# Server-Friendly Delta Compression for Efficient Web Access

***Anubhav Savant*** ***Torsten Suel****

Polytechnic University
Brooklyn, NY 11201

## Abstract

*A number of researchers have studied delta compression techniques for improving the efficiency of web page accesses over slow communication links. Most of these schemes exploit the fact that updated web pages often change only very slightly, thus resulting in very small sizes for the transmitted deltas. However, these schemes are only applicable to a minority of page accesses, and require web or proxy servers to retain potentially many different outdated versions of pages for use as reference files in the encoding. Another approach, studied by Chan and Woo [4], encodes a page with respect to similar files located on the same web server that are already in the client's browser cache.*

*Based on the latter approach, we study different delta compression policies for web access. Our emphasis is on web and proxy server-friendly policies that do not require the maintenance of multiple older versions of a page, but only use reference files accessed by the client within the last few minutes. We compare several policies for identifying appropriate reference files and evaluate their performance on a set of traces. We show that there are very simple policies that achieve significant benefits over* gzip *compression on most web accesses, and that can be efficiently implemented at web or proxy servers. We also investigate the potential of file synchronization techniques such as* rsync *[28] for efficient web access.*

## 1 Introduction

Delta compression (delta encoding) is the process of encoding a *target file* with respect to one or several, usually similar, *reference files*. This encoding, called a *delta*, describes the target file in terms of the reference files, and a recipient that receives the encoding and already knows the reference files can thus efficiently reconstruct the target. Delta compression has numerous applications in scenarios where there are several versions of a file or many similar files, such as software revision control systems, distribution of software updates, content distribution networks, or efficient storage of related files. Several tools for delta compression, such as *bdiff*, *vcdiff* [10, 13], *Xdelta* [14], and *zdelta* [25], are freely available. We refer to [23] for an overview of delta compression techniques and applications.

*Contact author. Email: suel@poly.edu

### 1.1 Delta Compression for Web Access

A number of authors have proposed the use of delta compression techniques to improve the efficiency of web access [1, 4, 7, 9, 16, 17, 21, 27, 29]. In particular, when web pages are updated, they typically do not change by much, and thus delta compression can be used to very succinctly encode the difference between a new version of a web page and an outdated version already in the client's browser cache. Most proposals focus on encodings between different versions located at the same URL, which results in small sizes for the deltas but is restricted to pages that have been previously visited by the client. One exception is the work by Chan and Woo [4], which proposes to use as reference files other pages on the same site recently visited by the client, which tend to have a significant degree of similarity due to common layout features and HTML structure. In general, delta compression schemes for web access can be distinguished along the following axes:

- **End-to-end vs. proxy-based:** A recent proposal [16] discusses how to integrate delta compression into the HTTP/1.1 standard. This enables end-to-end use of delta compression techniques between web servers and end clients, potentially leading to significant savings in bandwidth over the internet backbone. On the other hand, proxy-based schemes allow savings over a bottleneck link, say a dialup connection of an end user, without requiring changes in the HTTP/1.1 protocol to be adopted by millions of servers. A common architecture is the *dual proxy* architecture, where a pair of proxies, one located on each side of the bottleneck link, use a usually proprietary protocol incorporating compression, image transcoding, and various other optimizations to increase performance over this link. A number of such systems have for example been deployed by the major cellular network providers,

supplied by companies such as Bytemobile, Venturi Wireless, Slipstream Data, and others. In this paper, we focus on such dual-proxy architectures, and we do not discuss in detail how to integrate our approach with existing HTTP standards.

- **Standard vs. optimistic delta:** In the standard approach, both client and proxy have the same old version of a page. A proxy first waits for the updated version of the page to arrive from the server, and then transmits the delta between this and the old version to the client. In the optimistic delta approach in [1] only the proxy needs to hold the old version. Upon receiving a request, the proxy immediately starts sending the old version to the client, while waiting for the new version to arrive from the server; later, a delta between the versions is sent to the client. In a proxy-based environment, neither approach decreases the amount of traffic between server and proxy. The standard approach reduces the amount of data sent from proxy to client, thus reducing delay for low-bandwidth links. The optimistic delta approach does not reduce the amount of data sent to the client, but may decrease total delay in cases where server response delays are also significant compared with the transmission time over the bottleneck link. We focus on the standard approach.
- **Same URL vs. different URLs:** As described, we can limit delta compression to different versions of the same URL, or allow compression between pages corresponding to different URLs. In the latter case, clients and proxies need to choose appropriate reference pages, and this is our main focus.
- **Delta compression vs. file synchronization:** In the standard delta compression scenario, the proxy needs a copy of the old and the new version to compute a delta. File synchronization techniques such as *rsync* [28], on the other hand, allow the proxy to send a delta without knowing the old version in the client cache, based on only a set of hash values sent by the client as part of the request. File synchronization can be seen as a special, more restricted, case of delta compression, and usually achieves compression ratios that are significantly worse than those of the best delta compression tools [23], particularly on files that share only short common substrings. (Formally, any file synchronization protocol can be used to produce a delta by storing all messages sent from target file to reference file.) We consider both techniques.

### 1.2 Discussion of Known Approaches

Delta compression between different versions of the same page typically achieves a high compression ratio, but suffers from two major shortcomings. First, it only gives benefits for pages that have been previously visited and that have since been updated. A 1996 study [17] found that this only applies to about 30% of page accesses. Second, it imposes significant costs on the proxy, or the server in an end-to-end approach, which needs to retain older versions of each web page for potential use as reference files in future accesses, possibly for a significant amount of time. Also, additional disk accesses may be required to fetch the reference files.

Delta compression between different pages typically achieves a more moderate compression ratio. One problem is how to identify appropriate reference files for a requested page. A simple scheme for doing this is proposed in [4], based on the directory paths of the URLs. However, the scheme requires several reference files for best compression, which could seriously slow down the throughput of the proxy due to the costs of fetching and processing the reference files for each requested file.

A preliminary evaluation of *rsync* for web access was performed in [27] as part of the *rproxy* project. However, results are limited to different versions of the same URL, and only a few numbers are provided. As mentioned, file synchronization achieves a more limited compression ratio than delta compression. For the case of related files, the result might not be better at all than using *gzip*, due to the more limited similarity between the files. File synchronization techniques typically require the client to send a set of hash values or other information about the reference file to the server, and there is a trade-off between the amount of this data and the achieved compression in the other direction. The primary advantage of file synchronization is that no old versions of pages have to be stored and then fetched from disk by the proxy.

