To appear in the Usenix 2003 Annual Technical Conference, San Antonio, TX, USA.

Opportunistic Use of Content Addressable Storage for Distributed File Systems

Niraj Tolia[†‡], Michael Kozuch[‡], Mahadev Satyanarayanan[†‡], Brad Karp[‡],
Thomas Bressoud[‡⋆], and Adrian Perrig[†]
[‡]Intel Research Pittsburgh, [†]Carnegie Mellon University, and [⋆]Denison University

Abstract

Motivated by the prospect of readily available Content Addressable Storage (CAS), we introduce the concept of file *recipes*. A file's recipe is a first-class file system object listing content hashes that describe the data blocks composing the file. File recipes provide applications with instructions for reconstructing the original file from available CAS data blocks. We describe one such application of recipes, the CASPER distributed file system. A CASPER client opportunistically fetches blocks from nearby CAS providers to improve its performance when the connection to a file server traverses a low-bandwidth path. We use measurements of our prototype to evaluate its performance under varying network conditions. Our results demonstrate significant improvements in execution times of applications that use a network file system. We conclude by describing fuzzy block matching, a promising technique for using *approximately matching* blocks on CAS providers to reconstitute the *exact* desired contents of a file at a client.

1 Introduction

The exploding interest in distributed hash tables (DHTs) [6, 26, 28, 34] suggests that *Content Addressable Storage (CAS)* will be a basic facility in future computing environments. In this paper we show how CAS can be used to improve the performance of a conventional distributed file system built on the client-server model. NFS, AFS and Coda are examples of distributed file systems that are now well-entrenched in many computing environments. Our goal is to improve client performance in situations where a distant file server is accessed across a slow WAN, but one or more *CAS providers* that export a standardized CAS interface are located nearby on a LAN.

The concept of a file *recipe* is central to our approach. The recipe for a file is a synopsis that contains a list of data block identifiers; each block identifier is a cryptographic hash over the contents of the block. Once the data blocks identified in a recipe have been obtained, they can be combined as prescribed in the recipe to reconstruct the file. On a cache miss over a low-bandwidth network, a client may request a file's recipe rather than its data. Often, the client may be able to reconstruct the file from its recipe by contacting nearby CAS providers (including CAS services available on the client) to which it has LAN access. In this usage scenario, a recipe helps transform WAN accesses into LAN (or local client) accesses. Since a recipe is a first class entity, it can be used as a substitute for the file in many situations. For example, if space is tight in a file cache, files may be replaced by the corresponding recipes, which are typically much smaller. This is preferable to evicting the file entirely from the cache because the recipe reduces the cost of the cache miss resulting from a future reference to the file. If the client is disconnected, the presence of the recipe may make even more of a difference — replacing an unserviceable cache miss by file reconstruction.

It is important to note that our approach is *opportunistic:* we are *not* dependent on CAS for the correct operation of the distributed file system. The use of recipes does not in any way compromise attributes such as naming, consistency, or write-sharing semantics. Indeed, the use of CAS is completely transparent to users and applications. When reconstructing a file, some blocks may not be available from CAS providers. In that case, those blocks must be fetched from the distant file server. Even in this situation, there is no loss of consistency or correctness.

CAS providers can be organized peer-to-peer networks such as Chord [34] and Pastry [28], but may also be impromptu systems. For example, each desktop on a LAN could be enhanced with a daemon that provides a CAS interface to its local disk. The CAS interface can be extremely simple; in our work, we use just four calls `Query`, `MultiQuery`, `Fetch`, and `MultiFetch` (explained in more detail in Section 5.3). With such an arrangement, system administrators could offer CAS access without being required to abide by any peer-to-peer protocol or provide additional storage space. In particular, CAS providers need not make any guarantees regarding content persistence or availability.

As a proof of concept, we have implemented a distributed file system called CASPER that employs file recipes. We have evaluated CASPER at different bandwidths using a variety of benchmarks. Our results indicate that substantial performance improvement is possible through the use of recipes. Because CASPER imposes no requirements regarding the availability of CAS providers, they are free to terminate their service at any time. This encourages users to offer their desktops as CAS providers with the full confidence that they can withdraw at any time. In some environments, a system administrator may prefer to dedicate one or more machines as CAS providers. Such a dedicated CAS provider is referred to as a *jukebox*.

After summarizing related work in Section 2, we discuss the utility of recipes and propose an XML representation in Section 3. We next describe the architecture of CASPER in Section 4, and relate details of its implementation in Section 5. In Section 6, we evaluate the performance of CASPER and quantify its performance dependence on the availability of matching CAS blocks. We briefly review security considerations associated with using recipes in Section 7. We propose a technique in Section 8 for using similar blocks, not only exactly matching blocks, with CAS. We summarize our findings and conclude in Section 9.

2 Related Work

The use of content hashes to represent files in file systems has been explored previously. Single Instance Storage [2] uses content hashes for compression, to coalesce duplicate files, and Venti [24] employs a similar approach at the block level. The Low Bandwidth File System (LBFS) [19] operates on variable-length blocks, and exploits commonality between distinct files and successive versions of the same file in the context of a distributed file system. The Pastiche backup system [7] employs similar mechanisms. In all these systems, content hashes are used internally. The CASPER file system described herein uses similar techniques, but proposes a canonical representation of hash-based file descriptions (recipes), and promotes those descriptions to members of a first-class data type. CASPER introduces the concept of a portable recipe that can be used to support legacy applications. In addition, CASPER allows all nodes with storage to participate as CAS providers, whereas prior systems, such as LBFS, restrict their search for file system commonality to the individual peer client and server.

Much attention has been devoted to using overlay networks to form Distributed Hash Tables (DHTs). Recent work in this area includes Chord [34], Pastry [28], CAN [26], and Freenet [6]. We believe that these DHTs are very valuable in the CAS context, where they may act as content providers for our system.

File systems built atop these DHTs include CFS [8], PAST [11], and Ivy [20]. These systems, while completely decentralized, share the model that participants that join the overlay offer write access to their local storage to other participants, either directly or indirectly. In CAS, participants may offer to share the contents of their storage with others *without* agreeing to store others' data. Moreover, a CASPER client can function without connectivity to a widely dispersed collection of overlay members. A CASPER client always can fall back to requesting data from its file servers, and choose to exploit available DHT infrastructure when beneficial. With the exception of Ivy, peer-to-peer file systems have traditionally been read-only or single-publisher systems. CASPER, however, can be layered over any traditional network file system, without changing the original file system's semantics.

Both Fluid Replication [15] and Pangaea [30] also attack the distributed file system performance problem on wide-area networks. Fluid Replication differs from CASPER in its dependence on dedicated machines to aid update propagation; while not implemented in CASPER, these techniques could complement our approach. While CASPER provides a conventional access control model in a client-server architecture, Pangaea uses a decentralized model and is designed to allow for ad hoc sharing and replication of data.

There is also a significant body of work in the area of delta encoding [18, 36, 37], but these methods require a fixed reference point for data comparison. CASPER, on the other hand, is opportunistic, does not need fixed file references, and works correctly in the absence of any previous version of an object.

3 Recipes

File recipes provide instructions for the construction of files from CAS data blocks. A recipe lists the addresses of CAS blocks that compose the desired file and describes the arrangement of those component blocks. For example, a file object could be divided into a sequence of 4 KB blocks. By listing the SHA-1 [22] hash of each block in order, we derive one possible recipe for the file. Other possible recipes exist: a sequence of SHA-1 hashes of 8 KB blocks or a sequence of hashes of variably sized blocks (such as might be generated using Rabin fingerprints [19, 25]). Once an object's recipe is known, the object may be reconstructed by fetching the component CAS objects (the "ingredients") named in the recipe from any available source and combining them as specified in the recipe.

