Supporting Practical Content-Addressable Caching with CZIP Compression

KyoungSoo Park Sunghwan Ihm Mic Bowman Vivek S. Pai
Princeton University *Intel Research* *Princeton University*

Abstract

Content-based naming (CBN) enables content sharing across similar files by breaking files into position-independent chunks and naming these chunks using hashes of their contents. While a number of research systems have recently used custom CBN approaches internally to good effect, there has not yet been any mechanism to use CBN in a general-purpose way. In this paper, we demonstrate a practical approach to applying CBN without requiring disruptive changes to end systems.

We develop CZIP, a CBN compression scheme which reduces data sizes by eliminating redundant chunks, compresses chunks using existing schemes, and facilitates sharing within files, across files, and across machines by explicitly exposing CBN chunk hashes. CZIP-aware caching systems can exploit the CBN information to reduce storage space, reduce bandwidth consumption, and increase performance, while content providers and middleboxes can selectively encode their most suitable content. We show that CZIP compares well to standalone compression schemes, that a CBN cache for CZIP is easily implemented, and that a CZIP-aware CDN produces significant benefits.

1 Introduction

Content-based naming (CBN) refers to a naming scheme in which pieces of content are indexed by hashes over their data. By splitting the content into smaller-sized "chunks" and obtaining their chunk hashes using a one-way cryptographic hash function (e.g., MD5, SHA-1), any content can be represented as a list of chunk hashes. The main goal behind the scheme is to reduce storage space or network bandwidth consumption by eliminating redundant chunks. Redundant chunks can be found within a single file, across files (such as snapshots of the same file over time, or collections of files), or even across machines. These latter two scenarios require building a "CBN cache," which is a cache indexed by chunk hashes.

One of the main enablers of variable-sized chunking for CBN is the Rabin fingerprinting method, which breaks a stream of data into position-independent chunks, allowing similar content to be detected even when parts of files differ [20]. A number of research systems have been developed that use CBN internally, which range from distributed file systems [2, 13, 29] and Web caching [3, 12, 21] to a cross-application transfer service architecture [28]. The commercial sector has systems which apply the same concepts to disk blocks [8], file backup [6], and WAN-link accelerators [22]. Though the use of CBN has been demonstrated in both the research and commercial sectors, there remains no easy way of applying the concept in practice without invasively changing the target platform, or designing the platform with CBN integration from the start.

Our goal in this work is to develop a file format and system that allows users to opportunistically deploy CBN while keeping their current systems intact. For example, a system employing CBN could reduce the memory footprint of Linux distribution mirrors by eliminating redundant data, since the same content is served as a DVD ISO image as well as multiple CD ISO images. At the same time, one may not want less suitable formats (unrelated RPMs or text files which rarely share common chunks among them) to be served from the CBN cache, because that would only increase the overhead with no real gain. Another similar case is transferring multiple but slightly different virtual machine (VM) images with the same base operating system from a central location (e.g., the office) to one or more destinations (e.g., home machines or an off-site facility). With typical VM image sizes ranging from many hundreds of MBs to a few GBs, placing a CBN chunk cache near the destination can help reduce a significant amount of network bandwidth by only transferring the difference after the first VM image. If the updated images are transferred in each direction when the user commutes to/from the office, the CBN can cause just the updates to be sent.

In this paper, we consider how to selectively employ CBN without requiring any support from the underlying systems. We propose a generic compression scheme based on CBN called CZIP which provides chunking, naming, and compression, allowing CZIP-aware systems to eliminate redundant chunks across files. CZIP identifies unique chunks in the input file (or stream), and then compresses the chunk by existing compression methods, such as GZIP or BZIP2. CZIP exposes chunk content hashes in the header, and CZIP-aware systems can easily recognize the content and exploit CBN caching just by reading the header information.

This approach provides an appealing alternative to designing systems around CBN, and provides some advantages: (a) users or applications can better choose the set of content for CBN encoding without changing the existing environment, (b) because the file format is generic and independent of any particular system, different types of CBN caches can be utilized without sacrificing transparency, (c) even without a CBN cache, the compression scheme itself greatly reduces the content size where chunk commonality exists, and is comparable to other compression schemes in other cases.

We provide some examples of these benefits later in this paper, including the following highlights. In creating mirror servers for the Linux Fedora Core 6 distribution, CZIP reduces the data volume by a factor of 21-25 more than GZIP or BZIP2. For this kind of mirror, our server-side CBN cache provides a dramatic improvement in throughput by eliminating redundant disk reads and minimizing the memory footprint. We also integrate CBN support into the CoBlitz large-file content distribution network (CDN) [15], and show that it reduces the bandwidth consumption at the origin server by a factor of four, with no modification of the server or the client.

The rest of this paper is organized as follows: we provide some motivating examples for CBN and the CZIP format in Section 2. We then provide details about the design of CZIP in Section 3, and describe some typical deployment scenarios in Section 4. We perform some experiments on CZIP's effectiveness on different data types in Section 5, and evaluate the performance of two CZIP-augmented systems in Section 6. Finally, we discuss related work in Section 7 and then conclude.

2 Motivation

To illustrate the benefits of a common CBN-enabling format, we discuss a few candidate scenarios below. All of these examples are from systems we (and our colleagues) have built or are building at the moment, so an approach like CZIP, rather than just being theoretically interesting, actually stands to provide us with practical benefits.

2.1 Software Distribution

Software distribution over the network has been gaining popularity, especially as broadband penetration has increased, making download times more reasonable. Linux distributions are just one example of these kinds of systems, with popular projects like the Fedora Core distribution having over 100 mirror sites [1]. However, as users come to expect more capabilities, features, and packages bundled with the OS, the download sizes have increased, and the Fedora Core 6 distribution spans five CD-ROM images or one DVD-ROM image, at a total size of 3.3 GB. Since users may desire one format over another, any popular mirror site must keep both, requiring over 6 GB of space for just a single architecture. While this disk space is a trivial cost, the real problem is when this data is being served – it is larger than the physical memory of most systems, so it causes heavy disk access. The Fedora Core project has also been providing images for the 64-bit x86 architecture since the release of Fedora Core 2, and PowerPC since Fedora Core 4. While the 64-bit x86 extensions were originally available only in higher-end processors, their migration down the hierarchy to lower-end machines has also increased the demand for the x86/64 Fedora Core distribution.

Even if a mirror site provided only the two x86 distributions in both DVD and CD formats, the total size is over 12 GB. Unfortunately, this figure exceeds the physical memory size of most servers. Releases of new Fedora Core distributions tend to cause flash crowds – our own CoBlitz large-file distribution service experienced peak downloads rates of over 1.4 Gbps aggregate, and sustained rates over 1.2 Gbps [2]. In these scenarios, thousands of simultaneous users are trying to download from mirror sites, and between their sheer numbers, varying download rates, and different start times, virtually all parts of all of the files will be in demand simultaneously, causing significant memory pressure. Some popular mirror sites were unable to serve at their peak capacity due to the memory thrashing effects. One mirror site operator with 2 Gbps of bandwidth was only able to serve 500 Mbps since his system had only 2 GB of physical memory and was heavily thrashing [3].

