Hierarchical Substring Caching for Efficient Content Distribution to Low-Bandwidth Clients*

Utku Irmak
CIS Department
Polytechnic University
Brooklyn, NY 11201
uirmak@cis.poly.edu

Torsten Suel
CIS Department
Polytechnic University
Brooklyn, NY 11201
suel@poly.edu

ABSTRACT

While overall bandwidth in the internet has grown rapidly over the last few years, and an increasing number of clients enjoy broadband connectivity, many others still access the internet over much slower dialup or wireless links. To address this issue, a number of techniques for optimized delivery of web and multimedia content over slow links have been proposed, including protocol optimizations, caching, compression, and multimedia transcoding, and several large ISPs have recently begun to widely promote dialup acceleration services based on such techniques. A recent paper by Rhea, Liang, and Brewer proposed an elegant technique called *value-based caching* that caches substrings of files, rather than entire files, and thus avoids repeated transmission of substrings common to several pages or page versions.

We propose and study a hierarchical substring caching technique that provides significant savings over this basic approach. We describe several additional techniques for minimizing overheads and perform an evaluation on a large set of real web access traces that we collected. In the second part of our work, we compare our approach to a widely studied alternative approach based on delta compression, and show how to integrate the two for best overall performance. The studied techniques are typically employed in a client-proxy environment, with each proxy serving a large number of clients, and an important aspect is how to conserve resources on the proxy while exploiting the significant memory and CPU power available on current clients.

Categories and Subject Descriptors:
C.2.2 Computer-Communications Networks: Network Protocols: Applications C.2.4 Computer-Communications Networks: Distributed Systems: Client/server

General Terms:
Algorithms, Performance, Design, Experimentation

Keywords:
web caching, HTTP, web proxies, compression, WWW

*Work supported by NSF CAREER Award NSF CCR-0093400, NSF ITR Award IDM-0205647, and the Wireless Internet Center for Advanced Technology (WICAT) at Polytechnic University.

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

1. INTRODUCTION

The last few years have seen a significant increase in the available bandwidth on the internet, and many clients have upgraded to faster cable and DSL connections to the internet. However, there are still large numbers of clients that have only very slow connections, including wired dialup connections and also many cellular connections for new mobile clients. Thus, the gap between fast and slow clients may have actually increased, and there is a need for techniques that can bridge this gap in a transparent and (as much as possible) application-independent manner.

Caching and compression are maybe the most widely used techniques for dealing with slow networks, but both have clear limitations at least in their basic forms. Caching avoids the repeated transmission of identical objects, but only works if objects are completely unchanged and fails if they are even slightly modified. However, there are many scenarios where content is only gradually modified over time or there are significant similarities between different objects. When applied to individual objects, compression only exploits redundancies within that object. Compression techniques on streams, such as LZW type compression in modems, typically only exploit redundancies due to very recently seen data, and are not optimized for very long histories of transmissions.

A significant amount of recent work has focused on new techniques, many of them closely related to caching and compression, that can overcome these shortcomings and thus further reduce communication delays. For example, string decomposition techniques based on Karp-Rabin fingerprints have been used to remove redundant network transmissions in applications such as general network traffic [32], distributed file systems [25, 10], and web access [29]. File and data synchronization techniques, such as the `rsync` utility [38] or tools for handheld synchronization [2], exploit similarities between data to be transmitted and objects already stored at the recipient without the sender having the know those objects. Optimized delta compression techniques [33, 3, 20, 22, 35] can achieve significant additional savings by compressing data with respect to similar objects known to both sender and receiver. In addition to academic work, there are a large number of products and startup companies that heavily depend on these types of techniques.

Thus, there continues to be great interest in techniques for hiding the effects of slow communication links on application performance and user experience. While the techniques can be deployed in an end-to-end manner, they are most often used in a client-proxy architecture, where a proxy on

the high-bandwidth side of the internet communicates with one or often many clients on the other side of a slow link using an optimized protocol based on some of the above techniques. The most widely known example are the systems for accelerating dialup web access that are currently being offered by AOL (AOL 9.0 Velocity), NetZero (NetZero HiSpeed), EarthLink (EarthLink Accelerator), and Sprint PCS (Vision), among others.

In this paper, we mainly focus on the case of web access, for which we collected our data sets. However, our main contribution, an optimized hierarchical approach to substring caching, is applicable to other scenarios as well. The approach introduces a natural multi-resolution view of caching, where information about very recently accessed items is maintained in detail, while information about older items is only kept in very coarse-grained form. One important consideration in client-proxy systems is the potential bottleneck at the proxy. Since the cost of the service depends on the number of clients that can be supported by a single proxy, and many clients have significant spare memory and CPU capacity, it is desirable to use asymmetric techniques that minimize proxy load by moving as much work as possible to the client. We follow this approach, which fits well with content delivery scenarios such as web access where data primarily flows from proxy to client.

The paper is organized as follows. In the next section we give some background on proxy-based web acceleration and describe the *Value-Based Caching* technique of Rhea, Liang, and Brewer [29]. Section 3 summarizes the contribution of this paper. In Section 4, we describe our *Hierarchical Substring Caching* approach, compare it to *Value-Based Caching*, and propose and evaluate some additional optimizations. Section 5 compares the approach to another common approach based on delta compression, and gives a hybrid that outperforms both. Finally, Section 6 contains a brief discussion of related work, and Section 7 lists some open problems.

2. PRELIMINARIES

With the general perspective provided in the introduction, we now review previous work on optimizing Web access over slow links. After a general overview, we describe the *value based caching* technique in [29] and review previous work on delta compression of web content.

2.1 Web Access Over Slow Links

The problem of optimizing Web access over slow links has been studied almost since the beginning of the Web, with early work appearing in [5, 15, 14, 13]. Since then, a number of techniques have been studied in the literature and deployed in various commercial offerings. Before going into technical details, we note that there are two other issues that are often studied jointly with low-bandwidth web access in the literature: (1) The adaptation of content to small-display devices such as handhelds and cell phones, related to bandwidth issues in that smaller screens can only display low-resolution images and limited amounts of text (which enables additional opportunities for compression), and (2) the problem of serving diverse client populations with varying bandwidth, display, or computational constraints (i.e., broadband vs. cellular access and handheld vs. fast PC). In the following, we do not focus on these issues, and the reader may assume the common scenario of reasonably powerful clients with full displays connected via links with slow but fairly predictable bandwidth, such as is typical for dialup ISPs.

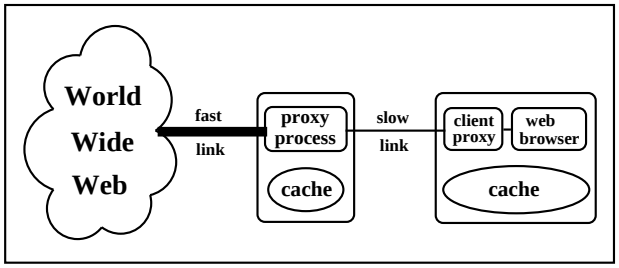

Figure 2.1: Client-Proxy Architecture for Web Access

A standard architecture for our scenario is the client-proxy architecture shown in Figure 2.1 where a proxy connected at high bandwidth to the internet is connected to a client via a slow link such as a modem connection. The proxy and client then communicate over the slow link using an optimized and usually proprietary protocol. In order to make the process transparent to applications on the client side, the client may also run a client-side proxy process that interacts with the proxy, or this functionality can be integrated into the application (web browser) on the client. Such proxy systems typically employ several approaches to optimize communication, in particular:

- **Protocol optimizations:** changes to HTTP or other protocols can avoid repeated opening of connections or eliminate roundtrips over the slow link.
- **Compression:** standard compression schemes such as *gzip* or optimized compressors for HTML, email, or other special file formats (e.g., MS Office formats) are used to decrease data size.
- **Transcoding:** images and video files are *transcoded* into smaller files of lower quality. In the best case, images are substantially reduced with no visible impact, though in practice quality is often affected.
- **Caching and Differential Compression:** caching is a standard technique in browsers and web proxies with known but limited benefit. Significant additional benefits can be obtained by exploiting similarities between different version of the same page or similar pages, as discussed in the introduction, and this has been a focus of a lot of research.