An interesting alternative called *value-based web caching* was very recently proposed in [21]. As in file synchronization techniques, bandwidth savings are obtained by using hash values to refer to blocks of data already known to the client. However, the technique does not require the client or proxy to choose a particular set of reference files, but exploits similarity between the requested page and previously transmitted content independent of file boundaries. This is achieved by using Karp-Rabin fingerprints [11] to identify block boundaries in a consistent manner independent of position as proposed in [15], and keeping a limited amount of state

for each client. Similar techniques have also recently been used in the *Low Bandwidth File System* [18] and the *Pastiche* distributed backup system [6].

Comparing *value-based web caching* to *rsync*, we would expect similar performance in cases where we can reliably select the best reference file, but better performance in cases where it is not clear which previously accessed file contains the similar content. In particular, *value-based web caching* resolves problems due to aliasing, i.e., different URLs returning the same content. Another advantage comes from caching hashes at the proxy, although one could also use an *rsync*-based approach to do the same (see Subsection 3.3).

### 1.3 Our Approach and Contributions

Our main focus is on studying web and proxy server-friendly schemes for delta compression between different pages that achieve good compression without adversely affecting throughput. In fact, we believe that delta compression between different pages has more practical potential than the more commonly studied case of delta compression between different versions of the same page. On the other hand, we conjecture that file synchronization techniques may be the most appropriate approach for delta compression between different versions of the same page, at least in a proxy environment. In this paper, we discuss techniques and provide experimental results for web access based on delta compression and file synchronization. In particular:

- We compare several policies for selecting appropriate reference files for delta compression, and evaluate the achieved compression ratio. Using a large set of *plausible* site visits by clients distilled from NLANR proxy traces, we show that significant average improvements over *gzip* are achieved by the best policy. Moreover, we show that very simple policies achieve close to optimal compression using only one or two reference files visited within the last few minutes, thus allowing for an extremely efficient main-memory based implementation in a web or proxy server.
- We discuss and evaluate the use of file synchronization techniques for efficient web access. We show that they are of limited use for delta compression between different pages, and currently studied improvements in file synchronization techniques are unlikely to change this. On the other hand, we show that file synchronization techniques have potential for delta compression between different versions of the same page, and we discuss how file synchronization tools could be reengineered for additional improvements.

While the chance for a broad adoption of the proposed delta compression schemes at clients and servers is very small, we believe that there is a significant potential for using such techniques in the context of proprietary proxy systems such as those deployed by wireless or dialup service providers. We note that the benefits of the techniques are limited to `text/html` files and do not apply to images and other multimedia objects, though there are other specialized techniques for such objects. According to the traces, `text/html` files made up $40$ to $50\%$ of the data going through the proxy, and about the same amount was due to image files. (Common statistics indicate that embedded images make up almost $70\%$ of all data in a displayed page; however, images have better caching behavior and thus make up a relatively smaller part of the traffic over the internet.) The benefits are in addition to any benefits due to client-side caching, which do not appear in our traces. The schemes we propose are applicable to about $68\%$ of all accesses to `text/html` files going through the proxy, and for those eligible pages we get up to a factor of $2.9$ average improvement over *gzip*. For all pages, the average improvement over *gzip* is up to $1.7$. Benefits are lower if duplicate URLs are removed from site visits; these duplicates in the proxy traces may be due to a changed page or a server that does not support the IF-MODIFIED-SINCE header.

In the next section, we study reference file selection policies for delta compression of different pages. In Section 3 we investigate the utility of file synchronization techniques for web access. Finally, Section 4 provides some concluding remarks.

## 2 Delta Compression Schemes for Site Visits

This section contains the main results of this paper. We are interested in delta compression schemes that encode a requested page in terms of other pages that the client already has in its cache. We restrict ourselves to reference files from the same site that have been very recently accessed by the client as part of the current *site visit*. We define a *site visit* of a client as a set of consecutive accesses to pages on the same site. We show that even with this limitation on the choice of reference files, we can obtain significant average compression benefits over *gzip*, by a factor of $1.7$ for all pages and $2.9$ for "eligible" pages. One challenge in the experimental evaluation is that it is not easy to obtain client traces that can be used to evaluate our schemes, due to privacy concerns.

In our case, we need a large number of site visits that are very recent, so that we are still able to obtain the pages from the origin servers. Since we were unable to get a representative set of end client traces, we decided to try to "distill" a plausible set of client visits from the publicly available NLANR proxy traces, as described later. We then fetched and stored all pages in those visits, and ran simulations based on the stored pages.

We used version 2.0 of the *zdelta* tool [25] to perform delta compression between pages.[1] The *zdelta* tool is carefully optimized for both compression and speed, and supports the use of up to four reference files. We note that in principle a similar compression ratio could probably also be obtained with the *vcdiff* compressor [13] provided that separate Huffman coders are applied to the different fields of the generated instruction stream; however, *vcdiff* currently only supports a single reference file. Both tools achieve throughputs of several MB per second for encoding and up to tens of MB for decoding, similar to the speed of *gzip*.

### 2.1 Reference File Selection Policies

We now define a few simple policies for choosing reference files that we investigate. The policies are very simple and we make no claims of algorithmic originality. Our primary goal is to evaluate their behavior on a large set of traces. Suppose that the client is performing the $i$th page access in the current site visit. The policies for choosing reference files from among the pages previously visited during the same site visit are as follows:

- **last-k:** choose the $k$ pages most recently visited.
- **longest match-k:** choose the $k$ pages whose URL has the longest directory path match with the requested URL; ties are broken by taking the more recently visited page.
- **best-k:** choose the $k$ pages that each individually achieve the best compression when used as the only reference file for the requested page.
- **best set-of-k:** choose the set of $k$ pages that provides the best compression.

We also consider two simple improvements over *last-k* and *longest match-k*, called *last-k*$^+$ and *longest match-k*$^+$, that add the following two rules: (1) if the requested page was previously accessed, then that page is always selected as a reference file, (2) we make sure that all selected reference files are distinct, since there is no benefit in using two identical reference files. Note that if we eliminate duplicates in the traces, then these policies will be identical to the basic ones.

Of course, we can choose from at most $i-1$ reference files for the $i$th page; thus for $i \leq k$ all policies are the same. The *last-k* and *last-k*$^+$ policies were chosen as the simplest heuristics we could come up with, and they also tend to minimizes the time a server or proxy has to retain pages for later use as reference files. The *longest match-k* and *longest match-k*$^+$ policies are almost identical to the policy proposed by Chan and Woo [4], and should achieve very similar performance. The *best set-of-k* policy is clearly optimal, but inefficient since there are $\binom{i-1}{k}$ possible choices of sets. (Finding the best set is NP Complete due to a reduction from Set Cover, under reasonable assumptions about the delta compressor.) The third policy, *best-k*, is not guaranteed to be optimal, but more efficient to implement.

