# Cache-based Compaction: A New Technique for Optimizing Web Transfer

Mun Choon Chan Thomas Y.C. Woo
Networking Software Research Department
Bell Laboratories
{munchoon,woo}@research.bell-labs.com

*Abstract*—**In this paper, we propose and study a new technique, which we call *cache-based compaction* for reducing the latency of Web browsing over a *slow* link. Our compaction technique trades computation for bandwidth. The key observation is that an object can be coded in a highly compact form for transfer if *similar* objects that have been transferred earlier can be used as *references*.**

**The contributions of this paper are: (1) an efficient *selection* algorithm for selecting similar objects as references, and (2) an *encoding/decoding* algorithm that reduces the size of a Web object by exploiting its similarities with the reference objects. We verify the efficacy of our proposal through detailed experimental evaluations. Our compaction technique significantly generalizes previous work on optimizing Web transfer using compression or differencing, and provides a systematic foundation that ties together caching, compression and prefetching.**

## I. Introduction

Despite the phenomenal growth of the Internet, the advance in the speed of access to the Internet has not caught up. In particular, the majority of *last hops* are still using traditional modem, with bandwidth up to only 56kbps. Separately, the use of wireless channel as the last hop is gaining popularity. Again, the raw bandwidth available on most wireless channels is low (e.g., 19.2kbps for CDPD). The bandwidth can be further reduced by multiple-access contention and protocol overhead. For example, the effective application layer throughput of CDPD is about 8kbps without contention. In a nutshell, Web browsing behind slow (wireline or wireless) access links will persist for years to come.

From an end user's perspective, her primary measure of browsing performance is response time or latency. Strictly speaking, latency is an end-to-end quantity, which consists of two main components, namely, *processing delay* and *transport delay*. The former refers to the processing time incurred in the origin server and all the intermediate proxy cache servers (see Figure 1), while the latter refers to the time spent in traversing all the interconnecting links.

In this paper, we consider the case where the transport delay is dominated by the delay incurred in the last hop. Our objective is to reduce the overall latency by reducing the transport delay, or specifically the last hop delay.

The key innovation behind our cache-based compaction technique is as follows. Instead of "coding" the requested object on its own, a more compact encoding is performed by leveraging other objects that are already available in the client's possession. In particular, if a client already possess "similar" objects in its cache, then those objects (called *reference* objects) is used as an extended "dictionary" based on which the newly requested object may be coded. The more "similar" the reference objects are to the requested object and the more such "similar" reference objects are available in the client's possession, the smaller is the resulting transfer.

Our approach is a compression technique in that it uses a dictionary-based compression technique. However, unlike standard compression techniques such as *gzip*, a set of "similar" objects, not just the requested object itself, is used as the compression dictionary.

Our approach can also be viewed as a differential transfer technique in that it compares objects, and transfers mainly the differences. However, unlike existing differential transfer techniques, comparison is not restricted to just objects from earlier versions. Our approach can potentially leverage multiple objects (with completely different URLs) in the client's cache.

To be accurate, our cache-based compaction idea represents a general approach rather than a specific algorithm. At a high-level, it consists of two key components: (1) a *selection* algorithm for choosing reference objects, and (2) an *encoding/decoding* algorithm that encode and decode a new object using a collection of reference objects. A specific compaction technique is obtained by providing concrete implementations of the selection and encoding/decoding algorithms.

In this paper, we examine an instantiation of the cache-based compaction idea. It uses: (1) an efficient selection heuristic based on the structure of URL as the selection algorithm, and (2) a *gzip*-like dictionary-based compression scheme as the encoding/decoding algorithm. We demon-

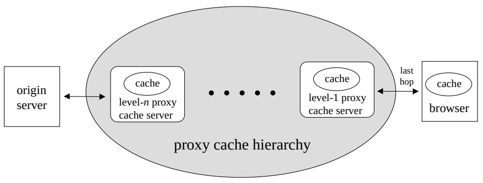

Fig. 1. System Model

strate that this instantiation provides significant improvement over existing techniques. We validate our ideas using both random sampling on different Web sites, as well as actual user traces.

The balance of the paper is organized as follows. In the next section, we review related work addressing similar problems. In Section III, we provide a detailed description of the various algorithms in our compaction technique. In Section IV, we evaluate our proposal by presenting extensive experimental results. Apart from its use for optimizing Web transfer, the cache-based compaction idea can also be viewed as a systematic foundation that ties together the three most-used, yet completely decoupled, techniques — caching, compression, and prefetching — for improving Web browsing. Finally, we conclude in Section V.

## II. Related Work

The major techniques used for optimizing Web transfer are compression, caching, differencing and prefetching.

**Compression** can be divided into lossy and lossless. Lossy compression is usually applied to graphical and audio objects, and lossless compression is applied to text and binary objects. The benefits of using lossless data compression algorithms such as gzip (which is based on LZ77 [15]) and vdelta [10] to compress non-video and non-audio objects is studied in [12].

The use of data-specific technique for reducing object size is described in [6]. Reduction was achieved by lossy compression, for example by reducing resolution and/or color of a graphics object. The *Mowgli* architecture [1] uses compression and prefetching for reducing Web access latency. The idea of content-type specific compression is similar to [6].

**Caching** is frequently used to improve the performance of distributed systems. Caching algorithms search for identical object. This topic has been studied extensively in the literature, see for example [4], [5], [8] and [14].