In our work, we focus on the last approach which we believe to have potential for significant improvements. We note that a complete system needs to include several or all of these approaches to be competitive with the best commercial systems. Protocol optimizations and standard compression are well understood. Specialized compression and transcoding is highly dependent on the data type, and significant amounts of industrial effort have been put into transcoding and into compression of common formats such as MS Office files. The techniques we focus on here are primarily applicable to non-multimedia data types, although some limited benefits can also be obtained for image data as we will show.

2.2 Fingerprints and Value-Based Caching

We now describe the recently proposed *value-based caching* technique of [29] in detail. The main idea is very simple. First, it is sometimes beneficial to identify an object not by

its name, but by a hash of its content. In the case of the web, there is a significant amount of aliasing, i.e., identical pages are returned for different URLs. This may be due to replicated servers or due to session IDs or other data being appended to URLs [19]. This limits caching if objects are identified by URL, and it may be better to identify objects by a hash of their content.

Value-based caching goes beyond this by partitioning files into blocks of some reasonable size, and caching these blocks separately, each identified by a hash of its content. This means that a page that is similar to a page previously viewed by a client (i.e., contains long common substrings) may already be partially cached in the client cache, and we only need to send those blocks that are not yet known to the client. The main challenge is that in order to identify the common substrings, the new file and the previously viewed file need to be partitioned in a consistent manner such that the common substring corresponds to a block in both files. Simply partitioning files into fixed-size blocks, e.g., of 500 bytes, does not work since a single extra character at the start of the new file would misalign all block boundaries between the files.

This problem is solved through the use of Karp-Rabin fingerprints [18]. In particular, a smaller window (e.g., of size 20 bytes) is moved over each file that is processed. For each byte position of the window, we hash the content using a simple random hash function. If the hash value is 0 mod b (say, $b = 512$), then we introduce a block boundary at the end of the current window. Figure 2.2 shows an example for a window size of 4 bytes and $b = 8$.

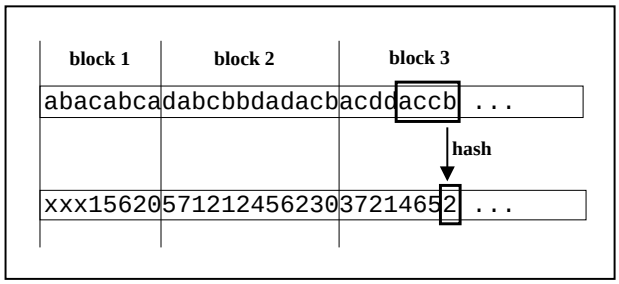

Figure 2.2: Use of Karp-Rabin fingerprints to partition a file into blocks. In this case, a window of size four bytes is moved over the file and at each position a hash $h()$ of the window is computed. Hash values are in the range $\{0, \ldots, 7\}$, and a block ends whenever we have a hash value of $0 \bmod 8$. Thus, the expected block size is 8 bytes unless there are repetitive patterns in the file.

The resulting blocks are then cached and identified by a hash of their content. We note that the hash function for identifying blocks is independent of the hash function used for the small window. For example, we may decide to use MD5 or SHA1 for hashing blocks, while the hash function for the window is chosen such that it can be efficiently "rolled" over the file. In particular, for efficiency reasons we need to be able to compute an updated hash value in constant time from the previous hash value whenever the window is shifted by one character, which is one of the main benefits of the hashes described in [18]. The only purpose of the window is to define block boundaries in a content-dependent manner. We observe that when a substring in one file contains a block boundary according to the above process, then if the same substring also appears in another file, it is guaranteed to again contain the same block boundary. Similar ideas have been used in the networking and OS communities to decrease communication costs for repetitive data [25, 32, 10].

There are some technical issues with the above approach that need to be addressed. A naive view would be that we expect a boundary to occur approximately every $1/b$ bytes. However, many files contain very regular patterns of data that may result in very short or infinite size blocks (e.g., if the file consists of a single character that is repeated thousands of times). In fact, a malicious user could invert the random hash function, and create documents which result in very small or very large blocks. Although the consistency of the content is still ensured on the client, there may be a negative effect on performance. This problem is resolved in [29] by enforcing a minimum and maximum block size. For example for $b = 2048$ one might choose a minimum block size of 64 bytes by ignoring earlier 0 mod b hash values, and a maximum block size of 16384 bytes after which a boundary is forced. A possible alternative is the scheme in [31] though it is unlikely to change results much.

Thus, in the proxy architecture shown in Figure 2.1, both clients and proxy would partition each file they encounter into blocks, and build a lookup structure on the hash of each block. Note that the proxy would only have to keep this lookup structure with the hash values in order to encode a file that is sent to the client; the actual files do not have to be cached at the proxy. When sending a new file for the client, the proxy uses the hashes to check whether any of the blocks have already been sent to the same client and if so, the hash is sent instead of the block. The other, unmatched blocks are either sent verbatim or compressed via *gzip*. On the client side, both hashes and files are stored, with pointers from the hashes to the file positions where they occur, to allow the client to identify the blocks corresponding to the received hashes.

We note that this approach fits ideally into the framework of one proxy interacting with many fairly powerful clients, since most of the data caching is done at the clients while only a small amount of data per client is stored at the proxy. For example, in [29], only a 16-byte hash is stored at the proxy for each block of expected size 2048 bytes. For concurrency reasons, each file is also briefly retained at the proxy until it is sure that it has been correctly received at the client, but this does not change the overall picture. On the other hand, the computational load of the algorithm at the proxy is low enough so that the incoming data can be processed at a rate of multiple MB/s even with a moderate CPU [29]. This allows the proxy to serve a large number of clients simultaneously. Our goal is to obtain significant bandwidth savings over the approach in [29] while preserving the client-centric space and work distribution between the two sides.

2.3 Delta Compression Approaches

Value-based caching provides an alternative to another, more extensively studied approach to exploiting similarity between versions of pages or different pages, called *delta compression*. A delta compressor takes two files as input, a *reference file* and a *target file*, and produces a *delta* that describes the difference between the two files in a minimum amount of space. Given the reference file and delta, the corresponding decompressor can recreate the target file. Delta

compression has been widely studied over the last few years, see [33] for an overview, and a number of optimized delta compressors based on LZ-type approaches are freely available [35, 20, 22].

Under the client-proxy scenario in Figure 2.1, delta compression can be applied as follows. Both proxy and client cache previously transmitted files, and if a new file has to be transmitted, then it is delta compressed by selecting a similar file in the cache (either an older version or a different page with similar content or structure) as a reference file. Several approaches for using delta compression in web proxies have been proposed, including approaches limited to using old versions of the same page as reference files [4, 24, 11, 23], and others that attempt to select the most similar reference file among all cached pages [7, 30]. In the first case, a drawback is that pages need to be stored at the proxy for a longer period of time in case of a revisit, and aliased pages cannot be captured. In the second case, we need an efficient way to identify the best reference file, and several approaches for this are studied in [7, 30]. On the positive side, results in [30] indicate that even if pages are retained for brief periods, some decent benefits can be obtained.

