces the server into accepting the locally computed hashes, overwriting whatever hashes the server currently has. The `use_hcache` parameter indicates whether the policy desires to use the H-Cache. The `hcache_coherence` parameter is a flag that indicates to the server the need for maintaining a coherent H-cache on all the clients that may have stale entries. The `delay_commit` indicates whether the commits due to writes should be delayed (buffered) at the client. The `num_hashes` parameter specifies how many hashes to fetch from the meta-data server at a time. These parameters can be changed by the user by defining a `pre_open` callback in the plug-in (Figure 3.6). This function returns a handle, which is cached by the client and is used as an identifier for the file. This handle is passed back to the user plug-in in `post_open` and other subsequent callbacks until the last reference to the file is closed. For instance, a plug-in implementing an AFS session like semantics [53] would fetch all hashes at the time of open, delay the commits till the time of a `close`, set the `force_commit` flag and commit all the hashes of a file at the end of the session.

Prefetching and Caching: Prior to a read, the client daemon invokes the `pre_read` callback (if registered). We envision that the user might desire to check H-Cache and D-Cache and fill them using the appropriate hcache_get/dcache_get API (Figure 3.6) exported by the client daemon. This callback might also be used to implement prefetching data, hashes, and the like.

Delayed commits: A user might overload the `pre_write` callback routine to implement delayed commits over specific byte ranges. One possible way of doing this is to have the `pre_write` callback routine set a timer (in case a policy wishes to commit every "t" units of time) that would invoke the `post_write` on expiration. But for the moment, `pre_write` returns a value for `delay_wc` (Figure 3.6) to indicate to the core daemon that the write commits

need to be delayed or committed immediately. Hence, on getting triggered, the `post_write` checks for pending commits and then initiates them by calling the appropriate core daemon API (`commit`). The `post_write` could also handle operations such as flushing or clearing the caches.

Summary: The callbacks provide enough flexibility to let the user choose when and how to implement most known optimizations (delayed writes, prefetching, caching, etc.) in addition to specifying any customized consistency policies. By passing in the offsets and sizes of the operations to the callback functions such as `pre_read, pre_write`, plug-in writers can also use more specialized policies at a very fine granularity (such as optimizations making use of MPI derived data-types [41]). This description details just one possible way of doing things. Users can use the API in a way that suits their workload, or fall back on standard predefined policies. Note that guaranteeing correctness of execution is the prerogative of the plug-in writer. Implementation of a few standard policies (Sequential, SESSION-like, NFS-like) and others (Table 3.8 in Section 3.4) indicate that this step does not place an undue burden on the user. The above plug-ins were implemented in less than 150 lines of C code.

One must also note, that in this scenario all clients are co-operative and mount the same file-system with same parameters. In case the same file or file-system is mounted (or opened) with different parameters at different clients, results of any ensuing operations could be unpredictable.

3.4 Experimental Results

Our experimental evaluation of CAPFS was carried out on an IBM pSeries cluster. with the following configuration. There are 20 compute nodes each of which is a dual hyper-threaded

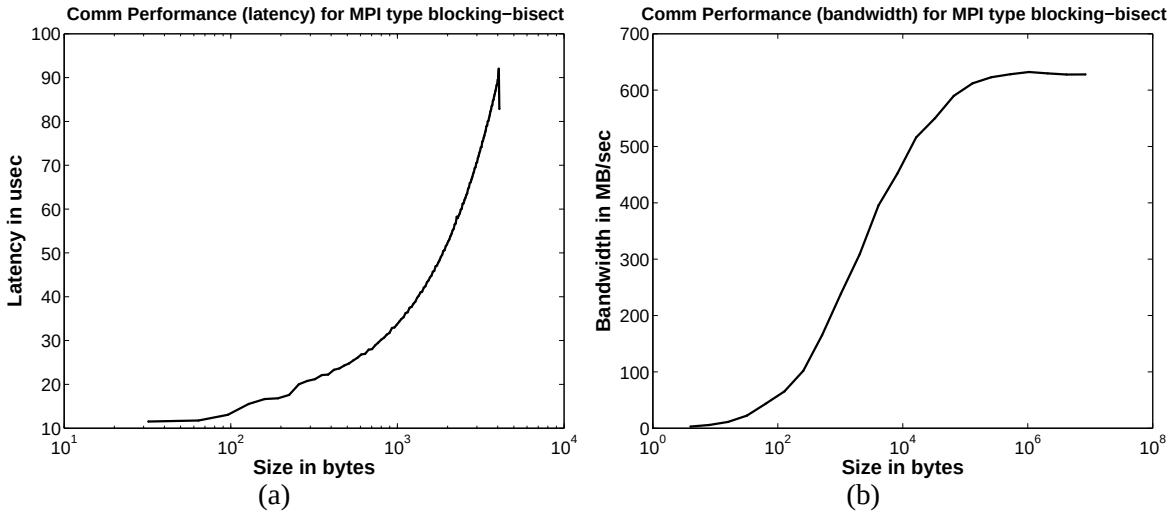


Fig. 3.7. Myrinet (a) Point-to-point latency in μsec (b) Bisection bandwidth in MB/s

Xeon clocked at 2.8 GHz, equipped with 1.5 GB of RAM, a 36 GB SCSI disk and a 32-bit Myrinet card (LANai9.0 clocked at 134 MHz). The nodes run Redhat 9.0 with Linux 2.4.20-8 kernel compiled for SMP use and GM 1.6.5 used to drive the Myrinet cards. Our I/O configuration includes 16 CAS servers with one server doubling as both a meta-data server and a CAS server. All newly created files are striped with a stripe size of 16 KB and use the entire set of servers to store the file data. A modified version of MPICH 1.2.6 distributed by Myricom for GM was used in our experimental evaluations.

3.4.1 Network Performance

We first evaluate the network performance of the clusters, by using the *mpptest* program supplied by the MPICH distribution. This test evaluates the point-to-point message latencies and bisection bandwidth for varying message sizes and the results are shown in Figures 3.7 (a) and (b). The bisection bandwidth serves as a useful yardstick to compare the aggregate bandwidths

that our system achieves (the bisection bandwidth was computed over the same 16 node subset that house the CAS servers). We find that our cluster's bisection bandwidth peaks around 630 MB/s for a message size of 1 MB, and the point-to-point latencies for a message size of 4 KB is around 90 microseconds. The command line used for measuring the bisection bandwidth is shown below,

```
mpirun -np <np> -machinefile <machinefilename> \
./examples/perftest/mpptest -bisect -logscale
```

3.4.2 Aggregate Bandwidth Tests

Since the primary focus of parallel file systems is aggregate throughput, our first workload is a parallel MPI program(*pvfs_test.c* from the PVFS distribution), that determines the aggregate read/write bandwidths and verifies correctness of the run. The block sizes, iteration counts, and number of clients are varied in different runs. Consequently, this workload demonstrates concurrent-write sharing and sequential-write sharing patterns, albeit not simultaneously. Times for the read/write operations on each node are recorded over ten iterations and the maximum averaged time over all the tasks is used to compute the bandwidth achieved. The graphs for the above workload plot the aggregate bandwidth (in MB/s) on the y-axis against the total data transferred to or from the file system (measured in MB). The total data transferred is the product of the number of clients, block size and the number of iterations.

We compare the performance of CAPFS against a representative parallel file system – PVFS (Version 1.6.4). To evaluate the flexibility and fine-grained performance tuning made

possible by CAPFS' plug-in infrastructure, we divide our experimental evaluation of into categories summarized in Table 3.8. Five simple plug-ins have been implemented to demonstrate the performance spectrum.

The values of the parameters in Table 3.8 — (*force_commit*, *hcache_coherence* and *use_hcache*) dictate the consistency policies of the file system. The *force_commit* parameter indicates to the meta-data server that the commit operation needs to be carried out without checking for conflicts and being asked to retry. Consequently, this parameter influences write performance. Likewise, the *hcache_coherence* parameter indicates to the meta-data server that a commit operation needs to be carried out in strict accordance with the H-cache coherence protocol. Since the commit operation is not deemed complete until the H-cache coherence protocol finishes, any consistency policy that relaxes this requirement is also going to show performance improvements for writes. Note that neither of these two parameters is expected to have any significant effect on the read performance of this workload. On the other hand, using the H-cache on the client-side (*use_hcache* parameter) has the potential to improving the read performance because the number of RPC calls required to reach the data is effectively halved.

Policy Name	Use Hcache	Force Commit	Hcache Coherence
SEQ-1	0	0	X
SEQ-2	1	0	1
FOR-1	0	1	X
FOR-2	1	1	1
REL-1	1	1	0

Fig. 3.8. Design space constituting a sample set of consistency policies: SEQ-1, SEQ-2 implement sequential consistency; FOR-1, FOR-2 implement a slightly relaxed mechanism where commits are forced; REL-1 implements an even more relaxed mechanism. The X in rows 1 and 3 denotes a don't care for the variable's value.

The first two rows of Table 3.8 illustrate two possible ways of implementing a sequentially consistent file system. The first approach denoted as SEQ-1, does not use the H-cache (and therefore H-caches need not be kept coherent) and does not force commits. The second approach denoted as SEQ-2, uses the H-cache, does not force commits, and requires that H-caches be kept coherent. Both approaches implement a sequentially consistent file system image and are expected to have different performance ramifications depending on the workload and the degree of sharing.

The third and fourth rows of Table 3.8 illustrate a slightly relaxed consistency policy where the commits are forced by clients instead of retrying on conflicts. The approach denoted as FOR-1, does not use the H-cache (no coherence required). The approach denoted as FOR-2, uses the H-cache and requires that they be kept coherent. One can envisage that such policies could be used in mixed-mode-environments where files are possibly accessed or modified by non-overlapping MPI jobs as well as unrelated processes.

The fifth row of Table 3.8 illustrates an even more relaxed consistency policy denoted as REL-1, that forces commits, uses the H-cache, and does not require that the H-caches be kept coherent. Such a policy is expected to be used in environments where files are assumed to be non-shared among unrelated process or MPI-based applications or in scenarios where consistency is not desired. Note that it is the prerogative of the application-writer or plug-in developers to determine whether the usage of a consistency policy would violate the correctness of the application's execution.

Read Bandwidth: In the case of the aggregate read bandwidth results (Figures 3.9(a) and 3.9(b)), the policies using the H-cache (SEQ-2, FOR-2, REL-1) start to perform better in comparison to both PVFS and policies not using the H-cache (SEQ-1, FOR-1). This tipping

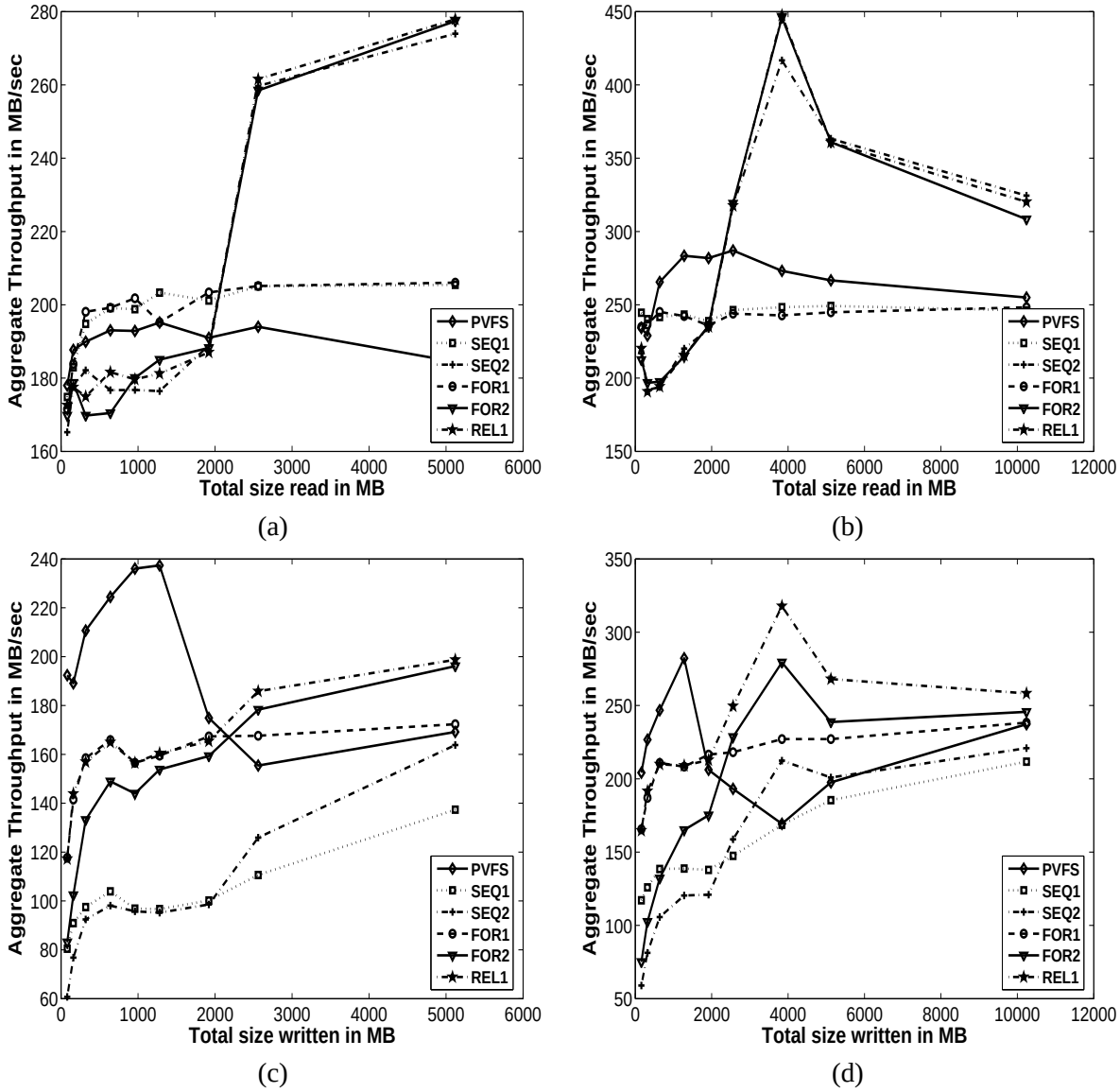


Fig. 3.9. Aggregate Bandwidth in MB/s with varying block sizes: CAPFS vs. PVFS for (a) read-8-clients, (b) read-16-clients, (c) write-8-clients, (d) write-16-clients.

point occurs when the amount of data being transferred is fairly large (around 3 GB). This is intuitively correct, because the larger the file, the greater the number of hashes that need to be obtained from the meta-data server. This requirement imposes a higher load on the server and leads to degraded performance for the uncached case. The sharp drop in the read bandwidth for the H-cache based policies (beyond 4 GB) is an implementation artifact caused by capping the maximum number of hashes that can be stored for a particular file in the H-cache.

On the other hand, reading a small file requires proportionately fewer hashes to be retrieved from the server, as well as fewer RPC call invocations to retrieve the entire set of hashes. In this scenario, the overhead of indexing and retrieving hashes from the H-cache is greater than the time it takes to fetch all the hashes from the server in one shot. This is responsible for the poor performance of the H-cache based policies for smaller file sizes. In fact, a consistency policy that utilizes the H-cache allows us to achieve a peak aggregate read bandwidth of about 450 MB/s with 16 clients. This is almost a 55% increase in peak aggregate read bandwidth in comparison to PVFS which achieves a peak aggregate read bandwidth of about 290 MB/s. For smaller numbers of clients, even the policies that do not make use of the H-cache perform better than PVFS.

In summary, for medium to large file transfers, from an aggregate read bandwidth perspective, consistency policies using the H-cache (SEQ-2, FOR-2, REL-1) outperform both PVFS and consistency policies that do not use the H-cache (SEQ-1, FOR-1).

Write Bandwidth: As explained in Section 3.3.3, write bandwidths on our system are expected to be lower than read bandwidths and these can be readily corroborated from Figures 3.9(c) and 3.9(d). We also see that PVFS performs better than all of our consistency policies for smaller data transfers (upto around 2 GB). At around the 1.5–2 GB size range, PVFS

experiences a sharp drop in the write bandwidth because the data starts to be written out to disk on the I/O servers that are equipped with 1.5 GB physical memory. On the other hand no such drop is seen for CAPFS. The benchmark writes data initialized to a repeated sequence of known patterns. We surmise that CAPFS exploits this commonality in the data blocks, causing the content-addressable CAS servers to utilize the available physical memory more efficiently with fewer writes to the disk itself.

At larger values of data transfers (greater than 2 GB), the relaxed consistency policies that use the H-cache (REL-1, FOR-2) outperform both PVFS and the other consistency policies (SEQ-1, SEQ-2, FOR-1). This result is to be expected, because the relaxed consistency semantics avoid the expenses associated with having to retry commits on a conflict and the H-cache coherence protocol. Note that the REL-1 scheme outperforms the FOR-2 scheme as well, since it does not perform even the H-cache coherence protocol. Using the REL-1 scheme, we obtain a peak write bandwidth of about 320 MB/s with 16 clients, which is about a 12% increase in peak aggregate write bandwidth in comparison to that of PVFS, which achieves a peak aggregate write bandwidth of about 280 MB/s.

These experiments confirm that performance is directly influenced by the choice of consistency policies. Choosing an overly strict consistency policy such as SEQ-1 for a workload that does not require sequential consistency impairs the possible performance benefits. For example, the write bandwidth obtained with SEQ-1 decreased by as much as 50% in comparison to REL-1. We also notice that read bandwidth can be improved by incorporating a client-side H-cache. For example, the read bandwidth obtained with SEQ-2 (FOR-2) increased by as much as 80% in comparison to SEQ-1 (FOR-1). However, this does not come for free, because the policy may require that the H-caches be kept coherent. Therefore, using a client-side H-cache

may have a detrimental effect on the write bandwidth. All of these performance ramifications have to be carefully addressed by the application designers and plug-in writers before selecting a consistency policy.

We now turn our attention to the performance of two well-structured, parallel scientific applications on our file system to measure the performance impact of consistency policies. The first application is a tiled visualization code obtained from [95] that simulates the I/O access patterns of parallel visualization tools. and the second application is the NAS BTIO [7] (Version 2.4) benchmark from NASA Ames Research Center. that simulates the I/O access patterns of a time-stepping flow solver that periodically dumps its solution matrix.

3.4.3 Tiled I/O Benchmark

Tiled visualization codes are used to study the effectiveness of today's commodity-based graphics systems in creating parallel and distributed visualization tools. In this experiment, we use a version of the tiled visualization code [95] that uses multiple compute nodes, where each compute node takes high-resolution display frames and reads only the visualization data necessary for its own display.

We use nine compute nodes for our testing, which mimics the display size of the visualization application. The nine compute nodes are arranged in the 3 x 3 display as shown in Figure 3.10, each with a resolution of 1024 x 768 pixels with 24-bit color. In order to hide the merging of display edges, there is a 270-pixel horizontal overlap and a 128-pixel vertical overlap. Each frame has a file size of about 118 MB, and our experiment is set up to manipulate a set of 5 frames, for a total of about 600 MB.