**Differencing** compares an earlier version of an object to the current version. Usually, only two objects of the same URL or output of CGI script with different parameters are considered. Some of the differencing algorithm used are UNIX *diff* and *vdelta* [10]. In [2], the issue of what objects should be used in differencing was mentioned as an open question. This is a question which we provide an answer to in this paper.

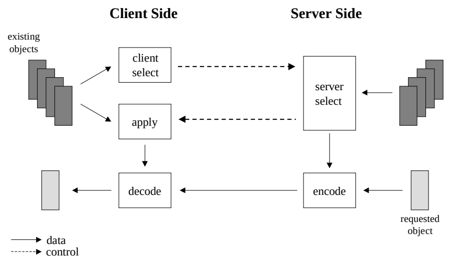

Fig. 2. System Overview

The benefits of delta coding is also studied in [12]. The authors found that differencing worked for 10% of all “status 200” response at the proxy level. The *WebExpress* system described in [9] included a number of techniques, the relevant ones being file caching and forms differencing. Object comparison was based on the object’s URL as well as a digital signature of the object. Differencing was applied mainly to output of CGI scripts.

In **Prefetching**, an object that might be needed in the future is fetched in advance. The utility of prefetching is studied in [13] using a statistical algorithm described in [7]. The bounds of latency reduction from caching and prefetching, based on search for objects with the same URL, is studied in [11]. The authors found that caching and prefetching could reduce latency by at best 26% and 57% respectively.

## III. Our Cache-based Compaction Technique

In the following, we first give an overview of our proposed technique. Then in Sections III-B and III-C, we present the specific algorithms we have used in this study.

### *A. Overview*

We first provide a generic description of our technique. Let $C$ be a client and $S$ a server.[1] Let $C$ contains a set of objects denoted by $C.cache$, and that $C$ would like to obtain a new object $o$ from $S$. Let $S$ contains a set of objects denoted by $S.cache$, where $o \in S.cache$.[2]

Instead of sending $o$ to $C$, $S$ computes a new object $o'$ using $o$ and $o_1, \ldots, o_n$ where $\{o_1, \ldots, o_n\} \subseteq C.cache \cap S.cache$, and sends $o'$ to $C$. We call $o_1, \ldots, o_n$ the *reference* objects. On receiving $o'$, $C$ recovers $o$ from $o'$ and $o_1, \ldots, o_n$. The computation of $o'$ by $S$ is the encoding step, while the reconstruction of $o$ by $C$ is the decoding step. The algorithms used in the encoding and decoding steps satisfy the following relationship:

$$o = \text{decode}(\text{encode}(o, o_1, \ldots, o_n), o_1, \ldots, o_n)$$

[1] $C$ and $S$ can be any of the entities in the process chain shown in Figure 1. $C$ and $S$ need not even be adjacent if some form of “tunneling” is available. The most interesting case would be when $C$ is the browser and $S$ the first level proxy cache server.

[2] The requirement that $o \in S.cache$ is a simplification. $S$ can fetch $o$ on demand if necessary.

```
function encode ( $o, o_1, o_2, \ldots, o_n$, threshold ) {
    $i$ = 1;
    $o'$ = empty string;
    while ( $o[i..]$ is non-empty ) {
        $o_0 = o[..i-1]$;
(1)     $CS$ = $\{(\ell, j) \mid \ell$ is a prefix of $o[i..]$ and
        $\ell$ occurs in $o_j, 0 \leq j \leq n\}$;
        if ( $CS = \emptyset$ )
            $L = o[i]$;
        else
(2)         pick $(L, k) \in CS$ such that $\forall(\ell, j) \in CS : |L| \geq |\ell|$;
        if ( $|L| \leq$ threshold )
            append to $o'$ the character token $L$;
        else
            append to $o'$ the triplet token
            ( $k$, starting position of $L$ in $o_k$, $|L|$ );
        $i = i + |L|$;
    }
    return o';
}
```

Fig. 3. Encoding Algorithm

When $n$ = 0, it reduces to essentially a compression technique. When $n$ = 1 and $o_1$ is an earlier version of $o$, it reduces to previously studied delta encoding technique. In other words, our compaction technique subsumes most existing proposals and is most interesting when $n > 1$ and $o_1, \ldots, o_n$ are not simply variations of $o$.

In this paper, we use a dictionary-based compression scheme as our encoding and decoding algorithms. Specifically, we uses $o_1, \ldots, o_n$ as "extended" dictionaries for compression of $o$. The objects $o_1, \ldots, o_n$ are determined via a *selection* algorithm which tries to identify objects that are "similar" to $o$. The measure of similarity is the number and length of common substrings.

Obviously, saving is possible with our compaction technique if and only if

$$t_{\text{select}} + t_{\text{encode}} + t_{\text{decode}} + \frac{|o'|}{s} < \frac{|o|}{s}$$

where $|o|$ denotes the size of an object $o$ and $s$ is the bit transfer rate on the link between $C$ and $S$. A necessary condition for this is $|o'| < |o|$, and the absolute reduction in latency is proportional to the size of $o$ and inversely proportional to $s$. Hence, our compaction scheme will make most sense when transferring Web responses in the last hop, where $o$ is of reasonable size and $s$ is typically small.