In the second part of this paper, we compare our hierarchical substring caching approach to delta compression, and derive a hybrid that improves on both approaches. Compared to value-based caching and our hierarchical substring caching, delta compression seems to put more load onto the proxy as it requires files to be retained by the proxy for a certain amount of time. An alternative approach called *file synchronization*, implemented in the *rsync* tool [38] and used for web access in [37], requires no data at all to be maintained at the proxy. In fact, this approach is related to value-based caching, and we will discuss this relationship and a potential application. In a nutshell, delta compression requires full knowledge of reference data, while value-based caching works with knowledge of only a few hash values and file synchronization requires no knowledge of the reference data at all.

3. OUR CONTRIBUTIONS

In this paper, we describe and evaluate techniques for improving the efficiency of content delivery over slow links in a client-proxy environment. While our techniques are optimized for web access over dialup links, they are also potentially applicable to other scenarios. Our main contributions are as follows:

(1) We propose a hierarchical substring caching technique that improves and generalizes the value-based caching technique of [29]. The technique introduces a natural multi-resolution approach to cached content, where recently seen content is remembered in detail while items seen some time ago are only remembered at very coarse granularity.

(2) We describe several additional techniques for reducing overheads in our approach.

(3) We perform an evaluation of the techniques using a large set of traces that we collected from our institution over several weeks.

(4) We perform an experimental comparison of our approach with delta compression, and propose a hybrid algorithm that improves on both approaches.

4. HIERARCHICAL SUBSTRING CACHING

We now describe hierarchical substring caching as a generalization of value-based caching. Recall that in value-based caching, files are partitioned into blocks in a content-based manner, with the average block size determined by the parameter b in the algorithm. In [29], an expected block size of 2048 is used, and it is natural to ask if significant additional savings could be obtained with a smaller block size. This question was answered in [29] in the negative based on an experimental evaluation.

We follow a slightly different approach. Instead of simply decreasing the block size, we propose to maintain blocks of different sizes on multiple levels. While this basic idea is rather simple, it leads to some interesting perspectives and requires several non-trivial optimizations to minimize overhead. In particular, we define a hierarchical k-level partitioning of a file, which is determined by parameters $b_0, b_1, \ldots, b_{k-1}$ where b_{i-1} is a multiple of b_i. (Usually, we choose the b_i as powers of two.) We divide a file again by sliding a window of a certain size over it, and declare a level i boundary whenever the window content hashes to 0 mod b_i. Thus, a level i boundary is also a level $i+1$ boundary, resulting in a hierarchical partitioning of the file. A simple example of a two-level partitioning is shown in Figure 4.1.

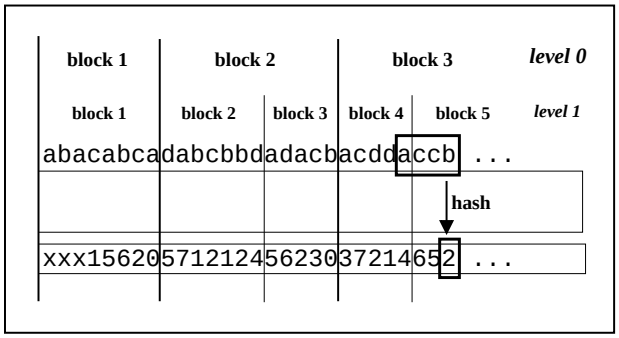

Figure 4.1: A simple example of hierarchical partitioning with two levels. A window of size four bytes is moved over the file and at each position a hash $h()$ of the window is computed, with hash values in the range $\{0, \ldots, 7\}$. In this case, a level 0 block ends whenever we have a hash value of 0 mod 8 and a level 1 block ends whenever we have a hash value of 0 mod 4. Note that not every level 0 block is partitioned into two level 1 blocks, but some may be partitioned into more than two and some into only one level 1 block.

Such a hierarchical partitioning leads to a natural multi-resolution view of caching. Hashes at the highest level $k-1$ allow us to exploit very fine-grain matches with small strings that have previously appeared in other files, while hashes at the lowest level 0 allow succinct referencing of very large matches. Of course, since we have to retain a hash value for each block, the blocks at level $k-1$ will take up a disproportionate amount of the caching space at the proxy. This in turns leads to these hashes being evicted fairly soon, while the relatively few hashes at level 0 are retained for an extended period of time. Thus, the cache maintains a multi-resolution view of previously seen data, with more fine-grained knowledge of recently seen data and a very coarse-grained view of much older data. An interesting question that we will address later is how long hashes at each level

should be kept, or equivalently, how much space should be allocated to each level of blocks.

Given this view, one might be tempted to first insert only the level $k-1$ hashes into the cache, and to then combine several level i hashes into one level $i-1$ hash at eviction time. However, this makes it more expensive to express long matches in the beginning, and it also requires overhead to remember which level i hashes make up one level $i-1$ hash. For this reason, we immediately insert hashes for all levels. This would result in a factor of 2 overhead in space versus only inserting level $k-1$ hashes, in the case where $b_i = 2 * b_{i+1}$ for all i and where all entries are kept for the same amount of time, but actually results in much lower overhead since entries in higher levels are evicted much sooner.

Before proceeding, we discuss the computational cost of hierarchical substring caching. During the process of identifying the block boundaries, the algorithm checks if the small window hashes to 0 mod b_i, starting from the highest level (smallest blocks). If so, we then check whether this is also a boundary for the next level, until either a non-zero modb_j hash is obtained or level 0 is reached. Thus, hierarchical substring caching introduces no significant extra CPU cost to identify the block boundaries, compared to value-based caching. On the other hand, at first glance it seems that the overhead for computing the hash values of the blocks themselves increases with the number of levels. However, it is possible to achieve better performance as follows: The algorithm computes the hashes only for the highest level (smallest) blocks using a Karp-Rabin hash function instead of MD5. Since Karp-Rabin hash functions are composable, the parent hash value can be computed from those of the children. Thus, the overhead for computing the hash values is in fact independent of the number of levels. Overall, performance similar to that of value-based caching can be achieved by hierarchical substring caching. In our current implementation, we use an MD5 hash function for convenience, but one could substitute this with a faster one to improve the computational performance.

Next, we describe the data sets that we used and our implementation of the proxy and client components. We then first evaluate a very basic version of our hierarchical substring caching approach. Then we present and evaluate various optimizations which obtain significant improvements over the basic version.

4.1 Traces and Experimental Methodology

For our experimental evaluation, we collected live web access traces of several static IP subnets with single-user workstations within our university network. We collected these traces using `tcpdump` [17] from November 7, 2003 to November 21, 2003, with several short interruptions. Individual responses were then reconstructed from the raw packet traces using `tcpflow` [12]. In our evaluation, we ignored responses that are already encoded (e.g., compressed with gzip) for technical reasons, though these were only a very small fraction of the total data. We also excluded media content (i.e., audio and video), software downloads, and any responses larger than 1 MB. It is expected that for real low-bandwidth clients, such requests would make up a far smaller percentage of the total traffic than in our LAN-based campus environment.