Further, a recipe may include multiple recipe choices. Each choice is one possible method for describing the original file. With more than one choice available, a recipe-based application (*e.g.,* the CASPER file system) may select the most appropriate recipe choice for the situation. Suppose, for example, that CAS providers on one campus only support SHA-1 hashes of 4 KB blocks while CAS providers on a neighboring campus only support MD5 [27] hashes on 8 KB blocks. If a file recipe contains two choices, one based on SHA-1, 4 KB hashes and the other based on MD5, 8 KB hashes, the corresponding file could be reconstructed by recipe-based applications on either campus.

Each recipe maps a set of ingredients from the CAS namespace to a higher-level namespace, such as a file-system namespace. However, recipes are themselves first-class objects and may be stored in the higher-level namespace. One benefit of their first-class status is that recipes can be cached, and the consistency of recipes may be maintained by traditional coherency mechanisms. For example, in the CASPER file system, file recipes are stored as ordinary files

```
<?xml version="1.0"?>
<recipe type="file">
  <metadata>
    <length>125637</length>
    <last_modified>11/12/2002 16:24:37</last_modified>
    <file_system type="Coda">
      <name>/coda/projects/shared/casper.pl</name>
      <fid>312567 0 678</fid>
      <version>6 7 3 4 9</version>
    </file_system>
  </metadata>

  <recipe_choice>
    <hash_list hash_type="SHA-1" block_type="fixed"
               fixed_size="4096" number="31">
      <hash>09d2af8dd22...</hash>
      <hash>e5fa44f2b31...</hash>
      .
      .
    </hash_list>
  </recipe_choice>

  <recipe_choice>
    <hash_list hash_type="SHA-1" block_type="variable"
               number="36">
      <hash size="3576">7448d8798a4...</hash>
      <hash size="1278">a3db5c13ff9...</hash>
      .
      .
    </hash_list>
  </recipe_choice>

  <recipe_choice>
    <hash_list hash_type="MD5" block_type="fixed"
               fixed_size="125637" number="1">
      <hash>9c6b057a2b9...</hash>
    </hash_list>
  </recipe_choice>

</recipe>
```

Figure 1. Sample File Recipe

and may be cached in the file system cache. Consistency between cached recipes and the server version of the recipe is maintained by the cache coherency protocol. Consistency between a file and its recipe is maintained lazily by maintaining version numbers for all files.

We have adopted XML (Extended Markup Language) [4] as the language for expressing recipes. An example file recipe is shown in Figure 1. Of course, the main motivation for employing XML is portability. We believe that recipes are a generally useful abstraction, and therefore, by encoding recipes in a portable format, various applications will be able to make use of the same infrastructure.

The sample recipe in Figure 1 describes a file of 125637 bytes. While generating the recipe, the application that created the recipe included additional metadata such as the last modified time, the type of file system where the file was found, and file-system-specific data. Naturally, the recipe metadata could be extended with other information such as the file ownership and access permissions. Such information, while not required for CASPER, may benefit other applications that leverage file recipes. Following the metadata XML element are three possible recipe choices (`recipe_choice` elements) for recreating the data that constitute the file.

The first recipe choice is a list of SHA-1 hashes corresponding to 4 KB blocks. In our grammar, a `hash_list` is used to denote a series of hashes whose contents should be concatenated to form a region. In this case, the hashes compose the entire file. In fact, the 31 4 KB blocks described in the hash list compose an object slightly larger than the length given in the metadata. In such cases, after assembly, the resultant object must be truncated to the proper-length.

The second recipe choice is a `hash_list` comprising SHA-1 hashes of variable-length blocks, as might be generated by employing Rabin fingerprints to find block boundaries. Again, to reconstitute the file, the blocks corresponding to the hashes in the hash list need only be concatenated to form the original file.

The third recipe choice is a `hash_list` comprising a single MD5 hash of the entire file. This hash may be used as a final checksum to ensure (statistically) end-to-end file integrity. When assembling a file from constituent blocks, the recipe-based application may fail to find all the requested CAS components. The missing components must be retrieved through another mechanism, and once obtained, the complete file is assembled. The file-wide hash enables the CAS application to provide confidence that the file was assembled correctly.

Note that because the various choices provided in the recipe are typically orthogonal, in some cases the CAS data can be fetched from more than one CAS source. For example, suppose that after attempting to reconstruct the file from the 4 KB hash list, the recipe-based application determines that two of the 31 blocks could not be found. The client could then query CAS providers to determine if the appropriate blocks from the variable-length list that cover the missing two blocks can be found. If so, the data from those can be used to fill in the missing regions of the file.

4 The CASPER Distributed File System

CASPER is a distributed file system that employs file recipes to reduce the volume of data transmitted from a file server to its clients. CASPER clients cache files with whole-file granularity, and relies on centralized file servers to guarantee data persistence and file consistency. The novel aspect of CASPER is that clients make opportunistic use of nearby CAS providers to improve performance and scalability.

If the available client-server bandwidth is low during a file fetch operation, the client requests a recipe rather than the contents of the file. Using the hashes contained in the recipe, CASPER attempts to reconstruct the file by fetching components from nearby CAS providers. Any components not found near the client are fetched from the CASPER server, which is responsible for maintaining a master copy of every file it serves.

Small files (our implementation classifies files smaller than 4 KB as "small files") often do not enjoy the benefits of CAS acceleration due to recipe metadata and network overhead costs. Consequently, CASPER does not employ the recipe mechanism when transferring small files. Instead, the data composing a small file is transferred directly.

CASPER caches data at a whole-file granularity to provide the file-session oriented, open-close consistency model of AFS [13] and Coda [32, 33]. Consequently, once the file is reconstructed, it is placed in the client cache. For efficiency, CASPER clients treat their own caches as CAS providers. Before requesting a component from nearby, external CAS providers, clients inspect their own caches to determine if the component is also part of a previously cached file. In this way, CASPER mimics the fetch behavior of LBFS.

Throughout this paper, we are primarily concerned with client reads. However, we intend to extend our current implementation to accommodate client write operations by adopting a similar local cache lookup mechanism on the server-side. To leverage the recipe-based mechanism, we view client writes as server reads of the modified file. When sending file modifications to the server, a client sends a recipe of the modified file rather than the file contents. The server will then peruse its own data for instances of the components, then request components that are not found locally from nearby CAS providers, and finally retrieve any remaining components from the client directly.

As recipes in CASPER are treated as hints, the consistency between files and their recipes is managed in a lazy fashion. The file system maintains a version number for each file. When a recipe is generated, the recipe includes the version of the file from which it was derived. Because CASPER clients always check the expected version of the file against the version stored in the recipe metadata, a cached stale recipe will never be used to reconstruct a file. When a recipe is determined to be stale, CASPER triggers the creation of a new recipe by the server.

5 Implementation

Our implementation of CASPER is derived from the Coda distributed file system, and we have adopted the modular, proxy-based layering approach described in the Data Staging work [12]. Figure 2 depicts the organization of our system. The *Coda Client* and *Coda File Server* modules are unmodified releases of the Coda client and server, respectively. The *Proxy* module is responsible for intercepting communication between the client and server and determining whether or not to activate CASPER's CAS functionality. The Coda client and proxy together act as a CASPER client. Likewise, the Coda file server and *Recipe Server* together act as the CASPER server. The recipe server is the component responsible for forming responses to recipe requests.

The proxy-based design enables us to prototype new file-system features such as the CAS-based performance enhancements without modifying Coda. In this design, the proxy provides the client with an interface identical to the Coda server interface. The proxy monitors network conditions and determines when to use an available CAS provider. Under optimal conditions, the CASPER system behaves identically to an unmodified Coda installation without the proxy. After detecting low bandwidth network conditions, however, the proxy registers with the *recipe server* and a CAS provider. The provider in our example is a *jukebox*.

While low-bandwidth conditions persist, the proxy intercepts all file requests from the client and asks for the corresponding recipe from the recipe server. The recipe server is responsible for generating a recipe for the current version of the file and delivering it to the proxy. The proxy then attempts to retrieve the data blocks named in the file from nearby CAS providers (including the client's own file cache). The proxy will request that the recipe server also deliver any blocks not found on the CAS provider(s). Once the file reconstruction is complete, it is passed back to the Coda client, which places the file in its file cache. No other traffic, such as writes, is currently intercepted by the proxy; instead it passes directly to the file server.