### 2.2 Experimental Setup

To evaluate the policies, we downloaded traces from the NLANR proxy servers and partitioned the traces into "plausible" site visits as follows. In the traces, clients are identified by IDs that are unique during a given day and for a particular proxy. However, these clients are typically not end clients (browsers) but populations of users in a particular organization, and for our purposes we need access patterns for end clients. We defined a site visit with timeout $t$ as a sequence of accesses to the same site by users with a common ID, with at most $t$ minutes between any two consecutive accesses. If $t$ is large, then a site visit to a popular web site is likely to consist of accesses by different end clients with a common ID. However, if we assume some degree of independence between the access times of the different users throughout the day, then by decreasing $t$ we expect to eventually split up most of these visits into shorter visits by different users. We found that after decreasing $t$ to less than an hour, subsequent decreases of $t$ to a few minutes do not result in many additional splits, indicating that most of the site visits at this point are probably due to a single end user. Of course, we cannot claim that the site visits thus obtained are all due to single individuals, but we believe that our heuristic is close enough to allow an evaluation. We note that such *sessionizing* problems have been extensively studied, see, e.g., [22].

We downloaded all NLANR proxy traces for April 22, 2003, and extracted site visits with timeout 10 minutes. We then took a sample of site visits with a total of 74434 pages, which we downloaded and stored on April 24. We excluded any dynamic pages (with parameters in the URL), as those parameters are removed

[1] Available at `http://cis.poly.edu/zdelta/`.

by NLANR for privacy reasons. We see no reason to believe that the approach would not work on dynamic pages; in fact, such pages may be particularly suitable for delta compression as they frequently change in minor ways and share a lot of content with other pages on the site. However, we have not yet obtained good traces to evaluate this. We made sure that our sample has the same distribution of site visit lengths (number of page accesses) as the entire set. Note that the proxy traces did not contain entries for accesses that were already satisfied by client-side caches; this is fine for our purpose since we are interested in benefits beyond those already obtained by client side caching.

The generated site visits did contain a significant number of duplicates, i.e., repeat accesses to the same URL in a single visit with return code 200. These accesses may be due to changed pages or due to servers that are not set up to support the IF-MODIFIED-SINCE feature in HTTP. While some of the duplicates may belong to different end clients, we believe most are due to the same client. In the following, we report results with duplicates included, and later discuss how the results are impacted if they are excluded.

We compare our results to *gzip* as a baseline compressor. Note that *gzip* is not optimized for HTML, and recent grammar-based approaches based on [12] can achieve significantly better results. (In fact, proprietary versions of these algorithms are the basis of the web acceleration system designed by Slipstream Data and deployed by NetZero.) These approaches can also be adapted to exploit similarity between different pages.

### 2.3 Experimental Evaluation of Policies

In Figures 1 and 2 we see the size reductions obtained by the *last-k* and *longest match-k* policies, respectively, for different values of $k$. For the first page access in a site visit, all policies and also *gzip* achieve the same reduction, to about 22% of the original size on average.[2] For the second access, pages are on average reduced to 10% of their uncompressed size. As there is only one available reference file, all policies except *gzip* achieve the same performance. On the third access, policies with $k \geq 2$ start outperforming those with $k = 1$, and on the fourth access policies with $k = 4$ start outperforming those with $k = 2$. Compression improves to about 5% of the uncompressed size after a few pages. (However, since most site visits are fairly short, the impact on the average benefit over all pages decreases as we move to the right, as shown later.) We also see that the policies *last-1*$^+$ and *longest match-1*$^+$ perform significantly better than the base policies. (Note that the base policy *longest match-k* only tries to match the directory part of the URL and does not distinguish between different pages in the same directory; otherwise, *longest match-1* and *longest match-1*$^+$ would be identical.)

[2] If no reference file is given, *zdelta* becomes identical to *zlib*, which itself has a similar performance as the closely related *gzip*.

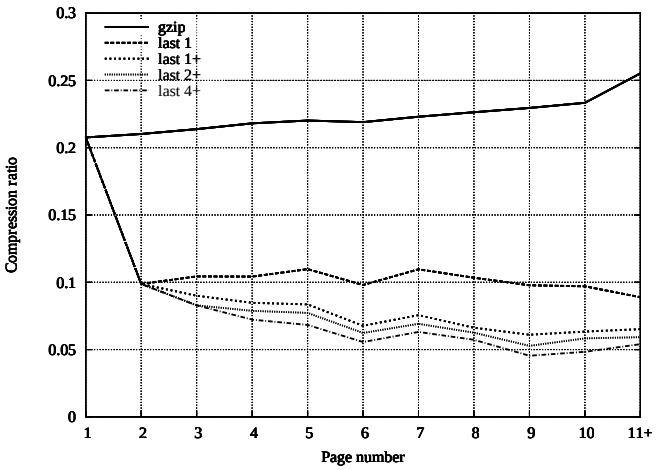


**Figure 1.** Average compressed size for *last-k* policies.

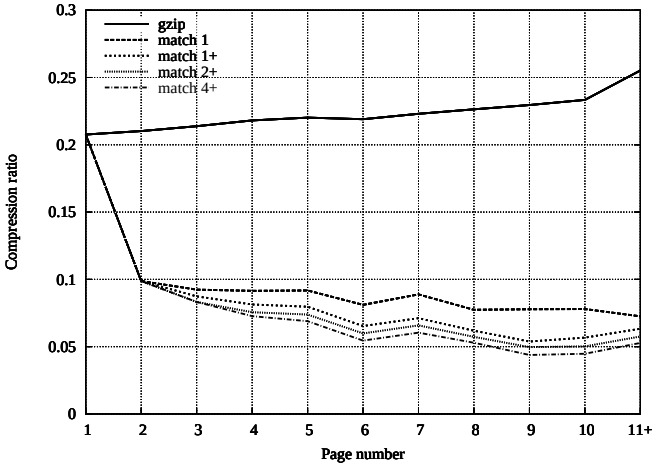


**Figure 2.** Average compressed size for *longest match-k* policies.

In summary, there is significant benefit in using delta compression versus *gzip*, and the benefit increases with the number of reference files and the length of the site visit. One curious detail is that *gzip* compression appears to get slightly worse as more pages on a site are visited; this is primarily due to a decrease in average page size as discussed further below.