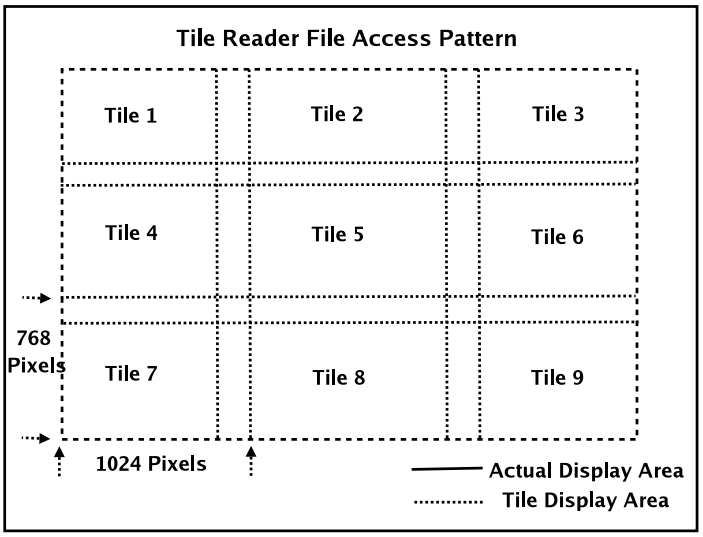


Fig. 3.10. Tile reader file access pattern: Each processor reads data from a display file onto local display (also known as a tile).

This application can be set up to run both in collective I/O mode [41], wherein all the tasks of the application perform I/O collectively, and in non-collective I/O mode. Collective I/O refers to an MPI I/O optimization technique that enables each processor to do I/O on behalf of other processors if doing so improves the overall performance. The premise upon which collective I/O is based is that it is better to make large requests to the file system and cheaper to exchange data over the network than to transfer it over the I/O buses. Once again, we compare CAPFS against PVFS for the policies described earlier in Table 3.8. All of our results are the average of five runs. A sample command line is,

```
 mpirun -np 9 -machinefile emp ./mpi-tile-io-gm --nr_tiles_x 3
--nr_tiles_y 3 --sz_tile_x 1024 --sz_tile_y 768 --sz_element 24
--overlap_x 270 --overlap_y 128 --filename foo
```

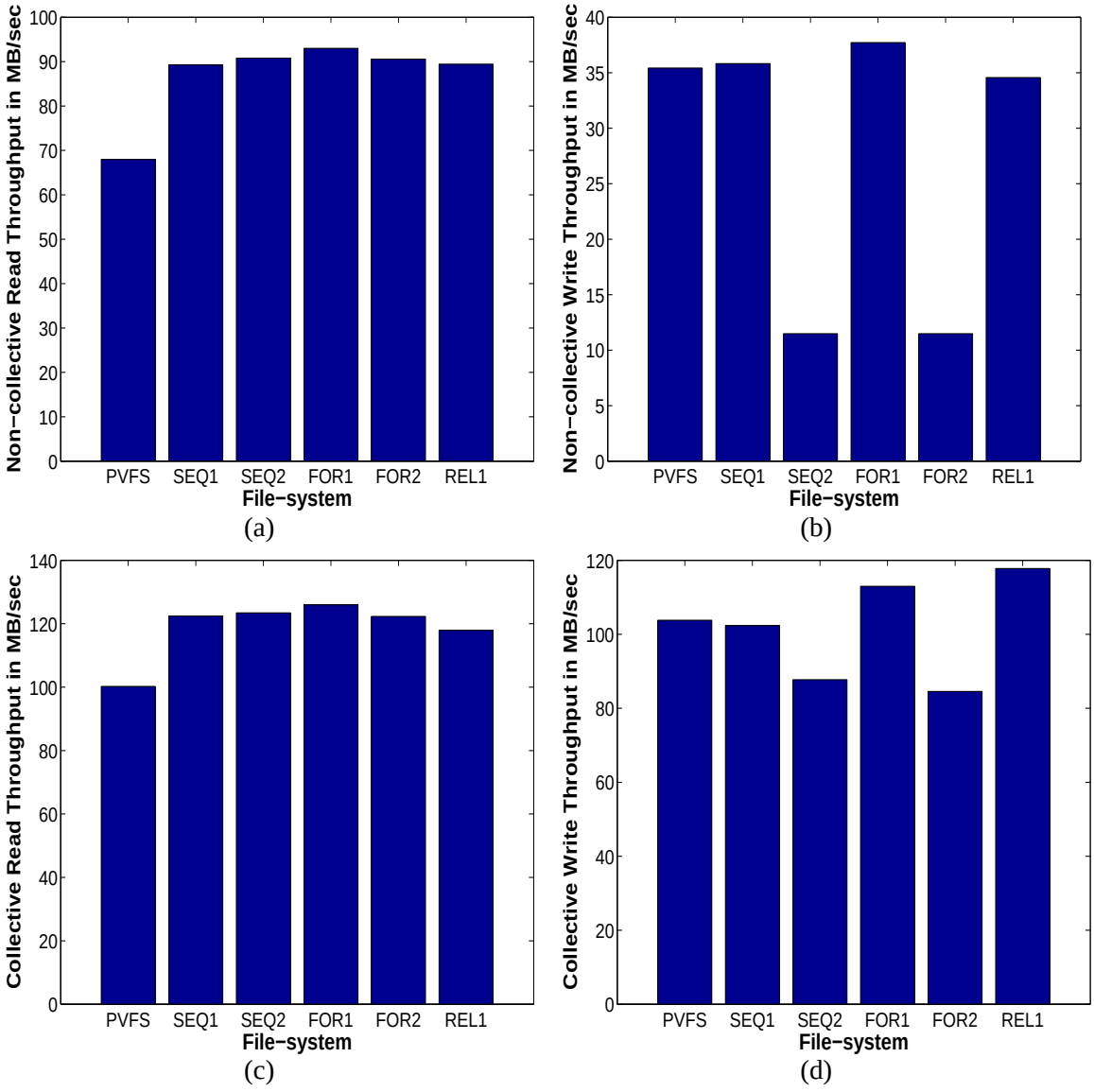


Fig. 3.11. Tile I/O benchmark bandwidth in MB/s: (a) non-collective read, (b) non-collective write, (c) collective read, (d) collective write.

Read Bandwidth: The aggregate read bandwidth plots (Figures 3.11(a) and 3.11(c)), indicate that CAPFS outperforms PVFS for both the non-collective and the collective I/O scenarios, across all the consistency policies. Note that the read phase of this application can benefit only if the policies use the H-caches (if available). As we saw in our previous bandwidth experiments, benefits of using the H-cache start to show up only for larger file sizes. Therefore, read bandwidths for policies that use the H-cache are not significantly different from those that don't in this application. Using our system, we achieve a maximum aggregate read bandwidth of about 90 MB/s without collective I/O and about 120 MB/s with collective I/O. These results translate to a performance improvement of 28% over PVFS read bandwidth for the noncollective scenario and 20% over PVFS read bandwidth for the collective scenario.

Write Bandwidth: The aggregate write bandwidths paint a different picture. For non-collective I/O, Figure 3.11 (b), the write bandwidth is very low for two of our policies (SEQ-2, FOR-2). The reason is that both these policies use an H-cache and also require that the H-caches be kept coherent. Also, the non-collective I/O version of this program makes a number of small write requests. Consequently, the number of H-cache coherence messages (invalidates) also increases, which in turn increases the time it takes for the writes to commit at the server. One must also bear in mind that commits to a file are serialized by the meta-data server and could end up penalizing other writers that are trying to write simultaneously to the same file. Note that the REL-1 policy does not lose out on write performance despite using the H-cache, since commits to the file do not execute the expensive H-cache coherence protocol. In summary, this result indicates that if a parallel workload performs a lot of small updates to a shared file, then any consistency policy that requires H-caches to be kept coherent is not appropriate from a performance perspective.

Figure 3.11(d) plots the write bandwidth for the collective I/O scenario. As stated earlier, since the collective I/O optimization makes large, well-structured requests to the file system, all the consistency policies (including the ones that require coherent H-caches) show a marked improvement in write bandwidth. Using our system, we achieve a maximum aggregate write bandwidth of about 35 MB/s without collective I/O and about 120 MB/s with collective I/O. These results translate to a performance improvement of about 6% over PVFS write bandwidth for the non-collective scenario and about 13% improvement over PVFS write bandwidth for the collective scenario.

3.4.4 NAS BTIO Benchmark

The BTIO benchmark (Version 2.4) [7] from NASA Ames Research Center simulates the I/O required by a time-stepping flow solver that periodically writes its solution matrix. The solution matrix is distributed among processes by using a multi-partition distribution in which each process is responsible for several disjoint sub-blocks of points (cells) of the grid. The solution matrix is stored on each process as C three-dimensional arrays, where C is the number of cells on each process (the arrays are actually four dimensional, but the first dimension has only five elements and is not distributed). Data is stored in the file in an order corresponding to a column-major ordering of the global solution matrix.

The access pattern in BTIO is non-contiguous in memory and in file and is therefore difficult to handle efficiently with the UNIX/POSIX I/O interface. Therefore, we used the "full MPI-IO" version of this benchmark, which uses MPI derived data-types to describe non-contiguity in memory and file and also uses a single collective I/O function to perform the entire I/O. We ran the Class A problem size, which uses a 64x64x64 element array with a total size of 400 MB

and tests were run using 4, 9, and 16 compute nodes (the benchmark requires that the number of compute nodes be a perfect square).

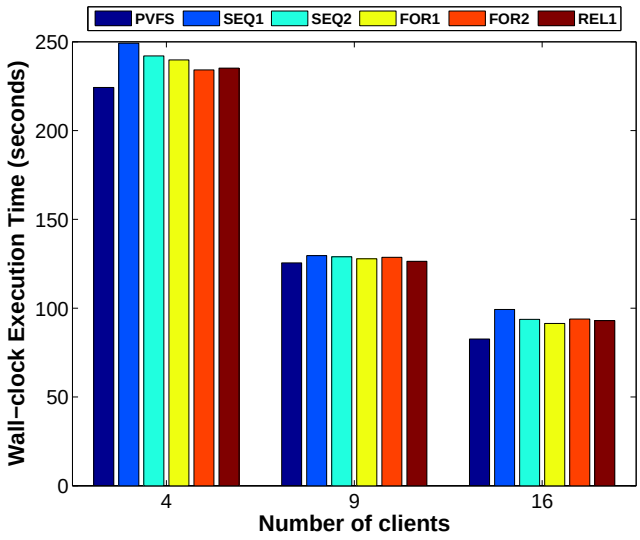


Fig. 3.12. Execution time for the NAS BTIO benchmark.

Figure 3.12 shows the wall-clock time for execution of this benchmark for the different file system configurations and number of compute nodes. We notice that as the number of clients increases, the overall execution time decreases for both the file systems. Further, we also see that all our consistency policies perform almost as well as PVFS with minimal overheads. With REL-1 which is the most relaxed policy in our system, the execution time of the application is about 10% slower than that of PVFS for 16 processors, and with SEQ-1 which is the most strict policy in our system, the execution time of the application is about 20% slower than that of PVFS for 16 processors. For this workload, the performance benefits with using a relaxed policy (such as

REL-1) over a stringent policy (such as SEQ-1) is higher for a smaller number of clients than for a larger number of clients indicating that this workload can benefit with an even more relaxed policy (such as a delayed commit policy) for a larger number of clients.

3.5 Related Work

Many efforts have sought to solve the consistency problem in the context of distributed file systems. Fundamentally, any approach to providing a consistency needs to address two issues: write atomicity/serialization (ensuring that writes appear atomic and are seen in the same order by all clients, especially in a system with multiple copies of data and across blocks that may span nodes) and write propagation issues (ensure that writes are visible to other processes). Therefore, we discuss past efforts to solve this problem based on these two categories.

Distributed file systems such as AFS [53], NFS [104] and Sprite [8, 85] have only a single server that doubles in functionality as a meta-data and data server. Because of the centralized nature of the servers, write atomicity is fairly easy to implement. Client-side caches still need to be kept consistent however, and it is with respect to this issue (write propagation) that these approaches differ from the CAPFS architecture. The Sprite distributed file system [85] uses a stateful server approach that keeps track of concurrently open sessions of every file in the system. Sessions that have opened the file in read mode are informed by means of a callback from the server that their caches are stale in case the file has been written to by some other client. If the server detects that there are concurrent write sessions, then it informs the relevant clients to write-through, effectively disabling the cache for the entire period of the operation. The venerable Network File System [104] implements a mostly stateless server, and hence the onus is on the client to periodically ensure that state of their cache is correct. AFS [53] implements

write-on-close semantics (a variant of the session semantics discussed earlier), where the server keeps track of clients caching a file, and when the files are closed the changes are written back to the server, which then notifies any other clients that may have cached the file. Coda [61], a descendant of AFS also uses callbacks which is basically a guarantee provided by the servers that clients would be notified when their cached copies are no longer valid. Coda differs from AFS in that it allows for server replication, that allows volumes to have read-write replicas at more than one server which is referred to as the volume storage group. Since, there are multiple servers, the write atomicity problem is solved by having modifications propagated in parallel to all available volume storage groups, and eventually to those that missed the updates. However, with the exception of Sprite, none of the other file systems offer a sequentially consistent file system image for the sake of performance and due to the different domains of deployment.

Since the beginning of distributed computing, there have been many efforts to build distributed file-servers all of which support mechanisms for concurrency control. Many of the earliest distributed file servers use variants of locks for concurrency control such as XDFS [119], Felix [42] and Alpine [20], while some such as SWALLOW [123, 97] use timestamps. Parallel file systems such as GPFS [110] and Lustre [15] employ distributed locking to synchronize parallel read-write disk accesses from multiple client nodes to its shared disks. The locking protocols are designed to allow maximum throughput, parallelism, and scalability, while simultaneously guaranteeing that file system consistency is maintained. Every file system operation acquires an appropriate read/write lock to synchronize with conflicting operations. In addition to a centralized lock manager, each node in the cluster also runs a local lock manager. The global, centralized lock manager coordinates locks between local lock managers by handing out tokens, which can be revoked at a later point on a conflicting access. Although such algorithms can

be highly tuned and efficient, failures of clients can significantly complicate the recovery process. Hence any locking-based consistency protocol needs additional distributed crash recovery algorithms or lease-based timeout mechanisms to guarantee correctness. Frangipani [125] and Petal [70] together implement a distributed storage system. Frangipani is a distributed file system built to operate on top of a distributed virtual disk interface provided by the Petal system. Analogous to GPFS, coherence in Frangipani for data and meta-data is maintained with the help of a distributed lock server that provides multiple-reader/single-writer locks to clients on the network, and uses leases [49] to handle with client failures. Likewise, the Global File System (GFS) [91, 92] (a shared-disk, cluster file system) also uses fine-grained SCSI locking commands, lock-caching and callbacks for performance and synchronization of accesses to shared disk blocks, and leases, journalling [92] for handling node failures and replays. The CAPFS file system eliminates much of the client state from the entire process, and hence client failures do not need any special handling.

Providing a plug-in architecture for allowing the user to *define* their own consistency policies for a parallel file system is a contribution unique to CAPFS file system. Tunable consistency models and tradeoffs with availability have been studied in the context of replicated services by Yu et al. [142, 143]. Swarm [122] provides the user with a choice of consistency policies for a wide-area object store.

Sprite-LFS [103] proposed a new technique for disk management, where all modifications to a file system are recorded sequentially in a log, which speeds crash recovery and writes. An important property in such a file system is that no disk block is ever overwritten (except after a disk block is reclaimed by the cleaner). Content-addressability helps the CAPFS file system gain this property, wherein updates from a process do not overwrite any existing disk or

file system blocks. Recently, content-addressable storage paradigms have started to evolve that are based on distributed hash tables like Chord [116]. A key property of such a storage system is that blocks are addressed by the cryptographic hashes of their contents, like SHA-1 [87]. Tolia et al. [127] propose a distributed file system CASPER that utilizes such a storage layer to opportunistically fetch blocks in low-bandwidth scenarios. Usage of cryptographic content hashes to represent files in file systems has been explored previously in the context of Single Instance Storage [11], Farsite [2], and many others. Similar to log-structured file systems, these storage systems share a similar no-overwrite property because every write of a file/disk block has a different cryptographic hash (assuming no collisions). CAPFS uses content-addressability in the hope of minimizing network traffic by exploiting commonality between data block, and to reduce synchronization overheads, by using hashes for cheap update based synchronization. The no-overwrite property that comes for free with content addressability has been exploited to provide extra concurrency at the data servers.

In the context of multi-processor systems, Herlihy proposed Lock-free and wait-free synchronization algorithms [52] that makes use of the load-linked and store-conditional instructions [58, 75, 112] provided by the micro-processor that effectively bounds the number of steps before which an operation to a shared region will complete. In the context of databases, optimistic concurrency algorithms [3, 126] have been proposed that focuses on delaying the locking of shared regions to achieve better scalability. An implication of the above property in the optimistic model is that transactions could fail and hence would need to be retried, while traditional pessimistic models obtain locks prior to updates and hence would always succeed. The proposed system makes use of a similar property by detecting conflicting operations and retrying them.

3.6 Chapter Summary

In this chapter, we have presented the design and implementation of a robust, high-performance parallel file system that offers user-defined consistency at a user-defined granularity using a client-side plug-in architecture. To the best of our knowledge CAPFS is the only file system that offers tunable consistency that is also user-defined and user-selectable at run-time. Rather than resorting to locking for enforcing serialization for read-write sharing or write-write sharing, CAPFS uses an optimistic concurrency control mechanism. Unlike previous network/parallel file system designs that impose a consistency policy on the users, our approach provides the mechanisms and defers the policy to application developers and plug-in writers.

We now look into more traditional uses of CAS – for saving storage space and network bandwidth. We next present a study of data collected from execution of real world application benchmarks, to analyze the pros and cons of using CAS.

Chapter 4

Content Addressable Storage : Pros and Cons

4.1 Introduction

Content Addressable Storage or CAS has been an increasingly popular tool in recent systems literature, often used as a silver bullet to manage large datasets by reducing their storage and network bandwidth requirements [12, 4, 93, 134, 83]. The above savings are achieved by eliminating duplicate instances of data *chunks* (blocks). Applications that manage large datasets are potential beneficiaries. We illustrate our case with two examples.

First, imagine a data-center where multiple virtual machines are hosted on the same server [1, 55, 76]. On powering up, each virtual machine first fetches its own virtual disk from which it boots its own guest operating system. Presumably, a significant number of the virtual disks have the same operating system. The storage backend that houses all the virtual disks could use CAS to exploit the *commonality* between the virtual disks, thereby over-committing the storage resource, similar to the way virtual machines often over-commit memory. The compute node on the other hand could use a CAS based store locally and exploit commonality across the multiple virtual disks that are hosted locally, thereby allowing more virtual machines to be hosted on a single node. The use of a CAS based cache, could reduce startup time (potentially dominated by the time required to fetch the virtual disk) from ‘a few minutes’ [55] to a smaller amount by exploiting commonality between data to be fetched and data available locally.

Or take the case of a grid application that periodically checkpoints its data for recovery or visualization purposes. For example, the TeraShake project generates close to 50 TB of data from one run [89]. Of this about 43 TB comprised of data generated iteratively, each time step. Such behavior is not uncommon for scientific applications. In these applications another requirement is to drain the low capacity local scratch storage volume so that the data from the next iteration can be stored. CAS can once again be useful here by exploiting any commonality across iterations. The savings achieved in the storage and network bandwidth requirements can potentially change a task (like 'on the fly' graphics for TeraShake) from the realm of infeasible to feasible.

Key to this usage of CAS, is the assumption of finding commonality in data. Commonality however, is an intrinsic property of the dataset itself. Very few analyses are available into the commonality inherent to datasets themselves. These are in the context of user data like webpages[100], home-directories[80, 12], source-code[128], virtual-machine image snapshots[83], or storing employee's file-systems [93]. We wish to find out how much commonality exists in data from real world applications and what overheads can be expected in exploiting this commonality using CAS. To the best of our knowledge, this is the first study into the use of CAS on real world data from live applications. What has made this work more challenging is that common, publicly available block level or file level traces do not contain enough information about the actual content of the data. Hence we prepared and compiled the applications and used the content addressable file-system platform developed in Chapter 3 to run the application benchmarks *live* and generate traces containing the relevant information.