In the next three sections, we describe the design and implementation of the recipe server, proxy and jukebox in more detail.

5.1 Recipe Server

As the name indicates, the recipe server generates the recipe representation of files. However, this module is also responsible for responding to requests for missed data blocks and forwarding callbacks.

The recipe server is a specialized user process that accesses file-system information through the standard file-system interface. Our implementation maintains generated recipes as files in the CASPER file system. For user files in a directory, `zeta`, the recipe server stores the corresponding recipes in a sub-directory of `zeta` (*e.g.,* `zeta/.shadow/`).

In Figure 2, we show the recipe server co-located with the Coda server. However, any machine that is well-connected to the server may act as the recipe server. In fact, if a user has a LAN-connected workstation, that workstation may make an excellent choice for the location of the user's primary recipe server because it is the most likely machine to have a warm client cache.

When a recipe request arrives at the recipe server, the recipe server first determines if the recipe file corresponding to the request exists in the file system. If so, the recipe is read, either from the recipe server's cache or from the Coda file server, and checked for consistency. That is, the versioning information in the recipe is compared to the current version of the file. If the recipe is stale or does not exist, the recipe server will generate a new one, send it to the proxy, and store it in the shadow directory for potential reuse.

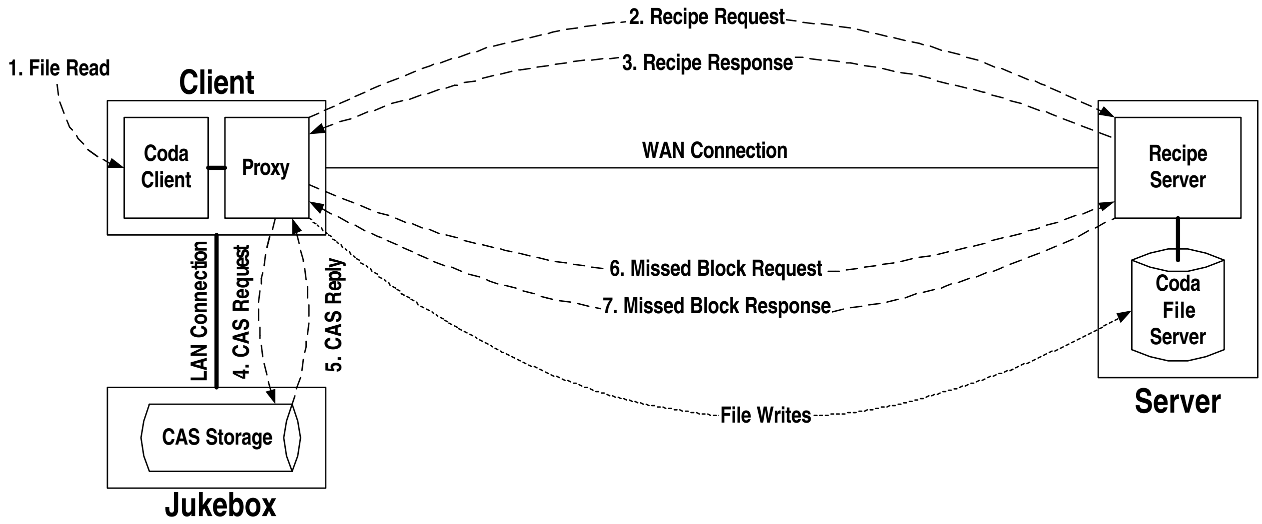


Figure 2. System Diagram

As mentioned in Section 4, the recipe server responds to requests for recipes of small files with the file's contents rather than the file's recipe. The threshold in the current implementation is 4 KB. The reason for this special handling is that the transfer time of small messages in low-bandwidth networks is often dominated by the network round-trip time.

Data blocks that are not found on any of the queried CAS providers are referred to as *missed blocks*. In our implementation, the proxy fetches missed blocks from the recipe server. Our recipe server supports requests for byte extents. That is, a missed block request is of the form `fetch(file, offset, length)`. To reduce the number of transmitted requests, missed blocks that are adjacent in the file are combined into a single extent request by the proxy, and we employ the further optimization of combining multiple fetch requests into a single multi-fetch message. In alternative implementations, missed block requests could be serviced by the file server. We chose to use the recipe server instead to reduce the file server load in installations where the recipe server is not co-located with the file server.

The recipe server also contributes to maintaining consistency by forwarding callbacks from the file server to the proxy for files that the proxy has reconstructed via recipes. That is, if the recipe server receives an invalidation callback for a file after sending a recipe for the file to a client, the callback is forwarded to the requesting client via the proxy.

Because the recipe server is a user process and not a Coda client, the recipe server does not receive the consistency callbacks directly. Instead, the recipe server communicates with the Coda client module through *codacon* [31], Coda's standard socket-based interface, which exports callback messages. Consequently, implementation of the callback-forwarding mechanism did not require modification of the client or server. Mechanisms that gather similar information, such as the SysInternals Filemon tool [35], could potentially serve the same function should CASPER be ported to network file systems that do not provide the necessary interface.

5.2 Proxy

The proxy is the entity that transparently adds recipe-based file reconstruction to distributed file systems without modifying the file system code. This arrangement is not necessary for the operation of CASPER but eased our implementation of the prototype. The two main tasks that it performs are fetching recipes and reconstructing files.

Because the proxy acts as the server for the client, the proxy intercepts all file read requests originating from the client. Client cache misses induce fetch requests which are caught by the proxy. The fetch operation causes the proxy to send a request for the file's recipe to the recipe server; other file system operations are forwarded directly to the Coda server. Assuming that a recipe fetch request is successful, the proxy compares the version in the received recipe to the expected version of the file. If the recipe version is more up-to-date than the proxy's expected version number, the proxy contacts the Coda server to confirm the file's version number. If the recipe version is older, there might be a problem with the recipe server (*e.g.,* the recipe server may have become disconnected from the Coda server), and therefore, the request is redirected to the file server. The request is also redirected to the file server if any unexpected recipe server errors occur.

If the version numbers match, the proxy selects one of the available recipe choices in the recipe file, and the proxy proceeds to reconstruct the file. After interpreting the XML data it received, the proxy queries both the client's local cache as well as any nearby jukeboxes for the hashes present in the recipe. Matching blocks, if any, are fetched from the jukeboxes, and the remaining blocks are fetched from the recipe server. Finally, the proxy assembles all constituent pieces and returns the resulting file to the client.

```
Retval Query(in Hash h, out Bool b);
Retval MultiQuery(in Hash harray[],
       out Bool barray[]);
Retval Fetch(in/out Hash h, out Bool b,
       out Block blk);
Retval MultiFetch(in/out Hash harray[],
       out Bool barray[], out Block blks[]);
```

The `Retval` is an enumerated type indicating whether or not the operation succeeded. Each operation also requires the identification of a channel as an input argument (not shown).

Figure 3. The Core CAS API

Note that we disabled Nagle's TCP/IP algorithm [21] for all recipe and hash requests issued by the proxy because the algorithm induced a performance overhead for these small requests.

5.3 Content Addressable Storage API

The CAS API defines the interface between *CAS consumers* and *CAS providers*. CAS consumers request data blocks by specifying a hash of the blocks' contents, and providers are storage devices that can respond to those requests. This API is what the proxy, a CAS consumer, uses to communicate with jukeboxes, which are CAS providers.

At the interface level, CAS consumers address providers through handles known as *channels*. This abstraction provides a common interface to all providers whether they are peer-to-peer networks, jukeboxes, or other devices. The CAS API provides a mechanism for obtaining the *characteristics* of a channel. While we do not currently use this mechanism, we envision that consumers will use it to determine such channel characteristics as the type of hashes supported, the channel's bandwidth, and some measure of the channel's topological proximity to the consumer.