The underlying observation is that dictionary-based compression scheme (the most well known being the LZ77 [15] and LZ78 [16] family) works because of the recurrence of common sub-strings within a document. The basic idea in our proposal is to exploit this notion of *similarity* among multiple documents for reducing transfer. If a number of similar documents have already been transferred from the web server to the client, transfer of the next similar document can be done in an efficient manner.

```
function decode ( $o', o_1, o_2, \ldots, o_n$ ) {
    $o$ = empty string;
    while ( $o'$ is non-empty ) {
        remove first token $t$ from $o'$;
        if ( $t$ is a character token )
            append $t$ to $o$;
        else {
            /* $t$ must be a triplet token */
            let $t = (k, pos, l)$;
            append to $o$ the substring in $o_k$ starting at
position $pos$ of length $l$;
        }
    }
    return $o$;
}
```

Fig. 4. Decoding Algorithm

## B. Encoding and Decoding Algorithms

The encoding and decoding algorithms are based on the universal compression algorithm described in [15].

The encoding algorithm is shown in Figure 3. An object $o$ is essentially viewed as a byte array, with $o[i]$ denoting the $i$-th byte of $o$ (we start counting from 1), $o[..i]$ denoting the part of $o$ from the beginning up to and including the $i$-th byte, and $o[i..]$ denoting the part of $o$ beginning at $i$-th byte of $o$ to the end.

The parameter threshold should be set to at least the encoded size of a triplet (whose size is at least 1) to ensure that the size of the compressed result is smaller than the original.

The steps (1) and (2) represent the searching of the longest common substring between the part of $o$ currently being processed and the part of $o$ that has been processed ($o_0$ specifically) together with the $n$ reference objects $o_1$, $\ldots$, $o_n$. The more similar the reference objects are to $o$, the more common substrings there are, and the better is the compression. This is the most time consuming part of the encoding algorithm, and is implemented using hash tables in our case.

In general, compression gets better with larger $n$, though the marginal improvement diminishes. The case when $n = 0$ is basically the *LZ77* algorithm. In that case, an ordered pair token, instead of a triplet, is sufficient.

Decoding is straightforward and is comparatively much faster. Its detail is shown in Figure 4, and should be self-explanatory.

It should be clear that the above encoding algorithm is lossless. Though it works for any objects, it is most applicable to text (e.g., plain ascii, HTML) objects. For graphical objects that are already in compressed form (e.g., GIF, JPEG), the amount of non-trivial similarity among objects is minimal. Lossy compression techniques can drastically reduce the size of a graphical object while retaining most of its "visible" quality. Thus in the sequel, we consider the use of compaction on text objects only.

We defer to Section IV to provide the performance numbers for the above encoding and decoding procedures.

### C. Selection Algorithm

In order to obtain good compression result, the selection algorithm needs to be able to pick a set of reference objects that are similar to the requested object. While examining the content of the objects is the only sure way of deciding if they are similar, this process is too computationally expensive, especially when the collection of objects is large (e.g., all the objects in a cache). Therefore, we are left with using heuristics based on other attributes of the objects.

A natural choice for selection parameter is the name or the Uniform Resource Locator (URL) of the object. Generally, the URL does not tell much about an object's content. We argue though that the structure of a URL may provide good enough hints.

By treating URL as path name, a collection of objects can be viewed as leaves in a forest, with all objects from the same site represented in a distinct tree. We observe that a large majority of sites tend to follow a consistent design style, which translates into the use of similar structure and formating instructions. Additionally, the hierarchy is often structured in terms of related topics, and objects pertaining to similar topics tend to share common content.

In summary, we conjecture that Web documents that are "close" together in the hierarchy formed by their URLs tend to be more similar than those that are "far apart." A degenerate case of this is used in the differencing scheme described in [2], [9], [12], where an older version of a document with the same URL is used to compute the delta for transferring a newer version of the document.

To precisely specify our heuristics, we first introduce some notations.

**Notations.** Let $u$ and $u'$ be two URLs.[3] They are written respectively as $h/p_1/p_2/\ldots/p_n$ and $h'/q_1/q_2/\ldots/q_m$. Each of the $h, h', p_i$ and $q_i$ is called a *segment*. The *length* of an URL is defined as the number of segments in the URL. Thus, $|u| = n + 1$. We define an *enumerator* function $[\cdot]$ for URL as follows: $u[0] = h$, and for $1 \leq i \leq n$, $u[i] = p_i$. Additionally, $u[i..j]$ is the path $u[i]/\ldots/u[j]$ $p$ is a *prefix* of $u$ if for all $0 \leq i \leq |p|$, $p[i] = u[i]$. $p$ is a *common prefix* of $u$ and $u'$ if $p$ is a prefix of both $u$ and $u'$. □

We define the *path similarity* between two URLs $u$ and $u'$ as *path-sim*$(u, u')$ be the length of the longest common prefix of $u$ and $u'$ and their *path difference* as *path-diff*$(u, u')$ = $|u|$ + $|u'|$ - 2 * *path-sim*$(u, u')$

[3]Since we consider only HTTP URLs, for ease of our disposition, we assume the protocol part has been omitted.