In this chapter, we investigate, i) the commonality present in real-world data; ii)the storage space savings using CAS; and iii) the network bandwidth savings when using a CAS repository and a local content addressable cache. A CAS based store has hidden overheads affecting not just the amount of storage space required, but also the error resilience of data. By removing duplicate chunks of data, and hence reducing the redundancy of data, CAS magnifies the loss inflicted on storage when data corruption occurs. We comment on these overheads and also on the run-time overheads that are seen when using a CAS-based file system.

We introduce our CAS evaluation methodology and benchmarks in Section 4.2 and evaluate the benefits and challenges posed by CAS in Section 4.3 and Section 4.4 respectively. Section 4.5 has a discussion on the lessons learned from this analysis that might help a design a CAS based system and/or decide suitability of a CAS based system for the application in question. Section 4.6 outlines the related work before summarizing our results in Section 4.7.

4.2 Methodology for Evaluating the Efficacy of CAS

To undertake an analysis of CAS on data generated by an application, one would require a trace-log of all the read/write requests. The requests should contain information about either the content being read/written so that it's CAS name (SHA1 hash) can be generated, or the CAS name (the SHA1 hash) itself. Unfortunately, this is not true of any of the commonly available public traces that we know of. In fact, most traces record just the meta-data information, not the *content* of the requests, which is essential for this study. Even if such a trace with the chunk names were to exist, in order to analyze the application data at a different chunksize, one would need to re-run the application on a file-system using the new chunksize – a simple extrapolation would lead to inaccuracies.

Hence we needed a mechanism to run the application(s), calculate and record the SHA1 hashes of the data and all the I/O requests. For this reason we chose to use our experimental content addressable file-system – CAPFS (described in Chapter 3. As detailed in the previous chapter, each CAS server (data server) exports two primitives, i) *get(hash)*, and ii) *put(hash,data)*. To implement these primitives, the CAS server manages, i) a simple in-memory database of *hash, location* tuples for chunks housed on the server, and ii) the data chunks themselves. In our experimental setup CAPFS was used with POSIX consistency semantics on a single CAS server. The CAS server was configured to use an in-memory hash table as the database, indexed by the 20 byte SHA1 hashes. Using a single CAS server simplified sequencing of the get/put requests in the trace log. Similar trace logging was enabled at all the client nodes running the application benchmark. The trace thus generated at the client and CAS server was sufficient to reconstruct the execution of the benchmark without running it again. We ran CAPFS with various chunksizes and the traces thus obtained were analyzed. Instrumentation of the CAPFS client and server daemons gave valuable insight into CAS based performance issues such as SHA1 hash generation overhead, as described later.

The experiments were run on an IBM xSeries 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.

4.2.1 Applications used as Benchmarks

A synthetically generated dataset may either contain too little commonality (e.g. a dataset made from random data) or might have too many regular patterns (e.g. a dataset made from

Name	Type	Description	Size
gene-dbase	Static Dataset	Dataset containing genes from GenBank	103 MB
btio	Live Application	Part of the NAS parallel benchmark set	400 MB
bssn	Live Application	The BSSN PUGH benchmark from Cactus	1.6 GB
heat-solver	Live Application	A generic heat-solver	106 MB
dbt2	Live Application	OSDL Database Test 2 - a TPCC-like benchmark	306 MB

Fig. 4.1. Benchmarks used

repeating different entries). In essence, commonality is an inherent property of an application's data, and a study of CAS would be highly dependent on the amount of commonality in data. With this factor in mind, we use the five datasets listed in Table 4.1 as benchmarks for studying the performance of CAS.

The *gene-dbase* dataset contains a set of 10 randomly chosen genomes from the NCBI GenBank database [84]. The genomes were uncompressed and un-tarred after download to form the dataset. We were unable to compile a suitable application that uses this dataset. Hence this dataset was statically analyzed as-is for inherent commonality. The other four datasets are the result of the execution of a live application, and are hence indicated as such Table 4.1. The BTIO application [82] 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 application using the full-mpiio version. The *bssn* application is a Cactus [23] benchmark application of a numerical relativity code using finite differences on a uniform grid [21]. It uses the CactusEinstein infrastructure to evolve a vacuum space-time, using the BSSN formulation of the Einstein equations. The *heat-solver* is a standard heat solver written in Fortran using MPI. *btio, bssn* and *heat-solver* 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. *dbt2* is the OSDL Database Test 2 benchmark [35]. It is a TPCC-like benchmark that loads tables into a Mysql server at start-up and runs queries on these 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 it's *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 the execution of all the *live* applications by instrumenting the CAPFS filesystem. The trace logs thus obtained help us to do post-mortem analysis on the data. The above five datasets cover data from three scientific applications, one on-line transaction processing (OLTP) benchmark and one dataset containing archival data (*gene-dbase*). The datasets thus acquired are from different sources and of varied sizes.

4.3 CAS: Pros

In this section we investigate two advantages offered by CAS - i) savings in storage space, and ii) savings in network bandwidth from use of a CAS-based cache at a client.

4.3.1 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 log-based file-system (no in-place writes).

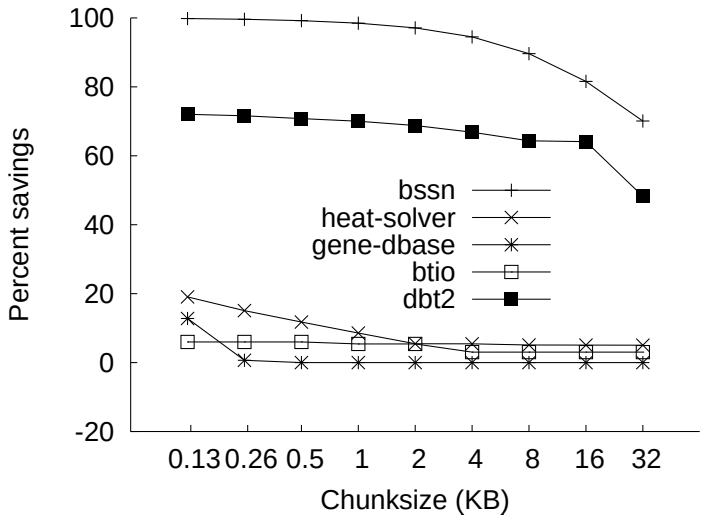


Fig. 4.2. Savings in storage space as a function of chunksize

4.3.1.1 Impact of chunksize

The chunksize in use for a CAS-based store is a tunable parameter that affects its performance. The curves in Figure 4.2 depict the effect of chunksize on the space required to store the datasets under consideration. As one might expect, the savings in storage space are higher at smaller chunksizes. Figure 4.2 shows that all *live* applications benefit from CAS. This in itself is a very surprising result indicating that applications that manage floating point data (scientific applications) and binary records exhibit some commonality.

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 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, falling precipitously from there on. The reason behind the gentle decline in savings till the 16 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. This is not too surprising considering that this dataset contains binary data. The *gene-dbase* savings values when compared with the others, indicates that there could be other reasons for finding commonality at higher chunksizes in the other applications. We investigate this next.

4.3.1.2 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 stay un-modified across iterations (*iterative commonality*). The three scientific applications (*bssn, heat-solver, btio*) have clear, well-defined iterative behavior in data generation. The *dbt2* and *gene-dbase* datasets do not have such iterative data generation patterns and hence any gains from the use of CAS are realized by 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, as shown below.

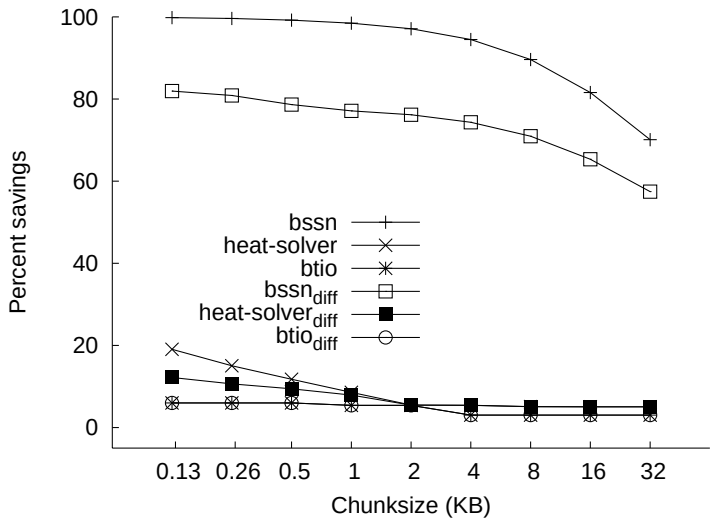


Fig. 4.3. Identifying commonality in data due to iterative behavior

In order to quantify commonality resulting from the iterative nature of an application, we applied a *diff* like filter and compared the data generated across iterations. This was accomplished by comparing the list of hashes for data belonging to one iteration and the next. The savings thus obtained (shown in Figure 4.3) are obtained from data remaining unchanged across iterations. The curves labeled as *bssn*, *heat-solver,* and *btio* are the identical to similarly labeled curves in Figure 4.2, while the curves obtained by applying the *diff* filter appear with a *diff* subscript.

We observe that there is significant iterative commonality. For example, the two curves for *btio and* $btio_{diff}$ are identical indicating that all the savings for this application is due commonality across iterations. Similarly, a large fraction of the savings for the other two applications (*bssn* and *btio*) come from their iteration based behavior. The incidental commonality within data chunks of an iteration is the difference in values between a curve and its diff-based

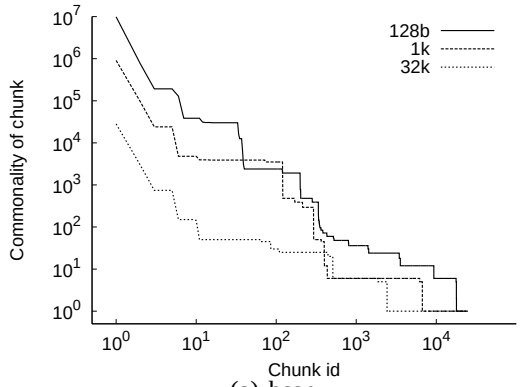
version in Figure 4.3. This value varies not just with the specific application itself, but also with chunksize. For *bssn*, this value is as high as 21% at a 1 KB chunksize. In the case of *heat-solver*, this difference is as high as 7% at a 128-byte chunksize, but tapers down to less than a percent at a 1 KB chunksize, while *btio* hardly has any commonality between chunks of the same iteration.

This brings us to the conclusion that, i) commonality from older iterations provides significant savings, which may be be realized by the use of a non-CAS, diff-based mechanism; and, ii) applications also have commonality across data chunks within the same iteration. Savings from the latter can only be exploited by CAS based schemes. The extent varies not just from application to application, and also with chunksize.

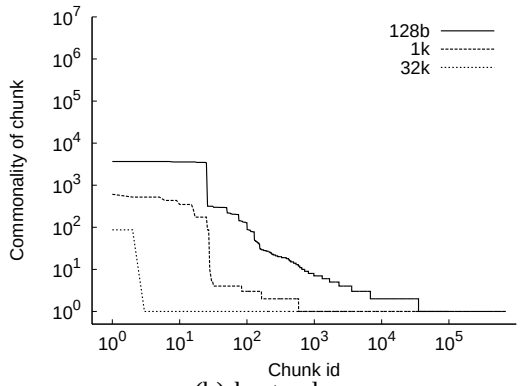
4.3.1.3 Commonality Profile

We now analyze the commonality profile of the data housed at the CAS store for possible clues to the source of commonality in the respective datasets. Figure 4.4 examines a CAS-based store that houses the data generated from an entire run of an application benchmark. The x-axis lists each chunk in the system and its commonality is shown on the y-axis. The chunks are numbered in decreasing order of commonality. The *btio* data (Figure 4.4(c)) stands out for having a low and flat profile, indicating that the dataset has hardly any commonality. The observable commonality of 40 for a large number of chunks, indicates that certain portions of the dataset, never change in value throughout the 40 data-generation iterations of the application.

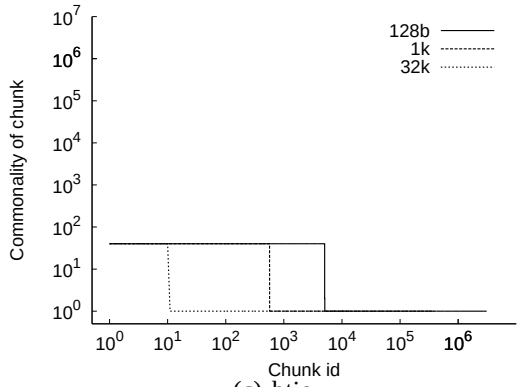
The left-most chunk or chunk #1 is the chunk that occurs most frequently in the data. The *dbt2* data (Figure 4.4(d)) is unique in that if few leftmost chunks are ignored, then the remaining dataset exhibits very little commonality. For example, at 128-byte chunksize, the commonality of the chunks (in order) are 3074290, 2429, 1276, 1249, 474; while at a 1 KB chunksize the



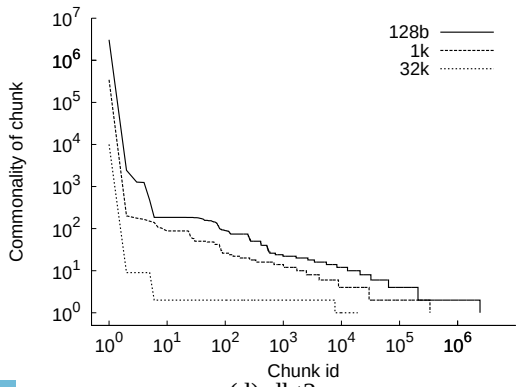
(a) bssn



(b) heat-solver



(c) btio



(d) dbt2

Fig. 4.4. Commonality profile.

commonality of the chunks are 343846, 201 ... and at a 32 KB chunksize the first chunk has a commonality of 10048 followed by a commonality of just 9. Clearly the first chunk brings a *huge* amount of savings. We manually inspected the data to find that this chunk is the *zero chunk* (a chunk composed entirely of zeroes). This is perhaps a result of the database's tablespace allocation policy where 64 contiguous pages (each of 16 KB size) are allocated together to reduce fragmentation. If this chunk is ignored, the *dbt2* data looks surprisingly similar to the *heat-solver* data (Figure 4.4(b)). Similarly, for the *bssn* data (Figure 4.4(a)), at a 128-byte chunksize, the most common chunk occurs over 9 million times. This too was found to be the zero chunk, indicating that the *bssn* data contains sparse matrices. The zero-block provides 76% of the space savings out of the 99% savings shown in Figure 4.2. The remaining 23% savings obtained from non-zero blocks compares favorably with the 19% savings for the *heat-solver* benchmark (Figure 4.2). The *heat-solver* and *btio* data were found to contain no zero blocks at all.

An important lesson here is that applications that use sparse matrices or tables, out of obvious program design or as a result of internal data allocation policies, can benefit tremendously from the use of CAS.

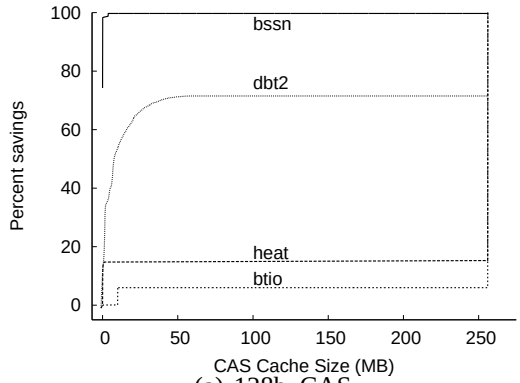
4.3.2 Content Addressable Caching: Savings in Network Bandwidth

The use of CAS impacts the amount of data to be sent over the network in two ways. First, CAS can be used as a compression mechanism. By internally chunking a large read or write into smaller parts, CAS removes the repeated chunks and sending only the unique chunks over the network. Second, CAS can be viewed as a caching technique to eliminate not just reads but also writes to a remote repository. For example, a write may not require the actual data to be written out if the chunk to be written already exists in the CAS repository. At the same time,

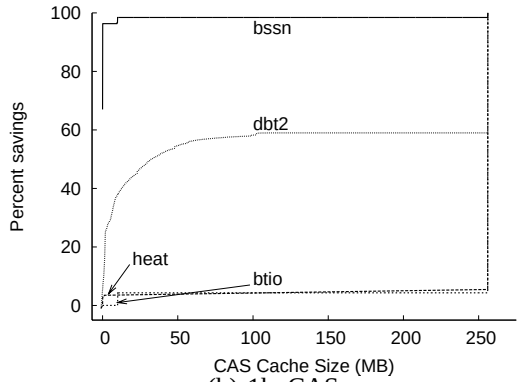
similar to a more traditional data cache, a read need not be requested from the repository if it can be satisfied by another chunk with the same content, available from a locally cached pool of chunks fetched earlier. The results in this section indicate the benefits of using CAS for reducing network network data.

For this analysis, the trace of the read/write requests described in Section 4.2 were used to evaluate the performance of a local cache on the client node, with LRU being the eviction policy. The percentage savings obtained with a buffer of a size S in Figure 4.5 indicates, i) savings when using a CAS based cache of size S bytes 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) represents the size of the data to be sent when no caching used. We compare the effectiveness of the CAS based caches against non CAS caches which would cache data based on file offsets.

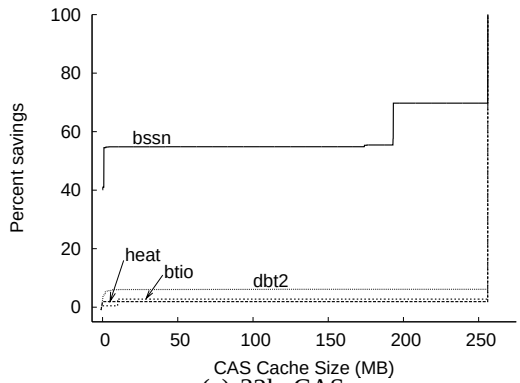
The applications have a mostly sequential write behavior and hence have minimal temporal locality. Hence, the spatial locality of reads accounts for the savings obtained with a traditional (non CAS) cache. We observe in Figure 4.5(d) that the three scientific applications perform very poorly when using a traditional LRU-based cache, achieving less than 0.5% savings even in the best case (32 KB chunksize). The CAS based schemes perform impressively under the same conditions. In fact, we can reason that a CAS based LRU policy will always outperform an LRU based cache – a CAS based cache exploits not just temporal and spatial locality, but also locality of content. For example, a request for a *cold* chunk (a chunk that has never been accessed before), which is not spatially co-located to chunks being accessed recently, can still be satisfied from a CAS based cache, provided another chunk with the same content already exists



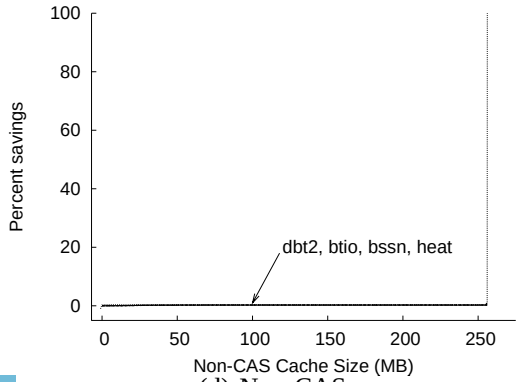
(a) 128b, CAS



(b) 1k, CAS



(c) 32k, CAS



(d) Non-CAS

Fig. 4.5. Percentage savings in network I/O when using a content addressable cache

in the cache ! In cases where there is absolutely no commonality in the data, a CAS based cache will perform as well as a non-CAS LRU cache.