The total size of the remaining web traffic was 2.13 GB, which belonged to 67 clients with various access patterns. Around 737 MB of this content consists of HTML and text responses (any mime type that contains the string text), 1.08 GB was image content (any mime type that contains the string image), and the remaining content was in various application formats (e.g., pdf, ps, doc). There were a total of 401168 responses with 134 mime types.

In the following, we first focus on the HTML and text responses and evaluate the benefits of various techniques by considering the bandwidth savings that are achieved on this data. To efficiently deal with image data, it is necessary to employ appropriate transcoding techniques as discussed, e.g., in [8]. Such transcoding can, for example, be implemented by converting images to the jpeg2000 format [1] and reducing image quality appropriately. We evaluate the potential benefits of combinations of substring caching and image transcoding in Section 5.

4.2 Proxy Server Cache Implementation

As described earlier, the proxy server moves a small window over the requested content to define the block boundaries for various levels. Each identified block is then represented in the *substring cache* by a small entry consisting of the following items: the MD5 hash value of the block, a timestamp (a sequence number that defines a relative ordering), the level of the block, and the ID of the client that the block was sent to. Although the length of an MD5 hash is 16 bytes, we used only 8 bytes in our implementation. We investigate the effect of this decision below, and also describe a recovery scheme for possible collisions on the client side – though these are very unlikely even with 8-byte hashes.

When the proxy cache is full, entries are evicted based on a *Least Recently Used* (LRU) policy. Since we have a hierarchy of multiple levels, we assign a certain fraction of the available space to each level and use an LRU policy for each level separately (the default is to give the same space to all levels). For efficiency reasons, whenever the cache is full we evict some small fraction of the entries in the cache (usually the oldest 10%) at once, instead of evicting single entries. Note that there is no fixed allocation of space to each client, but the cache is shared among all clients. Thus, an active client would usually have more entries in the proxy cache than a less active client. In Section 4.5, we revisit and evaluate this design decision.

4.3 Performance of Basic Hierarchical Scheme

We now look at the performance of a basic implementation of Hierarchical Substring Caching. In this basic scheme, the proxy uses the following approach to encode the requested content efficiently before sending it to the client. As the block boundaries are identified, the proxy server performs lookups into the cache to determine matched and unmatched blocks. Matched blocks are represented efficiently by an array of 8-byte MD5 hash values, one per block. Since we have various levels of blocks, the proxy server always looks for the largest possible matches (i.e., matches on the lowest possible level), to minimize the number of hashes that are sent. The remaining unmatched blocks are concatenated to form the *unmatched literals*, which are then compressed with *gzip*. To allow the client to reconstruct the response from the received byte stream, the proxy server also sends the total size of the compressed unmatched literals, the number of hashes in the array, and the byte offsets of the matched blocks in the response. The first two items are encoded with

a simple variable-byte code, while the array of byte offsets is encoded using Golomb coding (see, e.g., [39]) of the offset from the end of the previous match.

Note that with this scheme, data can be streamed to the client as it is received by the proxy, except that the proxy server needs to wait until the next level 0 boundary is encountered before sending the data on. Choosing a smaller expected size for the lowest level (largest block) might thus result in better streaming characteristics. For every response, the proxy server also sends a 16-byte MD5 of the entire content that allows the client to check the correctness of the response. Thus, any collisions due to the use of 8-byte MD5 hashes is detected by the client, who can then request the content again from the proxy. (On our data, we did not observe any collisions due to the use of 8-byte hashes.)

In Figure 4.2, we compare the bandwidth usage of this basic implementation of hierarchical substring caching to value-based caching from [29]. We use a proxy cache of 10 MB for our data set of 737 MB of content. We show results for value-based caching with 8- and 16-byte hashes, and for simple *gzip*. Consistent with [29], value-based caching with 16-byte hashes benefits only slightly from block sizes less than 2048 bytes. Some additional benefits are obtained by using 8-byte hashes with block size 256 bytes, but performance degrades for block sizes less than 128 bytes. The basic hierarchical approach already obtains significant additional improvements over value-based caching, particularly with 5 or more levels and minimum expected block sizes of 128 or less. As expected, *gzip*, which is independent of block size, performs worst.

A clarification concerning our use of the term *expected block size* for the parameter b_i in the definition of a hierarchical partitioning. Due to the enforced minimum and maximum block sizes (20 and 8192 bytes), the actual expected block sizes even under random content are slightly different. At the low end, $b_i = 16$ really results in an average block size close to $16 + 20$ bytes. For a real block size of 16 bytes, no savings over *gzip* would be obtained by replacing it, say, by a hash of 8 bytes.

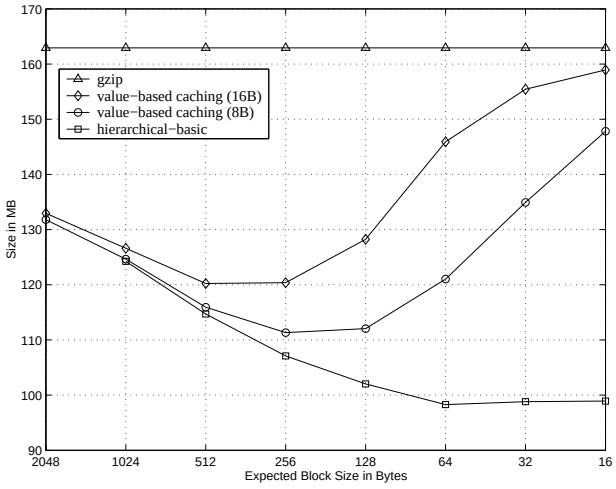

Figure 4.2: Bandwidth usage of value-based caching and a basic version of hierarchical substring caching, for various expected block sizes. For value-based caching, there is only one level. For hierarchical substring caching the lowest level has an expected block size of 2048 and we vary the block size in the highest level from 1024 to 16.

Prefix	Matches	Lookups	>10K	Gain
13	4.65346	637121.67	77180	6.88%
14	2.91242	9694.25	18429	6.75%
15	2.05026	240.77	1213	6.63%
16	1.61506	18.72	18	6.51%
17	1.39754	4.94	5	6.39%
18	1.28594	3.13	4	6.27%

Table 4.1: Evaluation of the average number of matches, number of lookups, and compression gains using various prefix lengths, for $k = 8$ and a postfix length of 32 bits.

4.4 Optimizations over the Basic Scheme

Next, we describe and evaluate three optimizations over the basic hierarchical caching technique in Figure 4.2, which result in significant additional improvements.

Shorter Hashes with XOR: In the first optimization, the proxy server partitions each 8-byte hash into a prefix of a certain size (say, the first 20 bits of each hash, though the best choice depends on the size of the cache), and a postfix with the rest of the hash (in practice, the next 32 bits following the prefix usually suffice). Instead of sending the 8-byte MD5 hash value for each matched block, the proxy now sends only the prefix of each hash. In addition, for every *batch* of k (say $k = 8$) matched blocks, the proxy server computes and sends the XOR of the k postfixes. The client, upon receiving the hash data, uses an ordered data structure to identify all blocks whose MD5 hashes start with the received prefixes. If there is only a single match for each prefix, the client checks the validity of the matches using the XOR data. If one or more of the prefixes result in more than one match, then the client resolves this ambiguity with the help of the XOR bits, as follows. The client checks every combination of matches for the $k - 1$ prefixes with the fewest numbers of matches, and for each combination uses the postfixes of the matches to determine the postfix of the k-th hash that is needed to obtain the received XOR. Then a lookup is performed to see if any of the matches of the k-th prefix have the required postfix. If any lookup is successful, then we have found the right combination of matches. Thus, the number of lookups performed is the product of the number of matches for the $k-1$ prefixes with the fewest numbers of matches. In the following, we look at how to choose the size of the prefix to keep the number of lookups at a reasonable level. Note that these lookups are only performed at the client, which usually has ample CPU power to decode data arriving at dialup speed. In contrast, the proxy does not need to maintain and query an ordered data structure on the hash values, but can use a faster hash-based or similar organization.