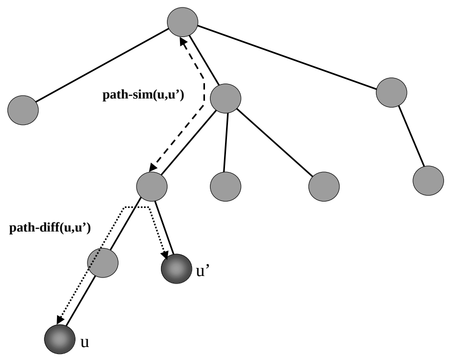

Fig. 5. Path Similarity and Path Difference

These definitions are graphically illustrated in Figure 5. As an example, the URLs http://www.cnn.com/US/news/abc and http://www.cnn.com/US/def have a path similarity of 2 and a path difference of 3.

When the path similarity is 0, it means that the two URLs refer to objects from different Web sites. When path similarity is at least 1, the path difference indicates the number of "hops" it takes to go from one URL to the other in the tree hierarchy. In particular, if the path difference is 2, it means that the two URLs belong to the same directory.

With the above, we are now ready define a similarity relationship. Given a URL $u$, the *similarity ordering* relative to $u$, denoted by $\sqsupseteq_u$, is defined as follows:

$$u_1 \sqsupseteq_u u_2 \quad \text{iff} \quad \begin{array}{c} \textit{path-sim}(u, u_1) \geq \textit{path-sim}(u, u_2) \\ \text{and} \\ \textit{path-diff}(u, u_1) \leq \textit{path-diff}(u, u_2) \end{array}$$

This essentially says that a URL $u_1$ is considered more *similar* to URL $u$ than another URL $u_2$ if $u$ and $u_1$ share a longer common prefix and it takes fewer hops to go from $u$ to $u_1$ than from $u$ to $u_2$. In other words, $u$ and $u_1$ share more common path and fewer disjoint hops. It is easy to see that $\sqsupseteq_u$ is a partial ordering.

A precise selection policy requires a total ordering. Thus, we extend $\sqsupseteq_u$ to a total ordering $\sqsupseteq_u^t$ as follows. If $u_1$ is related to $u_2$ under $\sqsupseteq_u$, then they are related in the same way under $\sqsupseteq_u^t$. Otherwise, $u_1 \sqsupseteq_u^t u_2$ iff *path-diff*$(u, u_1)$ is at most *path-diff*$(u, u_2)$. Clearly, there are multiple ways to extend a partial ordering to a total ordering. This particular definition gives priority to the path difference, and is the one studied in this paper.

Once $\sqsupseteq_u^t$ is defined, the selection algorithm is straightforward. It basically will select the top $n$ most similar URLs from all the URLs available for selection. The pseudo code for the selection algorithm is given in Figure 6. We note that $C$ is the set of URLs that are available for selection,[4] and $n$ is the maximum number of most similar URLs needed. In summary, the selection heuristic

[4]This is typically a subset of the cache.

```
function select ( $u, C, n$ ) {
    /* filter out objects from different sites */
(1) $C = C - \{u' \in C \mid \textit{path-sim}(u, u') = 0\}$;
    $S = \emptyset$;
    while ( $|S| < \min(n, |C|)$ ) {
        $T$ = subset of $C - S$ with minimum path
        difference with $u$;
        $S$ = $S$ + subset of $T$ with maximum path
        similarity with $u$;
    }
    return $S$;
}
```

Fig. 6. Selection Algorithm

tries to minimize the path difference, while maximizing the path similarity. The filtering step (step (1)) removes all the URLs that do not belong to the same site as $u$.

## IV. Experimental Results

In this section, we present the results of our experimental evaluation of our proposed compaction technique. Our experiments are broken down into 3 sets, each of which is intended to establish a distinct claim.

**Set 1.** In the first set of experiments (Section IV-A), we examine if the similarity ordering $\sqsupseteq_u^t$ introduced in Section III-C (or equivalently the selection algorithm shown in Figure 6) does actually pick out "good" reference objects that are useful in the encoding procedure (Figure 3). In other words, we would like to verify our conjecture that similarity in URL implies certain degree of similarity in their content.

For comparison purposes, we perform the same experiments with a standard compression scheme, namely, *gzip* [12], and a standard differencing scheme, namely, *diff -e* | *gzip* (abbreviated as *diff* in the sequel) [2], [9], [12].[5]

**Set 2.** From experiments in Set 1, we demonstrate that objects high in the similarity ordering serves better as reference objects than those low in the similarity ordering. The remaining question to ask is, in a real-life browsing session, how "high" in the similarity ordering can the selection algorithm typically find objects at. In order words, we study the actual distribution of path difference and path similarity in a typical browsing session.

We perform this set of experiments (Section IV-B) using actual client-side access log. Our objective is to demonstrate that typical browsing patterns of actual users contain sufficient locality such that reference objects with high content similarity (as defined by the similarity ordering) are frequently available in the client cache, and hence can be selected.

[5]*gzip* is like compaction with $n$ = 0, but with a number of additional optimizations. *diff* is like compaction with $n$ = 1. Strictly speaking, the differencing schemes that have been proposed and studied apply only to objects with the same URL. We relax this for our comparison.

| Section | mean $\delta$ | mean $\sigma$ | Selected subset of $\Omega$ |
|---|---|---|---|
| IV-A.1 | 0 | $\geq 1$ | traces from periodic downloading |
| IV-A.2 | 2 | $\geq 1$ | traces from CGI output |
| IV-A.3 | 2 | $\geq 1$ | random selection from Web sites |
| IV-A.4 | $> 2$ | $\geq 1$ | random selection from Web sites |
| IV-A.5 | $\geq 0$ | $\geq 0$ | random selection from Web sites |

Fig. 7. Summary of Experiments Performed

**Set 3.** Finally, in the last set of experiments (Section IV-C), we "follow" real user access trace to perform actual downloading of Web objects using compaction. We compute the actual savings and compare that with the results of identical downloading under *gzip* and *diff*. We also compute and compare the average latency incurred by compaction, *gzip* and *diff* under various link bandwidth.