We observe that the use of CAS tremendously benefits *bssn* (Figure 4.5), in-spite of this being a sequential, mostly write-only workload. This result illustrates the ability of a CAS based cache to reduce write traffic by removing writes containing same content, even when temporal locality is absent (sequential workload). As a result, the amount of data to be written by the *bssn* benchmarkcan be reduced by almost 100% at a 1 KB chunksize, using a very small CAS based cache. This is clearly due to the large commonality in *bssn* data, as seen previously.

Overall, the use of CAS brings tangible results for all the three scientific applications. For an iterative application if the CAS based cache is large enough to hold the data from an entire iteration, the next iteration can then exploit commonality across iterations. This is corroborated by Figure 4.5(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 a file of size 2 MB per iteration, a 2 MB sized 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 much smaller 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 4.2, 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 shows remarkable benefits from the use of a CAS based cache, indicating that eliminating redundant data brings very significant gains (Figure 4.5(b)). At a 32

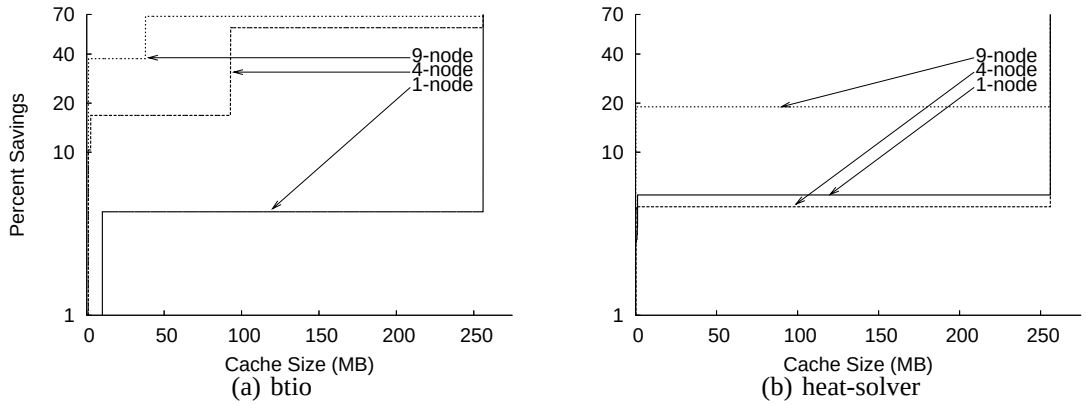


Fig. 4.6. Percentage savings in network I/O with a CAS based cache on a multi-node experiment

KB chunksize, the performance drops noticeably due to the smaller 16 KB internal page size used by the application.

The *heat-solver* application creates new data every iteration as a new file. As a result, a traditional is helpless, while the CAS based cache still has a reasonable hit-rate. This stems from the ability of a CAS based cache to look at *all* the data encountered, even across file-names. However, for a fair comparison, we did not disadvantage the LRU-based non-CAS cache in this manner when calculating the curve for *heat-solver* in Figure 4.5.

Multi-node experiments : We also evaluate the savings obtained with a CAS based cache when running the scientific applications on multiple client nodes (*dbt2* runs on a single node). Figure 4.6 shows the savings obtained on node 0, when running the application on multiple nodes, each using a CAS based cache. Figure 4.6(a) indicates that when each node deals with a specific subset of the whole data, there might be significantly more commonality to be exploited. 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. In fact,

if a larger cache size were (100 MB in the 4-node case, or 50 MB in the 9-node case), then the benefits would be even higher. As the number of nodes increase, the data generated per iteration per node by *btio* decreases, hence a progressively smaller cache size is required to realize the benefits. On the other hand, poor data-partitioning across nodes leads to minimal or negative increase in savings, with increase in number of client nodes, as visible in Figure 4.6(b). Here, in the case of *heat-solver*, on increasing the number of client nodes from 1 to 4, the savings decrease marginally from 5.5% to 4.6%. However, it takes a much smaller cache size (0.5 MB) to achieve these savings with 4-nodes. On going up to 9-nodes, the data savings increase to 19%. The *bssn* data has close to 100% savings for multi-node experiments and hence is not shown.

In summary,

- The use of a content addressable caches is always beneficial over a non-CAS based cache.
- For optimal performance from a CAS based cache with an iterative application, the cache size allocated should be larger than the amount of unique data generated per iteration.
- CAS based caching performs even better when applications partition data across nodes to do the computation in a distributed manner.

4.4 CAS: Cons

In this section we look at the demerits of using CAS. We investigate 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.

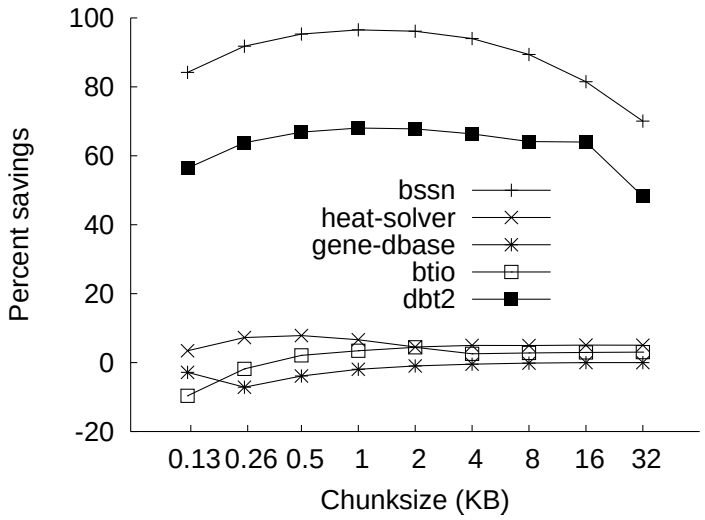


Fig. 4.7. Savings in storage space as a function of chunksize, including meta-data overheads

4.4.1 Meta-Data Overheads

The preceding discussion from Section 4.3.1 indicates that a CAS based store has the potential for significant space savings. As described in Chapter 2, these savings have an added cost – that of maintaining an additional mapping from file offset (chunk number) to the hash (the name) of the chunk. This meta-data, referred to as the *recipe*, stores a hash value for each N bytes of the file (N being the chunksize of the CAS file-system). On deducting the cost of storing the hashes (20 bytes per hash for SHA1), the net savings obtained from the use of CAS are shown in Figure 4.7.

For *bssn* data, the savings increase from their 128-byte value of 84% to a peak of 96.5% at at a 1 KB chunksize. The savings 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 versus the 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 diminishes the savings from CAS, from a value of almost 99% in Figure 4.2 to 84% . As the chunksize increases to 1 KB and 2 KB from the 128 byte value, the commonality decreases marginally (less than 1% in Figure 4.2), 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 trend is also observed for the *heat-solver* and *btio* applications. The *heat-solver* application peaks at a 512-byte chunksize, saving 7.8% storage space and then falls marginally to 6.6% at 1 KB. Beyond that the curve falls further. For small chunksizes, the savings for the *btio* application starts in negative territory – at a 128-byte chunksize, the CAS store requires 9.6% *more* storage than a conventional data store ! Clearly, this is due to the added overhead of having to maintain recipes, while at the same time, not finding any space savings due to commonality. With an increase in chunksize, from a 128-byte value to a 2 KB value, commonality stays relatively constant at around the 5% mark (Figure 4.2), while the meta-data overhead drops 16-fold. As a result, the curve has its highest savings of 4.4% at a 2 KB chunksize. The *gene-dbase* data does not exhibit any commonality (Figure 4.2) and the added meta-data overhead makes the CAS store inefficient.

The above study indicates that small chunksizes of around 1 KB are best for storage space savings. At smaller chunksizes meta-data overheads can even overwhelm any savings from commonality.

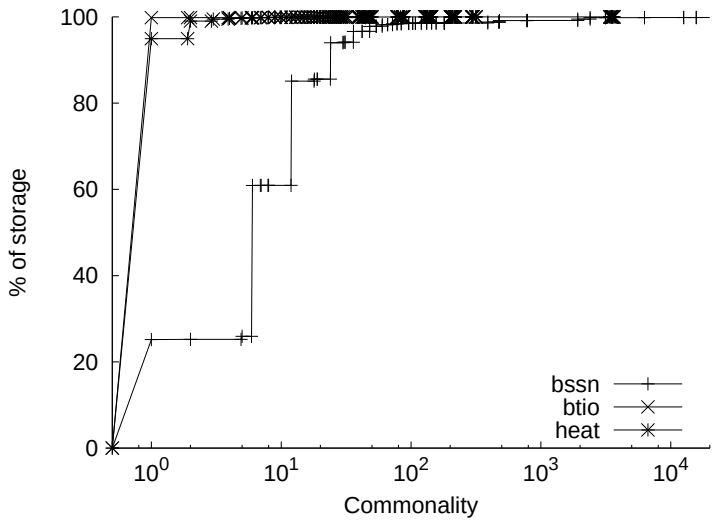


Fig. 4.8. Storage profile for 128-byte chunks

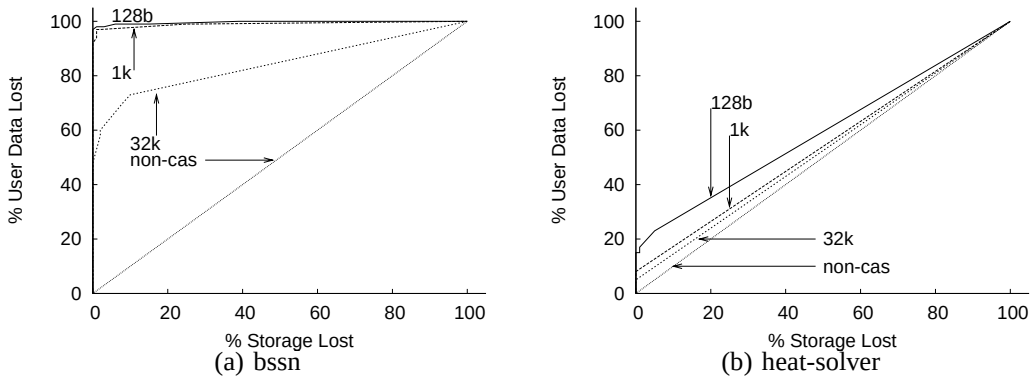
4.4.2 Decreased Error Resilience

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 as the amount of file data (user data) lost upon losing a single chunk in the repository, then for a traditional data store, each chunk would be equally valuable. Under similar conditions, in a CAS based store, a chunk with a higher commonality would be more valuable, implying that it's loss can cause much more damage than in a traditional data store. Using the data from Figure 4.8, 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 user data lost upon destruction of chunk i in the storage system is given by

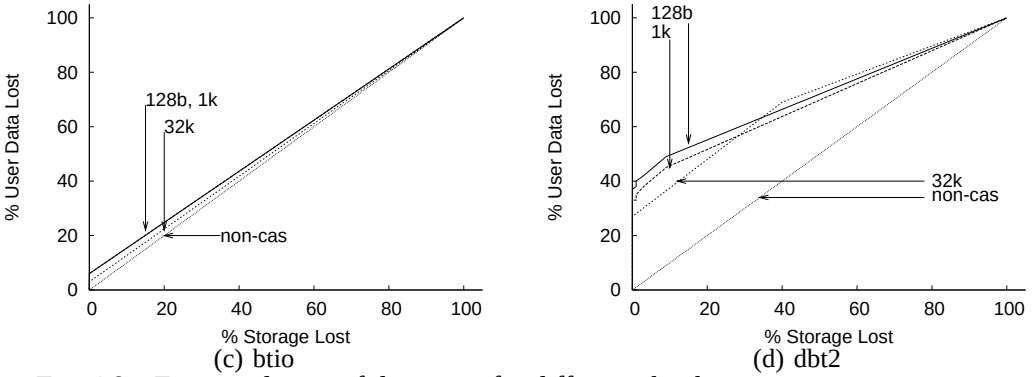
$$loss_i = commonality_i * chunksize \tag{4.1}$$

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, and choosing this sequence for progressive loss of storage data generates Figure 4.9. The $y = x$ line shows the baseline case – the non CAS store. A slope of one indicates that losing B bytes of storage would cause a loss of exactly B bytes of user data in the non CAS case. For CAS data, as expected, higher commonality in data leads to poorer error resilience. From Figure 4.9 we observe the corresponding trend — the smaller the chunksize (more commonality), the farther the curve from the non CAS case, and hence higher damage on loss of single chunk. This stems from the tendency of smaller chunksizes to expose more commonality in the data. The *bssn* and *dbt2* data, which have a large amount of commonality, also have the worst error resilience. At a 1 KB chunksize, losing a few percentage of the storage space can destroy close to 100% of the user data for *bssn*. Increasing the chunksize 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 chunksize brings the loss probability close to the 32 KB chunksize loss probability.

On the upside , CAS exposes *some* information about the value of a chunk, to make informed replication choices. A suitable replication policy could be chosen for a chunk, perhaps based on the 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 error resilience of the *right* chunks, or the dataset as a whole, without wasting too much space. For example, a policy could choose to allocate a certain amount of storage entirely for replication of data. Figure 4.10 shows the effect of one such replication policy on error resilience of the *bssn* dataset for a 1 KB chunksize. In this policy, the chunks are replicated in a greedy manner depending on their value as calculated in equation 4.1. The percentage number indicated in the graph indicates what



(a) bssn (b) heat-solver



(c) btio (d) dbt2

Fig. 4.9. Error resilience of data store for different chunksizes: worst-case scenario.

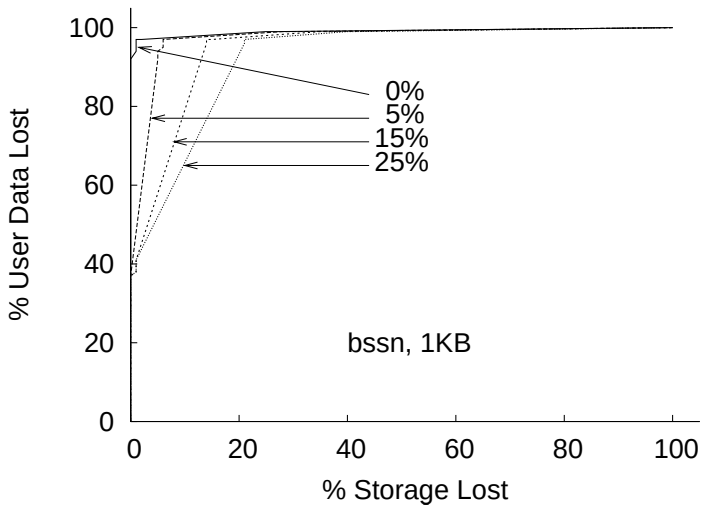


Fig. 4.10. Effect of replication on error resilience.

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.

4.4.3 CAS Performance Overheads

Data in a CAS based chunk store undergoes more processing than in a traditional file system. For example, as shown in in Figure 3.1, 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, a data server first looks up it's database of hashes to find if the chunk already exists. If it is a new chunk then an entry is added to the database, indicating the name and the assigned disk position for the chunk. 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 (thus increasing the network transfer size to the meta-data server), causes inefficient network messaging and poor disk throughput (due to small writes) at the data server. Unlike a traditional filesystem, where large, contiguous writes are sent to disk, a data server processes one chunk at a time. Hence, the largest contiguous write that a data server commits equals the chunksize. The number of chunks to be processed (which depends on the chunksize) affects not just the hash generation time, but also 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 required to store 200 MB of data into CAPFS. In this experiment, we are interested in identifying the overheads of various components of a CAS based system. In our experiment we ran CAPFS with the data server and the meta-data server housed on the client itself to eliminate any network related costs. 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 filesystem was noted and averaged over multiple runs. Between each run the filesystem was un-mounted, cleaned of pre-existing data and re-mounted again.

Figure 4.11 shows the results of this experiment. The *SHA1* curve indicates the time required to generate the SHA1 hashes. The *lookup* curve indicates the time required to process the original data (generate chunks, generate SHA1 hash and then query the CAS database for chunk name entry) upto the point where the CAS database is looked up for the hash. The *total* curve indicates the total time required (including the above lookup time) to complete the operation including disk I/O.

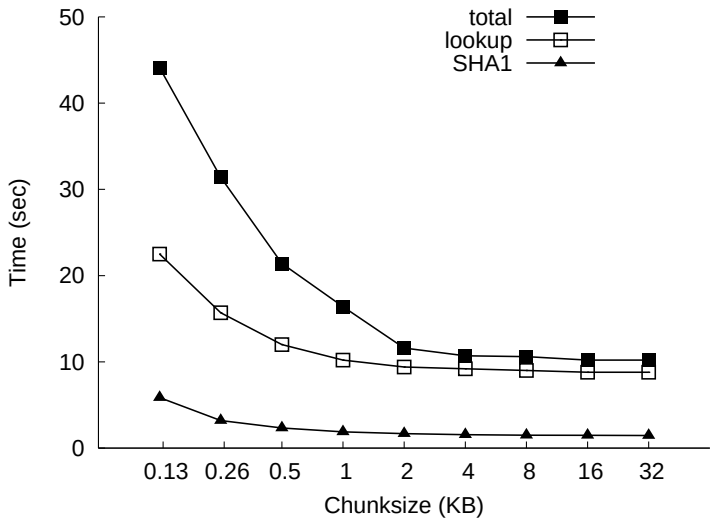


Fig. 4.11. Time required to write 200 MB of data to a CAS store

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. Hence on excluding this cost, and the SHA1 hash generation cost, we are left with some constant miscellaneous costs and the database querying overhead. This 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. We surmise that at small chunksizes the database component is significant due to the larger number of hashes stored in the database. The original CAPFS project [134] used an inefficient implementation of the database where this cost was much higher for small chunks.

Other overheads (not evaluated here) include space reclamation or garbage collection of unused chunks resulting from over-writes or deletes at the data server. One can imagine that a garbage collection daemon would periodically wake up and delete data. If such behavior is undesirable, then this could perhaps be done as part of an *fsck* operation. This feature was disabled in our tests and the cost has been ignored in this study.

Our conclusions are as follows,

- A 1 KB chunksize or larger yields the 'best' file system performance.
- Commonality information of chunks can be used to implement simple replication policies, that use little additional space, but considerably reduce the error-proneness of CAS data
- SHA1 hash generation costs are not significant. When ignoring network transmission times, these costs come to less than 15%.
- The most significant overhead is querying of CAS repository for the presence of a chunk. This overhead grows significantly with increase in the number of chunks housed on the repository.

4.5 Discussion

When to use CAS : As seen in Section 4.3.1, commonality varies from application to application. In certain applications, the presence of commonality may be due to the iterative

data generation behavior of the application. A good example for this case are the scientific applications that periodically checkpoint their results and/or generate data iteratively. Other cases where commonality can be hoped for, include applications that manipulate large sparse matrices, bitmaps or tables. Development and experimental platforms often compile and execute the same workload multiple times till confidence is attained. Such platforms would incur negligible storage and network I/O costs after the very first run, since the data would already exist on the CAS server. When commonality due to data generation or usage behavior is not obvious, space savings should not be the primary motivation behind the use of CAS.

As seen in Section 4.3.2, use of a content addressable cache is a good design choice. In the worst case of no extractable commonality, a CAS based cache will perform as well as a non-CAS cache with added computational 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 use of this property of CAS for de-linking the chunk name from filename, a system can achieve exploit caches without concerns of consistency as outlined in [134, 128].

What chunksize to use : Our results indicate that at small chunksizes the overheads outweigh any gains from the use of CAS. Use of a small chunksize of 1 KB or even 2 KB provides a good trade off between gains from space savings/caching, meta-data overheads, error resilience concerns and performance.

4.6 Related Work

The Farsite distributed file-system from Microsoft was perhaps the first study [13] 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.

Muthitacharoen et. al. proposed the Low Bandwidth File System [80] that eliminates commonality in the network data stream across variable sized chunks. Rabin [96] introduced the idea of generating variable sized chunks by detecting natural block boundaries. Broder et. al. present applications of this algorithm [17] and Chan et. al. provide an implementation for the same [26].