In Table 4.1, we evaluate this scheme for an 8-level hierarchy with expected block sizes from 2048 to 16 bytes for various prefix lengths. The second column shows the average number of matches observed for each hash, and the third column shows the average number of lookups per batch that are performed to resolve collisions in the prefixes. The fourth column shows the number of batches that resulted in more than $10,000$ lookups, and the last column shows the overall improvement in compression over the basic hierarchical scheme. We use $k = 8$ and a postfix length of 32 bits, which works well in practice. We counted a total of $281,030$ XOR batches, and the cache contained up to $309,341$ entries belonging to several dozen clients.

As one would expect, choosing a size of the prefix much smaller than the logarithm of the client cache size results in a large number of collisions. (In addition, performing many lookups also requires a larger postfix to achieve the same degree of certainty about the correctness of the match.) Choosing a prefix size close to the logarithm of the proxy cache size and thus slightly larger than the logarithm of the largest client cache size, say 17 or 18 in this case, gives a moderate number of lookups with good gains in compression. In our subsequent experiments, the proxy server sends a 17-bit prefix for each match, which works well for our range of cache sizes. Note that in this setting the likelihood of a false positive collision is slightly higher than with true 8-byte hashes. However, choosing slightly more than 32 bits for the XOR would remedy this problem at little extra cost.

Using Deltas for Literals: Applying *gzip* to compress the literal data allows us to exploit repeated patterns within the literals. However, matched data and unmatched literals also share many repeated patterns since they are part of the same content. Thus, if we compress unmatched literals by themselves, we fail to exploit this redundancy between literals and matched data. To resolve this issue, we use a delta compressor to compress the literal data. Recall that a delta compressor encodes a target file with respect to one or more reference files, by essentially encoding how to reconstruct the target file from the reference files. In this case, we concatenate the matched blocks into a reference file that is used to encode the literals using the *zdelta* delta compressor [35]. Note that due to limitations in the current version of *zdelta*, we have to wait until we have processed the entire file before running the delta compressor. However, this is not a fundamental limitation of the approach, and could be resolved with a delta compressor that allows both reference and target data to be applied in a streaming manner. Note that a modification to *gzip* proposed in [36] could also be used instead.

Using Overlapping Blocks: Recall that a block boundary occurs whenever a small window of a certain size hashes to 0 mod b. Under the definition in [29], a block contains the small window that defines its end, but not the one that defines its beginning. However, any match between two blocks is likely to extend to this small window, since both blocks have to be preceded by a small window that results in a boundary. Thus, we propose to redefine the blocks to contain the small windows on both sides. As illustrated in Figure 4.3, every block now contains both of the small windows, and thus consecutive blocks on the same level are overlapping. While this may result in some missed matches (if two blocks are preceded by different boundary windows to the left under the old definition), there are two advantages. First, while we miss a few matches, we are also able to match additional literals when an unmatched block is followed by a matched block. In particular, on our trace the total number of matches dropped by about 4.5%, mostly due to missed small matches, but the total size of the matched literals stayed almost the same (minus 0.5%). Thus, fewer bits are needed to communicate the matches, and we see overall bandwidth savings of about 1.75%.

The second advantage of using overlapping blocks is that it allows the proxy server to send less bits for consecutive matched blocks. Recall that each block is represented by a 64-bit MD5 hash value in the cache. Now we redefine this representation as follows. For each block, we replace

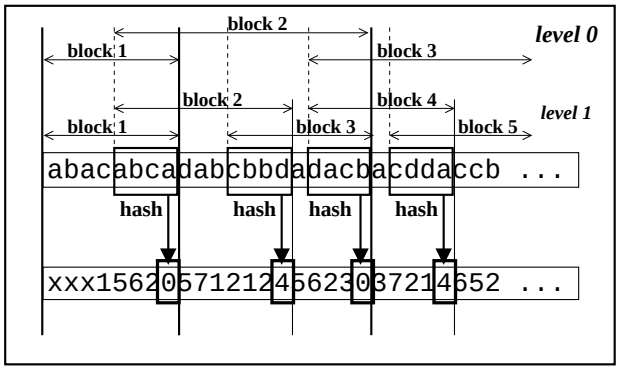

Figure 4.3: Overlapping block boundaries.

the first, say, 8 bits of its MD5 with a hash of the small window on the left edge of the block (the starting window). We keep the other 56 bits from the MD5 of the entire block. For any two consecutive matched blocks in the response, we can now skip the transmission of the initial 8 bits of the second hash since these bits can be computed by the client from the previous block. However, we might expect more collisions since the small initial window is likely to be repeated occasionally in the content, and the XOR collision resolution also needs to be modified since ambiguities in the first match in a batch now result in different hashes for subsequent blocks. We tried various choices for the number of initial bits that are taken from the starting window hash, and found good performance for 8 bits with a prefix of 17 bits as before. The two optimizations, when combined, yield an improvement of about 3%, though the second one comes at some cost in code complexity that may not always be worthwhile.

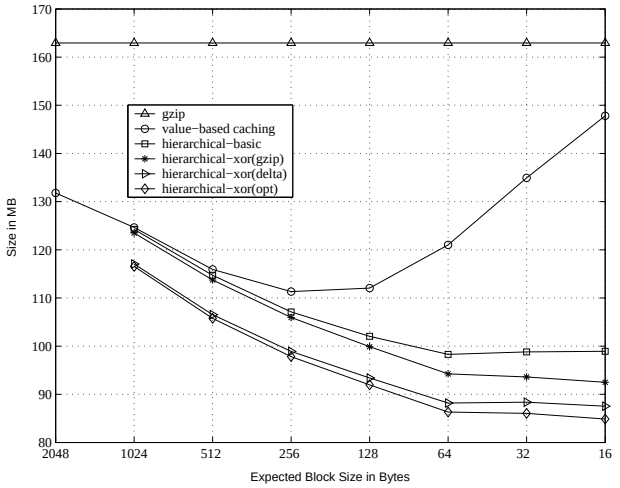

Figure 4.4: Total bandwidth use of hierarchical substring caching with optimizations.

Performance Evaluation of Optimizations: In Figure 4.4, we show the benefits obtained by these optimizations on our trace. We show from top to bottom the performance of *gzip*, value-based caching with 8-byte hashes, the basic hierarchical scheme from Figure 4.2, and three optimized codes. The optimizations from top to bottom are (i) the XOR technique, (ii) use of XOR and *zdelta*, and (iii) XOR, *zdelta*, and overlapping blocks. As we see, the opti-

α	1	1.25	1.5	1.75	2	2.25	2.5
L0	21.29	22.51	23.14	23.81	24.40	24.74	25.05
L1	11.25	11.57	12.09	12.28	12.23	12.35	12.41
L2	12.48	12.85	13.14	13.29	13.49	13.62	13.71
L3	13.84	14.13	14.16	14.18	14.24	14.25	14.22
L4	12.28	11.91	11.67	11.46	11.28	11.16	11.09
L5	10.52	10.11	9.74	9.53	9.30	9.15	9.05
L6	9.42	8.91	8.50	8.17	8.02	7.87	7.78
L7	8.92	8.01	7.56	7.28	7.04	6.86	6.69
Size	90.11	88.85	86.73	85.46	84.87	84.65	84.88

Table 4.2: Total bandwidth use in MB (bottom) and percentage contribution of each block level to the total matched data, for various values of α.

mizations give an additional reduction of almost 15% percent in bandwidth use over the basic hierarchical scheme. Comparing the best optimized result to *gzip* we get an improvement from 163 to 84 MB (48%), and comparing to the best setting value-based caching, we get an improvement from 112 to 84 MB of bandwidth usage (25%).