In Figure 3 we show the performance of the *best-k* and *best set-of-k* policies, and in Figure 4 we show the performance of a selection of all policies. All of the *best-k* and *best set-of-k* policies have almost the same performance, independent of $k$. Note that while results are only plotted for $k = 1$ and $k = 2$, the benefit from adding a third file would clearly be less than that for adding the second file (which itself is close to zero),

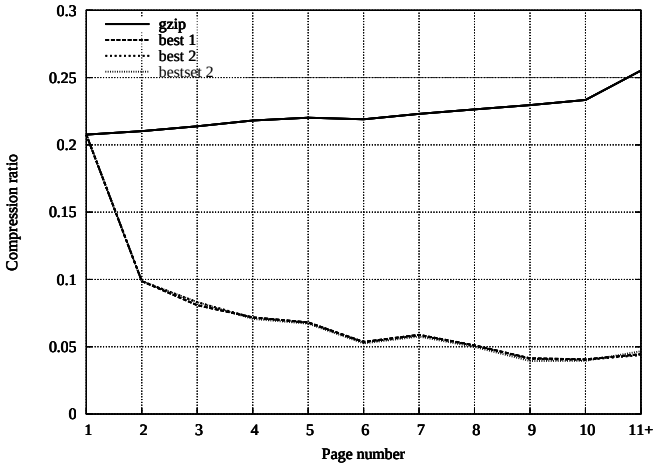


**Figure 3.** Average compressed size for *best-1*, *best-2*, and *best set-of-2* policies. The three policies result in almost identical graphs.

due to the way current delta compressors works. The interpretation of these results is important. In general, using more than one reference file can improve delta compression for two reasons: it could be that each reference file contributes to the compression of the target file (e.g., a target file could be similar to one reference file in the first half, and to another reference file in the second half), or it could be that by using several files we simply have a better chance of including the one reference file that contributes most of the benefit.

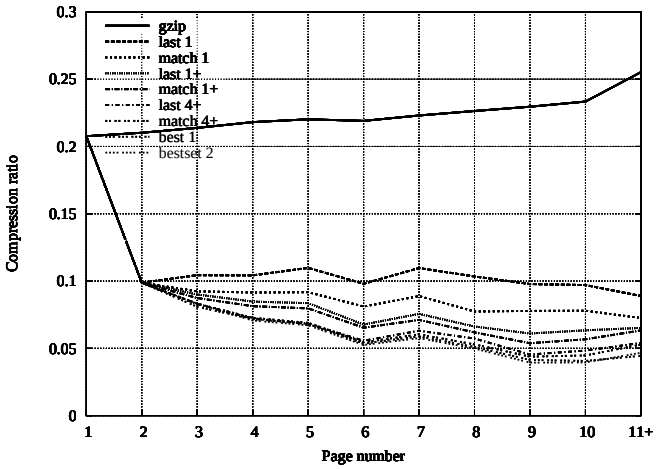


**Figure 4.** Average compressed size for various policies. The order of the graphs is (from the top) *gzip*, *last-1*, *longest match-1*, *last-1*$^+$, *longest match-1*$^+$, *last-4*$^+$ and *longest match-4*$^+$, and *best-1* and *best set-of-2*.

The above results implies that for our application it is primarily the second reason, since by choosing the best reference file we are getting essentially all of the possible benefit. A similar observation was also recently made in [8] in the context of file system compression. The *last-k* policy, and also the *longest match-k* policy proposed in [4], are not really good at identifying the best reference file, but by increasing $k$ we improve our chances of including the best reference file in the list of selected pages. For efficiency it would be preferable if we could directly identify the best reference file, rather than use up to 4 reference files or try all different files as in the *best-1* policy; we address this issue later.

Comparing *last-k* and *longest match-k*, we see that *longest match-1* is better than *last-1* at identifying good candidates; however, this advantage largely disappears if we use the improved policies *last-k*$^+$ and *longest match-k*$^+$ instead. Thus, it seems that in practice the directory-based heuristic proposed in [4] really does not do much better than a trivial heuristic such as *last-k*$^+$.

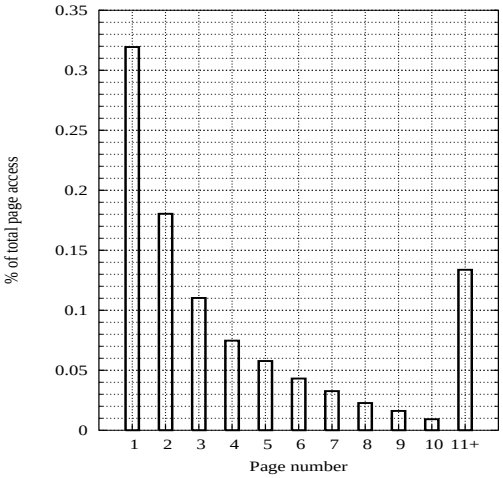


**Figure 5.** Distributions of page accesses in our traces.

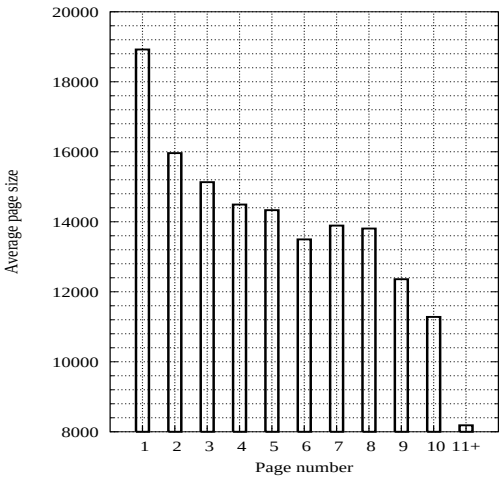


**Figure 6.** Distributions of page sizes in our traces.

Figures 5, 6, and 7 show the distribution of page accesses, page sizes, and transmitted bytes over the different classes plotted in the earlier charts. From the left chart, we see that 32% of all page accesses are first accesses in a site visit, 18% are second accesses, and finally about 13% are 11th and higher accesses in a visit. Thus, most site visits are fairly short. Since delta com-