The reason that CBN is important in these scenarios is because much redundant data exists, both across media formats (CD vs DVD) as well as across distributions for different architectures (x86 vs x86/64). The reason for the former is simple – the same files are simply being rearranged and placed on media of different sizes. The reason to expect similarity across architectures is that not all of the files in any distribution are executables. So, while the executable files may have virtually no similar chunks across different architectures, all of the support files, including documentation (PDFs, HTML pages, GIF and JPEG images, etc.) will likely be the same, as will many of the program resource files (configuration files, skins/textures, templates, sample files, etc.). We can expect savings at both of these levels, driving down the memory footprint required for serving multiple architectures. In an ideal scenario, we would expect no overhead for serving both the CD and DVD images, and each additional architecture would only expand the memory requirements by the size of the executable files.

[1] http://fedora.redhat.com/download/mirrors.html

[2] http://codeen.cs.princeton.edu/coblitz/

[3] mirror-list-d@redhat.com, Oct 26, 2006

2.2 Virtual Machine Image Mobility

Another area where large amounts of similar content are expected to occur is in the handling of virtual machine images. While virtual machines have been popular for server consolidation and management, they are also being explored for providing mobility and management. In the management scenario, VMWare has created a library of “appliances,” pre-configured VM images for certain tasks [4]. Most of these images will be very similar, since they use the same base operating system.

In the mobility area, virtual machines are being used to transport user environments. Rather than having users work on laptops that they take with them, this approach relies on servers that keep a virtual machine image of the user’s environment, which can be moved to whatever machine the user has available. In this way, users are not tethered to any particular physical machine even if they may use the same office and home machines repeatedly in practice. This work has been explored in the Internet Suspend/Resume (ISR) Project [25].

In such an environment, we would expect three sources of similar content – common chunks between an image and the same image at a different point in time, common chunks across images of the same or similar operating systems, and common chunks within an image. While the first two sources are easy enough to understand, the last source can arise from practices such as page-level granularity for copy-on-write – multiple instances of the same program may have only differences in their globals and heap, but these differences would have required duplicating the pages where they reside. By eliminating redundant content from all of these sources, we can expect faster time to download/upload image snapshots, as well as less memory pressure on the machine serving the images. While the ISR project has an accompanying content-addressable storage system, it uses fixed-size chunks in the range of 4-16KB [14], making it likely to only find common page-aligned content, such as executables. However, program data, etc., which may be allocated in slightly different portions of memory from image to image, may be missed even when commonality is high. Using a more sophisticated CBN approach is likely to find more commonality, and to increase the benefits from redundancy elimination. We note that the ISR project intentionally chose fixed-size chunks due to concerns about start-up times using variable-sized chunks. Later in the paper, we discuss what features the CZIP format has to support these kinds of environments efficiently.

2.3 Uncacheable Web Content

Web proxy caches have been the focus of much research, and the increasing capacities of disk and physical memory now enable proxies to store and index large data volumes, and to achieve high cache hit rates limited only by the HTTP-specified cacheability of the data stream [24]. However, even when content providers specify the data should not be cached, it may be the case that the data is slowly changing, and amenable to caching. For example, most news sites are only updated a handful of times per day, and even the updates leave most of the page the same – only a few articles on a page are likely to change during an update, with the rest of the page staying the same. Even sites with user-editable content, such as Wikipedia or bulletin boards, are unlikely to completely change from edit to edit.

In these scenarios, a CBN scheme can exploit the slowly-changing nature of the data to reduce bandwidth consumption, whereas a standard HTTP proxy would be prevented from caching the content at all. The prohibitions on caching are specified by the content providers via HTTP headers returned with the request. While providers can benefit from reduced bandwidth consumption when users cache content, enabling caching for these kinds of dynamically-generated content is problematic – providers often do not know when the next modification will occur, so allowing content to be cached would result in users seeing stale versions of the page.

A CBN-aware cache can work with an HTTP proxy to share responsibilities, since each is better suited for certain portions of the workload. For example, when an HTTP proxy is allowed to cache content, it need not contact the origin server during the caching period. Only when the content expires and a client requests it does the proxy need to contact the server to re-validate. The CBN cache, in contrast, must always contact the origin server when fetching dynamic content, but it may be able to avoid actually downloading the data if it is found to have not changed. A similar approach has been proposed at the router level [26], and works without any explicit HTTP-level cooperation. Later in the paper, we will discuss why exposing HTTP-level details can help optimize these kinds of transfers, and how an explicit CZIP-aware proxy can take advantage of the extra information not available at the router level.

3 Design & Implementation

CZIP is conceptually a very simple compression format that detects and eliminates redundant chunks in the input file or stream. It exposes each chunk’s information in the header of the output file and compresses the chunk data itself using existing compression schemes such as GZIP or BZIP2. The overall format is shown in Figure 1. Be-

[4] http://www.vmware.com/vmtn/appliances/directory/

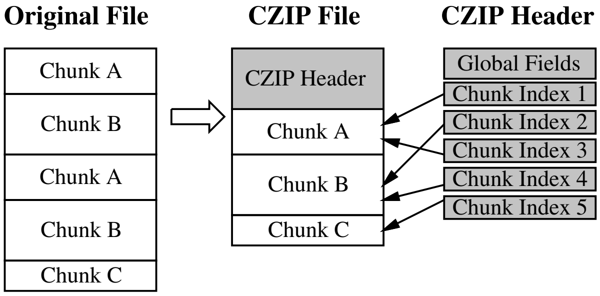


Figure 1: CZIP File format: The CZIP encoder creates the chunk index structures in the header and stores only unique chunks in the body. The stored chunk is compressed and convergently encrypted, if requested.

low, we first describe the various fields, and then provide a more detailed rationale for the design of the file format.

3.1 Header Format

The CZIP header consists of two parts – a fixed-size header that describes the overall file, and then a variable-sized header that contains the chunk information. The header is designed to be efficient and flexible, allowing applications to download what they need, and to make CZIP processing as efficient as possible. At the same time, it is designed to allow random access, even when chunk sizes are unpredictable. The specific fields of the header are:

- **Magic Cookie** – just a well-known value used to identify the CZIP format
- **Version** – which version of CZIP is used, for future expansion
- **Footer** – does this file use a footer instead of a header. If this field is set, the chunk array and real header occurs at the end of the file, not the head. The rationale is described below.
- **Sizes** – sizes of original data, total CZIP file size, and total header size.
- **Hash Format** – which hash function is being used, such as MD5, SHA-1, etc.
- **Compression Format** – which compression scheme is used, such as GZIP, BZIP2, etc.
- **Chunk Hash Size** – size of hash values in chunk array and file hashes, used to calculate positions in chunk hash array
- **Encrypt** – encryption schemes used for convergent encryption, such as DES, AES, etc. (see below)
- **Num Chunks** – number of chunks in the file
- **Header CRC** – CRC value over total header, used to detect corruption
- **File Hashes** – hash value over original and CZIP files, used to detect corruption
- **Hash Array Pointer** – the offset of the start of the chunk hash array, mostly used when the file has a footer instead of a header. Otherwise, finding the start of the hash array would be problematic.