The enumeration of available channels is the responsibility of a CAS API module running on the client system. The general topic of service discovery over a network is a rich area and will not be discussed here. In the context of the CAS API, the role of service discovery is to enumerate the set of available local and remote services and relay their associated characteristics to the consumer. The underlying mechanisms employed may be varied and may range from static specification and local operating system mechanisms to network-based protocols such as LDAP [14] or Universal Plug 'n Play [38].

Figure 3 summarizes the operations of the CAS API. The core operation of the CAS API is the retrieval of a CAS data block given the block's hash through the `Fetch` call. The consumer specifies the `hash` of the object to be retrieved (the hash data structure includes the hash type, block size, and hash value) and a channel. `Fetch` returns a boolean indicating if the block was found on the specified channel and, if so, the block itself. A consumer may instead make a single request for multiple blocks through the `MultiFetch` function, specifying an array of hashes and a channel. This function returns an array of Booleans and a set of data blocks. The CAS API also provides associated `Query` and `MultiQuery` operations for inquiring after the presence of a block without allocating the buffer space or network bandwidth to have the block returned.

We are currently developing a Content Addressable Storage wire protocol as a companion to the CAS API. With a common wire protocol, multiple CAS providers could be discovered and used in a consistent manner without requiring rewriting of the consumer-side CAS library code.

5.4 Jukebox

To evaluate our CASPER prototype, we implemented a jukebox CAS provider that conforms to the API described in Figure 3. The consumer-provider (proxy-jukebox) communication that supports the `Query` and `Fetch` functions is currently implemented by using a lightweight RPC protocol.

The jukebox, a Linux-based workstation, uses the native `ext3fs` file system to store the CAS data blocks. Currently, the system administrator determines a set of files to be exported as CAS objects. The jukebox makes use of recipes to track the location of data blocks within the exported files. The jukebox creates an in-memory index of the data at startup. The index, keyed by the hash, provides an efficient lookup mechanism.

6 Evaluation

We measured the performance benefit of recipe-based file reconstruction using three different benchmarks. We evaluated each benchmark under several combinations of network bandwidth, network latency and jukebox hit-ratio. We describe the experimental methodology, discuss the three benchmarks, and present their results.

6.1 Experimental Methodology

Our experimental infrastructure consisted of several contemporary, single-processor machines. The jukebox and client were workstations with 2.0 GHz Pentium® 4 processors. While the client had 512 MB of SDRAM, the jukebox had 1 GB. The file server contained a 1.2 GHz Pentium® III Xeon™ processor and 1 GB of SDRAM.

The jukebox and client ran the Red Hat 7.3 Linux distribution with the 2.4.18-3 kernel, and the file server ran the Red Hat 7.2 Linux distribution with the 2.4.18 kernel. All machines ran version 5.3.19 of Coda with a large enough disk cache on the client to prevent eviction during the experiments. The recipe server process was co-located with the file server. To discount the effect of cold I/O buffer caches, we ran one trial of each experiment before taking any measurements. However, we ensured that the Coda client cache was cold for all experiments.

To simulate different bandwidths and latencies, we used the NIST Net [23] network emulator, version 2.0.12. The client was connected to the jukebox via 100 Mb/s Ethernet, but the client's connection to the server was controlled via Nist Net. All benchmarks ran at three different client-server bandwidths: 10 Mb/s, 1 Mb/s and 100 Kb/s. We do not report results for a 100 Mb/s client-server connection, because CASPER clients should fetch data from the server directly when the client-server bandwidth is equal to, or better than, the client-jukebox bandwidth. Extra latencies of 10 ms and 100 ms were introduced for the 1 Mb/s and 100 Kb/s cases, respectively. No extra latency was introduced for the 10 Mb/s case.

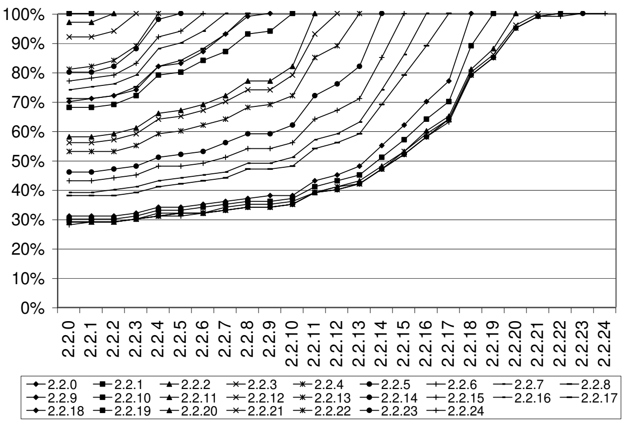
The effectiveness of recipe-based reconstruction in CASPER is highly dependent on the fraction of data blocks the client can locate on nearby CAS providers. We used sensitivity analysis to explore the effect of the CAS provider hit rate on CASPER performance. For each of the experiment and bandwidth cases, the jukebox was populated with various fractions of the requested data: 100%, 66%, 33%, and 0%. That is, in the 66% case, the jukebox is explicitly populated with 66% of the data that would be requested during the execution of the benchmark. The chunks for population of the jukebox were selected randomly. The two extreme hit-ratios of 100% and 0% give an indication of best-case performance and the system overhead. Our baseline for comparison is the execution of the benchmarks with an unmodified Coda client and server.

The recipes employed by CASPER for these experiments included a single type of recipe choice: namely, SHA-1 hashes of variable-size blocks with an average block size of 4 KB (similar to LBFS). During the experiments, none of the recipes were cached on the client. Every client cache miss generated a recipe fetch for the file. Further, because the client cache was big enough to prevent eviction of any reconstructed files, the client never requested a file or a recipe more than once.

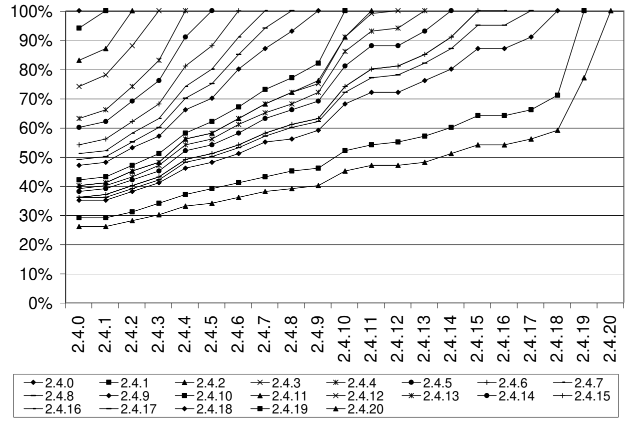
6.2 Measured Commonality

The hit rate of requests to CAS providers is highly dependent on the commonality in data between the file server and the CAS providers. Further, we expect the degree of commonality to depend on the applications that are trying to exploit it. Some applications, like the virtual machine migration benchmark used in our evaluation, would be well-suited to a CASPER approach. This benchmark reconstructs the data, including common operating system and application code, found on typical disk drives. A preliminary study by the authors of several user systems and prior work [3, 7] suggest that a high degree of commonality may be expected for this application.

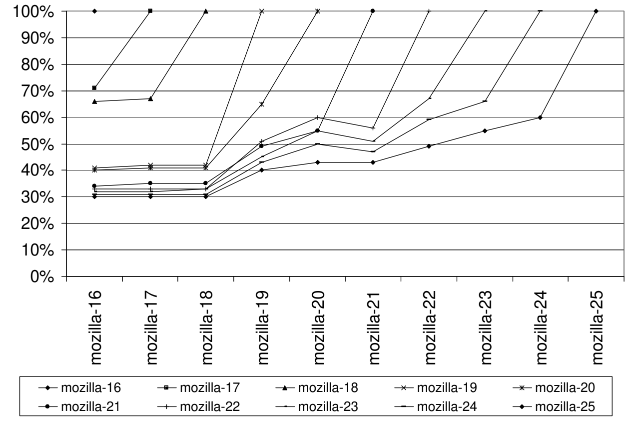
Commonality may also be expected when retrieving common binary or source-code distributions. For a software developer, different versions of a source tree also seem to exhibit this trend. For example, we compared the commonality between various releases of the Linux kernel source code. For this study, we define commonality as the fraction of unique blocks in common between pairs of releases.