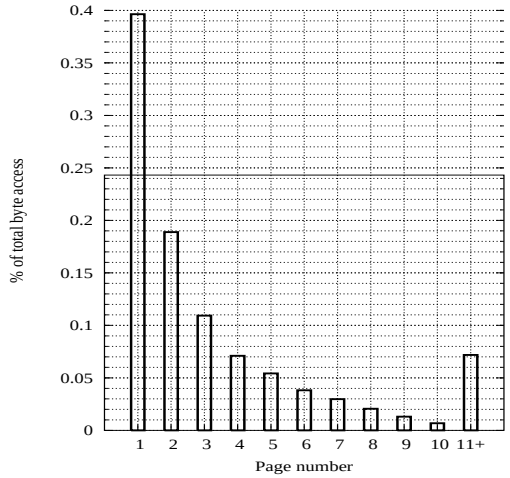


**Figure 7.** Distributions of total bytes in our traces.

pression is not possible for the first page access under our schemes, this means that about 68% of all page accesses are eligible for delta compression. From the middle chart, we see that page size significantly decreases for subsequent page accesses in a visit; this is the reason why the compression achieved by *gzip* shown in Figures 1 to 4 deteriorates slightly on longer site visits. (We are somewhat surprised by this phenomenon and do not have a good explanation yet.) In the right chart, we see the distribution of total bytes over the page accesses; due to the skew in page size the first pages in the site visits together account for about $40\%$ of all bytes transmitted. Thus, about $60\%$ of all transmitted bytes are eligible for delta compression.

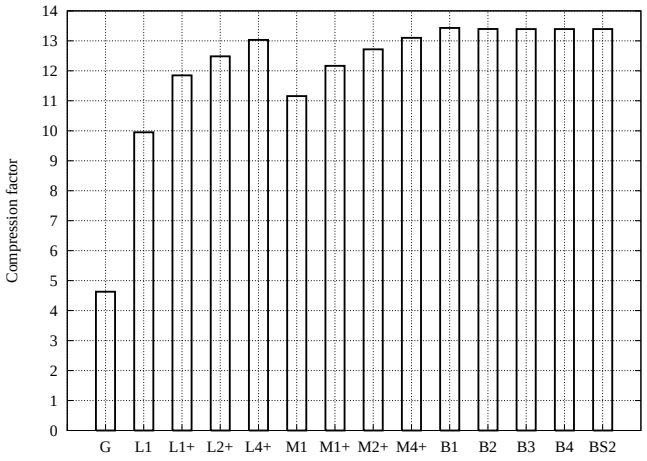


**Figure 8.** Average compression for policies on eligible pages. The policies are *gzip*, *last-1*, *last-k*$^+$, *longest match-1*, *longest match-k*$^+$, *best-k*, and *best set-of-2*.

In Figures 8 and 9, we show the average compression factor, in terms of total transmitted bytes, over all eligible page accesses and all page accesses achieved by the various policies. Compression ranges from a factor of $4.5$ for *gzip* to more than $13$ for the best policy for eligible pages. We note that the various policies are actually much closer in terms of average benefit than suggested by Figures 1 to 4, since most transmitted bytes fall into the left half of those figures. In particular, even *last-1*$^+$ and *longest match-1*$^+$ perform quite well.

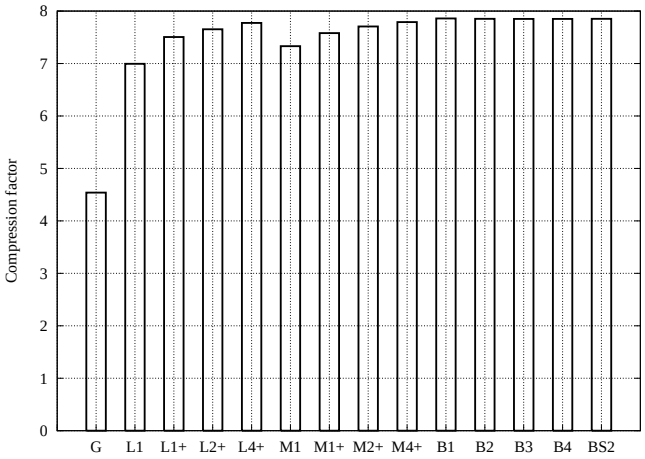


**Figure 9.** Average compression for policies on all pages.

### 2.4 An Efficient and Nearly Optimal Protocol

We now address the problem of how to efficiently identify the reference file that provides the best compression ratio. Our proposed solution is very simple, and uses random hash functions to create fingerprints for files according to a technique proposed by Broder in [2]. This technique has been previously used to cluster documents in [3], and was shown in [8, 20] to provide reasonable estimates for the size of a delta between two files. In particular, we hash all substrings of length $4$ in a file to integer values, and then retain the $s$ smallest hash values. (The method does not seem to be very sensitive to the length of this substring.) We then estimate the similarity of two files by intersecting their samples. Given a page request, we select the reference file that is most similar to the requested page under this measure.

In the first experiment, we used this sampling technique to approximate the *best-1* policy, which as we saw earlier is essentially as good as the best policy, *best set-of-k*, in practice. In Figure 10 we see the results for the standard *best-1* policy and for the sampling based approach with sample sizes of 10, 30, and 100. We find that even with sample size 10, the compression performance is almost as good as that of the standard *best-1* policy. This this technique can also be combined with the other policies to give the following simple protocol:

- The proxy stores each page that it sends to the client for a limited period of time, together with a fingerprint of each page. Pages are kept in

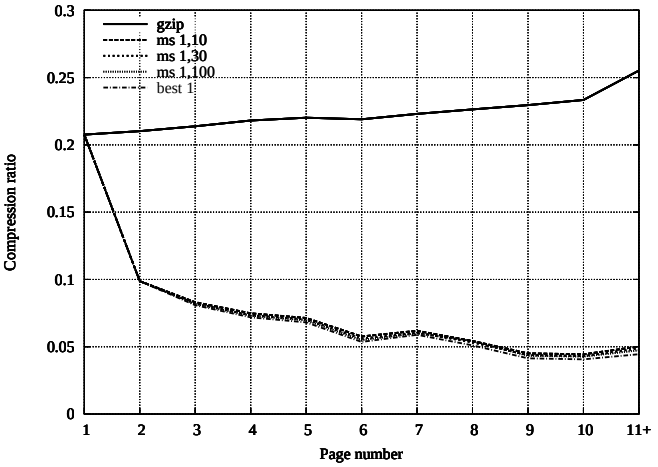


**Figure 10.** Performance of a sampling based approach for determining the most similar file, for sample sizes 10, 30, and 100.