This fixed-size header is followed by an array of per-chunk headers. These contain information about each chunk, and are designed to allow easy access.

- **Chunk Sizes** – original and compressed size of the chunk
- **Offsets** – locations of this chunk in the original and CZIP formats
- **Hash Values** – hash values for the original and processed data in this chunk

3.2 Format Rationale

The CZIP format is designed to be easily usable at several levels – including applications that need to just get general data about the file, applications that need to process the entire file, and applications that need to randomly access the file. It is also designed to be relatively easy to generate, given the constraints inherent in compressing a file using CBN. Some of the considerations involved are described below:

Planning Tools – Some tools may not care about the exact data in a CZIP file, but may be interested in knowing how much space the decompressed file requires. These tools could read just the fixed-size header to get this information.

Random Access – If a CZIP file is being used to represent a file with sparse access, programs need only read the full header to get the array of chunk headers. Since the offsets in the original file are recorded, a binary search of the chunk header array can be performed quickly, without the need to calculate offsets by adding all preceding chunk sizes.

Streamability – For most applications, having the file summary and chunk array information at the head of the file is the most useful. However, when the CZIP file is to be streamed as it is created, having all of the data in the header would require a full pass over the file, which would require buffering/creating the entire CZIP file before sending it. In this scenario, the “Footer” flag can be set, and any filled values in the file’s header are viewed as advisory in nature. When a footer is used, the arrangement of the CZIP file is as follows: advisory header, chunks, chunk hash array, file hashes, fixed-size header. In this manner, the fixed-sized header can still be found quickly when the file is retrieved from storage.

Creation Flexibility – If a variable-sized chunking scheme is used, the exact number of chunks may not be known in advance, but an approximate number can be used. If the file is being written to disk as it is created, space can be left after the chunk hash array to allow some extra space beyond the expected number of chunks. Since the per-chunk information specifies offset in the file, actual chunk data does not need to immediately follow the end of the chunk hash array. Otherwise, the creation process would have to create the header file and compress chunks separately, and then merge them. By allowing extra space between the array and the start of the chunks, the output file only has to be created once. Obviously, if footers are being used, this approach is not needed.

Encryption – A convergent encryption scheme (using DES or AES) may be applied to each chunk. Convergent encryption encrypts each chunk with its content hash as an encryption key so that the encoded chunk can be shared among authorized users [7]. The keys, which are the original content hashes, are again encrypted by a public cryptographic algorithm (such as RSA) and delivered to the authorized users. The default methods in CZIP are SHA-1 for hashing, GZIP for compression, and no encryption for the chunk content.

Integrity – The CZIP format has several mechanisms for integrity. The chunk content hash is calculated after compression and encryption are applied to the original content. This allows applications to check the integrity of a CZIP file without decompressing it.

3.3 Chunking Specifics

CZIP supports two chunking methods: fixed and variable-sized chunking. Fixed-sized chunking is simplest but if an update causes content to get shifted slightly, all previously-detected chunks after the modification point would become useless. To address the problem, the Rabin fingerprinting method is often used. Rabin's fingerprints use a random polynomial called a Rabin function with n consecutive bytes as input. A chunk boundary is determined when the function's output value modulo *average chunk size*, M, is equal to a predefined value, K (K is an integer, $0 \leq K < M$). Say M is 32 KB, and K is 17. Because the output values modulo M are well distributed over [0..2^{15}-1], the probability of the output value being 17 is close to 2^{-15}, which means the chunk boundary is formed every 32 KB on average when the Rabin function is evaluated at each byte. One advantage of Rabin's fingerprints is that even if the content is modified, that does not affect the chunking boundary beyond the modified chunk and its neighbors. Thus, most of the previously detected chunks can be reused regardless of local updates. One drawback is that the chunk size is variable, making it harder to know in advance exactly how many chunks a given piece of content produces.

By default, CZIP uses Rabin fingerprinting with a 32 KB average chunk size and GZIP compression, but these parameters can be adjusted by command-line options. For most of our workloads, 32 KB is small enough to expose most chunk commonalities and big enough to fully utilize the network socket buffers. GZIP, the default chunk compression scheme used by CZIP, finds redundant strings within a 32 KB sliding window [16], so a larger chunk size may not produce significant extra compression from GZIP.

4 Deployment Options

The overall goal of CZIP is to recognize the value of CBN by proposing an easily-handled format that provides a migration path from current compression schemes to content-based naming without requiring invasive changes or significant system redesign. While basic support for the CZIP format provides its own benefits, the design of CZIP can easily enable other benefits when used with infrastructure that is CBN-aware. In this section, we describe some deployment scenarios to maximize the benefits from CZIP. We examine what can be obtained with a CZIP-aware server, a CZIP-aware client, or both.

In this discussion, we focus on deployment using HTTP, but any appropriate protocol could be used. In particular, FTP and RSYNC servers would also be good candidates for CZIP support. Our focus on HTTP is due to its widespread adoption, and our assumption that more people know its protocol details than other protocols.

4.1 CZIP-Aware Server

A CZIP-aware server makes more efficient use of its memory when serving high-similarity content by using a CBN-based cache to reduce its working set size. In this scenario, shown in Figure 2, content providers distribute similar files encoded using CZIP, and their clients decompress the downloaded files like they would handle any other compressed formats. The CZIP files received by the client are fully self-contained. However, on the server, a chunk cache is maintained that is used across files, reducing the working set size when similar files are being served. Whenever a CZIP-file request is received, the server reads the chunk index structures from the requested file's header, and checks to see if the chunks are already loaded in the CBN cache. Any cache misses are served from the requested file, with the CBN cache also receiving the data. In this manner, the server avoids polluting main memory with redundant data, and

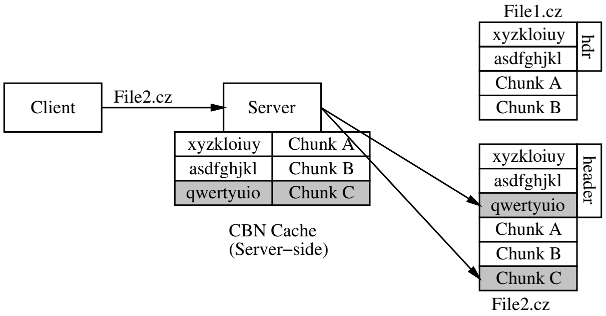


Figure 2: A CZIP-aware server needs to read only the chunk index structures and then the missing chunk C to serve File2.cz to the client. Chunk A and B are already in the server's CBN cache from a previous fetch of File1.cz.