4.5 Assignment of Proxy Cache Space

Next we evaluate the proper assignment of cache space to the different levels of blocks. We note that each assignment corresponds to an expected time that an entry will remain in the cache under some simplifying assumptions. For example, if we assign the same amount of space to all levels, then a naive view would be that the entries for the largest blocks might stay in the cache about twice as long as the entries for the next level, since there are only half as many of the larger blocks that are produced. In Table 4.2, we look at the effect of different memory allocation policies for the levels of the cache. In each policy, space is allocated such that an entry at level i is expected to stay by a factor of α longer in the cache than an entry at level $i+1$ under this naive view. As we see, it is obviously better to allow larger blocks to stay in the cache for a longer period of time, but the precise choice of α does not seem to make a huge difference and our default choice of $\alpha = 2.0$ (same space for each level) is almost optimal.

There are two benefits in starting the lowest level at a fairly large expected block size, such as 2 KB in our case. First, these entries allow us to transmit large matched areas in the content more efficiently, with fewer hashes. The second benefit is due to the fact that we only have a limited cache size. Thus, by using a small amount of space to keep larger blocks for a longer period of time, we can identify matches that would have already been evicted if they had been stored only as smaller blocks.

However, one could argue that it might be better to get rid of the lowest levels (current level 0), and to only use 7 levels from 1024 down to 16 bytes, or 6 levels from 512 down to 16 bytes, and to give more space to each remaining level. We analyze this issue in Figure 4.5, where we vary the expected block size from 1024 to 32 for the lowest level. In the figure, we present the resulting changes in bandwidth consumption compared to our default setting of 8 levels from 2048 to 16 bytes. In each case the same amount of total cache space (10 MB) is divided evenly among the levels. As we see, there is actually a slight benefit in using only 7 levels from 1024 to 16 bytes, but for initial block sizes of 256 or less there is a significant cost in terms of missed matches due to evictions (black bar) and additional encoding overhead due to more

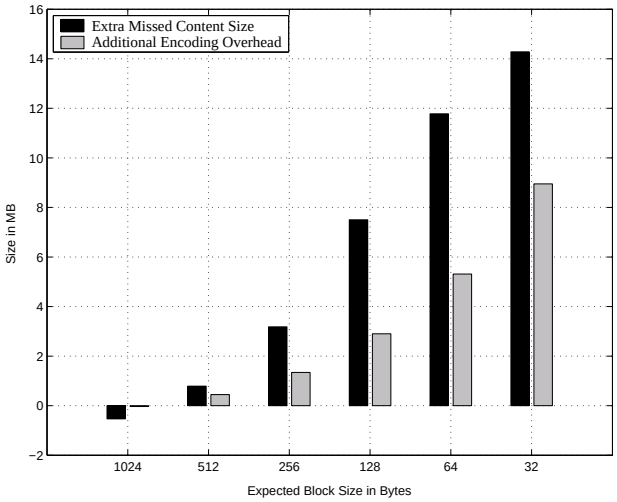

Figure 4.5: Performance with varying numbers of levels, identified by largest expected block size. Black bars show the size of the matched literals that would be missed compared to an 8-level hierarchy starting at 2048-byte block size, and grey bars show extra encoding overhead.

hashes that need to be sent (grey bar).

In Figure 4.6, we compare *gzip*, value-based caching, basic hierarchical substring caching, and the optimized version on different cache sizes from 1 to 20 MB. As we see, all of the techniques already do quite well even on fairly moderate proxy cache sizes; this is important since in a realistic dialup scenario, each proxy will usually be responsible for a larger number of active clients than in our trace of a few subnets.

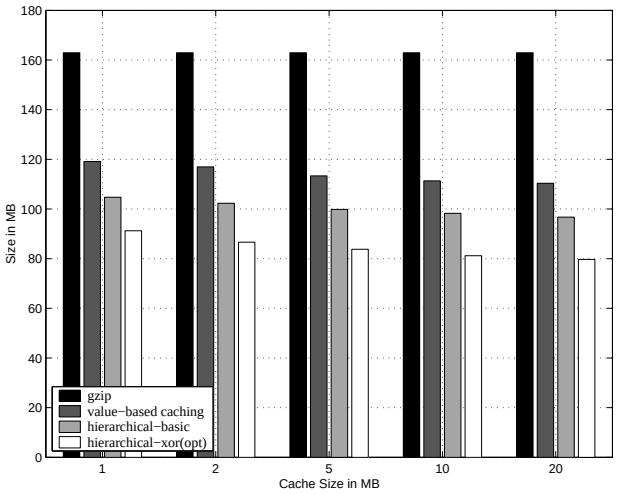

Figure 4.6: Comparison of schemes for various cache sizes.

Shared Cache vs Fixed Memory Allocation: Recall that in our default cache design, the space is shared among all clients, thus allowing active clients to utilize a larger share of the cache space. Another approach would be to consider a client active as long as it has an entry in the cache, or has made an access in the last x minutes, and to assign equal space to all active clients. In Figure 4.7, we compare these two designs and show the benefits of a shared cache design over a fixed allocation for each client. As we

see in the figure, the shared cache design offers significant benefits to the most active users (the leftmost bars) while not significantly impacting the less active users. Overall, the shared allocation achieves about a 6.25% reduction in bandwidth usage.

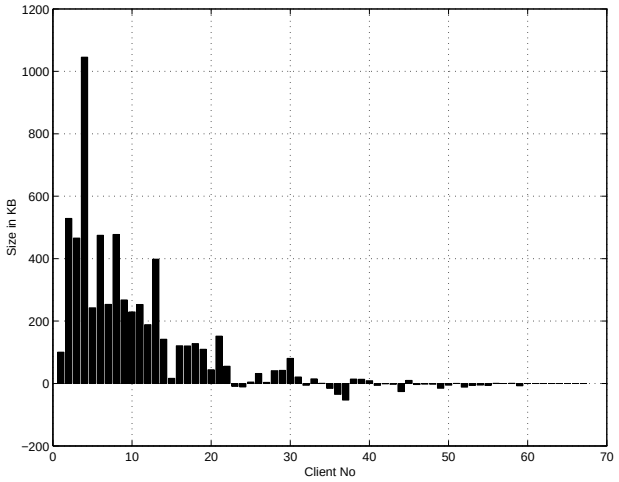

Figure 4.7: Benefit of shared cache design over fixed allocation. Clients are ordered from left to right by the total amount of content that they retrieve, with the y-axis showing the total bandwidth savings of the client compared to a fixed allocation of cache space (a negative value means a client would benefit from a fixed allocation).