(a) Commonality between Linux 2.2 Kernel releases



(b) Commonality between Linux 2.4 Kernel releases



(c) Commonality between Mozilla nightly binary releases

Each data series represents the measured commonality between a reference version of the software package and all previous releases of that package. Each point in the data series represents the percentage of data blocks from the reference version that occur in the previous release. The horizontal axis shows the set of possible previous releases, and the vertical axis relates the percentage of blocks in common. Each data series peaks at 100% when compared with itself.

Figure 4. Linux Kernel and Mozilla Commonality

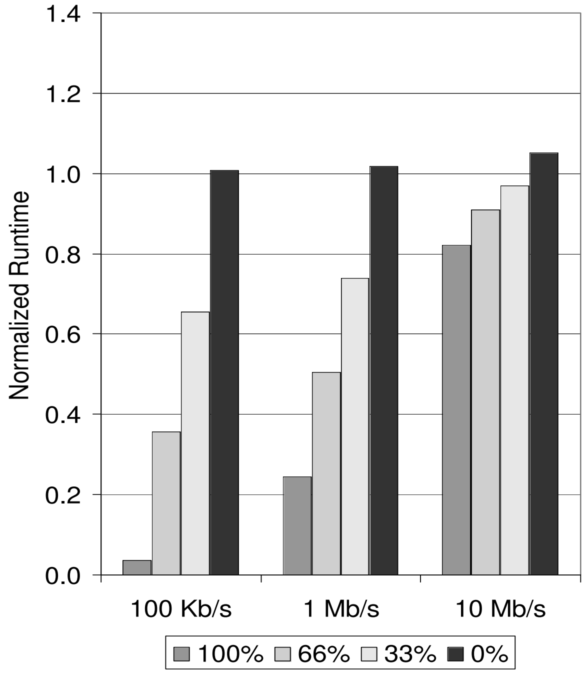
Figures 4 (a) and (b) show the measured commonality between versions of the Linux 2.2 and 2.4 kernels. We applied the block-extraction algorithm used in CASPER to each kernel's source tree and collected the unique hashes of all blocks in the tree. We then compared the set of hashes derived from each release to the set of hashes derived from each previous release in its series. The results show that commonality of 60% is not uncommon, even when the compared source trees are several versions apart. The minimal commonality we observe is approximately 30%. We will show in Section 6.3.3 that even this degree of commonality can lead to significant performance benefits from CAS in low-bandwidth conditions.

As small changes accumulate in source code over time, the corresponding compiled binary will diverge progressively more in content from earlier binaries. To assess how commonality in binaries changes as small source-level changes accumulate, we examined the nightly Mozilla binary releases from March 16th, 2003 to March 25th, 2003. These measurements are presented in Figure 4 (c). A binary on average shares 61% of its content in common with the binary of the preceding revision. In the worst case, where the CAS jukebox only has the binary from March 16th, but the client desires the binary from the 25th, we observe a commonality of 42%. We performed the same analysis on major Mozilla releases, but observed significantly less commonality for those binaries, because of the great increase in code size and functionality between releases. One interesting exception concerned release 1.2.1, a security fix; this release had 91% commonality with the previous one.

These measurements of cross-revision commonality for both source code and binaries are promising, but somewhat domain-specific. The degree of commonality in users' files bears future investigation. Highly personalized data are unlikely to be held in common by multiple users. But there may be cases where users share large objects, such as email attachments sent to many users in the same organization. An investigation of cross-user commonality is beyond the scope of this paper, as it requires a study of a substantial user population. However, we demonstrate with measurements that even in the presence of relatively little commonality, CASPER offers performance improvements to clients with a low-bandwidth connection to their file server. Moreover, our measurements show that CASPER adds very little overhead to file system operations when blocks cannot be found at a jukebox.

6.3 Benchmarks

We evaluated three different benchmarks on the CASPER file system: Mozilla software installation, an execution trace of a virtual machine application, and a modified Andrew Benchmark. The descriptions of these benchmarks together with experimental results is presented in the next three sections.



Mozilla install times with different Jukebox hit-ratios at 100 Kb/s, 1 Mb/s, and 10 Mb/s with 100 ms, 10ms and no added latency respectively. Each bar is the mean of three trials with with the maximum standard deviation observed as 1%.

Figure 5. Results: Mozilla Install

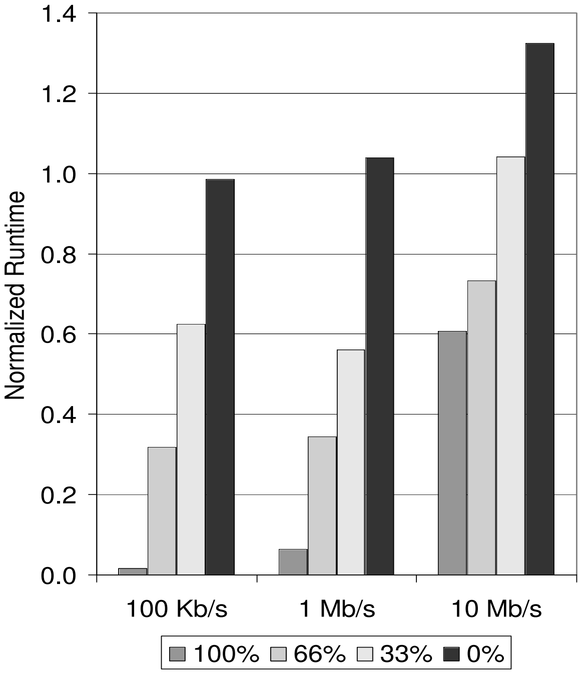
6.3.1 Mozilla Install

Software distribution is one application for which CASPER may prove effective. Often when installing or upgrading a software package, a previous version of the package may be found on the target machine or another machine near the target; the previous version may provide a wealth of matching blocks.

To evaluate software distribution using CASPER, we measured the time required to install the Mozilla 1.1 browser. The installation was packaged as 6 different RPM Package Manager [29] (RPM) files ranging from 105 KB to 10.2 MB with a total size of 13.5 MB. All the files were kept in a directory on the Coda File System and were installed by using the `rpm` command with the `-Uhv` options.

Figure 5 reports the time taken for the Mozilla install to complete at various network bandwidths. The times shown are the mean of three trials with a maximum standard deviation of 1%. To compare benefits at these different settings, the time shown is normalized with respect to the time taken for the baseline install with just Coda. The observed average baseline install times at 100 Kb/s, 1 Mb/s, and 10 Mb/s are 1238 seconds, 150 seconds, and 44 seconds, respectively. Note that apart from the network transfer time, the total time also includes the time taken for the `rpm` tool to install the packages.

As expected, the gain from using CASPER is most pronounced at low bandwidths where even relatively modest



Internet Suspend/Resume benchmark times with different Jukebox hit-ratios at 100 Kb/s, 1 Mb/s, and 10 Mb/s with 100 ms, 10ms and no added latency respectively. Each bar is the mean of three trials with the maximum standard deviation observed as 9.2%.

Figure 6. Results: ISR Benchmark

hit-rates can dramatically improve the performance of the system. For example, the reduction in benchmark execution time is 26% at 1 Mb/s with a 33% jukebox population. Further, while we do not expect the CASPER system to be used on networks with high bandwidth and low latency characteristics as in the 10 Mb/s case, the graph shows that the worst-case overhead is low (5%) and a noticeable reduction in benchmark time is possible (approximately 20% in the 100% population case).

6.3.2 Internet Suspend/Resume

Our original motivation for pursuing opportunistic use of Content Addressable Storage arose from our work on Internet Suspend/Resume (ISR) [16]. ISR enables the migration of a user's entire computing environment by layering a virtual machine system on top of a distributed file system. The key challenge in realizing ISR is the transfer of the user's virtual machine state, which may be very large. However, the virtual machine state will often include installations of popular operating systems and software packages – data which may be expected to be found on CAS providers.