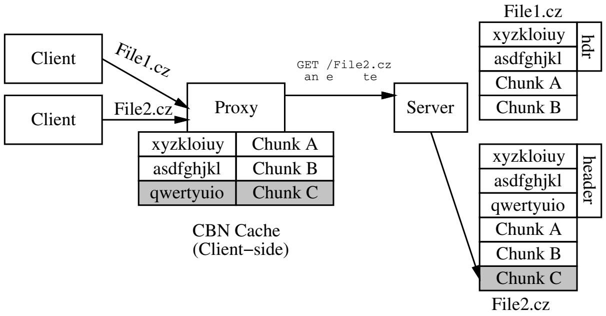


Figure 3: A CZIP-aware proxy first fetches the header of File2.cz, and downloads only chunk C. The chunk request includes the chunk's byte-range and its content hash. We assume that another client previously downloaded File1.cz.

the server's effective working set size is the union of all the unique chunks in the CZIP'ed files being served.

Since the data stream received from the server is just another file, no changes are required on any intermediary devices or at the client in order to download the CZIP file. Obviously, the client must be able to uncompress the CZIP format, which can be achieved via stand-alone programs, as a browser plug-in, as a helper application, or even with integrated browser support, as is done with GZIP'ed objects.

If this support is not feasible at the client side, the server could un-CZIP the data as it is being sent to the client, or even re-encode it using GZIP or ZLIB/deflate depending on what the client specifies in the HTTP "Accept-Encoding" header. A clever system may be able to take GZIP-encoded chunks from the CZIP file and serve them to GZIP-capable clients without a full decompression step, but this approach requires knowing some low-level details of the ZLIB stream format, and is beyond the scope of this paper. In any case, it is easy to see that a CZIP-aware server could still obtain memory footprint benefits even with a completely unmodified client.

4.2 CZIP-Aware Client

If the server is only a standard Web server with no special support for CZIP, a CZIP-aware client can still independently exploit CZIP-encoded files using only the standard HTTP protocol. The advantage for the client is lower bandwidth consumption and faster download times. In this scenario, one or more clients maintain a CBN cache that stores recently-downloaded chunks. When clients want to download a CZIP-encoded file, they ask for only the CZIP headers using the HTTP byte-range support that has been present in Apache and IIS since 1996. The clients ask for at least as many bytes as the fixed-size header, and if the response does not contain the full chunk hash array, another request can be sent to get the rest of the variable-sized header. Note that in the unlikely event that the CZIP files are stored with footers, in streaming order, the byte-range support can also ask for bytes at the end of the file.

Once it has the header, the client knows the chunk hashes, so it can try to fetch the chunks from its local CBN cache. For any chunks it does not have, the chunk hash array also contains the byte positions of the chunks within the CZIP file, so the client can use the byte-range support to just ask for specific portions of the file containing the chunks it needs. These chunks are also inserted into the CBN cache.

Alternatively, this level of support could be added to a client-side proxy server, as shown in Figure 3 so that clients themselves do not have to be aware of the CZIP format. If the connection between the client and the proxy is faster than the speed of the wide-area network, the download will still be faster than if the client had contacted the server directly.

4.3 CZIP-Awareness at Both Endpoints

The greatest benefit using CZIP arises when both endpoints are CZIP-aware and utilize CBN caching. In this scenario, the client's first request to the server retrieves the full CZIP header for the file. After consulting with its local CBN cache to determine which chunks it does not have, the client contacts the server to request the chunks by their chunk hash information instead of requesting range requests of the CZIP-encoded file. In this manner, both the client and server are only dealing with chunks rather than files when serving the body of the request, reducing both bandwidth consumption and the server's working set. With persistent connections and request pipelining, any gaps between serving individual chunks can be minimized.

This scenario requires more infrastructural change than the two previous scenarios, but even these changes could be incorporated into proxy servers. Proxy servers are often deployed as "server accelerators" or "reverse

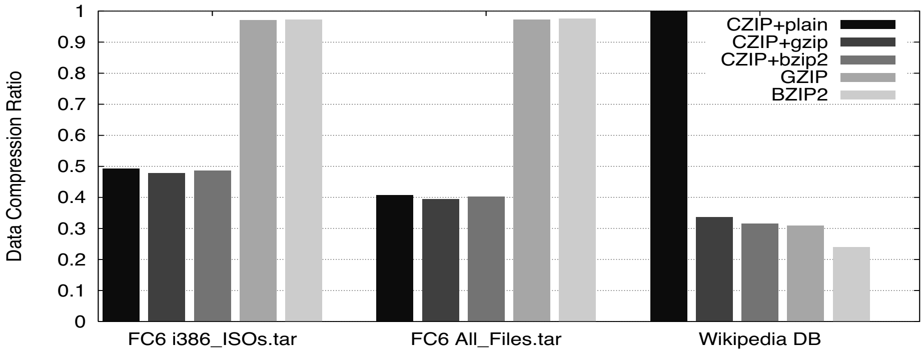


Figure 4: Data Compression Ratio: CZIP vs. GZIP and BZIP2

	Orig Size	CZIP						GZIP		BZIP2	
		Size			Time						
		plain	gzip	bzip2	plain	gzip	bzip2	Size	Time	Size	Time
FC6 (i386)	6.65	3.27	3.18	3.23	428	765	2004	6.46	846	6.47	3964
Wikipedia	7.94	7.94	2.67	2.50	656	1419	3079	2.45	976	1.90	3151
FC6 (all)	49.73	20.27	19.63	19.86	323	5194	12767	48.34	6424	48.53	29004

Table 1: Compression Performance. All sizes in GB, and all times in seconds.

proxies," where the content provider will have incoming requests pass through a proxy server before reaching the actual Web server. This approach is used to offload requests from the Web server, since the proxy may be more efficient at serving static content. In this scenario, CZIP support merely needs to be added to the proxy server, and when a CZIP-enabled client-side proxy realizes it is communicating with a CZIP-enabled server-side proxy, it can use the CZIP-aware protocols for transfers. In this manner, the changes are more localized than requiring modifications to all Web servers.

5 Compressibility Experiments

In this section, we perform a number of experiments to demonstrate the effectiveness and performance of CZIP on a range of data types, focused on the scenarios we described in Section 2. CZIP reduces data volume by first finding and eliminating redundant data, and then passing the remaining data through existing compression schemes. Not surprisingly, CZIP is most useful where a high level of chunk commonality is expected, but its compression performance does not degrade much even when there is little commonality because each chunk is individually compressed. The experiments described below compare CZIP's compression performance with GZIP and BZIP2 in terms of compression ratio and speed.

5.1 Linux Distributions

Our first experiment examines CZIP in an environment where we can expect significant data commonality, serving the Fedora Core Linux distribution [17] across its three CPU architectures (i386, x86_84 and ppc) and two media formats, DVD and CD ISOs. The byte count for just the 32-bit i386 architecture is 6.7 GB, while all Fedora Core 6 (FC6) ISOs together is about 22 GB. The full FC6 mirror, including all source RPMs, is 49.7 GB.