5. COMPARISON TO DELTAS AND A HYBRID

In this section, we compare our hierarchical scheme to approaches based on delta compression, and then propose a hybrid that unifies and improves on both approaches. As shown in the previous section, compression improves when the unmatched literals are delta compressed with respect to the matched literals (as opposed to just using *gzip*). In the context of web access, delta compression is also commonly used to compress the requested pages with respect to previously transmitted similar pages. While several schemes limit themselves to delta compression between different versions of the same page [4, 24, 11, 23], others attempt to select the most similar reference file among all cached pages [7, 30]. In the following, we take the latter approach, which is more appropriate when files can only be cached for a fairly short amount of time as in our case where many clients share the proxy.

Thus, a limited amount of cache space is used at the proxy to store recently accessed pages for future use as reference files. We then implemented two techniques for identifying good reference files. The first technique is based on the sampling approach for estimating file similarities proposed by Broder in [6], also used in [30] and referred to as *shingles* in Figure 5.1. In our second technique, we split the cache into two parts, one for storing cached files, and one for hierarchical substring caching as previously described. However, we now use any block matches that we encounter to identify appropriate reference files. For each entry in the substring cache, we keep a list of pointers to those files in the file cache that contain this block. The proxy then follows the pointers whenever a match is found to identify the file in the file cache that contains the largest number of matched literals. This file is then used as a reference file for delta compression; the technique is referred to as *delta* in Figure 5.1.

From Figure 5.1, we see that the first technique, *shingles*, does slightly better than the second technique. Both techniques perform better than value-based caching for all cache sizes, but do not perform as well as our hierarchical substring caching technique.

Also shown in the figure is a hybrid technique, called Hierarchical Substring Caching with Delta Compression and labeled *HSC-delta*, that outperforms all other techniques and works as follows. As in *delta*, the proxy server splits its cache space into a file cache and a substring cache and uses block matches to identify the best reference file in its file cache. Then the proxy determines any matches in files other than the best reference file, and transmits them to the client through the use of hashes, as in Hierarchical Substring Caching. The remainder of the file, including the matches found in the best reference file, are then delta compressed with respect to the best reference file and a second reference file constructed from the matches found in other files. (The *zdelta* compressor supports up to 4 reference files.) This improves on Hierarchical Substring Caching by using an optimized delta compressor (with Huffman-coded offsets instead of inefficient hash values) to identify matched regions in the best reference file, which contains most of the matches.

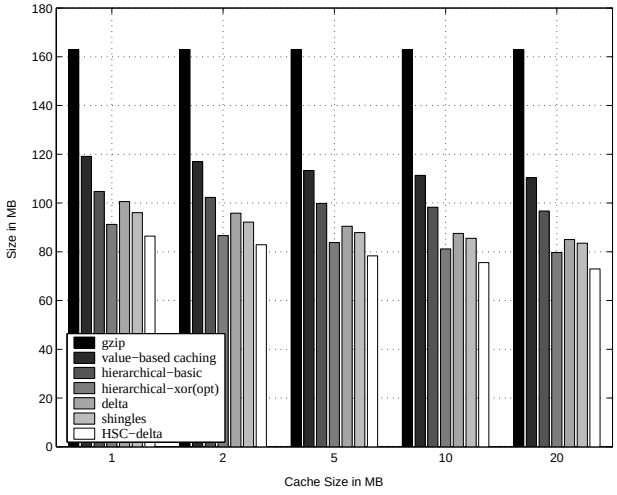

Figure 5.1: Comparison of delta encoding and substring caching

In order to find a good ratio between the sizes of the file cache and the substring cache, we experimented with various partitioning strategies. As shown in Table 5.1, devoting just a quarter of the cache to the file cache is enough; using a larger fraction decreases the benefits. The table also shows how much of the matched literals were available in the best reference file. As we see, for the best setting almost 25% of the matched literals were not contained in the best reference file; this makes HSC-delta perform better than any technique based only on deltas.

The cache sizes used for file caching may seem rather small compared to the sizes used in studies on web caching. This is so for several reasons. First, for performance reasons it is highly desirable to have the file cache in main memory,

	1/4	1/3	1/2	2/3	3/4
Encoding	77.99	78.27	79.17	80.47	81.42
Ref. Hit	257.68	264.05	271.53	274.87	275.46
Other Hit	62.33	59.57	52.85	45.80	41.52
Total Hit	320.01	323.62	324.38	320.67	316.98

Table 5.1: Total encoding size in MB with various allocations of space for the file cache, with total cache size 10 MB. The second row shows the size of the matched literals contained in the best reference file, while the third row shows the size of the matched literals only available in other files.

since otherwise each delta compression step requires an extra disk access, resulting in a potential bottleneck if many clients are currently connected to the proxy. Secondly, as shown in [30] and confirmed by the above numbers, significant benefits are available with delta compression even if files are only retained for a fairly short period of time, such as a few minutes [30].

In this case, most of the benefit comes from similarities between the requested page and other pages from the same site that have just been visited. Capturing similarities between different versions of the same page visited several days or weeks apart would either require a very large cache on disk, or would possibly be better handled using hierarchical substring caching or file synchronization techniques. In particular, if a page is revisited after all the block hashes from the previous visit have already been evicted from the proxy, but the client still has the old page in his disk cache, then there is a very simple and elegant way to utilize file synchronization techniques similar to *rsync* in this context. The client can simply partition the old version of the page into blocks, say of expected size 512, 1024, or 2048 bytes, and include the hashes of these blocks in its HTTP request. The proxy then intercepts these hashes, inserts them into its substring cache, and processes the new file in the normal way as it arrives.

5.1 Impact of Images and Image Transcoding

As mentioned earlier, significant speedups over dialup links can be obtained by employing transcoding techniques on image files. On the other hand, if no transcoding is performed, then our techniques will be limited in benefit, as they are primarily suitable for the text/html fraction of the web traffic. In Figure 5.2, we show the benefits of our HSC-delta hybrid on both text/html and image data when combined with image transcoding techniques. For the 4 bars on the left we assume that transcoding decreases image file size by a factor of 2 on average, while for the bars on the right we assume a factor of 3. In each group, the leftmost bar represent the bandwidth usage when all images are transcoded to jpeg2000 format without any quality loss (but often with a slight decrease in size due to better compression in jpeg2000), and text content is compressed simply with *gzip*. The second bars show the bandwidth usage when duplicate files are identified and encoded by 8-byte MD5 hashes (a simple form of hash-based alias detection). The third bars assume transcoding techniques on images and *gzip* on text/html, while the last bars assume transcoding on images and the HSC-delta hybrid on text/html. As we see, any approach that focuses on one type of data, such as text/html, has to be combined with techniques for other data types in order to make a real impact in performance.

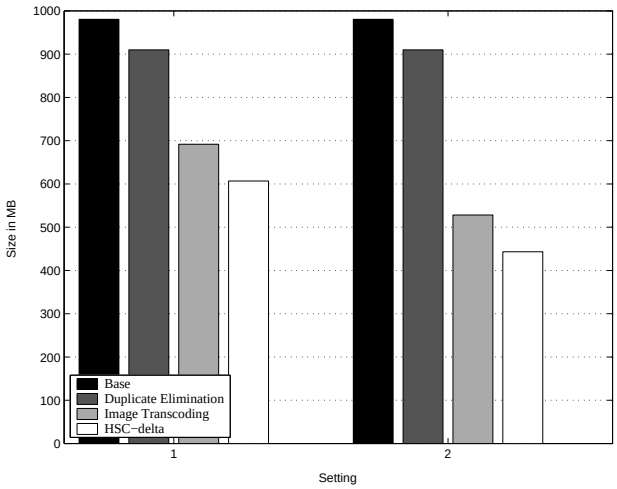

Figure 5.2: Combined benefits of image transcoding and HSC-delta on our traces.