To investigate the potential performance gains obtained by using CASPER for ISR, we employ a trace-based approach. For our workload, we traced the virtual disk drive accesses observed during the execution of an ISR system that was running a desktop application benchmark in demand-fetch mode. The CDA (Common Desktop Application) benchmark uses Visual Basic scripts to automate the execution of common desktop applications in the Microsoft Office suite executing in a Windows XP environment.

By replaying the trace, we re-generate the file-access pattern of the ISR system. The ISR data of interest is stored as 256 KB files in CASPER. During trace replay, the files are fetched through CASPER on the client machine. During the benchmark, approximately 1000 such files are accessed. The trace does not include user think time.

Figure 6 shows the results from the Internet Suspend/Resume benchmark at various network bandwidths. The time shown is again the mean of three trials with a maximum standard deviation of 9.2%. The times shown are also normalized with respect to the baseline time taken (using unmodified Coda). The actual average baseline times at 100 Kb/s, 1 Mb/s, and 10 Mb/s are 5 hours and 59 minutes, 2046 seconds, and 203 seconds, respectively.

By comparing with the previous Mozilla experiment, we can see that the benefit of using CASPER is even more pronounced as the quantity of data transferred grows. Here, unlike in the Mozilla experiment, even the 10 Mb/s case shows significant gains from exploiting commonality for hit-ratios of 66% and 100%.

Note that in the ISR experiment that used a 1 Mb/s link and a 33% hit rate, CASPER reduced execution time by 44%. The interaction of application reads and writes explains this anomaly. Every time an application writes a disk block after the virtual machine resumes, the write triggers a *read* of that block, if that block has not yet been fetched from the virtualized disk. Writes are asynchronous, and are flushed to disk periodically. If CASPER speeds execution, fewer writes are flushed to disk during the shorter run time, and fewer write-triggered reads occur during that execution. Conversely, if CASPER slows execution, more writes are flushed to disk during the longer run time, and more write-triggered reads occur during that execution. This interaction between writes and reads also explains the higher-than- expected overhead in the 33% and 0% hit-ratio cases on the 10 Mb/s link.

6.3.3 Modified Andrew Benchmark

We also evaluated the system by using a modified Andrew Benchmark [13]. Our benchmark is very similar to the original but, like Pastiche [7], uses a much larger source tree. The source tree, Apache 1.3.27, consists of 977 files that have a total size of 8.62 MB. The script-driven benchmark contains five phases. Beginning with the original source tree, the scripts recreate the original directory structure, executes a bulk file copy of all source files to the new structure, stats every file in the new tree, scans through all files, and finally builds the application.

To isolate the effect of CAS acceleration on file fetch operations, the modified Andrew Benchmark experiments were executed in write disconnected mode. Thus, once the Copy

Jukebox hit-ratio	Network Bandwidth		
	100 Kb/s	1 Mb/s	10 Mb/s
100%	261.3 (0.5)	40.3 (0.9)	17.3 (1.2)
66%	520.7 (0.5)	64.0 (0.8)	20.0 (1.6)
33%	762.7 (0.5)	85.0 (0.8)	21.3 (0.5)
0%	1069.3 (1.7)	108.7 (0.5)	23.7 (0.5)
Baseline	1150.7 (0.5)	103.3 (0.5)	13.3 (0.5)

Results from the Copy Phase of the modified Andrew Benchmark run at 100 Kb/s, 1 Mb/s, and 10 Mb/s with 100 ms, 10ms and no added latency respectively. Each result is the mean of three trials with the standard deviation given in parentheses

Figure 7. Results: Modified Andrew Benchmark Copy Phase

phase was complete, a local copy of the files allowed all remaining phases to be unaffected by network conditions; only the Copy phase will show benefit from the CASPER system.

The modified Andrew benchmark differs from the Mozilla and ISR benchmarks in that the average file size is small and more than half of the files are less than 4 KB in size. As described in Section 5.2, the current system is configured such that all recipe requests for files less than 4 KB are rejected by the recipe server in favor of sending the file instead. Thus while the jukebox might have the data for that file, CASPER fetches it across the slow network link.

Note that as all operations happened in write disconnected state, MakeDir, ScanDir, ReadAll and Make were not affected by bandwidth limitations, as they were all local operations. As Figure 8 shows, these phases are not affected by network conditions in the 1 Mb/s case. We observed very similar numbers for the 100 Kb/s and 10 Mb/s cases.

Figure 7 therefore only presents results for the Copy phase for all experiments, as Copy is the only phase for which the execution time depends on bandwidth. The figure shows the relative benefits of using opportunistic CAS at different bandwidths. The time shown is the mean of three trials with standard deviations included within parentheses. Interestingly, the total time taken for the 100 Kb/s experiment with an hit-ratio of 0% is actually less than the baseline. We attribute this behavior to the fact that some of the data in the Apache source tree is replicated and consequently matches blocks in the client cache.

The results for the 10 Mb/s case show a high overhead in all cases. This benchmark illustrates a weakness of the current CASPER implementation. When many small files are transferred over the network, the performance of CASPER is dominated by sources of overhead. For example, our proxy-based approach induces several additional inter-process transfers per file relative to the baseline Coda installation. Further, the jukebox inspection introduces multiple additional network round-trip time latencies associated with the jukebox query messages and subsequent missed-block requests.

Normally, these sources of overhead would be compensated for by the improved performance of converting WAN accesses to LAN accesses. However, for this particular benchmark the difference in peak bandwidth between our LAN connection and simulated WAN connection was not sufficient to compensate for the additional overhead. In practice, when CASPER detects such a high-bandwidth situation, the system should revert to baseline Coda behavior. Such an adjustment would avoid the system slowdown illustrated by this example.

7 Security Considerations

While the focus of our investigation has largely been on distributed file system performance, there are security considerations raised by the use of recipes. In this section, we briefly comment on the interaction of recipes and file system security. We consider the following security properties: data confidentiality, data authenticity, data integrity, data privacy, and user access control.

The foremost security concern raised by recipes is that querying a jukebox for a file poses a privacy risk: the jukebox learns the checksums of the blocks a user desires. With a large database of common blocks and their checksums, a malicious jukebox or eavesdropper could deduce the *content* a user requests. This privacy risk appears to be fundamental to our approach. An efficient solution remains an open problem.

Conventional encryption techniques also pose a challenge to recipe-based systems. If each user encrypts and decrypts her data with a distinct secret key, the same plaintext data would be encrypted into different ciphertext by users with different encryption keys. In such a case where commonality is obscured by encryption, a user could share blocks with herself alone, with marginal benefit at best. However, if we use *convergent encryption* [1, 9] (*i.e.* encrypting a block using a hash of the block as the key), recipes would be applicable—as identical data are encrypted into identical ciphertext, a recipe would again identify matching blocks.

On the other hand, recipes easily coexist with data authenticity. If data authenticity or non-repudiation is required, a file will simply contain a message authentication code or digital signature. As the authentication data are simply part of the file, recipes describe them as well. Likewise, recipes ensure data integrity. As a recipe provides strong cryptographic checksums of a file's contents, the user has a strong guarantee of a file's integrity when all its block checksums match.

A recipe-based system should enforce access control requirements for recipes that are identical to the requirements for the corresponding files. Even though recipes contain one-way hashes of file data, they still reveal information useful to a malicious user who wishes to learn the file's contents. If an attacker has an old version of a file and the up-to-date recipe for that file, he can compute many changed versions of the old file by brute force, and use checksums from the

AB phase	100%	66%	33%	0%	Baseline
MakeDir	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
Copy	40.3 (0.9)	64.0 (0.8)	85.0 (0.8)	108.7 (0.5)	103.3 (0.5)
ScanDir	2.0 (0.0)	2.0 (0.0)	2.0 (0.0)	2.3 (0.5)	2.0 (0.0)
ReadAll	3.7 (0.5)	3.7 (0.5)	4.0 (0.0)	3.7 (0.5)	3.7 (0.5)
Make	15.3 (0.5)	16.0 (0.0)	15.7 (0.5)	16.0 (0.0)	15.7 (0.5)
Total	61.3 (0.9)	85.7 (1.2)	106.7 (0.5)	130.7 (0.5)	124.7 (0.5)

Modified Andrew Benchmark run at 1 Mb/s. Each column represents the hit-ratio on the jukebox. Times are reported in seconds with standard deviations given in parentheses

Figure 8. Results: Modified Andrew Benchmark (1 Mb/s)

recipe to identify the correct version of the new file. To prevent such attacks, CASPER assigns recipes identical permissions to the files they summarize.