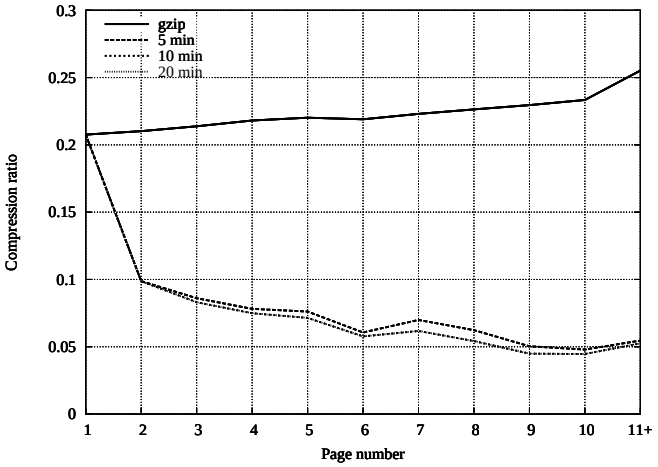


**Figure 11.** Performance of a sampling based approach for determining the most similar file, for histories of length 5, 10, and 20 minutes. The latter two cases are almost identical.

main memory indexed by MD5 hash, and may be deleted by the proxy at any point in time.

- A client sends each page request to the proxy accompanied by several MD5 values of previously visited pages, called *candidate pages*. These candidate pages can be selected by the client based on *last-k*$^+$, *longest match-k*$^+$, or any other policy.
- The proxy checks which of the candidate pages it still holds in main memory, and uses the most similar of those as reference file.

Thus, if the client uses the *last-4* policy to identify candidates, then the resulting compression performance will closely track that of the standard *last-4* policy, using only a single reference file. Note from Figures 8 and 9 that both *last-4* and *longest match-4* achieve average compression quite close to the optimal, making them viable approaches. For the proxy (or server in an end-to-end approach), this scheme is highly efficient as only a single reference file is used, and as pages only need to be retained for a few minutes.

In Figure 11 we show the performance for the case where only pages accessed in the preceding few minutes are used as candidate files; the achieved compression ratio is very close to optimal even for windows of 5 and 10 minutes. (While site visits were defined by us as having at most 10 minutes between consecutive accesses, most are actually much closer in time. This also supports our contention that our distilled site visits are mostly due to a single end client.)

We note two drawbacks of this scheme. First, the computation of the samples could become a bottleneck, though we believe this can be handled through careful optimization. Second, the most similar reference file can only be selected once the entire requested page has arrived at the proxy. In contrast, in policies such as *last-k* and *longest match-k* the proxy can select the reference files as soon as it receives the request. These can then be immediately inserted into the hash table of the delta compressor, and the requested page can be streamed through the compressor and forwarded as it arrives from the server.[3] This second drawback might make schemes such as *last-k*$^+$ or *longest match-k*$^+$ for $k = 1$ or $k = 2$ more attractive in many scenarios.

### 2.5 Impact of Duplicates

We now look at how results change if we remove any page accesses with return code 200 that go to a page previously accessed in the same site visit. Recall that such accesses may be due to changed pages or due to servers not supporting IMS. Table 1 shows the compression ratios achieved in this case. As we see, benefits of delta compression are reduced, but still significant. In particular, the average benefit over *gzip* is now a factor of about 1.4 instead of 1.7 for all pages, and 1.9 instead of 2.9 for eligible pages under the best policy.

### 2.6 Summary of Observations

The main observations from the experiments in this section are as follows: (1) even very simple schemes for selecting reference files achieve significant compression over *gzip* and come close to the optimum, (2) there seems to be only very limited benefit for the directory matching technique in [4] over other simple heuristics, (3) essentially all of the potential benefit can be achieved with a single reference file and there are sim-

[3] Under realistic assumptions about reference file size and with slight modifications to the interface of the *zdelta* delta compressor.

| | With duplicates | | Duplicates removed | |
|---|---|---|---|---|
| Policy | Eligible | All pages | Eligible | All pages |
| GZIP | 4.54 | 4.63 | 4.64 | 4.72 |
| L1 | 9.95 | 7.00 | 7.91 | 6.31 |
| L1+ | 11.85 | 7.51 | 7.91 | 6.31 |
| L4+ | 13.03 | 7.77 | 8.78 | 6.62 |
| M1 | 11.16 | 7.33 | 8.12 | 6.39 |
| M1+ | 12.17 | 7.58 | 8.12 | 6.39 |
| M4+ | 13.10 | 7.79 | 8.84 | 6.64 |
| MS 1,10 | 13.02 | 7.82 | 8.73 | 6.61 |
| B1 | 13.43 | 7.86 | 9.02 | 6.71 |

**Table 1.** A comparison of compression factors achieved with and without duplicates, for policies *last-1*, *last-k*$^+$, *longest match-1*, *longest match-k*$^+$, *most similar-1* with sample size 10, and *best-1*.

ple sampling based methods for identifying this file, (4) however policies such as *last-1*$^+$ and *longest match-1*$^+$ perform quite well and might be preferable in practice for other technical reasons, and (5) benefits are somewhat lower if page size distribution is taken into account and duplicates are removed.

## 3 Utility of *rsync* for Web Access

In this section, we study the potential for using *rsync* and other file synchronization techniques for web access. By file synchronization, we refer to techniques where the server has the requested pages, but not the reference file held by the client. The most widely used tool for file synchronization is *rsync* [28], which uses a single round-trip between client and server as follows: the client partitions the reference file into blocks of a few hundred bytes, and sends a hash value for each block to the server. (The hash value has a strength of 6 bytes for common file sizes.) The server then sends the requested file to the client, but replaces any block that hashes to one of the received hash values by a reference to the hash. This method is then combined with standard *gzip* compression. The cost is given by the size of the hashes and the size of the encoded page, with a trade-off between the two. The size of the encoded page sent to the client is clearly lower-bounded by the size of a delta.

In fact, measurements in [23] show that the size of the data sent from server to client in *rsync* is usually significantly larger than a delta. This raises the question of whether file synchronization techniques are efficient enough for web access. In the following, we present results for related pages and versions of the same page.

### 3.1 Experiments for Related Pages

Figure 12 shows results for related pages, using the *last-1* and *longest match-1* policies and several block sizes in *rsync*. We also show results for the *most similar-1* policy, with and without duplicates, to get an upper bound on the possible benefit, even though this policy is not realistic as the selection of reference files for *rsync* is performed at the client. (We used *most similar-1* instead of *best-1* for efficiency but the result should be very similar. One caveat is that we did not adapt the substring size in the sampling step to the block size of *rsync*; we plan to correct this in the final version.)