In the following, we refer to our compaction scheme as *npact*($n$), where $n$ is the number of reference objects used. While we have experimented with many different values of $n$, all experiments presented below used $n$ = 3. This value is selected because the performance of *npact*(3) is noticeably better than *npact*(1), while *npact*(7) is only slightly better than *npact*(3). Furthermore, because of the length limitation, we can only present results from selected sites and Web traces.

### A. Set 1: Usefulness of Similarity Ordering

We like to study the performance of *npact* when reference objects of different path differences and similarities are used. Different groups of experiments are performed, they are broken down by path differences and path similarities (see Figure 7 for a summary). To precisely state our results, we first introduce some notations.

**Notations.** Let $\Omega$ be a set of objects available for selection, and $\{u_1, \ldots, u_n\} \subseteq \Omega$. Let

$$\textit{path-sim}(u, \{u_1, \ldots, u_n\}) = \sum_{i=1}^{n} \textit{path-sim}(u, u_i)$$
$$\textit{path-diff}(u, \{u_1, \ldots, u_n\}) = \sum_{i=1}^{n} \textit{path-diff}(u, u_i)$$

Define the set $\Delta_\Omega(\delta, \sigma) = \{(u, u_1, \ldots, u_n) \subseteq \Omega \mid \textit{path-diff}(u, \{u_1, \ldots, u_n\}) = \delta$ and $\textit{path-sim}(u, \{u_1, \ldots, u_n\}) = \sigma\}$

20 Web sites were used in the experiments [6]. 6 of these sites were ranked in the top 25 most visited sites, and 14 were ranked in the top 500 sites. [7] The rest of the sites were chosen to include various categories. The category

[6]www.abcnews.com (news) www.aol.com (information) www.bofa.com (commercial) www.cisco.com (commercial) www.cnet.com (techncial) www.columbia.edu (academic) www.edmund.com (commercial) sportszone.espn.com (news) www.fcc.org (government) www.ibm.com (commercial) www.javasoft.com (technical) www.lucent.com (commercial) www.microsoft.com (commercial) www.netscape.com (commercial) www.nycvisit.com (information) www.techweb.com (technical) www.tripod.com (information) www.umass.edu (academic) www.usatoday.com (news) www.ustreas.gov (government)

[7]Source: MediaMetix (www.mediametix.com)

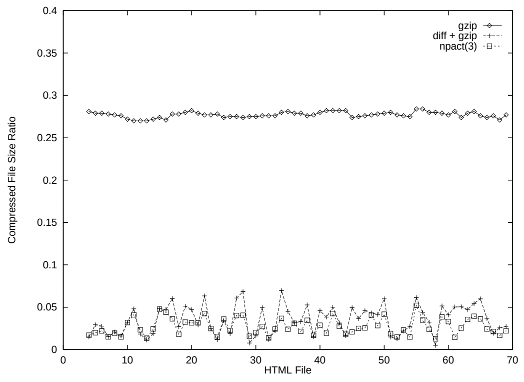

Fig. 8. Different Versions of http://www.abcnews.com/index.html

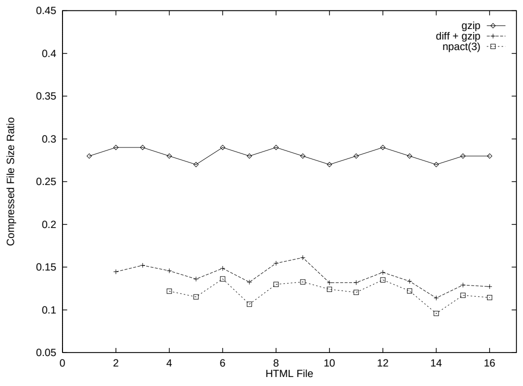

Fig. 9. Response from www.altavista.digital.com

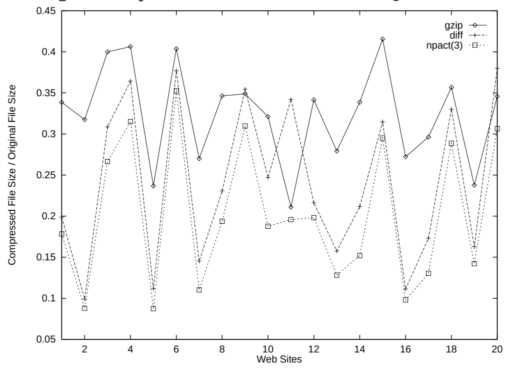

Fig. 10. Objects from the Same Directory

breakdown is 3 news sites, 3 information sites, 7 commercial sites, 3 technical sites, 2 academic sites and 2 government sites. For the files collected, all binary, graphics and audio files were removed. Also, only files with size between 1K and 64K were considered.

The experiments in this set operate as follows: For each site studied, we first pick a random object from the site. Then we try to simulate the transfer of the chosen object using compaction by selecting $n$ other objects (from same or different sites) to be used as reference objects. We compute the size of the encoded object, and tally this by path difference and path similarity values.