The Farsite project implements a Single Instance Store (a CAS store) for the Windows 2000 NTFS volume [12, 4]. The Plan 9 project from Bell Labs uses the Fossil file-system [94] to store snapshots of a live system on top of Venti [93], a content addressable backend. CAPFS or the Content Addressable Parallel File System [134] is a cluster file-system that exploits CAS for bandwidth savings. Various content based chunking methods have been used in Pasta[78], Pastiche[32] and REBL[67]. Ajtai et. al. [5] provide a comparison of the above methods.

Tolia et. al. first coined the term *recipes* [130] 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 [74, 18, 19]. The rsync protocol [131] uses MD5 hashes for comparing files. Cryptographic hashes have also been used to synchronize content across replicated collections [120, 56]. Chord [117], CFS [34] and Pond [99] exploit the global naming property of CAS. Sundr [71], Ivy [81], Plutus [57] and Tripwire [59] use 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 [62]. That study estimates the benefits of using CAS to house virtual machine snapshots.

Compare by hash, the underlying principle of CAS has been criticized in [51] for being prone to collisions (two or more blocks generating the same SHA1 hash). More recently however, the Monotone team [77] and Black [10] have shown that this may not be a large concern.

4.7 Chapter Summary

In this chapter, we have evaluated the pros and cons of content addressable storage for five real world datasets and found it to be beneficial (to varying degrees) for storing scientific data and data from a TPCC-like benchmark. 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%, when neglecting network I/O costs.

In this chapter, through our analysis of application benchmarks, we obtained a notion of the benefits and demerits of CAS. Unfortunately, this study could not take into account real world data usage patterns. For example, in the real world, applications are often run several times, either with the same parameters or different ones. Each execution of the application might generate exactly the same or somewhat similar data. For a traditional data store, the net I/O generated would equal the I/O generated per execution times the number of executions. If however, the system were to be run on a CAS based storage platform like CAPFS, then the I/O generated after N executions of the application could be significantly less than N times I/O generated per execution. This would depend on how much *commonality* exists between the data generated across different executions. If the data generated is identical (application is run with same parameters), then only $1/N^{th}$ of the total data will need to sent to the storage over the

network. To incorporate such usage behavior in the real world, we now turn our attention to a case study analysis of an application, deployed in the real world for a period of about seven months.

Chapter 5

Case Study: Internet Suspend/Resume

5.1 Introduction

The systems literature of recent years bears witness to a significantly increased interest in virtual machine (VM) technology. Two aspects of this technology, namely platform independence and natural state encapsulation, have enabled the application of this technology in systems designed to improve scalability [24, 40, 47, 98, 124, 144], security [43, 65, 139], reliability [9, 16, 28, 73, 135], and client management [27, 22, 64].

The benefits derived from platform independence and state encapsulation, however, often come with an associated cost, namely the management of significant data volume. For example, enterprise client management systems [27, 64] may require the storage of tens of gigabytes of data *per user*. For each user, these systems store an image of the user's entire VM state, which includes not only the state of the virtual processor and platform devices, but the memory and disk states as well.

While this cost is initially daunting, we would expect a collection of VM state images to have significant data redundancy because many of the users will employ the same operating systems and applications. Content addressable storage (CAS) [14, 80, 93, 107, 135, 141] is an emerging mechanism that can reduce the costs associated with this volume of data by eliminating such redundancy. Essentially, CAS uses cryptographic hashing techniques to identify data by its *content* rather than by name. Consequently, a CAS-based system will identify sets of identical

objects and only store or transmit a single copy even if higher-level logic maintains multiple copies with different names.

To date, however, the benefit of CAS in the context of enterprise-scale systems based on VMs has not been quantified. In this study, we analyze data obtained from a seven-month, multi-user pilot deployment of a VM-based enterprise client management system called Internet Suspend/Resume (ISR) [63, 108]. Our analysis aims to answer two basic questions:

Q1: By how much can the application of CAS reduce the system's storage requirements?

Q2: By how much can the application of CAS reduce the system's network traffic?

The performance of CAS depends upon several system parameters. The answers to Q1 and Q2, therefore, are analyzed in the context of the two most important of these design criteria:

C1: The *privacy policy*, and

C2: the *object granularity*.

The storage efficiency of a CAS system, or the extent to which redundant data is eliminated, depends upon the degree to which that system is able to *identify* redundant data. Hence, the highest storage efficiency requires users to expose cryptographic digests to the system and potentially to other users. As we shall see, the effects of this exposure can be reduced but not eliminated. Consequently, criterion C1 represents a trade-off between storage efficiency and privacy.

Object granularity, in contrast, is a parameter that dictates how finely the managed data is subdivided. Because CAS systems exploit redundancy at the object level, large objects (like disk images) are often represented as a sequence of smaller objects. For example, a multi-gigabyte disk image may be represented as a sequence of 128 KB objects (or *chunks*). A finer granularity

(smaller chunksize) will often expose more redundancy than a coarser granularity. However, finer granularities will also require more meta-data to track the correspondingly larger number of objects. Hence, criterion C2 represents the trade-off between efficiency and meta-data overhead.

The results obtained from the ISR pilot deployment indicate that the application of CAS to VM-based management systems is more effective in reducing storage and network resource demands than applying traditional compression technology such as the Lempel-Ziv compression [145] used in *gzip*. This result is especially significant given the non-zero run-time costs of compressing and uncompressing data. In addition, combining CAS and traditional compression reduces the storage and network resource demands by a factor of two beyond the reductions obtained by using traditional compression technology alone.

Further, using this real-world data, we are able to determine that enforcing a strict privacy policy requires approximately 1.5 times the storage resources required by a system with a less strict privacy policy. Finally, we have determined that the efficiency improvements derived from finer object granularity typically outweighs the meta-data overhead. Consequently, the disk image chunksize should be between 4 and 16 KB.

Sections 5.4 and 5.5 will elaborate on these results from the pilot deployment. But first, we provide some background on ISR, content addressable storage, and the methodology used in the study.

5.2 Background

5.2.1 Internet Suspend/Resume

Internet Suspend/Resume (ISR) is an enterprise client management system that allows users to access their personal computing environments from different physical machines. The system is based on a combination of VM technology and distributed storage. User computing environments are encapsulated by VM instances, and the state of such a VM instance, when idle, is captured by system software and stored on a carefully-managed server. There are a couple of motivations for this idea. First, decoupling the computing environment from the hardware allows clients to migrate across different hosts. Second, storing VM state on a remote storage repository simplifies the management of large client installations. The physical laptops and desktops in the installation no longer contain any hard user-specific state, and thus client host backups are no longer necessary; the only system that needs to be backed up is the storage repository.

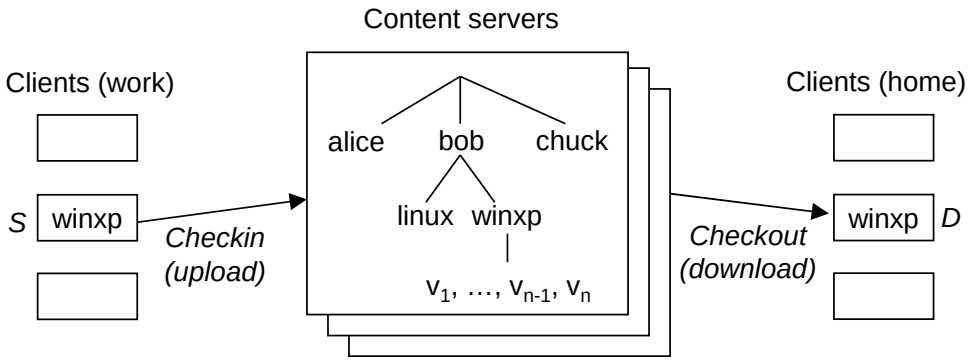


Fig. 5.1. An ISR system.

Figure 5.1 shows the setup of a typical ISR system. The captured states of user environments are known as known as *parcels* and are stored on a collection of (possibly) distributed *content servers*. For example, in the figure, Bob owns two parcels. One environment includes Linux as the operating system, and the other includes Windows XP. Each parcel captures the complete state of some VM instance. The two most significant pieces of state are the *memory image* and the *disk image*. In the current ISR deployment, memory images are 256 MB and disk images are 8 GB. Each memory image is represented as a single file. Each disk image is partitioned into a set of 128 KB *chunks* and stored on disk, one file per chunk.

For each parcel, the system maintains a sequence of checkpointed diff-based *versions*, $v_1, \ldots, v_{n-1}, v_n$. Version v_n is a complete copy of the memory and disk image. Each version v_k, $1 \leq v_k \leq v_{n-1}$, has a complete copy of the memory image, along with the chunks from the v_k version of the disk image that changed between version v_k and v_{k+1}.

Each client host in the ISR system runs a *VM monitor* that can load and execute any parcel. ISR provides a mechanism for suspending and transferring the execution of these parcels from one client host to another. For example, Figure 5.1 shows a scenario where a user transfers the execution of a VM instance from a source host S at the office to a destination host D at home.

The transfer occurs in two phases: a *checkin* step followed by a *checkout* step. After the user suspends execution of the VM monitor on S, the checkin step uploads the memory image and any dirty disk chunks from S to one of the content servers, creating a new parcel version on the server. The checkout step downloads the memory image of the most recent parcel version from the content server to D. The user is then able to resume execution of the parcel on D (even

before the entire disk image is present). During execution, ISR fetches any missing disk chunks from the content server *on demand* and caches those chunks at the client for possible later use.

5.2.2 Content Addressable Storage

Content addressable storage (CAS) is a data management approach that shows promise for improving the efficiency of ISR systems. CAS uses cryptographic hashing to reduce storage requirements by exploiting commonality across multiple data objects [39, 67, 90, 130, 134, 141]. For example, to apply CAS to an ISR system, we would represent each memory and disk image as a sequence of fixed-sized chunk files, where the filename of each chunk is computed using a collision-resistant cryptographic hash function. Since chunks with identical names are assumed to have identical contents, a single chunk on disk can be included in the representations of multiple memory and disk images. The simplest example of this phenomenon is that many memory and disk images contain long strings of zeros, most of which can be represented by a single disk chunk consisting of all zeros. A major goal of this study is to determine to what extent such redundancy exists in realistic VM instances.

5.3 Methodology

Sections 5.4 and 5.5 present our analysis of CAS technology in the context of ISR based on data collected during the first 7 months of a pilot ISR deployment at Carnegie Mellon University. This section describes the deployment, and how the data was collected and analyzed.

5.3.1 Pilot Deployment

The pilot deployment (pilot) began in January, 2005, starting with about 5 users and eventually growing to 23 active users. Figure 5.2 gives the highlights. Users were recruited from

Number of users	23
Number of parcels	36
User environment	Windows XP or Linux
Memory image size	256 MB
Disk image size	8 GB
Client software	ISR+Linux+VMware
Content server	IBM BladeCenter
Checkins captured	817
Uncompressed size	6.5 TB
Compressed size	0.5 TB

Fig. 5.2. Summary of ISR pilot deployment.

the ranks of Carnegie Mellon students and staff and given a choice of a Windows XP parcel, a Linux parcel, or both. Each parcel was configured with an 8 GB virtual disk and 256 MB of memory. The *gold images* used to create new parcels for users were updated at various times over the course of the pilot with security patches.

The content server is an IBM BladeCenter with 9 servers and a 1.5 TB disk array for storing user parcels. Users downloaded and ran their parcels on Linux-based clients running VMware Workstation 4.5.

5.3.2 Data Collection

During the course of the pilot, users performed numerous checkin operations, eventually creating 817 distinct parcel versions on the content server. In August, 2005, after 7 months of continuous deployment, a snapshot of the memory and disk images of these parcel versions was taken on the content server. In uncompressed form, the snapshot state would have consumed about 6.5 TB. However, due to ISR's diff-based representation and gzip compression, it only required about 0.5 TB of disk space. This snapshot state was copied to another server, where it was post-processed and stored in a database for later analysis.

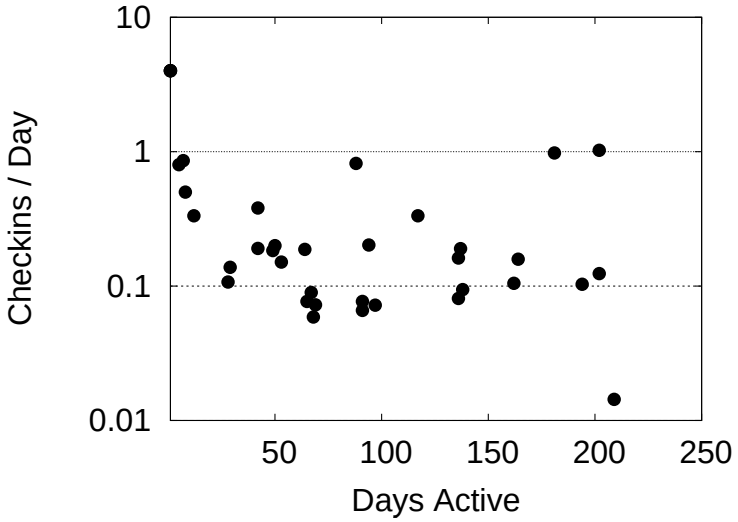


Fig. 5.3. Observed parcel checkin frequency

Figure 5.3 summarizes parcel usage statistics for the deployment data. Each point in the figure represents a single parcel and indicates the number of days that parcel was active as well

as its checkin frequency (average number of checkins per day). Parcels could be active for less than the entire duration of the deployment either because the parcel was created after the initial deployment launch or because a user left the study early (e.g. due to student graduation or end-of-semester constraints). Since new users were added throughout the course of the pilot, during post-processing we normalized the start time of each user to day zero. No extrapolation of data was performed, thus the usage data for a user who has used the system for n days appears in the first n days worth of data in the corresponding analysis. We also removed several parcels that were used by developers for testing, and thus were not representative of typical use.

5.3.3 Analysis

The August 2005 snapshot provided a complete history of the memory and disk images produced by users over time. This history allowed us to ask a number of interesting "what if" questions about the impact of different design choices, or policies, on the performance of the ISR system. In particular, we explored three different storage policies: a baseline non-CAS *Delta* policy and two different CAS policies called *IP* and *ALL*. These are summarized in Figure 5.4. In each approach, a parcel's memory and disk images are partitioned into fixed-sized chunks,

Policy	Encryption	Meta-data
Delta	private per-parcel key	none
IP	private per-parcel key	(tag) array
ALL	convergent encryption	(tag, key) array

Fig. 5.4. Storage policy encryption technique summary.

which are then encrypted, and optionally compressed using conventional tools like gzip.

As will be shown in sections 5.4 and 5.5, differences in the storage and encryption of data chunks affect not only the privacy afforded to users but also dramatically alter the resources required for storage and network transmission. For our evaluations, we chose chunksizes of 4KB (a typical disk-allocation unit for most operating systems) and larger.

Delta policy. In this non-CAS approach, the most recent disk image v_n contains a complete set of chunks. For each version $k < n$, disk image v_k contains only those chunks that differ in disk image v_{k+1}. Thus, we say that Delta exploits *temporal redundancy* across the versions.

Chunks in all of the versions in a parcel are encrypted using the same per-parcel private key. Individual chunks are addressed by their position in the image (logical block addressing), hence no additional meta-data is needed. Memory images are represented in the same way. Delta is similar to the approach used by the current ISR prototype (the current prototype only chunks the disk image and not the memory image). We chose it as the baseline because it is an effective state-of-the-art non-CAS approach for representing versions of VM images.

IP (intra-parcel) policy. In this CAS approach, each parcel is represented by a separate pool of unique chunks shared by all versions, $v_1, \ldots, v_n$, of that parcel. Similar to Delta, IP identifies temporal redundancy between contiguous parcel versions. However, IP can also identify temporal redundancy in non-contiguous versions (e.g., disk chunk i is identical in versions 4 and 6, but different in version 5), and it can also identify any *spatial redundancy* within each version.

As with Delta, each chunk is encrypted using a single per-parcel private key. However, each version of each disk image (and each memory image) requires additional meta-data to record the sequence of chunks that comprise the image. In particular, the meta-data for each

image is an array of *tags*, where tag i is the SHA-1 hash of chunk i. This array of tags is called a *keyring*.

ALL policy. In this CAS approach, all parcels for all users are represented by a single pool of unique chunks. Each chunk is encrypted using *convergent encryption* [37], where the encryption key is simply the SHA-1 hash of the chunk's original plain-text contents. This allows chunks to be shared across different parcels and users, since if the original plain-text chunks are identical, then the encrypted chunks will also be identical.

As with IP, each version of each disk image (and each memory image) requires additional keyring meta-data to record the sequence of chunks that compose the image, in this case an array of (tag, key) tuples, where key i is the encryption key for chunk i, and tag i is the SHA-1 hash of the encrypted chunk. Each keyring is then encrypted with a per-parcel private key.

The IP and ALL policies provide an interesting trade-off between privacy and space efficiency. Intuitively, we would expect the ALL policy to be the most space-efficient because it identifies redundancy across the maximum number of chunks. However, this benefit comes at the cost of decreased privacy, both for individual users and the owners/operators of the storage repository. The reason is that ALL requires a consistent encryption scheme such as convergent encryption for all blocks. Thus, individual users are vulnerable to dictionary-based traffic analysis of their requests, either by outside attackers or the administrators of the systems. Owner/operators are vulnerable to similar analysis, if, say, the contents of their repository are subpoenaed by some outside agency.

Choosing appropriate chunk sizes is another interesting policy decision. For a fixed amount of data, there is a tension between chunk size and the amount of storage required. Intuitively, we would expect that smaller chunk sizes would result in more redundancy across

chunks, and thus use less space. However, as the chunk size decreases, there are more chunks, and thus there is more keyring meta-data. Other chunking techniques such as Rabin Fingerprinting [74, 96, 114] generate chunks of varying sizes in an attempt to discover redundant data that does not conform to a fixed chunk size. However, as described in Chapter 2, the evaluation of non-fixed-size chunk schemes is beyond the scope of this study.

The remainder of the study uses the data from the ISR deployment to quantify the impact of CAS privacy and chunksize policies on the amount of storage required for the content servers, and the volume of data that must be transferred between clients and content servers.

5.4 Results: CAS & Storage

Because server storage represents a significant cost in VM-based client management systems, we begin our discussion by investigating the extent to which a CAS-based storage system could reduce the volume of data managed by the server.

5.4.1 Effect of Privacy Policy on Storage

As expected, storage policy plays a significant role in the efficiency of the data management system. Figure 5.5 presents the growth in storage requirements over the lifetime of the study for the three different policies using a fixed chunksize (128 KB). As mentioned in Section 5.3.2, the graph normalizes the starting date of all users to day zero. The growth in the storage from thereon is due to normal usage of disks and storage of memory checkpoints belonging to the users. The storage requirement shown includes both the disk and memory images.

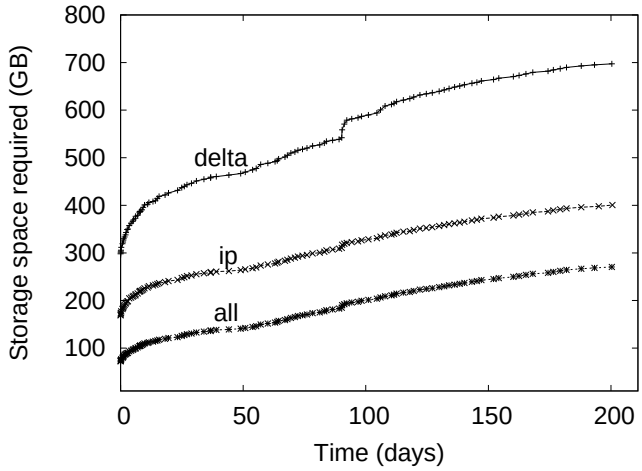


Fig. 5.5. Growth of storage needs for Delta, IP, and ALL.