We prepare two sets of files, just the i386 DVD/CD ISOs, and all FC6 mirrored files, including the source RPMs. We *tar* each set into a file, and apply CZIP, GZIP and BZIP2 each on a machine with a 2.8 GHz Pentium D processor and 2 GB of memory. Each file is compressed with just CZIP alone (no chunk compression), CZIP used with GZIP or BZIP2 as a per-chunk compressor, and then GZIP and BZIP2 used alone as standalone compressors. In all cases, whether running standalone or in conjunction with CZIP, the GZIP and BZIP2 compressors use their default compression parameters.

Figure 4 shows the data compression ratios, which are calculated as compressed size/original size. More details are shown in Table 1. The results show that all of the CZIP variants, even with no chunk-level compression, yield files less than half the size of GZIP and BZIP2, which show virtually no compression. The CZIP results are not surprising, because the DVD ISO contains all the data of the CD ISOs, and the ISOs again contain all the RPM contents. The CZIP'ed-ISO file (3.17 GB) is actu-

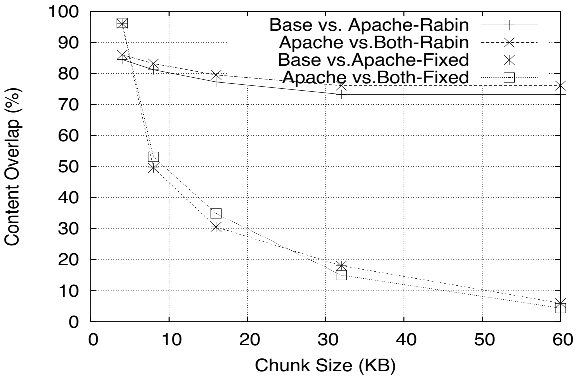


Figure 5: Content overlap over similar Xen Linux VMs

ally smaller than just one DVD ISO (3.28 GB) because of the compression of each chunk's data. On the other hand, GZIP and BZIP2 do not find the chunk-level commonality across the ISOs, nor can they compress the ISOs much further because most of their contents are already compressed. By the actual byte counts, CZIP saves 3.3 GB and 28.7 GB more disk space for each FC file than the other schemes. The 32KB chunk size performs well enough – dropping to 4KB using CZIP alone only compresses the file to 19.07 GB from 20.27 GB. The smaller chunk size does provide a small speed boost, reducing compression time to 2583 seconds from 3231, presumably due to processor-cache effects.

5.2 Wikipedia

Our next test involves a large set of human-generated content, an offline version of the Wikipedia [32] database containing all of its pages, which is intended for proxying in regions where bandwidth is limited. We download the latest version of the database file (produced on 11/30/2006), and tested each compression scheme on it.

The data compression ratio in Figure 4 shows that CZIP with no compression performs the worst, which is in contrast to the Fedora Core cases. The Wikipedia database file shows almost no chunk-level commonality ($< 0.001\%$) using the default average chunk size of 32 KB, while it is easily compressed with other methods. This file presents a worst case scenario for CZIP, since the chunking scheme presents smaller pieces of data for the individual compressors. The difference for CZIP+GZIP versus GZIP alone is an additional 2.8%, since GZIP uses a relatively small (32KB) window. BZIP2, however, uses a much larger compression window (900KB), and is able to gain an additional 7.6% over the combination of CZIP+BZIP2, since CZIP produces chunks smaller than BZIP2's window.

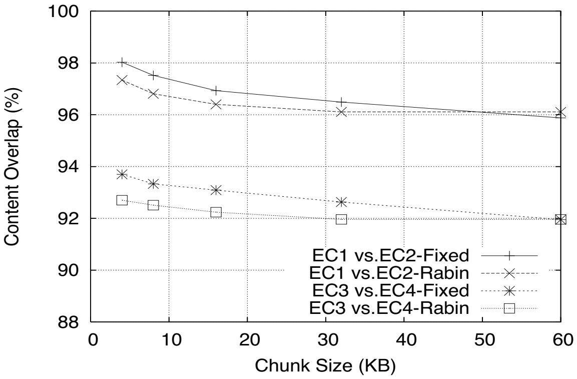


Figure 6: Content overlap over engineering computing Xen Linux VMs after three weeks of use

5.3 Virtual Machine Images

Handling multiple virtual machine (VM) images for the same operating system is another area where much cross-file commonality would be expected. We investigate this scenario using two sets of images – one that is server oriented, and another reflecting client machines.

The server test creates three Xen [4]-based Linux VM images: "Base," a minimum-functionality (with no redundant daemons) Fedora Core 4 image, "Apache," which adds the Apache Web server to the base image, and "Both," which adds the MySQL database server to Apache. Each image is created on a 2 GB file-based disk image, but the real content sizes, measured by unique chunks, are 734.8, 782.3 and 790.8 MB, respectively.

In Figure 5, we compare the content overlap ratio between Base and Apache, and between Apache and Both over different chunk sizes and chunking methods. We see more content overlapping with smaller chunks, because the granularity of comparison gets smaller. The performance of fixed-size chunking degrades significantly after 4 KB, the hardware page size. However, variable chunking degrades much more slowly and flattens after 16 KB. Using Rabin's fingerprinting method, we detect 73-86% of redundancy regardless of chunk sizes.

For the client test, we create five identical Xen Linux VMs (EC1-EC5) configured with a standard engineering computing (EC) for a large technology company. For three weeks, five different engineers extensively used the VMs for various tasks in the hardware design process. The image size is 4 GB each, but the real content per VM is 2.2 GB. In Figure 6, we show the overlap between two pairs of images, EC1 and EC2, and EC3 and EC4 (comparison with other pairs is similar). CZIP finds more than 90% redundancy between all pairs of VMs, and sometimes as much as 98%. Interestingly, fixed-size chunking degrades much less in this test, and even slightly outperforms Rabin fingerprinting. The reason is because most

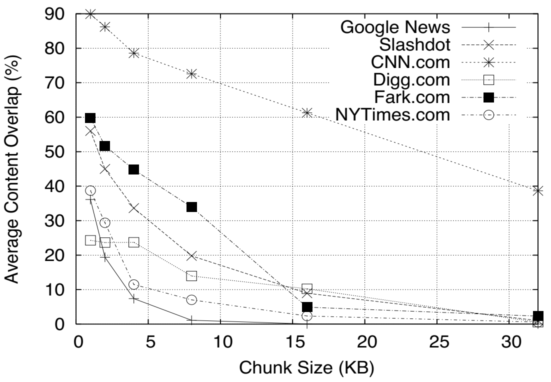


Figure 7: Average content overlap for uncacheable Web pages

of the content did not change over the three weeks, and if the layout of the content is aligned with multiples of the fixed chunk size, fixed-sized chunking can find more commonality.