### A.1 Objects with same URL (mean $\delta$ = 0, mean $\sigma \geq 1$)

For brevity, we present our results only for a representative site, www.abcnews.com. In this experiment, we collected objects from the Web site www.abcnews.com every hour, over a period of 5 days (from the May 23 1998 to May 28 1998). Different versions of objects with the same URL were grouped together and sorted in chronological order. For each sequence of objects, we apply *gzip*, *diff* (between the current and the last version), and *npact* (the 3 most recent versions) to determine the size of transfer. While comparisons were performed for a number of URLs, only the URL http://www.abcnews.com/index.html, which generated a total of 69 different objects, will be described here. Other URLs exhibit similar trends.

Figure 8 shows the ratio of the encoded and original size for all 69 objects.

The results show that for objects with same URL, which tend to have similar content, *diff* and *npact* performed much better than *gzip*. In addition, Figure 8(b) shows that *npact* is better in capturing similarity among less similar objects. For the set of objects selected every hour, the mean compression ratios are 0.2772 for *gzip*, 0.0358 for *diff* and 0.02718 for *npact*. When the set of objects is selected every 4 hour, the mean compression ratios are 0.2768 for *gzip*, 0.0703 for *diff* and 0.0473 for *npact*.

### A.2 Objects from CGI scripts with different parameters (mean $\delta$ = 2, mean $\sigma \geq 1$)

We submitted a number of queries to the search engine www.altavista.digital.com with different query strings. Figure 9 shows the output for 16 pages, the first 8 pages are for the query string *java* and the next 8 pages for the query string *network*.

The results show that both *diff* and *npact* perform very well (mean compression ratio of 15%), while *gzip* performs much poorer (mean compression ration of 30%). An interesting observation was that there was no significant difference in result when responses from different query strings were used for referencing. This implied that most of the similarity came from formating. Similar results were also obtained from requests to the electronic site like www.amazon.com.

In general, HTML pages that are being updated continuously (stock quote, sports scoreboard, newspaper headlines, weather, movie showtimes, etc.) can be transferred very efficiently under *npact*.

### A.3 Objects from the same directory (mean $\delta$ = 2, mean $\sigma \geq 1$)

For each of the 20 Web sites, the objects collected were filtered such that only objects in directories with 4 or more objects were extracted. After this filtering, the minimum number of objects left per site

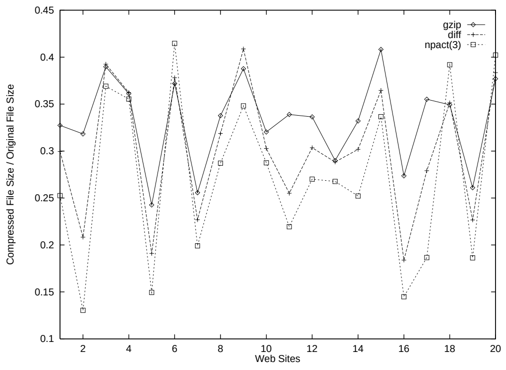

Fig. 11. Objects from the Same Site

was 61 (`www.columbia.edu`), maximum was 3,964 (`sportszone.espn.com`) and the average was 1,110.

The mean compression ratio for each Web site is plotted in Figure 11.

For all 20 Web sites, *npact* performs better than *gzip* on the average. Of the 20 sites, *npact* performs more than 50% better for 6 sites, $10\%$ to 50% better for 13 sites, and 10% better or less for only 1 site. The performance of *diff* tracked that of *npact*, though not in all cases. In 3 out of 20 sites, *diff* performed worse than even *gzip*. The mean compression ratios over all 20 site were 0.2012 for *npact*, 0.2419 for *diff*, and 0.3242 for *gzip*.

### A.4 Objects from the same Web site but different directories (mean $\delta > 2$, mean $\sigma \geq 1$)

In our experiments, the minimum, maximum, and average number of files per site was 250 (www.nycvisit.com), 5,550 (`sportszone.espn.com`), and 1,594.

Figure 11 shows the mean compression ratio for each of 20 sites. As expected, the results show that objects chosen randomly from the same Web site had a smaller amount of similarity. Nevertheless, out of the 20 sites, relative to *gzip*, *npact* performed 50% better in 2 sites, 10% to 50% better in 12 sites, less than 10% better in 6 sites.

In 3 sites though, *npact* performed worse than *gzip*. Two of them were academic sites and the third was a government site. Both academic sites contained a large number of objects from different departments and (probably) prepared by different people, with little common in style and formating. The government site contained a large number of plain text file with very minimum formatting. Since the current implementation of *npact* performed worse than *gzip* if used purely as a compression scheme, *npact* thus performed worse in these cases.

Note that when objects are very different, the mean compression ratios of *gzip* and *diff* will be very close because *diff* will simply output the requested object plus some overhead. Therefore, the observation that the mean compression ratios of *gzip* and *diff* were approximately the same for these 3 sites provides further evidence that the reference objects share little similarity with the requested object.

Over all 20 sites, the mean compression ratios were 0.2726 for *npact*, 0.3013 for *diff* and 0.3317 for *gzip*.

### A.5 Objects from different Web sites (mean $\delta > 0$, mean $\sigma > 0$)

In this case, *npact* did not perform as well as both *gzip* and *diff*. The mean compression ratios were 0.3145 for both *gzip* and *diff*, and 0.3455 for *npact*. This confirms that similarity among randomly selected objects is low.