CAS provides significant savings. As shown in Figure 5.5, adopting CAS with the IP policy reduces the required server resources at day 201 under the Delta policy by 306 GB, from 717 GB to 411 GB. This reduction represents a savings of 42%.

Recall that adopting CAS is a loss-less operation; CAS simply stores the same data more efficiently than the Delta policy. The improved efficiency is due to the fact that the Delta policy only exploits temporal redundancy between versions. That is, the Delta policy only identifies identical objects when they occur in the same location in subsequent versions of a VM image. The IP policy, in contrast, identifies redundancy anywhere within the parcel – within a version as well as between versions (including between non-subsequent versions).

Note that the 42% space savings was realized without compromising privacy. Users in a CAS-IP-backed system do not expose the contents of their data to any greater degree than users of a Delta-backed system.

Relaxing privacy introduces additional gains. In systems where a small relaxation of privacy guarantees is acceptable, additional savings are possible. When the privacy policy is relaxed from IP to ALL, the system is able to identify additional redundancy that may exist between different users' data. From Figure 5.5, we see that such a relaxation will reduce the storage resources required by another 133 GB, to 278 GB. The total space savings realized by altering the policy from Delta to ALL is 61%.

On comparing ALL with IP in Figure 5.5, we see that the curves are approximately parallel to each other. However, under certain situations, a system employing the ALL policy could dramatically outperform a similar system that employs the IP policy. Imagine for example a scenario where a security patch is applied by each of a large number, N, of users in an enterprise. Assuming that the patch affected each user's environment in the same way, by introducing X MB of new data, an IP server would register a total addition of NX MB. In contrast, an ALL server would identify the N copies of the patched data as identical and would consequently register a total addition of X MB.

The starting points of the curves in Figure 5.5 are also of interest. Because the X-axis has been normalized, this point corresponds to the creation date of all parcels. To create a new parcel account, the system administrator copies a gold image as version 1 of the parcel. Hence, we would assume that the system would exhibit very predictable behavior at time zero.

For example, under the Delta policy which only reduces redundancy *between versions*, the system data should occupy storage equal to the number of users times the space allocated to each user. In the deployment, users were allocated 8 GB for disk space and 256 MB for memory images. Thirty-six parcels should then require approximately 300 GB of storage space which is exactly the figure reported in the figure.

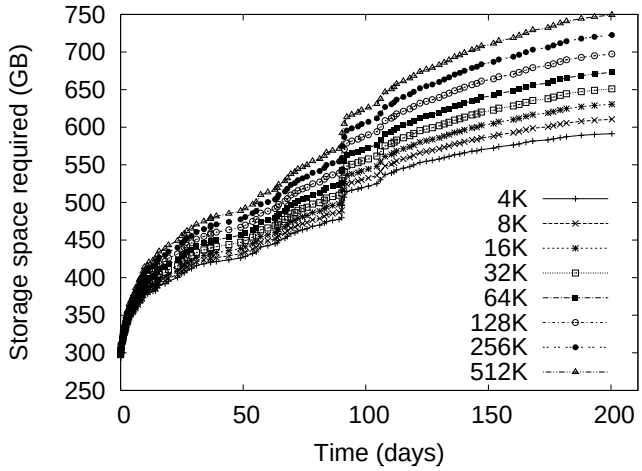
For the IP policy, one would also expect the server to support a separate image for each user. However, CAS had eliminated the redundant data within each of these images yielding an average image size of approximately 4 GB. The observed 171 GB storage space is consistent with this expectation.

Under the ALL policy in contrast, one would expect the system to store a single copy of the gold image shared by all users, yielding a total storage requirement of 8 GB plus 256 MB (closer to 4 GB, actually, due to the intra-image redundancy elimination). We were quite surprised, consequently, to observe the 72 GB value reported in the figure. After reviewing the deployment logs, we determined that this value is due to the introduction of multiple gold images into the system. To satisfy different users, the system administrators supported images of several different Linux releases as well as several instances of Windows images. In all, the administrators had introduced 13 different gold images, a number that is consistent with the observed 72 GB of occupied space.

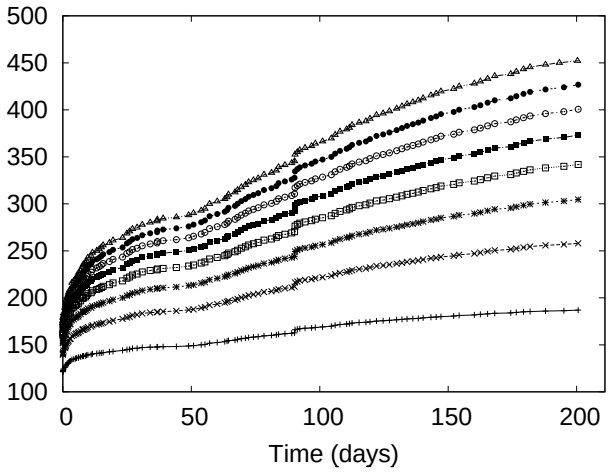
Another point of interest is a disturbance in the curve that occurs at the period around 100 days. We note that the disturbance is significant in the Delta curve, smaller in the IP curve, and almost negligible in the ALL curve. We've isolated the disturbance to a single user and observe that this anomaly is due to the user reorganizing his disk image without creating new data that did not already exist somewhere in the system. Hence, we conclude that this must have been an activity similar to defragmentation or re-installation of an operating system.

5.4.2 Effect of Chunksize on Storage

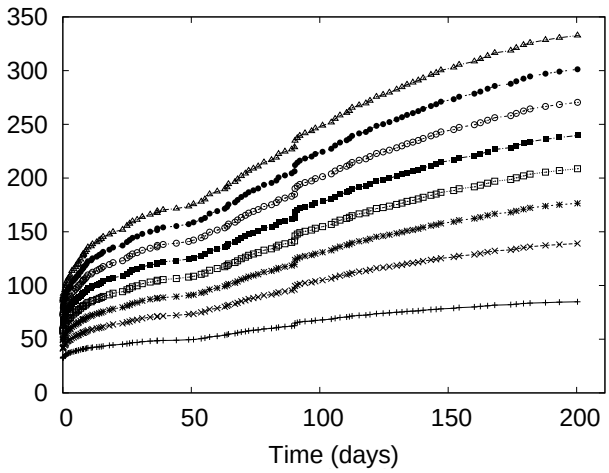
In addition to privacy considerations, the administrator of a VM-based client management system may choose to optimize the system efficiency by tuning the chunksize. The impact



(a) Delta Policy



(b) IP Policy



(c) ALL Policy

Fig. 5.6. Storage space growth for various chunksizes without meta-data overhead (y-axis scale varies).

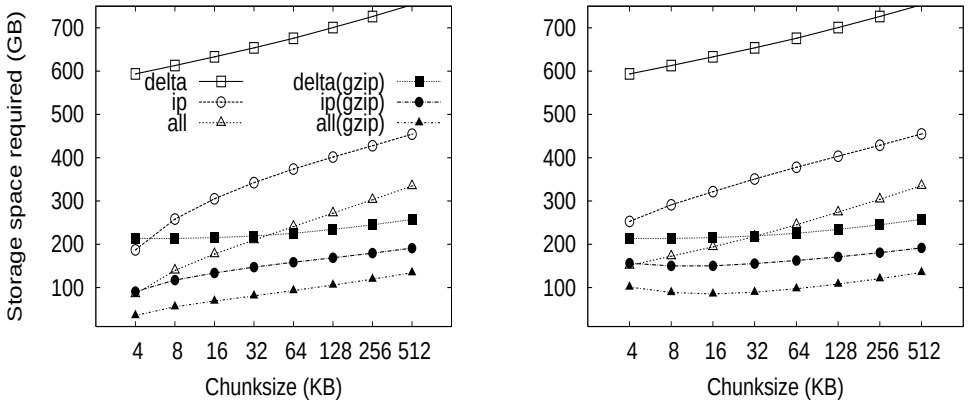
of this parameter on storage space requirements is depicted in Figure 5.6; in this figure, we present what the growth curves of Figure 5.5 would have been had we chosen different chunk-sizes.

Note that the effect of this parameter is not straightforward. Varying the chunksize has three different effects on efficiency.

First, smaller chunksizes tend to expose more redundancy in the system. As a trivial exercise, consider two objects each of which, in turn, comprises two blocks ($Object_1 = AB$ and $Object_2 = CA$). If the chunksize is chosen to be a whole object, the content addresses of $Object_1$ and $Object_2$ will differ and no redundancy will be exposed. If the chunksize is chosen to be a block, in contrast, the identical A blocks will be identified and a space savings of 25% will result.

Second, smaller chunksizes require the maintenance of more meta-data. With the whole-object chunksize from the example above, the system would maintain two content addresses, for $Object_1$ and $Object_2$. With the block chunksize, however, the system must maintain two sets of two content addresses so that $Object_1$ and $Object_2$ may each be properly reconstructed. Note further that this additional meta-data maintenance is required whether or not any redundancy was actually identified in the system.

Third, smaller chunksizes tend to provide a reduced opportunity for post-chunking compression. In addition to chunk-level redundancy elimination through CAS, intra-chunk redundancy may be reduced through traditional compression techniques (such as *gzip*). However, as the chunksize is reduced, these techniques have access to a smaller intra-chunk data pool on which to operate, limiting their efficiency.



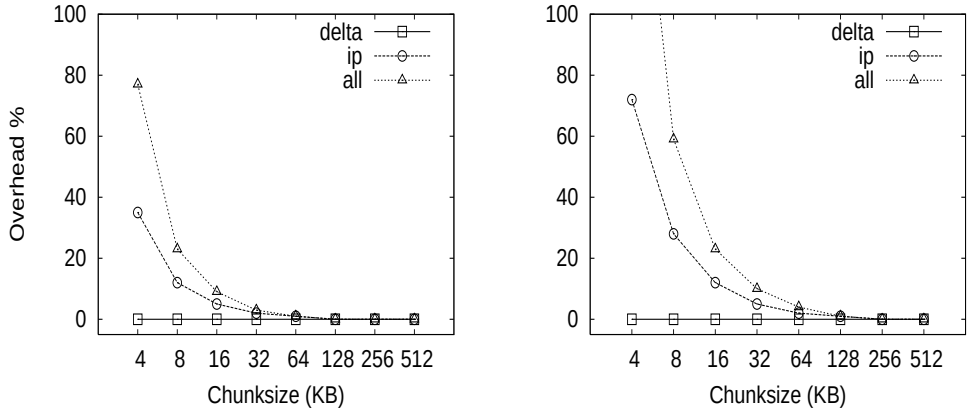
(a) Without meta-data overhead (b) With meta-data overhead

Fig. 5.7. Server space required, after 201 deployment days.

To better understand the effect of chunksize, we analyzed the deployment data for all three storage policies with and without compression under several different chunksizes. The results are shown in Figure 5.7.

All three effects of chunksize can be observed in this figure. For example, Figure 5.7(a), which ignores the increased meta-data required for smaller chunksizes, clearly indicates that smaller chunksizes expose more redundancy. These gains for small chunk sizes, however, are erased when the meta-data cost is introduced to the storage requirements in Figure 5.7(b). Finally, the reduced opportunities for compression due to smaller chunksize can be observed in Figure 5.7(b) by comparing the IP and IP(gzip) or ALL and ALL(gzip) curves.

CAS is more important than compression. In Figure 5.7(a), the Delta curve *with compression* intersects the IP and ALL curves *without compression*. The same is true in Figure 5.7(b) with respect to the ALL curve. This indicates, that given appropriate chunksizes, a CAS-based policy can outperform compression applied to a non-CAS-based policy.



(a) Uncompressed user data (b) Compressed user data

Fig. 5.8. Meta-data overhead expressed as a percentage of user data.

Considering meta-data overheads, the ALL policy outperforms Delta with compression for all the chunksizes less than 64KB. This is a very remarkable result. Compression in the storage layer may be a high latency operation, and it may considerably affect virtual disk operation latencies. By use of CAS, one can achieve savings that exceed traditional compression techniques! If additional space savings are required, compression can be applied after the application of content addressing.

Figure 5.7(a) shows that compression provides an additional savings of a factor of two to three. For example, the space demands for the ALL policy, drops from 87GB to 36GB when using 4KB chunks, and from 342GB to 137GB when using 512KB chunks.

Exposing redundancy outweighs meta-data overhead. Figure 5.8 shows the ratio of meta-data (keyring size) to the size of the data. We observe that this ratio is as high as 80% for ALL, and 35% for IP at 4KB chunksize without compression and even higher after compression is applied to the basic data. Yet, from Figure 5.7(b), we observe from the IP and ALL curves that

reducing chunksize always yields a reduction in storage requirements. This indicates that the gains through CAS-based redundancy elimination far exceed the additional meta-data overhead incurred from smaller chunksize.

The picture changes slightly with the introduction of traditional compression. The IP(gzip) and ALL(gzip) curves of Figure 5.7(b) indicate that the smallest chunksize is not optimal. In fact, we see from Figure 5.8 that the meta-data volume becomes comparable to the data volume at small chunksizes.

Small chunk sizes improve efficiency. With Figure 5.7(b), we are in a position to recommend optimal chunk sizes. Without compression, the optimal chunksize is 4 KB for the Delta, IP and ALL policies. With compression, the optimal chunksize is 8 KB for the Delta(gzip) policy and 16 KB for the IP(gzip) and ALL(gzip) policies.

5.5 Results: CAS & Networking

In a VM-based client management system, the required storage resources, as discussed in the previous section, represent a cost to the system administrator in terms of physical devices, space, cooling, and management. However, certain user operations, such as check-in and checkout, require the transmission of data over the network. While the system administrator must provision the networking infrastructure to handle these transmissions, perhaps the more significant cost is the user time spent waiting for the transmissions to complete.

For example, a common telecommuting scenario may be that a user works at the office for some time, checks-in their new VM state, travels home, and attempts to checkout their VM state to continue working. In the absence of CAS or traditional compression, downloading just the 256 MB memory, which is required before work can resume, over a 1 Mbps DSL line requires

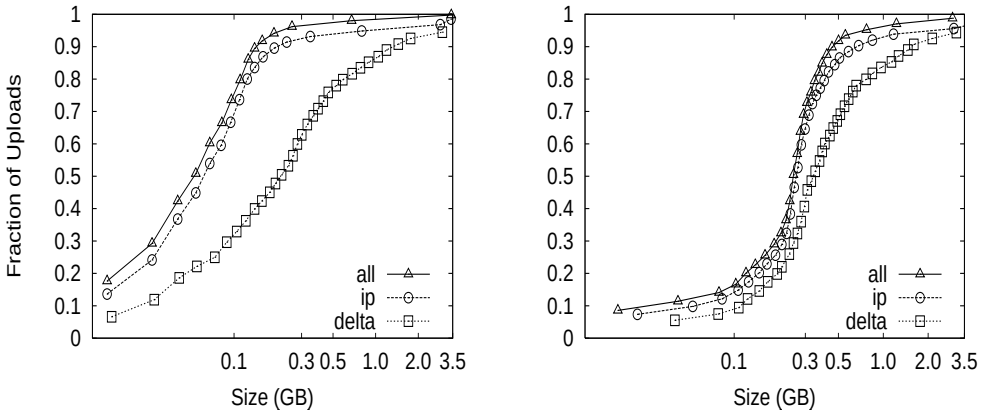
more than 30 minutes of wait time. After working at home for some time, the user will also want to checkin their new changes. Because the checkin image is typically larger than the checkout image, and because the upload speed of ADSL is often much slower than the download speed, the checkin operation can often require two hours or more.

Consequently, we devote this section to characterizing the benefits that CAS provides in terms of reducing the volume of data to be transmitted during typical upload (checkin) or download (checkout) operations.

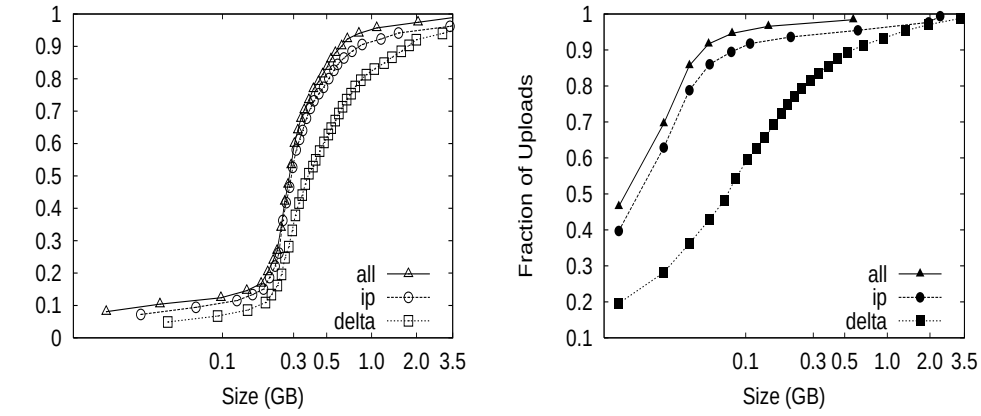
5.5.1 Effect of Privacy Policy on Networking

As with storage, we begin the discussion by considering the effect of privacy policy on networking. We note that our definition of privacy policy affects the representation of data chunks in storage, not the mechanics of chunk transmission. However, the chosen storage policy can affect the capability of the system to identify redundant data blocks that need not be sent because they already exist at the destination.

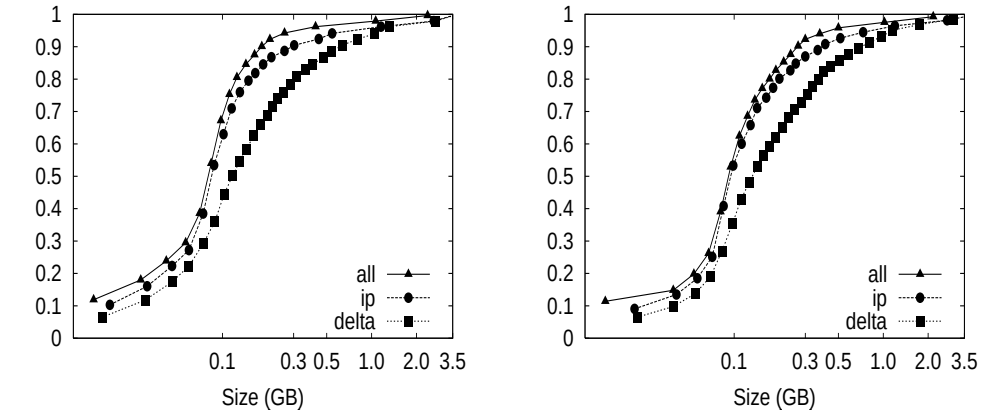
As an example, suppose that a user copies a file within their virtual environment. This operation may result in a virtual disk that contains duplicate chunks. Under the IP and ALL policies, at the time of upload, the client will send a digest of modified chunks to the server, and the server may respond that the duplicate chunks need not be sent because the chunks (identified by the chunks' tags) already exist on the server. Such redundant data can occur for a variety of reasons (particularly under the ALL policy) including the push of software patches, user download of popular Internet content, and the installation and compilation of common software packages.



(a) 4K, without compression (b) 128K, without compression



(c) 512K, without compression (d) 4K, with compression



(e) 128K, with compression (f) 512K, with compression

Fig. 5.9. CDF of upload sizes for different policies, without and with the use of compression.

	Redundancy Comparison	
	Upload (between client copy and...)	Download (between server version N and ...)
Delta	server version N-1	current client version
IP	server versions [1, N-1]	current client version
ALL	all versions/all parcels	current client version

Fig. 5.10. Search space for identifying redundant blocks during data synchronization operations. Note that for download, the system inspects the most recent version available at the client (which may be older than $N - 1$).