5.4 Dynamic Web Pages

Our final compressibility test examines the commonality between multiple snapshots of dynamic Web sites over time. We download the front pages of Google News, CNN, Slashdot, Digg.com, Fark.com and the New York Times (NYT) every 10 minutes for 18 days. All of these sites mark their front pages uncacheable with "no-cache," "no-store," or "private" in the "Cache-Control" response header, which would not only render shared HTTP proxy caches useless, but would also prevent browser caching in most of their cases. The data volumes for the HTML alone range from 120-360 MB for the entire period.

We run CZIP on each snapshot, and for each site, we compare the chunk overlaps on every pair of snapshots taken 10 minutes apart. Figure 7 shows the average content overlap during the 18-day period for each site, using CZIP runs with varying chunk sizes. As in the previous section, we see that the commonality decreases as the chunk size increases, but we see 24% to 90% average redundancy for 1-KB chunks. The particular pattern is also interesting – Google News shows the worst savings at the 4KB chunk size, since most of the blurbs are small and their positions are updated frequently. The entertainment site Fark.com uses much smaller blurbs and updates roughly 50 times per day, but shows high commonality, due to the blurbs getting added and removed from the front page in FIFO order. As such, the Rabin fingerprinting approach can still work with the shifted content. The per-site savings in bytes transferred using CZIP-aware systems is shown in Figure 8.

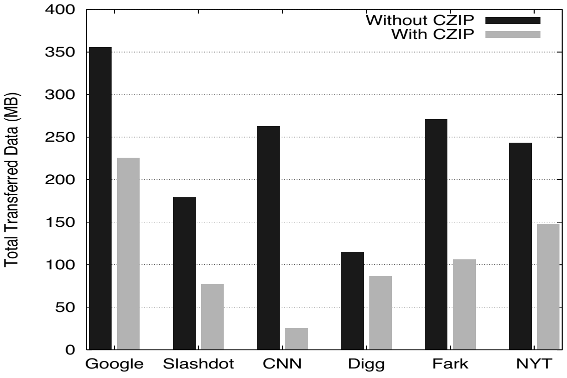


Figure 8: Potential transferred data savings using CZIP with 1-KB chunks

	CZIP			GZIP	BZIP2
	plain	gzip	bzip2		
FC6 (i386)	130	169	1299	229	2021
Wikipedia	146	200	786	218	1167
FC6 (all)	1465	1865	10009	1708	14971

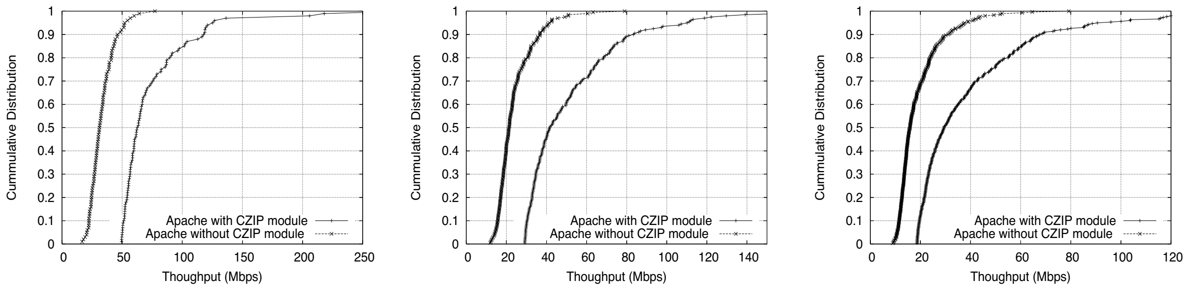
Table 2: Decompression Performance. All times in seconds.

5.5 Overheads

CZIP's compression and decompression speed are mostly comparable to GZIP and much better than BZIP2. Tables 1 and 2 show the times taken for compressing and decompressing FC6 and Wikipedia DB files. For the FC6 files, CZIP finishes 91 seconds (10.6%) and 1229 seconds (19.1%) earlier than GZIP in compression because it can avoid processing redundant chunks. But CZIP is 443 seconds (45.3%) slower than GZIP for the Wikipedia DB file. This is due to CZIP's chunking overhead and redundant file access for temporarily saving intermediate chunks before writing the header. Decompression is generally much faster than compression, and its performance is usually bounded by the disk write speed. CZIP's decompression speed is comparable to that of GZIP. In comparison with BZIP2, CZIP is 2.2 to 5.6 times faster in compression and 5.8 to 12 times faster in decompression, mostly due to BZIP2's CPU-heavy reconstruction process during decompression.

6 Performance Evaluation

In this section, we evaluate the performance benefits of CZIP-aware systems in two contexts – a server-side CBN cache, and CZIP integration with a content distribution network (CDN). The CBN cache is implemented as a module on the Apache Web Server, and the CDN support is integrated into the CoBlitz large-file CDN [15].



(a) 100 clients

(b) 200 clients

(c) 300 clients

Figure 9: Client throughput distribution when downloading large ISO image files

6.1 Server-Side CBN Cache

Our server-side CBN cache is evaluated for one of the deployment scenarios that we described earlier in Section 2 – a software distribution mirror handling a data set larger than its physical memory. In this scenario, the server can easily experience thrashing, and have its throughput bottlenecked by disk access performance. A server-side CBN cache can help avoid unnecessary disk reads and reduce the effective memory footprint of the server.

Our implementation consists of an Apache module which handles a CZIP file request by parsing the file's header and fetching the chunks from a CBN cache server. The CBN cache server is a user-level file chunk server on the same machine that caches chunks indexed by CBN. The module sends a chunk request with a file path, a byte-range and a chunk content hash. The CBN cache server finds the chunk in its cache or reads it from the file system on cache misses. It can be configured to recheck the chunk content hash for possible attacks or corruption. Because the CBN cache server is a separate process, any CZIP-aware servers on the same machine can benefit from the cache as well.

We use a server machine with a 2.8 GHz Pentium D processor, 2 GB memory, and two Gigabit Ethernet network interface cards (NICs). We compare Apache 1.3.37 with and without the CZIP module on a data set consisting of a 1.5 GB file extracted from the Fedora Core 6 DVD ISO and three 0.5 GB files whose contents overlap with the 1.5 GB file. This simulates the typical Linux mirror setup with one DVD and many CD ISOs. To include aliasing effects, we duplicate the set and place one copy in a different directory, raising the total content size 6 GB.