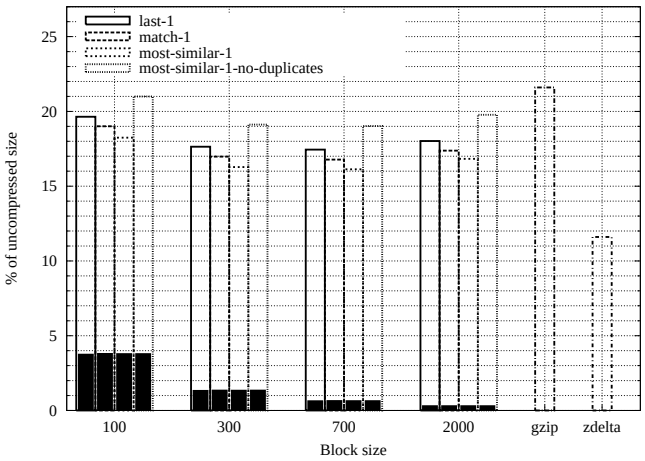


**Figure 12.** Performance of *rsync* under the *last-1*, *longest match-1*, and *most similar-1* policies for different block sizes.

We see from Figure 12 that *rsync* performs only slightly better than *gzip* if we consider the total data sent in both directions. Even for the *most similar-1* policy with block size 700, files only reduce on average to about 16% of their original size, compared with 22% for *gzip*. If we exclude duplicate URLs, then there is hardly any benefit over *gzip* at all even for the *most-similar-1* policy (about 19% vs. 22% for *gzip*). Performance is best for block sizes between 300 and 700 bytes.

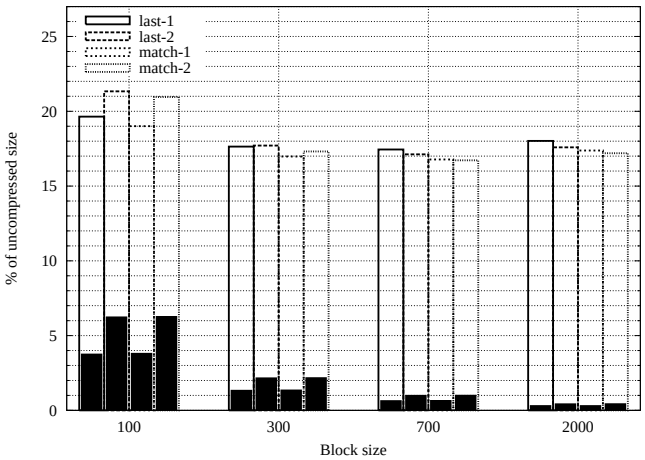


**Figure 13.** Performance of *rsync* for one and two reference files with different block sizes.

In Figure 13 we investigate the use of more than one

reference file in connection with *rsync*. To use *rsync* on two reference files, we concatenated the two files into one longer file. We do not show results for more than two reference files as there clearly would be no benefit in this. We note that the amount of data sent from client to proxy is proportional to the number of blocks, and thus increases with the inverse of the block size. We see that for block sizes of 700 and 2000 bytes, adding a second reference file gives some negligible benefits under the *last-k* and *longest match-k* policies. Benefits are small because the increase in *read size* cancels out most of the decrease in *write size*.

In general, the reason for the disappointing performance of *rsync* on related pages on the same site is that these pages, while overall similar, differ in a number of places. While there is some benefit for aliased and very similar pages, in general the fine granularity of differences does not allow *rsync* to find large matches at the level of blocks of several hundred bytes, while block sizes of 100 bytes or less would greatly increase the number of hashes sent by the client. We note that Tridgell, in Section 4.4.3 of [26], proposes to use the Burrows-Wheeler transform to increase the locality of changes in updated files. This approach works well when files are updated by replacing all occurrences of a string by another string (e.g., renaming of a variable or a consistent change in format such as line breaks), but it does not appear to work well at all in our scenario. In fact, we observed a significant decrease in performance under this approach, as it also has the reverse effect of spreading out blocks of changed bytes all over the transformed files. We conclude that simple *rsync* is not a good approach for compression between related files.

### 3.2 Experiments for Versions of a Page

We now consider the case of several versions of the same page. Our NLANR traces are not very useful for this case since the client IP hashes change from day to day. Since we did not have alternative traces, we instead used data obtained from repeated web crawls of a certain subset of pages. In particular, we chose pages at random from two large page collections of more than 100 million pages each, one from the Internet Archive and one from our own crawls. The selected pages were than crawled every night for several weeks in Fall 2001. In the following, we use four versions of a subset of 10000 of these pages: one base set and three updated versions crawled 2, 20, and 72 days later. In the experiments, we measure the benefit of using *rsync* to fetch an updated page given that the client already has an outdated version from the base set. We give average results for all pages and for only those that have changed.