6. RELATED WORK

Much of the relevant work was already reviewed in Sections 1 and 2, and hence we only briefly list here the most closely related results.

Our work is most closely related to, and an extension of, the value-based caching technique in [29], which itself employs Karp-Rabin fingerprints as introduced in [18]. Techniques similar to value-based caching have also recently been used by researchers in the networking and OS communities [32, 25, 10]. We are unaware of any previous hierarchical uses of these block partitioning techniques.

Delta compression tools and algorithms are described in [16, 3, 35, 20, 22, 33]. Delta compression for web access has been studied in [4, 24, 11, 23, 7, 30, 28, 26]. The first four papers assume that deltas are only performed between different versions of the same page (URL), while the last two assume that good reference files are identified by the content provider. Work in [7, 30] addressed the problem of how to select reference files from all possible files. We follow the main conclusions from that previous work by using simple heuristics to identify reference files and limiting delta compression to very recent reference files. While [24] evaluates delta compression between different versions of the same page over a large set of traces, we are not aware of any previous evaluation of delta compression with arbitrary reference files over longer periods of time.

The *rsync* file synchronization tool is described in [38]. There are several recent approaches to improve the bandwidth efficiency of file synchronization [9, 21, 27, 34], but all these approaches use multiple roundtrips and are thus not suitable for the relatively small file sizes encountered in web access. The utility of *rsync* for web access was studied in [37, 30]. One conclusion from that work is that file synchronization works best for different versions of the same page. In particular, file synchronization allows efficient revisiting of slightly changed pages even after a long period of time, without having to store anything at the proxy.

7. CONCLUDING REMARKS

While we have collected a reasonably large trace and run detailed experiments, it would be beneficial to have our results confirmed on a larger user population, and to conduct

experiments on user-perceived delays using the dummynet tool. It would also be interesting to measure the throughput of the proxy under load once the implementation is fully optimized. As discussed in Section 4, since all the fingerprinting and compression steps in our approach are simple, we expect throughputs of multiple MB/s. For mobile clients, we are confident that with the right parameter settings, these devices can also benefit from the techniques described.

In terms of open research questions, we see a lot of opportunities for future research on general bandwidth-efficient communication primitives, such as better file [38] and data [2] synchronization algorithms, transparent improvements at the networking level [32], and applications of such primitives, e.g., in peer-to-peer systems.

8. REFERENCES

[1] Jpeg2000 standard. http://www.jpeg.org/jpeg2000/.

[2] S. Agarwal, D. Starobinski, and A. Trachtenberg. On the scalability of data synchronization protocols for PDAs and mobile devices. *IEEE Network Magazine, special issue on Scalability in Communication Networks*, July 2002.

[3] M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the ACM*, 49(3):318–367, 2002.

[4] G. Banga, F. Douglis, and M. Rabinovich. Optimistic deltas for WWW latency reduction. In *1997 USENIX Annual Technical Conference, Anaheim, CA*, pages 289–303, Jan. 1997.

[5] H. Bharadvaj, A. Joshi, and S. Auephanwiriyakul. An active transcoding proxy to support mobile web access. In *Seventeenth IEEE Symp. on Reliable Distributed Systems*, pages 118–126, Oct. 1998.

[6] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29. IEEE Computer Society, 1997.

[7] M. Chan and T. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proc. of INFOCOM'99*, March 1999.

[8] S. Chandra, A. Gehani, C. S. Ellis, and A. Vahdat. Transcoding characteristics of web images. SPIE - The International Society of Optical Engineering, Jan 2001.

[9] G. Cormode, M. Paterson, S. Sahinalp, and U. Vishkin. Communication complexity of document exchange. In *Proc. of the ACM–SIAM Symp. on Discrete Algorithms*, Jan. 2000.

[10] L. Cox, C. Murray, and B. Noble. Pastiche: Making backup cheap and easy. In *Proc. of the 5th Symp. on Operating System Design and Implementation*, December 2002.

[11] M. Delco and M. Ionescu. xProxy: A transparent caching and delta transfer system for web objects. May 2000. unpublished manuscript.

[12] J. Elson. `tcpflow` – a tcp flow recorder, June 2001. http://www.circlemud.org/jelson/software/tcpflow/.

[13] A. Fox and E. Brewer. Reducing WWW latency and bandwidth requirements by real-time distillation. *Computer Networks and ISDN Systems*, 28(7–11):1445–1456, May 1996.

[14] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. Dynamic adaptation in an image transcoding proxy for mobile web browsing. *IEEE Personal Communications*, pages 8–17, Dec. 1998.

[15] B. Housel and D. Lindquist. WebExpress: A system for optimizing web browsing in a wireless environment. In *Proc. of the 2nd ACM Conf. on Mobile Computing and Networking*, pages 108–116, November 1996.

[16] J. Hunt, K.-P. Vo, and W. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7, 1998.

[17] V. Jacobson, C. Leres, and S. McCanne. `tcpdump`, June 1989. available via anonymous ftp to ftp.ee.lbl.gov.

[18] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[19] T. Kelly and J. Mogul. Aliasing on the World Wide Web: Prevalence and Performance Implications. In *Proceedings of the 11th International World Wide Web Conference*, Honolulu, Hawaii, May 2002.

[20] D. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *Proceedings of the Usenix Annual Technical Conference*, pages 219–228, June 2002.

[21] J. Langford. Multiround rsync. January 2001. Unpublished manuscript.

[22] J. MacDonald. File system support for delta compression. MS Thesis, UC Berkeley, May 2000.

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

[24] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proc. of the ACM SIGCOMM Conference*, pages 181–196, 1997.

[25] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 174–187, October 2001.

[26] M. Naaman, H. Garcia-Molina, and A. Paepcke. Evaluation of delivery techniques for dynamic web content. In *8th Int. Worksh. on Web Content Caching and Distribution*, 2003.

[27] A. Orlitsky and K. Viswanathan. Practical algorithms for interactive communication. In *IEEE Int. Symp. on Information Theory*, June 2001.

[28] K. Psounis. Class-based delta-encoding: A scalable scheme for caching dynamic web content. In *22nd Int. Conf. on Distributed Computing Systems Workshops (ICDCSW)*, pages 799–805, 2002.

[29] S. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *Proc. of the 12th Int. World Wide Web Conference*, May 2003.

[30] A. Savant and T. Suel. Server-friendly delta compression for efficient web access. In *8th Int. Workshop on Web Content Caching and Distribution*, 2003.

[31] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. of the 2003 ACM SIGMOD Int. Conf. on Management of Data*, pages 76–85, 2003.

[32] N. Spring and D. Wetherall. A protocol independent technique for eliminating redundant network traffic. In *Proc. of the ACM SIGCOMM Conference*, 2000.

[33] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In K. Sayood, editor, *Lossless Compression Handbook*. Academic Press, 2002.

[34] T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *Proc. of the Int. Conf. on Data Engineering*, March 2004.

[35] D. Trendafilov, N. Memon, and T. Suel. zdelta: a simple delta compression tool. Technical Report TR-CIS-2002-02, Polytechnic University, CIS Department, June 2002.

[36] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.

[37] A. Tridgell, P. Barker, and P. MacKerras. rsync in http. In *Conference of Australian Linux Users*, 1999.

[38] A. Tridgell and P. MacKerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.

[39] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.