8 Future Extension: Fuzzy Block Matches

Our experimental results clearly show that at a given bandwidth between client and server, the dominant factor in determining the performance benefit afforded by CAS is the block hit ratio at the jukebox. This relationship suggests that strategies for increasing the hit ratio may improve the performance of CAS-enabled storage systems. Toward this end, in this section we propose *fuzzy block matching*, a promising enhancement to CAS that is the topic of our continuing research and implementation efforts.

The two recipe block types we have considered thus far, `fixed`- and `variable`-length blocks, require *exact* matches between a block's hash in a recipe and a block's hash at the jukebox. Matching hashes prove, with extremely high probability, that the jukebox and "home" file server (which generated the recipe from the canonical instance of the file data) hold the same data for a block. However, an inexact match between block contents is also possible. Here, two blocks may share many bytes in common, but differ in some bytes. Fuzzy block matching allows a CAS client to request blocks with the *approximate* contents it needs from a jukebox, and to use these approximately matching blocks to reconstruct a file so that its contents *exactly* match the canonical version of the file.

One cannot simply use as-is a block whose contents only approximately match those desired. The resulting reconstructed file would differ from the canonical file stored at the home server. After Lee *et al.* [17], we view a block at a jukebox that approximately matches the desired block as an *errored* version of the desired block, received through a noisy channel. Lee *et al.* use error-correcting codes (ECCs) to correct short replacements in similar disk blocks. We note an analogous opportunity in CAS: if we could somehow *correct* the errors in the jukebox's block, we could produce the exact desired block.

Under fuzzy block matching, the specification of a block in a recipe includes three pieces of information:

- **An exact hash value** that matches only the correct block;
- **A fuzzy hash value** that matches blocks similar to the correct block;
- **Error-correcting information** that, when applied to a block similar to the correct block, may sometimes recover the correct block.

While the hit-rate increase offered by fuzzy block matching will improve with added error-correcting information, we note that these three items must be compact in their total size. For recipes to be a viable mechanism, they must be significantly smaller than the file contents themselves. We expect that a few hundred bytes will suffice to hold useful fuzzy hash values and error-correcting information.

Fuzzy block matching works as follows:

- The CAS client sends the jukebox the enhanced recipe entry for a block, described above.
- The jukebox first determines whether it holds a block whose hash matches the exact hash value; if so, it returns this block to the client.
- The jukebox next uses the fuzzy hash value to find any *candidate blocks* it holds that approximately match the block sought by the client.
- The jukebox applies the error-correcting information to each candidate block. If the corrected block's hash matches the exact hash value from the recipe, the jukebox returns the corrected block to the client.
- If none of the jukebox's candidate blocks can be corrected to match the exact hash, the CAS client has missed in the jukebox.

There are two algorithmic aspects to fuzzy block matching: identifying approximately matching blocks with fuzzy hashing, and error-correcting approximately matching blocks, to recover correct block. We now sketch the techniques we are pursuing to solve these two problems.

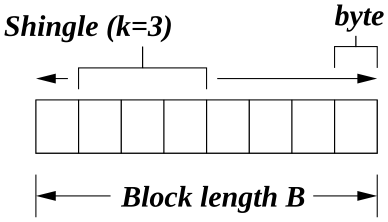


Figure 9. Structure of a disk block (length B) and shingle (k contiguous bytes). Here, $k = 3$.

8.1 Fuzzy Hashing: Shingling

The well-known technique of *shingling* [5] efficiently computes the *resemblance* of web documents to one another. Douglis and Iyengar [10] apply shingling to identify similar data objects in a file system, so that one may be delta-encoded in terms of the other. We briefly describe an adaptation of Broder *et al.*'s shingling to binary disk blocks in the CAS context.

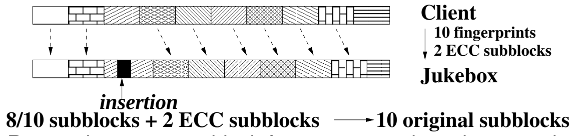
We assume variable-length blocks, whose boundaries are determined by Rabin fingerprints over block contents, as described in Section 3. Consider a block of length B bytes. Define a *shingle* to be a sequence of k contiguous bytes in a block. Shingles overlap; there are $B-k+1$ shingles in a block of length B bytes, one beginning at each of bytes zero through $B-k$. Figure 9 shows a shingle within a disk block. The horizontal arrows denote the "sliding" of the k-byte window within the block, to produce different shingles.

Compute a hash of each shingle in the disk block. Broder *et al.* use an enhanced version of Rabin fingerprints [25] for hashing, for reasons of computational efficiency. This procedure produces a set of integral shingle values. Sort these shingle values into numeric order, and select the m smallest unique values. These m values form a *shingleprint,* which represents the approximate contents of the disk block. Estimating the resemblance of two blocks is a simple computation involving the number of their m shingle values they share in common.

Under CAS, a shingleprint can be stored in a recipe as the fuzzy hash of a block. A client presents the shingleprint to a jukebox, which may use it to identify blocks similar to the desired block.

8.2 Using Similar Blocks: ECCs

The essence of our approach is to divide a variable-length disk block into fixed-length, non-overlapping subblocks (the last subblock may be shorter than the others), and to detect (possibly offset) subblocks that remain unchanged in content after insertions, deletions, or replacements in the full disk block. Subblocks that changed are treated as errored. By targeting a specific maximum number of changed subblocks, we can store enough ECC information to recover the original data for the whole block. In this scheme, replacements, insertions, and deletions within fixed-length blocks amount to corruptions of subblocks.



Recovering a correct block from an errored version stored at a jukebox using ECC data: the jukebox scans for unchanged subblocks, and uses them with ECC blocks to produce the correct block.

Figure 10. Example of Similar Block Correction

Again working with a disk block of length B, let us define a fixed subblock length L, where $L < B$. The client's recipe includes a Rabin fingerprint for each subblock. When the client requests similar blocks from a jukebox, it includes these Rabin fingerprints over the subblocks in the request.

For each candidate block it holds, the jukebox scans for the subblock Rabin fingerprints it receives from the client. By "scan," we mean that the jukebox computes a Rabin fingerprint on every subblock in the candidate block, scanning each contiguous region of length L. As described previously, Rabin fingerprints are computationally efficient to compute in a sliding window at every offset in a block of data. Even if the candidate block contains insertions, deletions, and replacements, it is likely that it will contain subblocks identical to subblocks of the correct block.

An example of this ECC procedure is provided in Figure 10. Here, the client desires a disk block, subdivided into ten subblocks. The client stores in its recipe the Rabin fingerprint for each of these ten subblocks. It also stores two ECC subblocks, each 128 bytes long, originally derived from the block's canonical contents. Let us assume that the jukebox has a candidate block found by shingling, but that contains an insertion of length shorter than one subblock. The jukebox computes the Rabin fingerprint for each contiguous 128-byte subblock of the candidate block, and identifies those subblocks whose Rabin fingerprints match those provided by the client; these subblock correspondences between the correct block and jukebox block are shown by vertical dashed arrows in Figure 10. Thereafter, the jukebox applies the ECC to these unchanged subblocks and the two ECC blocks, and overcomes the "erasure" of the modified subblock, to produce the exact desired block. As a final check, the jukebox computes an exact-match hash over the whole corrected block, and compares it with the exact-match hash requested by the client. We assume here that we've chosen an ECC robust to loss of two subblocks out of ten; in practice, the degree of redundancy chosen for the ECC will reflect the degree of difference between blocks from which one wishes to recover successfully. One pays for increased robustness to changes in a candidate block with increased storage for ECC blocks in recipes.