### A.6 Compression Ratio with respect to $\delta$ and $\sigma$

The previous experiments showed that *npact* performed well on the average for specific ranges of $\delta$ and $\sigma$ values. Due to space limitations, we cannot include the measurement plots. Instead, we will highlight observations drawn from these experiments.

1. The use of $\delta$ and $\sigma$ as selection parameters correctly selected objects with *similar* contents for the case of small $\delta$ and large $\sigma$. For example, choosing reference objects from the same directory generated encoded objects that were smaller than *gzip* consistently (on the average) for all sites studied. Therefore, the conjecture that "closeness" of URLs implies similarity in content was true for the case of small $\delta$ and large $\sigma$, but not true for large $\delta$ and small $\sigma$.
2. The parameter $\delta$ was better in predicting good performance for small $\delta$, as in the case of $\delta$ = 0 (same URL) and $\delta$ = 2 (same directory). However, for larger values, $\sigma$ may be a better indicator of good performance than $\delta$.
3. While the compression is good for small $\delta$ and large $\sigma$, it shows no clear trend when only one of the dimensions ($\delta$ or $\sigma$) is varied.

## *B. Set 2: Distribution of Similarity Ordering in Actual Traces*

Results from Section IV-A show that *npact* performed significantly better than *gzip* and *diff* if objects with high similarity ordering are used as references. The objective of this section is to show that in an actual user browsing session, our proposed selection algorithm is able to pick up reference objects with high similarity ordering most of the time.

To verify our claim, we made use of an actual user trace. We had two requirements for the trace. First, the requested URL must be retained in the trace in order to compute $\delta$ and $\sigma$. Second, the trace should record the behavior of the actual client making the request so that per-client statistics could be collected. The first requirement ruled out the use of most publicly available HTTP logs (e.g., UC Berkeley Home IP Web Traces[8] and Dig-

[8]http://www.cs.berkeley.edu/~gribble/traces/index.html

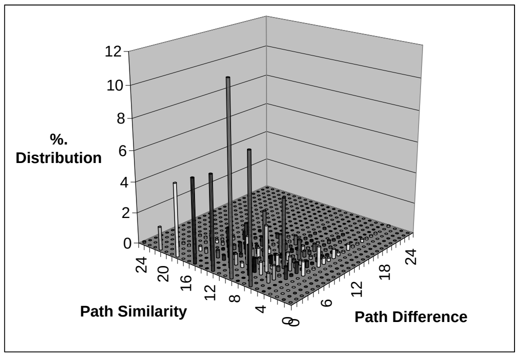

Fig. 12. Distribution of Path Difference and Path Similarity in Boston University Trace using $w$ = 64

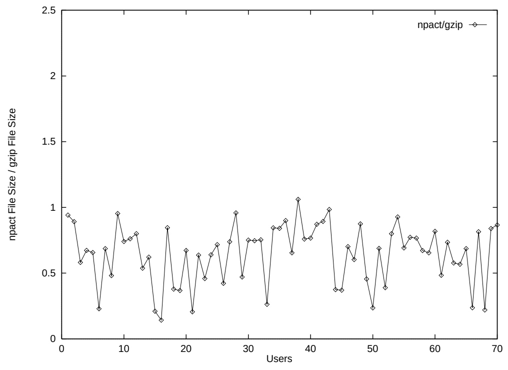

Fig. 13. Performance of *npact* using traces from www.bell-labs.com

ital's Web Proxy Trace[9] ) because the URLs had been anonymized for privacy reason. The second requirement ruled out the NLANR[10] cache access logs because the log entries were highly aggregated. With these limitations, we can only find an older log from Boston University [3] which satisfied our requirements.

The Boston University trace contains 762 unique users, and after removing URLs with extensions that indicated that they may be non-HTML or non-text objects (e.g., those with extension gif, jpeg etc.), 197,004 URLs were left. The maximum number of URLs per user is 4,412 and the minimum number of accesses per user is 16. 448 users have 100 or more URLs accesses. For each user's access log, we used the selection algorithm to select $n$ reference objects, whose aggregate path differences and path similarities are recorded as a $(\delta, \sigma)$ tuple. To simulate the effect of caching, we used a moving window size of $w$, where $w$ represents the cache size.

Figure 12 shows the density function of the $(\delta, \sigma)$ tuples for the case of $w$ = 64. Of all the requests, 78% found 3 or more URLs from the same site. Among these requests, 37.7% found 3 or more URLs with $\delta$ = 0 (same name), 76.7% had $\delta \leq 6$ , and 90.7% had $\delta \leq 10$. When $w$ is increased to 1024, the improvement only improves slightly.

The distribution of $(\delta, \sigma)$ was heavily concentrated in the regions of $\delta \leq 10$ and 80% of the tuples had $\delta \leq 6$ (i.e., mean $\delta$ = 2). Earlier results (Section IV-A) demonstrated that the region with mean $\delta$ = 2 corresponds to region of high object similarity. With $\delta = 6$, the 3 reference objects must either be all from the same directory (Section IV-A.3) or have at least one object with the same name (Section IV-A.1). From the results of Section IV-A, the first case has a 38% improvement over *gzip* and 17% improvement over *diff*, while the second case can have a 90% improvement over *gzip* and a 23% improvement over *diff*.