During download (checkout) operations, the client code will search through the existing version(s) of the user's data on that client to identify chunks that need not be retrieved from the server. As the system is only comparing the latest version on the server with the existing version on the client, the volume of data to be transmitted does not depend on the privacy policy. In contrast, the volume of data transmitted during upload (checkin) operations does depend on the privacy policy employed because, at the server, redundant chunks are only identified within that user's version history under the IP policy, but can be identified across *all* users' version histories under the ALL policy. These differences based on storage policy are summarized in Figure 5.10 and affect our discussion in two ways: (1) this section (Section 5.5.1), which investigates the effects of privacy policy, only considers the upload operation, and (2) Figures 5.12 and 5.13 in Section 5.5.2 contain curves simply labeled CAS that represent the identical download behaviors of the IP and ALL policies.

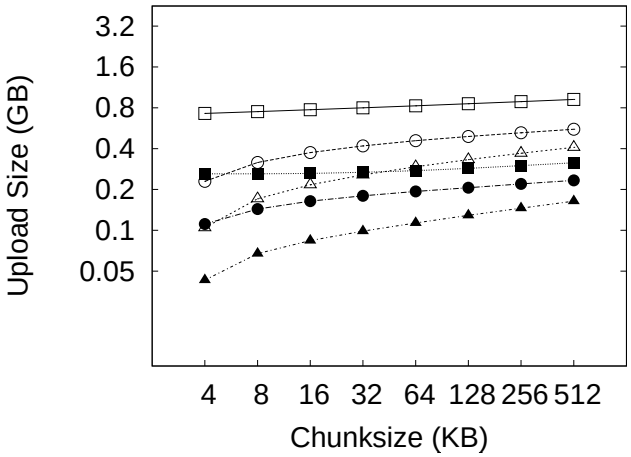
CAS is essential. The upload volume for each of the storage policies with and without compression is presented in Figure 5.9. Because the upload size for any user session includes the 256MB memory image and any hard disk chunks modified during that session, the upload

data volumes vary significantly due to user activity across the 800+ checkin operations collected. Consequently, we present the data as a cumulative distribution function (CDF) plots. In the ideal case, most upload sizes would be small; therefore, curves that tend to occupy the upper left corner are better. Note that the ALL policy strictly outperforms the IP policy, which in turn, strictly outperforms the Delta policy.

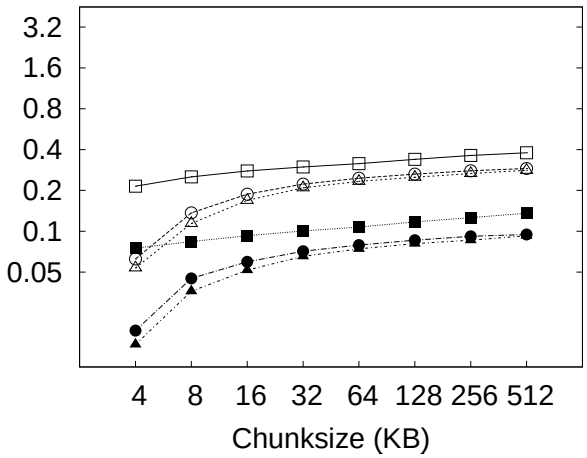
The median (50^{th} percentile) and 95^{th} percentile sizes from Figure 5.9 are presented along with average upload sizes in Figure 5.11. Note that the median upload sizes tend to be substantially better than the mean sizes, indicating that the tail of the distribution is somewhat skewed in that the user will see a smaller than average upload sizes for 50% of the upload attempts. Even so, we see from Figure 5.11(c) that the tail is not so unwieldy as to present sizes more than a factor of 2 to 4 over the average upload size 95% of the time.

Figure 5.11(a) shows that, for the 128 KB chunksize used in the deployment, the use of CAS reduces the average upload size from 880 MB (Delta policy) to 340 MB (ALL policy). The use of compression reduces the upload size to 293 MB for Delta and 132 MB for ALL. Further, CAS policies provide the most significant benefits where they are needed most, for large upload sizes. From Figure 5.11(b) we see that CAS improves small upload operations by a modest 20 to 25 percent, while from Figure 5.11(c), we see that CAS improves the performance of large uploads by a factor of 2 to 5 without compression, and by a factor of 1.5 to 3 with compression. Thus, we observe that CAS significantly reduces the volume of data transmitted during upload operations, and hence the wait time experienced at the end of a user session.

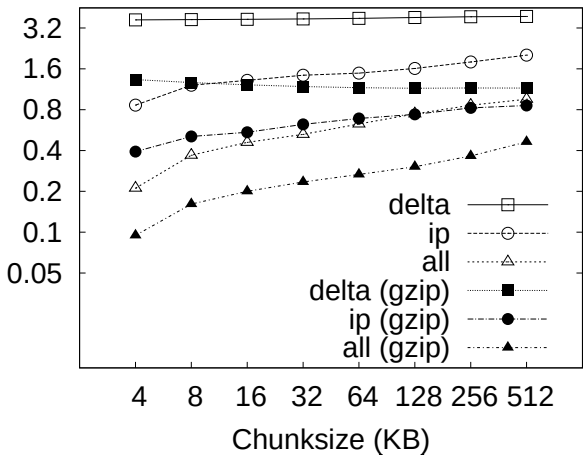
CAS outperforms compression. Figure 5.11(a) indicates that the ALL policy *without* compression outperforms the Delta policy *with* compression for chunk sizes less than 64 KB (as does the IP policy at a 4 KB chunk size). This shows that for our application, inter-chunk CAS



(a) Average



(b) 50^{th} Percentile



(c) 95^{th} Percentile

Fig. 5.11. Upload sizes for different chunksizes.

techniques may identify and eliminate more redundancy than traditional intra-chunk compression techniques. The difference may be substantial, particularly when the upload size is large. As Figure 5.11(c) shows, the ALL policy *without* compression (chunksize=4 KB) outperforms the Delta policy *with* compression (chunksize=512 KB) by a factor of 4.

IP identifies both temporal and spatial redundancy. For each of the components of Figure 5.9, we see that the IP policy consistently outperforms the Delta policy. Both of these policies restrict the search space for redundancy identification to a single parcel. However, the Delta policy only detects temporal redundancy between the current and last versions of the parcel, while the IP policy detects temporal and spatial redundancy across all versions of the parcel. The savings of IP over Delta indicate that users often create modified chunks in their environment that either existed at some point in the past, or in another location within the parcel.

ALL identifies *inter*-parcel savings. In all of Figure 5.9, the common observation between an IP and ALL comparison is that the ALL policy consistently outperforms the IP policy. This observation is consistent with our intuition that for upload operations, the ALL policy must perform *at least* as well as the IP policy because the ALL policy identifies redundancy within the set of blocks visible to the IP policy as well as blocks in other parcels. In fact, Figure 5.11(a) indicates that the ALL policy performs about twice as well as the IP policy for small chunk sizes and approximately 25 percent better at larger chunk sizes.

This difference shows the benefit of having a larger pool of candidate chunks when searching for redundant data. As mentioned, one source of this gain can be the "broadcast" of objects to many users (e.g. from software installation, patches, popular documents, big email attachments, etc.). In systems leveraging the ALL policy, therefore, operations that might be expected to impose a significant burden such as the distribution of security patches may result

in very little realized cost because the new data need only be stored once and transmitted once (across all users in the system).

5.5.2 Effect of Chunksize on Networking

The choice of chunksize will affect both the download size and upload size to a server. We continue our discussion of upload operations first, and then discuss the appropriate chunksize for download operations.

5.5.2.1 Effect on Upload Size

Smaller chunksize is better for CAS. Figure 5.11(a) shows very clearly that smaller chunksizes result in more efficient upload transmission for CAS policies. In fact, under the ALL policy, users with 4 KB chunk sizes will experience average upload sizes that are approximately one-half the average size experienced by users with a 128 KB chunk size (whether compression is employed or not).

Chunk sizes of 4 KB turned out to be optimal for all policies when considering the average upload size. However, chunksize plays a very limited role for the non-CAS (Delta) policy, and Figure 5.11(c) indicates that smaller chunk sizes may even be a liability for transfer size outliers under the Delta policy with compression.

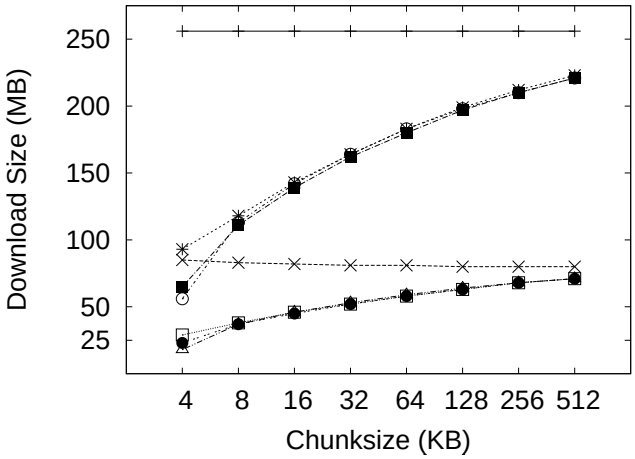
5.5.2.2 Effect on Download Size

Employing CAS techniques also potentially affects the volume of data transmitted during download operations in two ways. First, CAS can identify intra-version redundancy and reduce

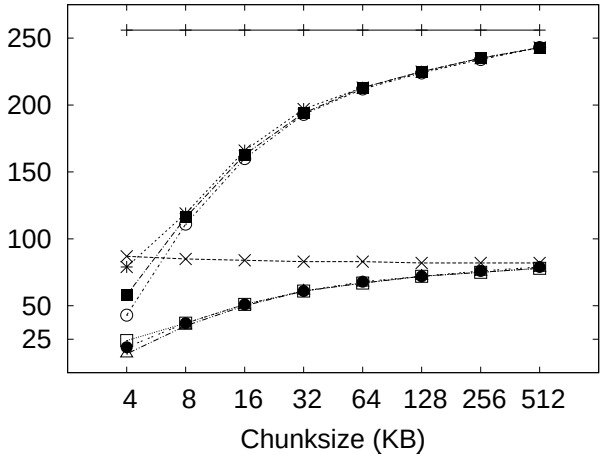
the total volume of data transmission. Second, when a user requests a download of their environment to a particular client, CAS has the potential to expose any chunks selected for download that are identical to chunks that happen to have been cached on that client from previous sessions.

To simplify our discussion we assume that the client has cached at most one previous version of the parcel in question, and if a cached version is present, it is the version prior to the one requested for download. This assumption corresponds to an expected common user telecommuting behavior. Namely, the user creates version $N-1$ of a parcel at home and uploads it to the server. The user then retrieves version $N-1$ at work, creates version N, and uploads that to the server. Our operation of interest is the user's next download operation at home; upon returning home, the user desires to download version N and modify it. Fortunately, the user may still have version $N-1$ cached locally, and thus, only the modified data that does not exist in the cache need be retrieved. Note that this CAS technique can be likened to a sub-set of the IP policy which inspects chunks of a single user, but only for a single previous version.

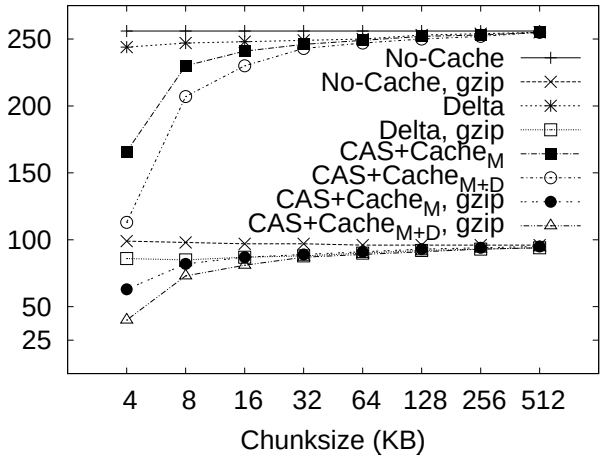
Our client management system, ISR, supports two basic modes for download: *demand-fetch* and *complete-fetch*. Demand-fetch mode instantiates the user's environment after downloading the minimum data needed to reconstruct the user's environment, essentially the physical memory image corresponding to the user's VM (256 MB in our test deployment). In particular, the largest portion of the VM image, the virtual disk drive, is *not* retrieved before instantiating the user's environment. During operation, missing data blocks (chunks) must be fetched on demand in a manner analogous to demand-paging in a virtual memory system. The complete-fetch mode, in contrast, requires that the entire VM image including the virtual disk image (8.25 GB in our test deployment) be present at the client before the environment is instantiated.



(a) Average



(b) 50^{th} Percentile



(c) 95^{th} Percentile

Fig. 5.12. Download size when fetching memory image of latest version.

Caching improves demand-fetch. To evaluate the effect of client-side caching on demand-fetch download volume, we calculated how much data would need to be transferred from the server to a client under various conditions and collected those results in Figure 5.12. The curve labeled "No-cache" depicts the volume of data that would be transmitted if no data from the previous version of the parcel were present in the client cache. Under the "Delta" policy, the chunks in the memory image are compared with the same chunks (those at the same offset within the image) in the previous version of the memory image to determine whether they match. The "CAS+Cache$_M$" policy compares the keyring for the new memory image with the keyring for the previous memory image to determine which chunks need to be transferred. The "CAS+Cache$_{M+D}$" policy is similar except that it searches all the data cached on the client (memory *and* disk) to identify chunks that are already present on the client. Each basic curve in Figure 5.12 also has a companion curve depicting the download volumes observed when compression is employed during the transfer.

As shown in Figure 5.12(a), introducing a differencing mechanism (either Delta or CAS) yields a reduction of approximately 20% (for the 128 KB chunk size) in the download size relative to the size when no cached copy is present. Using compression alone, however, is very effective– reducing the transfer size from 256 MB to approximately 75 MB in the absence of caching. Leveraging cached data in addition to compression yields a further 20% reduction.

Chunk size dramatically affects demand-fetch. Moving to a smaller chunk size can have a significant effect on the volume of data transmitted during a download operation, particularly if compression is not used, as shown in Figure 5.12. The average download size, in particular, is reduced by a factor of two (for Delta) to four (for "CAS+Cache$_{M+D}$") when the

chunk size is reduced from 128 KB to 4 KB when comparing the policies either with or without compression. Further, we see again that, with a 4 KB chunk size, the CAS policies *without* compression outperform the no-cache policy *with* compression.

The difference between the "CAS+Cache$_M$" and "CAS+Cache$_{M+D}$" policies is also most apparent with a 4 KB chunk size. At this size, in the absence of compression, leveraging the cached disk image in addition to the memory image reduces the average transfer size to 56 MB from the 65 MB required when leveraging just the memory image. A similar gain is observed when compression is employed; the transfer size is reduced from 23 MB (for "M") to 18 MB (for "M+D")– a savings of more than 20%.

However, the added benefit of inspecting additional cached data diminishes quickly as the chunk size increases beyond 4 KB. We believe this phenomenon is due, at least in part, to the fact that the 4 KB size corresponds to the size of both memory pages disk blocks in these VMs. Consequently, potentially redundant data is most likely to be exposed when chunks are aligned to 4 KB boundaries.

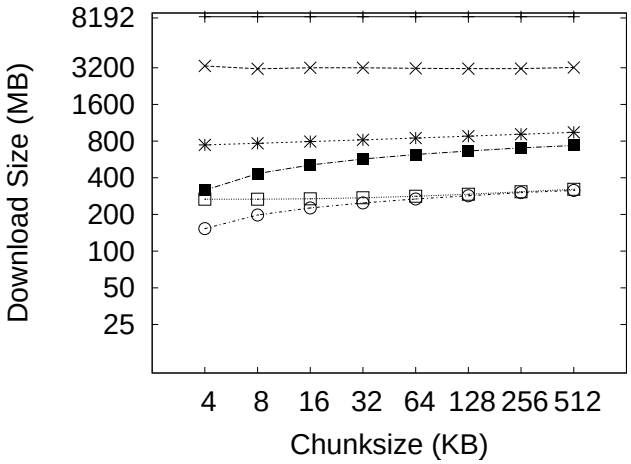
Caching significantly improves complete-fetch. The need for efficient download mechanisms is perhaps greatest in the complete-fetch mode due to the volume of data in question. In this mode, the user is requesting the download of the entire VM image, the most significant component of which is the virtual disk drive image. In our test deployment, the virtual disk drive was a very modest 8 GB in size. One can readily imagine that users might desire virtual disk drive spaces an order of magnitude larger. However, even with a modest size (8 GB) and a fast network (100 Mbps), a complete-fetch download will require at least 10 minutes. Consequently, reducing the volume of data to be transferred by at least an order of magnitude is essential to the operation of these client management systems.

The basic tools are the same as those mentioned for demand-fetch mode. That is, a cache of at least one previous version of the parcel is maintained at the client, if possible. Redundancy between the cached version and the current version on the server is identified and only non-redundant chunks are transferred during the download. Further, the transferred chunks are (optionally) compressed prior to transmission. One difference between our treatment of demand-fetch and complete-fetch is that the CAS policy for complete-fetch mode always compares the entire current server version with the entire cached client version. Consequently, Figure 5.13 includes a single "CAS" curve rather than the separate "M" and "M+D" curves of Figure 5.12.

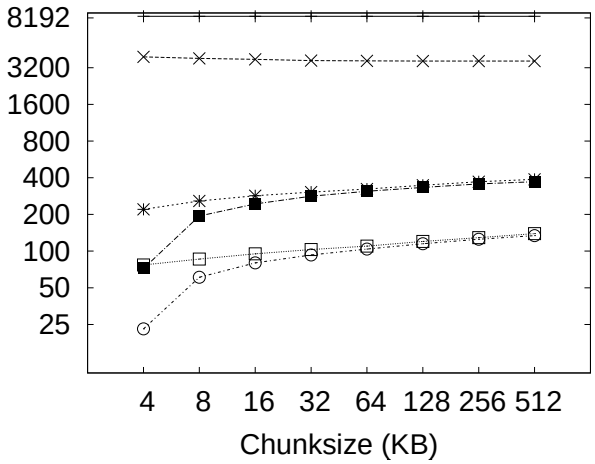
Figure 5.13(a) indicates that intelligent transfer mechanisms can, in fact, significantly reduce the volume of data transmitted during a complete-fetch operation. Compression reduces the average data volume from 8394 MB to 3310 MB, a factor of 2.7. In contrast, the Delta policy *without* compression yields a factor of 9.5 and a factor of 28.6 *with* compression, assuming a 128 KB chunk size. At the same chunk size, CAS provides even more impressive savings: factors of 12.6 and 29.5, without and with compression, respectively.

Small chunk sizes yield additional savings. While the slopes of the "CAS" and "CAS,gzip" curves are not as dramatic as in previous figures, reducing the chunk size from 128 KB to 4 KB still yields significant savings. At this chunk size, the average download size shrinks from the nominal 8+ GB size by a factor of 31.4 without compression and a factor of 55 (*fifty-five!*) by employing both CAS and compression.