In Figure 11, we detail the size of the four per-block components of recipes in the prototype implementation of fuzzy block matching we are currently developing. The prototype uses variable-sized blocks of mean size 4K, and subblocks 128 bytes in length. The SHA-1 exact-match hash

Content	Total Size (bytes)
SHA-1 exact hash	20
Shingleprint fuzzy hash	80
Subblock Rabin fingerprints	256
ECC information	256

Average per-block recipe storage cost in prototype supporting fuzzy block matching on variable-sized blocks (4KB mean block size).

Figure 11. Fuzzy Block Matching Recipe Storage

requires 20 bytes of storage. The shingleprint fuzzy hash consists of ten 64-bit Rabin fingerprints. The subblock fingerprints (which are also computed as Rabin fingerprints) require 64 bits each. Finally, our prototype includes two 128-byte ECC subblocks per block, and can thus successfully reconstruct a desired block from a similar block that differs in up to two 128-byte subblocks. The total recipe storage per block is 612 bytes, about 15% the size of a 4K block, but an order of magnitude longer than an exact-match SHA-1 hash alone. We note that this storage cost is conservative; fuzzy block matching with less recipe storage per block should be achievable. Truncating the Rabin fingerprints used in the shingleprint and per-subblock from 64 bits to 32 bits would reduce the storage requirement to 444 bytes, or 10.8% the size of a 4K block. Truncating these fingerprints increases the likelihood of collisions in fingerprint values, which can cause shingling to overestimate resemblance, and cause subblocks to be falsely identified as matching. Because the whole-block hash is always verified before returning a corrected block to a client, however, these collisions do not compromise the correctness of fuzzy block matching. Measuring the increase in hit rate afforded by fuzzy block matching is the subject of our continuing research.

9 Conclusion

In this paper, we have introduced the file *recipe* abstraction. File recipes are useful synopses of file contents that may be treated as first-class objects. Using this abstraction, we enhanced a distributed file system to retrieve file data from nearby Content Addressable Storage *opportunistically*. In cases where the network path between a client and its home file server offers low bandwidth, the resulting distributed file system, CASPER, offers significantly improved performance over a traditional Coda installation.

The efficiency of CASPER depends directly on the ability of the system to locate data blocks on CAS providers. Preliminary measurements of the overlap in data blocks across revisions of the Linux kernel sources and Mozilla nightly binary releases show significant commonality, even between non-consecutive pairs of releases. Finally, our ongoing work on a Fuzzy Block Matching algorithm holds promise for improving the data-block hit rate by using approximately matching blocks from CAS providers to reconstitute the exact desired contents of a file.

Acknowledgments

We would like to thank Jason Flinn and Shafeeq Sinnamohideen for their help and feedback with the design of CASPER, and Casey Helfrich for his help in setting up the test machines. We have benefited greatly from the work of Jan Harkes, Peter Braam and many other past contributors to Coda. Our shepherd, Amin Vahdat, and the anonymous reviewers gave valuable feedback and many suggestions for improving the paper. All unidentified trademarks used in this paper are properties of their respective owners.

References

[1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI '02)* (December 2002).

[2] BOLOSKY, W. J., CORBIN, S., GOEBEL, D., , AND DOUCEUR, J. R. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24. Seattle, WA (August 2000).

[3] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. 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). ISSN 0163-5999.

[4] BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., AND MALER, E. Extensible Markup Language (XML) 1.0 (Second Edition) (October 2000). http://www.w3.org/TR/REC-xml.

[5] BRODER, A., GLASSMAN, S., MANASSE, M., AND ZWEIG, G. Syntactic Clustering of the Web. In *Proceedings of the 6th International WWW Conference* (1997).

[6] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Lecture Notes in Computer Science*, 2009:46+ (2001).

[7] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making Backup Cheap and Easy. In *OSDI: Symposium on Operating Systems Design and Implementation* (2002).

[8] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. 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).

[9] DOUCEUR, J. R., ADYA, A., BOLOSKY, W. J., SIMON, D., AND THEIMER, M. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS 2002)* (July 2002).

[10] DOUGLIS, F. AND IYENGAR, A. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of the USENIX Annual Technical Conference*. San Antonio, Texas (June 2003).

[11] DRUSCHEL, P. AND ROWSTRON, A. PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In *HotOS VIII*, pages 75–80. Schloss Elmau, Germany (May 2001).

[12] FLINN, J., SINNAMOHIDEEN, S., TOLIA, N., AND SATYANARAYANAN, M. Data Staging on Untrusted Surrogates. In *Proceedings of the FAST 2003 Conference on File and Storage Technologies* (2003).

[13] HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1) (February 1988).

[14] HOWES, T. A. AND SMITH, M. C. A Scalable, Deployable, Directory Service Framework for the Internet. Technical report, Center for Information Technology Integration, Univerity of Michigan (1995).

[15] KIM, M., COX, L. P., AND NOBLE, B. D. Safety, Visibility and Performance in a Wide–Area File System. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*. Monterey, CA (January 2002).

[16] KOZUCH, M. AND SATYANARAYANAN, M. Internet Suspend/Resume. In *Fourth IEEE Workshop on Mobile Computing Systems and Applications*. Callicoon, New York (June 2002).

[17] LEE, Y.-W., LEUNG, K.-S., AND SATYANARAYANAN, M. Operation-based Update Propagation in a Mobile File System. In *Proceedings of the USENIX Annual Technical Conference*. Monterey, California (June 1999).

[18] MOGUL, J. C., DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *SIGCOMM*, pages 181–194 (1997).

[19] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. Chateau Lake Louise, Banff, Canada (October 2001).

[20] MUTHITACHAROEN, A., MORRIS, R., GIL, T., AND CHEN, B. 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).

[21] NAGLE, J. RFC 896: Congestion Control in IP/TCP Internetworks (January 1984).

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

[23] NIST Net. http://snad.ncsl.nist.gov/itg/nistnet/.

[24] QUINLAN, S. AND DORWARD, S. Venti: A New Approach to Archival Storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies* (2002).

[25] RABIN, M. Fingerprinting by Random Polynomials. In *Harvard University Center for Research in Computing Technology Technical Report TR-15-81* (1981).

[26] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A Scalable Content Addressable Network. In *Proceedings of ACM SIGCOMM 2001* (2001).

[27] RIVEST, R. The MD5 Message-Digest Algorithm. *RFC 1321* (1992).

[28] ROWSTRON, A. AND DRUSCHEL, P. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. Heidelberg, Germany (November 2001).

[29] RPM Software Packaging Tool. http://www.rpm.org/.

[30] SAITO, Y., KARAMANOLIS, C., KARLSSON, M., AND MAHALINGAM, M. Taming Aggressive Replication in the Pangaea Wide-Area File System. In *OSDI: Symposium on Operating Systems Design and Implementation* (2002).

[31] SATYANARAYANAN, M., EBLING, M. R., RAIFF, J., BRAAM, P. J., AND HARKES, J. *Coda File System User and System Administrators Manual*. Carnegie Mellon University (1995).

[32] SATYANARAYANAN, M., KISTLER, J., KUMAR, P., M.E., O., SIEGEL, E., AND STEERE, D. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4) (April 1990).

[33] SATYANARAYANAN, M. The Evolution of Coda. *ACM Transactions on Computer Systems*, 20(2) (May 2002).

[34] 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).

[35] SYSINTERNALS. http://www.sysinternals.com.

[36] TICHY, W. F. RCS - A System for Version Control. *Software - Practice and Experience*, 15(7):637–654 (1985).

[37] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. Ph.D. thesis, The Australian National University (1999).

[38] Universal Plug and Play. http://www.upnp.org/.