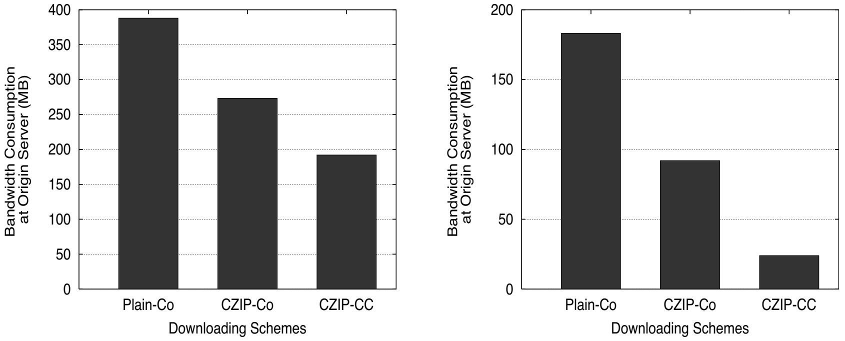
Our client workload is generated by six machines with one Gigabit Ethernet NIC each, split across two LAN switches. Each machine generates multiple simultaneous requests to the server, and we simulate 100-300 clients total. A new simulated client arrives on average every 3 seconds, up to the per-experiment client limit.

# of clients	Avg	Min	Median	90%
100	33.45	16.72	30.12	46.00
100 (w/CBN)	75.69	49.52	62.48	117.76
200	24.11	11.92	21.28	36.72
200 (w/CBN)	53.23	29.20	41.68	83.36
300	19.14	8.96	15.84	32.04
300 (w/CBN)	40.30	18.64	29.44	67.60

Table 3: Per-client throughputs (in Mbps) for serving a large data set with lots of commonality. We show the average, minimum, median, and 90th percentiles.

Figure 9 shows the client throughput CDF for serving 100, 200, and 300 simultaneous clients. The CZIP-enabled Apache outperforms the standard Apache by a factor of 2.11 to 2.26 (mean), and 1.84 to 2.07 (median). The means are higher than the medians because the CBN-cache case has a long tail – later requests do not overlap with many of the previous requests, so they compete less for the server's CPU cycles and network bandwidth. The CBN-cache case performs much better with physical memory cache hits than the non-cache case, whose throughput is bounded by the disk read bottleneck. This is observable in all three graphs by noticing that the horizontal gap between the two lines widens beyond the 50th percentile. Another interesting observation is that the worst-case throughput with the CBN cache beats 65% to 91% of non-cache throughputs.

The development effort for the components mentioned above were relatively modest. The Apache module consists of 700 lines of C, of which 235 are semicolon-containing. The CBN cache server is larger, with 613 semicolon lines out of 2061 total. However, the total development time for them was one week combined, and it was performed by a first-year graduate student. We believe that future development can leverage the effort here, especially of the stand-alone CBN cache.



(a) Downloading 100MB file

(b) Downloading 50MB file

Figure 10: Bandwidth Consumption at Origin Server: Plain-Co and CZIP-Co both use regular CoBlitz but Plain-Co downloads original file while CZIP-Co downloads CZIP'ed file. CZIP-CC uses C-CoBlitz to download the CZIP'ed file.

6.2 CBN-Aware Content Distribution

To build a CBN-Aware Content Distribution Network (CDN), we create C-CoBlitz by integrating CZIP support into the CoBlitz CDN, a scalable large-file HTTP CDN running on PlanetLab [15]. CoBlitz has been in production for over two years, and serves roughly 1 TB per day, with peaks as high as 5 TB/day and sustained bandwidth rates in excess of 1.2 Gbps.

CoBlitz already tries to reduce origin server bandwidth consumption, even when hundreds of edge proxies are serving multi-GB files. Rather than fetching whole files, CoBlitz nodes request and cache fixed-sized chunks of a file from the origin server using HTTP byte-range support. The nodes then cooperate with each other to reassemble the chunks in order and seamlessly serve the file to unmodified Web clients. Since CoBlitz does not examine the content of chunks, it does not identify redundant content. By integrating CZIP into CoBlitz, we can avoid fetching and storing redundant content, reducing origin server bandwidth consumption even further.

CBN cache integration with CoBlitz transparently provides the benefits described in Section 4.2 without any modification of the client or the server. CoBlitz names its fixed-size chunks using the original URL name and byte-range information. To add CZIP support, we add the content hash and chunk size to this name when handling CZIP-format files. The CBN cache integration requires only about 200 lines of new code.

The chunk naming scheme extends to the underlying CoDeeN content distribution network [31] on which CoBlitz is built. The benefit of this approach is that if a given chunk is not found at a CoDeeN node, the request is served from the peer CoDeeN node responsible for that given URL. The mapping of URLs to CoDeeN nodes is performed using the HRW consistent hashing scheme [27], so if a chunk is needed at several nodes, it will likely be fetched from a peer CoDeeN node instead of from the origin server.

To test how much bandwidth is saved using C-CoBlitz, we have 100+ U.S. PlanetLab nodes simultaneously download the first 100 MB and 50 MB of the Fedora Core 5 DVD ISO from a server at Princeton. For the origin server, we use lighttpd [9] on a 2.8 GHz Pentium D machine with 2 GB of memory. We download the 100 MB content first and then the 50MB content, for both the original and CZIP'ed versions. Using CZIP reduces the 100 MB and 50 MB files to 68.1 MB and 28.9 MB, respectively.

Figure 10 compares the bandwidth consumption at the origin server for the different schemes. Because the number of clients varies from 93 to 106 depending on the time of the tests, we normalize the bandwidth consumption for the 100-client case. Distributing the uncompressed 100 MB content to 100 nodes via regular CoBlitz consumes 388 MB of bandwidth at the origin server (3.8 copies of original content) while the CZIP'ed content needs 273 MB, a 29.6% reduction from serving the original content. This saving comes directly from the content size reduction by CZIP compression, and the same trend is seen in the case of the 50 MB file as well, which is reduced by 49.7%. Serving the CZIP'ed 50 MB content through C-CoBlitz shows the largest bandwidth reduction because most chunks were already cached while downloading the CZIP'ed 100 MB content. To serve the CZIP'ed file to 100 clients (2.9 GB of content or 5 GB uncompressed content), C-CoBlitz requires only 24 MB of origin server bandwidth, which is just 0.8% of the total size. Regular CoBlitz requires 3.83 times (92 MB) more bandwidth.

The other interesting comparison in Figure 10 is the bandwidth consumption drop from CZIP-CO (273MB) to CZIP-CC (191MB). The difference here is that CZIP-CO is just the CZIP'ed file being served over regular

CoBlitz, while CZIP-CC uses C-CoBlitz. Since this is the first transfer of the file to all 100 clients, these results should be the same, but C-CoBlitz still shows a 29.7% drop in bandwidth consumption. The C-CoBlitz system appears to be reducing the bandwidth burstiness and network congestion compared to regular CoBlitz, triggering fewer retries and causing each fetched chunk to be served to more peers. As a result, fewer nodes are fetching each chunk from the origin server, reducing the bandwidth consumption further.