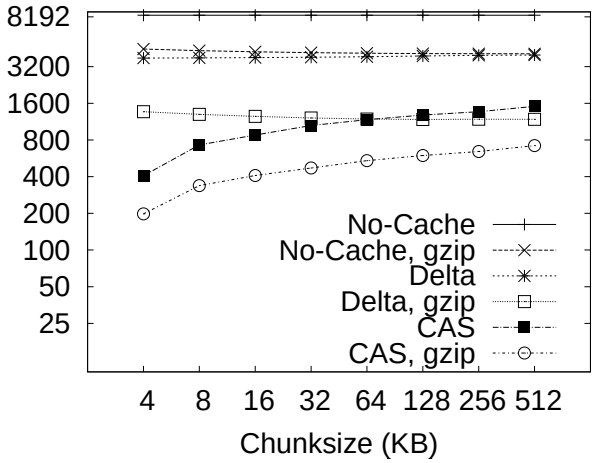
CAS has a big impact where it's needed most. Figure 5.13(c) indicates that the 4 KB "CAS,gzip" combination may be particularly effective for download operations that may otherwise have resulted in large data transfers. The performance gap between "CAS,gzip" and "Delta,gzip" is particularly large in this graph. In fact, for small chunk sizes "CAS" *without*



(a) Average



(b) 50^{th} Percentile



(c) 95^{th} Percentile

Fig. 5.13. Download size when fetching memory *and disk* of latest version.

compression significantly outperforms the Delta policy *with* compression. Note in particular that when employing the "CAS,gzip" policy with the 4 KB chunk size, the 95^{th} percentile upload sizes are not significantly larger than the average size, thus providing the user with better expected bounds on the time required for a complete-fetch download.

5.6 Related Work

Our results are most directly applicable to VM-based client management systems such as the Sun Microsystem's Sun Grid Compute Utility [76], the Amazon Elastic Compute Cloud [55], 3Tera's grid computing infrastructure [1], the Collective [27, 106], Soulpad [22], and ISR [63, 108], as well as systems that use VMs for Grid applications [29, 40, 66, 72, 121]. Further, our results also provides guidelines for the storage design of applications that need to version VM history. Examples include intrusion detection [38], operating systems development [60], and debugging system configurations [138]. Related applications include storage cluster and web services where VMs are being used for balancing load, increasing availability, and simplifying administration [86, 137].

The study could also help a large number of systems that use use CAS to improve storage and network utilization. Examples of CAS-based storage systems include EMC's Centera [39], Deep Store [141], the Venti [93], the Pastiche [32] backup system, the TAPER [56] scheme for replica synchronization and Farsite [12]. Other systems use similar CAS-based techniques to eliminate duplicate data at various levels in the network stack. Systems such as the

CASPER [130] and LBFS [80] file systems, Rhea et al.'s CAS-enabled WWW [100], etc. apply these optimizations at the application layer. Other solutions such as the DOT transfer service [129] and Riverbed's WAN accelerator [101] use techniques such as Rabin Fingerprinting [74, 96, 114] to detect data duplication at the transfer layer. However, most of these systems have only concentrated on the mechanism behind using CAS. Apart from Bolosky et al. [14] and Policroniades and Pratt [90], there have been few studies that measure data commonality in real workloads.

5.7 Chapter Summary

Managing large volumes of data is one of the major challenges inherent in developing and maintaining enterprise client management systems based on virtual machines. Using empirical data collected during seven-months of a live-deployment of one such system, we conclude that leveraging content addressable storage (CAS) technology can significantly reduce the storage and networking resources required by such a system (questions Q1 and Q2 from Section 3.1).

Our analysis indicates that CAS-based management policies typically benefit from dividing the data into very small chunk sizes despite the associated meta-data overhead. In the absence of compression, 4 KB chunks yielded the most efficient use of both storage and network resources. At this chunk size, a privacy-preserving CAS policy can reduce the system storage requirements by approximately 60% when compared to a block-based differencing policy (*Delta*), and a savings of approximately 80% is possible by relaxing privacy.

Similarly, CAS policies that leverage data cached on client machines reduce the average quantity of data that must be transmitted during both upload and download operations. For upload, this technique again results in a savings (compared to Delta) of approximately 70%

when preserving privacy and 80% when not. This technique also reduces the cost of *complete-fetch* download operations by more than 50% relative to the Delta policy (irrespective of CAS privacy policy) and by more than an order of magnitude relative to the cost when caching is not employed.

Leveraging compression in addition to CAS techniques provides additional resource savings, and the combination yields the highest efficiency in all cases. However, a surprising finding from this work is that CAS alone yields higher efficiency for this data set than compression alone, which is significant because the use of compression incurs a non-zero run-time cost for these systems.

This chapter provides an insight into the benefits of CAS, when applied on real usage data. We now present a few concluding remarks on this thesis.

Chapter 6

Conclusions

In this thesis we have explored the design of a content addressable file system – CAPFS. A key feature of CAPFS is incorporation of consistency/concurrency as part of the file system design. Integral to this is the use of CAS, and the recipe based representation of a file, allowing CAPFS to provide optimistic concurrency, thus boosting file system throughput. The use of CAS based storage on data servers de-links data operations from meta-data operations, and allows them to proceed in parallel, thereby increasing bandwidth further. We have also evaluated few real world application benchmarks on CAPFS, to gain insights into how CAS would perform on real-world data. The three most impotant insights were, i) a CAS based cache performs an outstanding job of saving network bandwidth; ii) CAS can save storage space for most live application workloads; iii) 1 KB is a good choice for chunksize. We noted that the largest bottleneck in a CAS based system is querying a CAS server for the presence of a chunk in its repository. We also find that SHA1 hash generation overheads are not significant, and are of the order of 15%. We concluded from the Internet Suspend/Resume study, that in a real world scenario, CAS can provide even more savings than in the case of application benchmark data. An important conclusion was that for such workloads, CAS provides better compression than *gzip*. We also noted remarkable storage space savings. Once again, the savings in network bandwidth makes a compelling case for the use of CAS based caches.

References

[1] 3tera. http://www.3tera.com/.

[2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, 2002.

[3] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.

[4] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, 2002.

[5] Miklos Ajtai, Randal Burns, Ronald Fagin, Darrell D. E. Long, and Larry Stockmeyer. Compactly encoding unstructured inputs with differential compression. *J. ACM*, 49(3):318–367, 2002.

[6] M. B. Alexander. *Assessment of Cache Coherence Protocols in Shared-Memory*. PhD thesis, University of Toronto, 2003.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, pages 63–73, Fall 1991.

[8] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery SIGOPS, 1991.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[10] John R. Black. Compare-by-hash: A reasoned analysis. In *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX'06)*, Boston, MA, June 2006.

[11] 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*, August 2000.

[12] William J. Bolosky, Scott Corbin, David Goebel, , and John R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, 2000.

[13] William J. Bolosky, John R. Douceur, David Ely, and Marvin 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.

[14] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. *ACM SIGMETRICS Performance Evaluation Review*, 28(1):34–43, 2000.

[15] P. J. Braam. The Lustre Storage Architecture, http://www.lustre.org/documentation.html, August 2004.

[16] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[17] 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.

[18] A. Broder. On the resemblance and containment of documents. In *SEQS: Sequences '91*, 1998.

[19] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, 2000.

[20] M. R. Brown, K. N. Kolling, and E. A. Taft. The Alpine file system. *ACM Transactions on Computer Systems*, 3(4), 1985.

[21] BSSN Pugh Benchmark. http://www.cactuscode.org/Benchmarks/bench_bssn_pugh.

[22] Ramon Caceres, Casey Carter, Chandra Narayanaswami, and Mandayam Raghunath. Reincarnating PCs with portable SoulPads. In *MobiSys '05: Proceedings of the 3rd International conference on Mobile Systems, Applications, and Services*, 2005.

[23] Cactus Code. http://www.cactuscode.org.

[24] Brad Calder, Andrew A. Chien, Ju Wang, and Don Yang. The entropia virtual machine for desktop grids. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution environments*, 2005.

[25] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[26] Calvin Chan and Hahua Lu. Fingerprinting using polynomial (rabin's method). Faculty of Science, University of Alberta, CMPUT690 Term Project, December 2001.

[27] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The Collective: A Cache-Based System Management Architecture. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.

[28] P.M. Chen and B.D. Noble. When Virtual is Better Than Real. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, 2001.

[29] Stephen Childs, Brian A. Coghlan, David O'Callaghan, Geoff Quigley, and John Walsh. A single-computer grid gateway using virtual machines. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications (AINA 2005)*, pages 310–315, Taipei, Taiwan, March 2005.

[30] R.E. Cohen, editor. *High-Performance Computing Requirements for the Computational Solid Earth Sciences*. 2005. http://www.geo-prose.com/computational_SES.html.

[31] Peter F. Corbett and Dror G. Feitelson. The Vesta Parallel File System. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. IEEE Computer Society Press and Wiley, 2001.

[32] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making Backup Cheap and Easy. In *OSDI: Symposium on Operating Systems Design and Implementation*, 2002.

[33] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing*. ACM Press, 1995.

[34] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[35] http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-2/. OSDL Database Test 2.

[36] J. M. del Rosario and A. N. Choudhary. High-performance i/o for massively parallel computers: Problems and prospects. *Computer*, 27(3):59–68, 1994.

[37] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 617, Washington, DC, USA, 2002. IEEE Computer Society.

[38] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *OSDI*, 2002.

[39] EMC Corp. *EMC Centera Content Addressed Storage System*, 2003. http://www.emc.com/.

[40] Renato J. Figueiredo, Peter A. Dinda, and Jos A. B. Fortes. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*, page 550, Washington, DC, USA, 2003. IEEE Computer Society.

[41] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 1997. http://www.mpi-forum.org/docs.

[42] M. Fridrich and W. Older. The Felix File Server. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, 1981.

[43] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[44] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

[45] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[46] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-Tolerant Erasure-Coded Storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, 2004.

[47] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: Resource Management using Virtual Clusters on Shared-Memory Multiprocessors. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, 1999.

[48] Susan Graham, Marc Snir, and Cynthia Patterson, editors. *Getting Up to Speed:The Future of Supercomputing*. 2004. http://www.nap.edu/catalog/11148.html.

[49] C. Gray and D. Cheriton. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 1989.

[50] Tim Grembowski, Roar Lien, Kris Gaj, Nghi Nguyen, Peter Bellows, Jaroslav Flidr, Tom Lehman, and Brian Schott. Comparative analysis of the hardware implementations of hash functions sha-1 and sha-512. In *ISC '02: Proceedings of the 5th International Conference on Information Security*, pages 75–89, London, UK, 2002. Springer-Verlag.

[51] Val 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.

[52] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Language Systems*, 15(5), 1993.

[53] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), 1988.

[54] IBM. Distributed Lock Manager for Linux, 2001. http://oss.software.ibm.com/dlm.

[55] Amazon.com Inc. Amazon Elastic Compute Cloud. http://www.amazon.com/gp/browse.html?node=201590011.

[56] Navendu Jain, Mike Dahlin, and Renu Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *USENIX Conference on File and Storage Technologies*, Dec 2005.

[57] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 29–42, Berkeley, CA, USA, 2003. USENIX Association.

[58] G. Kane and J. Heinrich. MIPS RISC Architecture, 1992. Prentice-Hall, Upper Saddle River, NJ.

[59] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM Press.

[60] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 1–15, Anaheim, CA, April 2005.

[61] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1991.

[62] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, 2002.

[63] Michael Kozuch and Mahadev Satyanarayanan. Internet Suspend/Resume. In *Fourth IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, New York, June 2002.

[64] Michael A. Kozuch, Casey J. Helfrich, David O'Hallaron, and Mahadev Satyanarayanan. Enterprise Client Management with Internet Suspend/Resume. *Intel Technology Journal*, November 2004.

[65] Ivan Krsul, Arijit Ganguly, Jian Zhang, Jose A. B. Fortes, and Renato J. Figueiredo. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[66] Ivan Krsul, Arijit Ganguly, Jian Zhang, Jose A. B. Fortes, and Renato J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 7, Washington, DC, USA, 2004. IEEE Computer Society.

[67] Purushottam Kulkarni, Fred Douglis, Jason D. LaVoie, and John M. Tracey. Redundancy Elimination Within Large Collections of Files. In *USENIX Annual Technical Conference, General Track*, 2004.

[68] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2), 1981.

[69] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[70] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[71] Jinyuan Li, Maxwell Krohn, David Mazierères, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 91–106, December 2004.

[72] Bin Lin and Peter Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC 2005)*, Seattle, WA, November 2005.

[73] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, 2004.

[74] U. Manber. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, CA, January 1994.

[75] C. May, E. Silha, R. Simpson, and H. Warren. The PowerPC Architecture: A Specification for a New Family of RISC Processors, 1994. Morgan Kaufman, San Francisco, CA, Second Edition.

[76] Sun Microsystems. Sun Grid Compute Utility. http://www.network.com/.

[77] http://www.venge.net/monotone/docs/hash-integrity.html.

[78] Tim D. Moreton, Ian A. Pratt, and Timothy 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.

[79] S. J. Mullender and A. S. Tanenbaum. A Distributed File Service based on Optimistic Concurrency Control. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1985.

[80] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

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

[82] NAS PARALLEL BENCHMARKS. http://www.nas.nasa.gov/Resources/Software/npb.html.

[83] Partho Nath, Michael Kozuch, David O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt 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.

[84] NCBI GenBank. http://www.ncbi.nlm.nih.gov/Genbank/.

[85] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. ACM Press, 1987.

[86] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX 2005 Annual Technical Conference*, Anaheim, CA, April 2005.

[87] NIST. Secure Hash Standard (SHS). In *FIPS Publication 180-1*, 1995.

[88] B. D. Noble and M. Satyanarayanan. An Empirical Study of a Highly Available File System. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems*, 1994.

[89] 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*.

[90] Calicrates Policroniades and Ian Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *USENIX Annual Technical Conference, General Track*, 2004.

[91] K. W. Preslan and A. P. Barry. A 64-bit, Shared Disk File System for Linux. Technical report, Sistina Software, Inc, 1999.

[92] K. W. Preslan, A. P. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S. Van Oort, D. Teigland, M. Tilstra, M. O'Keefe, G. Erickson, and M. Agarwal. Implementing Journaling in a Linux Shared Disk File System. In *IEEE Symposium on Mass Storage Systems*, 2000.

[93] 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.

[94] Sean Quinlan, Jim McKie, and Russ Cox. Fossil, an archival file-server. http://www.cs.bell-labs.com/sys/doc/fossil.pdf.

[95] R. B. Ross. Parallel I/O Benchmarking Consortium, http://www-unix.mcs.anl.gov/ rross/pio-benchmark.

[96] Michael Rabin. Fingerprinting by Random Polynomials. In *Harvard University Center for Research in Computing Technology Technical Report TR-15-81*, 1981.

[97] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Massachusetts Institute of Technology, Cambridge, MA, 1978.

[98] John Reumann, Ashish Mehra, Kang G. Shin, and Dilip D. Kandlur. Virtual Services: A New Abstraction for Server Consolidation. In *USENIX Annual Technical Conference, General Track*, 2000.

[99] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2003. USENIX Association.

[100] Sean Rhea, Kevin Liang, and Eric Brewer. Value-Based Web Caching. In *Proceedings of the Twelfth International World Wide Web Conference*, 2003.

[101] Riverbed Technology, Inc. http://www.riverbed.com.

[102] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. *Fast Software Encryption*, 2004.

[103] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), 1992.

[104] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, 1985.

[105] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, 1999.

[106] Constantine Sapuntzakis and Monica S. Lam. Virtual Appliances in the Collective: A Road to Hassle-Free Computing. In *Proceedings of the Ninth Workshop on Hot Topics in Operating System*, May 2003.

[107] 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*, December 2002.

[108] M. Satyanaranyanan, Michael A. Kozuch, Casey J. Helfrich, and David R. O'Hallaron. Towards Seamless Mobility on Pervasive Hardware. *Pervasive and Mobile Computing*, 1(2):157–189, July 2005.

[109] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*, 2002.

[110] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*, 2002.

[111] A. Siegel, K. P. Birman, and K. Marzullo. Deceit: A Flexible Distributed File System. Technical Report TR89-1042, Cornell University, 1989.

[112] R. L. Sites. Alpha AXP Architecture. *Digital Technical Journal*, 4(4):19–34, 1992.

[113] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata Efficiency in a Comprehensive Versioning File System. Technical Report TR CS4)2 145, Carnegie Mellon University, 2002.

[114] Neil T. Spring and David Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of ACM SIGCOMM*, August 2000.

[115] IEEE/ANSI Standard. 1003.1 Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language]., 1996.

[116] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Netw.*, 11(1), 2003.

[117] 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.

[118] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2000.

[119] H. Sturgis, J. Mitchell, and J. Israel. Issues in the design and use of a distributed file system. *SIGOPS Operating Systems Review*, 14(3), 1980.

[120] Torsten Suel, Patrick Noel, and Dimitre Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. *icde*, 00, 2004.

[121] Ananth I. Sundararaj and Peter A. Dinda. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 177–190, San Jose, CA, May 2004.

[122] S. Susarla and J. Carter. Flexible Consistency for Wide area Peer Replication. In *Twenty-fifth International Conference on Distributed Computing Systems*, June 2005.

[123] L. Svobodova. A reliable object-oriented data repository for a distributed computer system. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, 1981.

[124] Nut Taesombut and Andrew Chien. Distributed Virtual Computer (DVC): Simplifying the Development of High Performance Grid Applications. In *Proceedings of the Workshop on Grids and Advanced Networks (GAN'04)*, 2004.

[125] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[126] A. Thomasian. Checkpointing for Optimistic Concurrency Control Methods. *IEEE Transactions on Knowledge and Data Engineering*, 7(2), 1995.

[127] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003.

[128] Niraj Tolia, Jan Harkes, Michael Kozuch, and Mahadev Satyanarayanan. Integrating Portable and Distributed Storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004.

[129] Niraj Tolia, Michael Kaminsky, David G. Andersen, and Swapnil Patil. An architecture for internet data transfer. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006.

[130] 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.

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

[132] M. Uysal, A. Acharya, and J. Saltz. Requirements of I/O Systems for Parallel Machines: An Application-driven Study. Technical Report CS-TR-3802, University of Maryland, College Park, MD, 1997.

[133] Murali Vilayannur and Partho Nath. Technical Report and CAPFS Source code. *http://www.cse.psu.edu/∼vilayann/capfs/*.

[134] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. Providing Tunable Consistency for a Parallel File Store. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST'05)*, 2005.

[135] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Boston, MA, USA, 2002.

[136] Xiaoyun Wang, Yiqun L. Yin, and Hongbo Yu. *Finding Collisions in the Full SHA-1*, volume 3621. November 2005.

[137] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing Storage for a Million Machines. In *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS)*, 2005.

[138] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI*, 2004.

[139] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Dec 2002.

[140] M. Yang and Q. Koziol. Performance Comparison Study of Using Gzip and Bzip2 Data Compression Packages to NASA HDF-EOS Data and Other Scientific Data With Different Approaches. *AGU Fall Meeting Abstracts*, pages B185+, December 2002.

[141] Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '05)*, April 2005.

[142] H. Yu. TACT: Tunable Availability and Consistency Tradeoffs for Replicated Internet Services (poster session). *SIGOPS Operating Systems Review*, 2000.

[143] H. Yu and A. Vahdat. Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services. *ACM Transactions on Computer Systems*, 20(3), 2002.

[144] Yuting Zhang, Azer Bestavros, Mina Guirguis, Ibrahim Matta, and Richard West. Friendly Virtual Machines: Leveraging a Feedback-Control Model for Application Adaptation. In *VEE '05: Proc. 1st ACM/USENIX Inter. Conf. on Virtual Execution Envs*, 2005.

[145] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Vita

Partho Nath did his schooling from St. Francis de Sales School in New Delhi, India. Subsequently, he received his undergraduate B.Tech degree in Computer Science and Engineering from the Institute of Technology - Banaras Hindu University Varanasi, India in the year 2001. He expects to get his Ph.D degree in May 2007 from the Department of Computer Science and Engineering at the Pennsylvania State University. He spent the summers of 2003, 2004 and 2005 at Intel Research, Pittsburgh as a graduate intern. His research interests mainly include Operating Systems, Storage, and Virtual Machines. He is a student member of IEEE and ACM.