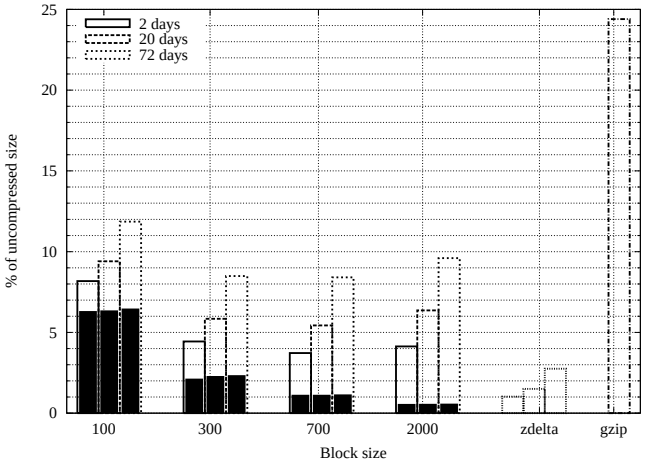


**Figure 14.** Performance of *rsync* on all pages for 3 different time intervals.

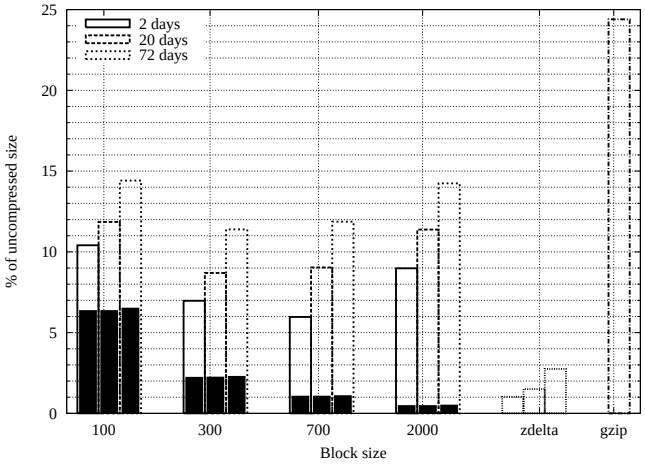


**Figure 15.** Performance of *rsync* on changed pages for 3 different time intervals.

Figure 14 and 15 show the result for the different block sizes. As we see, even for pages that are revisited 72 days later, we still get significant benefit over *gzip*, by about a factor of 2 even if unchanged pages are excluded. On the other hand, delta compression does even better then *rsync*, by another factor of about 4. We note that user surfing behavior is likely to be biased towards sites that are frequently updated, and thus our results for a random sample of pages may be overly optimistic. Nonetheless, there seems to be significant potential for using *rsync* between different versions of a page, as previously suggested by the *rproxy* project [27]. Note that examination of the pages shows that page updates tend to be highly clustered within a page; this explains the good performance of *rsync* in such cases.

### 3.3 Discussion and a Generalized Approach

The results indicate that file synchronization using *rsync* is efficient for different versions of the same page, but

performs only slightly better than *gzip* on related pages from the same site. Of course, this could be due to shortcomings in the *rsync* algorithm, and there are several proposals for improved file synchronization techniques [5, 19, 24]. However, these proposals rely on additional roundtrips and would not be appropriate for a web access scenario since modem latencies are high and objects are not very large. It appears difficult to achieve significant bandwidth savings over *rsync* without incurring more than the single roundtrip used by *rsync*. This suggests that an approach that uses delta compression between related pages in a site visit, and file synchronization between different versions of the same page, might be a good combination. In such a scheme, the server or proxy only needs to keep old files for a few minutes, and still achieves good performance for different versions of the same page using *rsync*. This leads us to the following ideas and observations:

- Instead of sending block hashes to the proxy with each request under the *rsync* protocol, we could allow the proxy to compute and store block hashes for the references files as they pass through the proxy. In this case, the approach becomes very similar to *value-based web caching* [21], except for the use of fixed rather than value-based block boundaries. We can estimate the benefit by deducting the dark portion of the bars from the charts in this section. Some additional benefits might arise since there is no need anymore to choose a small set of reference files for each access.
- In the case of the Low Bandwidth File System [18], the use of value-based block boundaries is crucial due to the fact that files have to be transmitted in both directions. In our scenario, where files are only sent from proxy to client, we can also use fixed boundaries based on position and then try to match these blocks with all positions in the file to be encoded, as done in *rsync*.
- When caching hash values at the proxy, it might be interesting to consider a "multi-resolution" approach for the block size: when sending a file to the client, we initially compute and store hash values for blocks of fairly small size, say 32 bytes. This takes a lot of space at the proxy, but allows compression of similar files that is much better than that achieved with block sizes of several hundred bytes as in *rsync*. To limit space consumption, instead of evicting hashes we can combine two adjacent hashes into one hash for a larger block, provided that the hash function is *composable* (i.e., the hash of a block can be computed from the hashes of the left and right half). Alternatively, we could for each file generate hashes at different levels of granularity, and first evict fairly old hashes of small size. (This takes at most a factor of 2 more space than a combining approach and removes some other complications.)

Based on these ideas, we plan to investigate an approach that combines delta compression, value-based caching, and synchronization, as follows: Transmitted files are cached for a short time to allow delta compression, while block hashes for the files are cached for longer periods based on the above multi-resolution approach. Thus, hashes for small blocks are kept for a shorter period of time, and hashes for large blocks for longer periods. In addition, we could also allow the client to send hashes in some cases, such as when a page is revisited after several days and the old hashes are likely to have been evicted at the proxy. There are a number of questions to explore in this context, such as the advantages and disadvantages of fixed and value-based block boundaries and the details of the multi-resolution approach, and we plan to address these in our future work.

## 4 Concluding Remarks

In this paper, we have studied the performance of simple delta compression and file synchronization schemes for efficient web access. Our focus was on web and proxy server-friendly schemes that do not require the storage and retrieval of old versions of web pages at the server. Our main conclusion is that there is significant benefit to using delta compression between pages on the same site, and we gave several low-overhead schemes that improve significantly over *gzip*. On the other hand, we found that single-roundtrip file synchronization techniques such as *rsync* obtain good compression between different versions of a page at low overhead, but do not significantly improve over *gzip* for related pages.

We are working on several extensions of this work. We plan to study the approach combining delta compression, value-based web caching, and file synchronization described in Subsection 3.3, and to integrate it into a dual proxy architecture called *SPAWN*[4] that we have implemented at Polytechnic University. We are also working on improved software tools for file synchronization and plan to evaluate these in the current context and for content distribution networks and large replicated document collections.

[4] See `http://cis.poly.edu/spawn/` for more details.

**Acknowledgements**

We thank Dimitre Trendafilov for help with experiments, and the anonymous referees for helpful comments. This research was supported by NSF CAREER Award NSF CCR-0093400, Intel Corporation, and New York State through the Wireless Internet Center for Advanced Technology (WICAT) at Polytechnic University. Proxy traces were provided by the IRCache Project, which is supported by the National Science Foundation (grants NCR-9616602 and NCR-9521745) and the National Laboratory for Applied Network Research.

## References

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

[2] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, pages 21–29. IEEE Computer Society, 1997.

[3] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Sixth Int. World Wide Web Conference*, 1997.

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

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

[6] 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*, 2002.

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

[8] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proc. of the USENIX Annual Technical Conference*, June 2003.

[9] 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.

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

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

[12] J. Kieffer and E. Yang. Grammar based codes: A new class of universal lossless source codes. *IEEE Trans. on Information Theory*, 46(3):737–754, 2000.

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

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

[15] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. of the 1994 Winter USENIX Conference*, pages 23–32, January 1994.

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

[17] 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.

[18] 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.

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

[20] Z. Ouyang, N. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. In *Third Int. Conf. on Web Information Systems Engineering*, December 2002.

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

[22] M. Spiliopoulou, B. Mobasher, B. Berendt, and M. Nakagawa. A framework for the evaluation of session reconstruction heuristics in web usage analysis. *INFORMS Journal on Computing*, 15, 2003.

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

[24] T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. July 2003. unpublished manuscript.

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

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

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

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

[29] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox. Removal policies in network caches for World-Wide Web documents. In *Proc. of the ACM SIGCOMM Conference*, 1996.