7 Related Work

The idea of exploiting chunk-level commonality has been widely applied in many systems, but these techniques have not been easily separable from the underlying system. The earliest work of which we are aware is the system proposed by Spring *et al.* to eliminate the packet-level redundancy by recognizing identical portions in IP packets [26]. They assume synchronized caches at both endpoint routers and detect identical chunks by finding anchors [10] in the packet and expanding the region of same content from that point. The packet is encoded with tokens representing the repeated strings in the cache. Because the approach is independent of application-level protocols, it can offset application-level caching, such as removing the redundancy among the uncacheable HTTP responses. In CZIP, we focus on separating the CBN techniques from the underlying system, allowing us to provide similar benefits using only user-level CBN caches.

Another system that appeared shortly thereafter was LBFS [13], and was the first system taking advantage of Rabin's fingerprints to reduce bandwidth consumption in a distributed file system. LBFS finds about 20% redundancy in a 384 MB set of file data. Similar ideas have since been applied in numerous other file or storage back-up systems such as Farsite [1], Pastiche [5], Venti [19], CASPER [29], and Shark [2]. File systems are an attractive place to implement CBN caching because file access patterns often reveal significant redundancy [23, 30]. However, the CBN support has been built into these systems, making it more difficult to select when it is appropriate, or to use it outside of the range of tasks handled by the system.

Some systems have been built to reduce duplicate data specifically in the context of the Web. Value-based Web Caching (VBWC) [21] reduces redundant chunk transfer on the Web by coordinating the browser's CBN cache and a client-side parent proxy. Its operation is similar to Spring's work [26] except it is specific to HTTP. They also describe a synchronization mechanism between these two caches. Duplicate Transfer Detection (DTD) [12] adds a message digest in the HTTP response header (without a message body), and allows the client to search its CBN cache for the message body. Only in the case of a cache miss does the client ask for the message body – this is called the "pure-proceed" model. It is similar to our client-only caching scenario in Section 4.2, but the content hash is based on the whole file (except for byte-range queries), which may make finding redundancy across parts of files difficult. It does not require cache synchronization as in VBWC, but at the cost of an extra RTT delay for every cache miss. CZIP-based requests can utilize partial content overlap but do not require an extra RTT even for client-side caching only.

Delta encoding has also been proposed for Web pages that partially change [11]. In this general approach, clients can specify what version of a page they have cached when asking the server if the page has been updated. The server can then send just the updated portions rather than the whole page. While this approach is well suited for static content that has a small set of easily-identifiable versions, it is much harder to adapt it for dynamic content, which can not be easily named or tagged. The extra overhead on the server created a higher barrier to adoption for this approach. CZIP-based schemes would not have to remember specific states of Web pages, since the chunks can come from any page and be used in any other page. As a result, while a CZIP-based approach may not produce deltas as small as other approaches, it can find commonality across files with relatively little server state.

More recently, a transparent transfer service architecture based on a CBN chunk cache has been proposed [28]. It asks for a chunk hash array exchange before actual delivery, which is similar to the case in Section 4.3. Our belief is that by making CZIP a standard format, the benefits of this kind of compression can be achieved end-to-end, instead of just by an enhanced system in the middle. This approach would also let the endpoints select when to use it, avoiding the overhead when serving content with little data commonality, like unrelated compressed files [16].

Similarity-Enhanced Transfer (SET) [18] exploits chunk-level similarity in downloading related files. It finds relatively high chunk-level similarity in popular music and video files. Much of the similarity comes from files with the same content but with slightly different metadata information in the header. In order to utilize the similarity, SET proposes to maintain the CBN information and chunk location in a DHT-like infrastructure so that a SET-based downloader can easily find the chunk location with a constant number of lookups. CZIP-aware CoBlitz provides similar benefits without maintaining separate mapping information since the chunk data itself is cached with its metadata at the same location. This approach avoids any possible staleness concerns.

8 Conclusion

Although content-based naming (CBN) has proven itself useful in a variety of systems, there has been no general-purpose tool to enable its use in a variety of systems that were not designed with in mind from the start. In this paper, we have shown a new compression format, CZIP, which can be used to efficiently support CBN with reasonable overheads in processing power and space consumption. We have demonstrated that CZIP can identify and eliminate redundant data across a range of useful scenarios, without being tied to any particular system.

CZIP provides a flexible combination of deployment paths, not only providing benefits by itself, but also by providing more benefits if CZIP-awareness is added to the client, server, or both endpoints. We have described how these deployment options can be implemented without invasive changes to the endpoints. To support these claims, we have implemented CZIP awareness in the Apache Web Server, and have shown that integration with a CBN cache reduces its memory footprint and dramatically improves client throughput. We have also added CZIP support to a deployed content distribution network, and have shown that it reduces origin server bandwidth consumption significantly.

We believe that this combination of flexibility, ease of integration, and performance/consumption benefits will make CZIP an attractive tool for those wishing to support content-based naming or develop new systems using this technique.

Acknowledgments

We thank Jeff Sedayao and Claris Castillo for providing the realistic VM images. We also want to thank Sanjay Rungta and Michael Kozuch for useful discussion about redundancy suppression in enterprise network traffic as well as in virtual machine migration. This work is supported in part by NSF grants CNS-0615237 and CNS-0519829.

References

[1] 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 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[2] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, May 2005.

[3] H. Bahn, H. Lee, S. Noh, S. Min, and K. Koh. Replica-aware caching for web proxies. *Computer Communications*, 25(3):183–188, 2002.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[5] L. Cox and B. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[6] Data Domain.
`http://www.datadomain.com/`.

[7] J. Douceur, A. Adya, W. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[8] EMC Corporation.
`http://www.emc.com/`.

[9] J. Kneschke. lighttpd.
`http://www.lighttpd.net`.

[10] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

[11] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. Delta encoding in HTTP. RFC 3229, January 2002.

[12] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, 2004.

[13] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'01)*, pages 174–187, 2001.

[14] P. Nath, M. A. Kozuch, D. R. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of USENIX Annual Technical Conference*, 2006.

[15] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '06)*, 2006.

[16] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of USENIX Annual Technical Conference (USENIX '04)*, 2004.

[17] F. Project.
`http://fedora.redhat.com/`.

[18] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th USENIX/ACM Symposium on Networked Systems Design and Implementation(NSDI'07)*, 2007.

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

[20] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, 1981.

[21] S. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *Proceedings of the Twelfth International World Wide Web Conference*, May 2003.

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

[23] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of USENIX Annual Technical Conference*, 2000.

[24] A. Rousskov, M. Weaver, and D. Wessels. The fourth cache-off. `http://www.measurement-factory.com/results/`.

[25] M. Satyanarayanan, M. Kozuch, C. J. Helfrich, and D. R. O'Hallaron. Towards seamless mobility on pervasive hardware. *Pervasive and Mobile Computing*, 1(2):157–189, July 2005.

[26] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, 2000.

[27] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, Feb. 1998.

[28] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for internet data transfer. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.

[29] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of USENIX Annual Technical Conference(USENIX '03)*, 2003.

[30] W. Vogels. File system usage in windows NT 4.0. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 1999.

[31] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[32] Wikipedia. `http://www.wikipedia.org/`.