In summary, the distribution of $(\delta, \sigma)$ in this trace contains a significant amount of reference locality such that our selection heuristics can find highly similar URLs. In fact, at least 80% of the accesses would benefit from the use of *npact*.

## *C. Set 3: Performance of* npact *based on Actual Traces*

### C.1 Object transfer size reduction

In the final set of experiments, we compute the actual amount of savings using *npact* by performing *npact* (i.e., selection, encoding and decoding) for actual access traces. The Boston University trace we used in Section IV-B could not be used here because the age (3 years old) of the logs meant that many of the URLs were outdated, and could no longer be fetched.

What we chose to do instead was to take a specific multi-day server log, divide it into per-user access traces, perform *npact* for each such trace, and compute the mean compression ratio. In the following, we presented our results based on the site www.bell-labs.com, where we had access to the detailed server log.

Specifically, we obtained the server log of www.bell-labs.com for 7 days from June 20 1998 to June 26 1998. In this log, we were able to extract 3,296 user access trace (based on unique IP addresses) that we "followed" using *npact*, *gzip* and *diff*.

Figure 13 plots the performance of *npact* relative to *gzip*. Logs were ordered by their number of access. Figure 13 shows the ratio for the first 70 users with the most number of accesses and Among the remaining 3,226 users, 2,831 out of 3,297 users had ratios smaller than 1 (*npact* outperforms *gzip*), and 69 out of first 70 users had ratio smaller than 1. The average ratio of *npact* over *gzip* for all 3,279 users was 0.6693.

In summary, 86% of all users would benefit from the use of *npact*. The larger the number of accesses, the more likely that *npact* would perform better. It can also be observed from Figure 13(b) that substantial savings were possible even for users with very few accesses.

The comparison with *diff* is similar. The performance of *npact* relative to *diff* for all users was 0.8170 on the average.

[9] ftp://ftp.digital.com/pub/DEC/traces/webtraces.html
[10] http://ircache.nlanr.net

| | $n$ = 0 | $n$ = 1 | $n$ = 3 | $n$ = 7 |
|---|---|---|---|---|
| $t_{\text{select}}$ (ms) | 0.00 | 0.17 | 0.20 | 0.23 |
| encode (kbyte/s) | 900 | 722 | 380 | 176 |
| decode (kbyte/s) | 2,830 | 6,011 | 6,255 | 6,750 |

Fig. 14. Select/Encode/Decode Processing Times

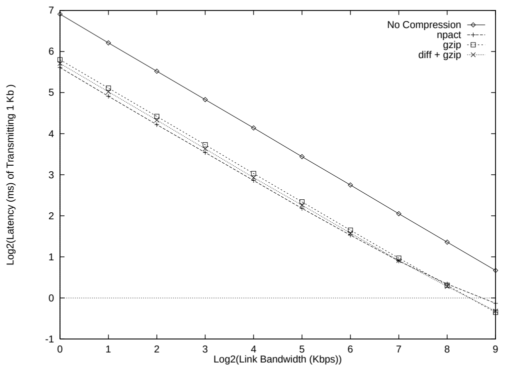

Fig. 15. Transfer Time (in ms/kbyte) vs. Effective Link Bandwidth (in kbyte/s)

### C.2 Latency reduction

Latency reduction is achieved in our scheme if $t_{\text{select}}$ + $t_{\text{encode}}$ + $t_{\text{decode}}$ + $\frac{|o'|}{s} < \frac{|o|}{s}$

In order to quantify the reduction in latency, the selection, encoding and decoding speed of *npact*, *gzip* and *diff* were measured. Figure 14 shows the average execution time of the *npact* with respect to $n$, averaging over 1,000 files. The measurements were done on a SUN Ultra2. The compression speed of *gzip* is 2,545 kbyte/s and *gunzip* decompresses at 12,270 kbyte/s. The encoding speed of *diff -e* is 961 kbyte/s.

To ease the comparison in latency, we normalize all the values to the inverse of rate (measured in ms/kbyte). The normalized transfer time, $T_{nor}$ (ms/kb) is defined as $T_{nor}$ = $t_{\text{encode-nor}}$ + $t_{\text{decode-nor}}$ + $\frac{|o'|}{s} * \frac{1}{|o|}$. (Selection time is negligible).

Figure 15 shows the values of *TTime* for various algorithms for effective link bandwidth ($s$) from 1 kbps to 512 kbps. The base case of plain transfer without compression is included for comparison. Despite its higher processing overhead, the normalized latency incurred by using *npact* is the lowest among all 4 curves till $s$ reaches 256kbps. This can be attributed to the higher compression achieved by *npact*. Beyond that, *gzip* performs the best due to its low overhead. For $s > 8096$ kbps, plain transfer incurs the smallest latency.

## V. Conclusion

We presented a technique which we call *cache-based compaction* for reducing the size (optimizing the latency) of Web transfer. The two key ideas behind our technique is: (1) an efficient selection heuristic, and (2) the use of an extended dictionary (specifically the client cache) for compression. Our compaction technique significantly generalizes previous work on optimizing Web transfer using compression or differencing.

Through experiments, we observe that our compaction technique provides significant improvement over previously proposed techniques for real-life user accesses.

The technique of compaction can be applied to other domains in addition to Web browsing. For example, we believe it is also applicable to electronic mail, and can potentially be a part of a generic wireless middleware layer.

## References

[1] Timo Alanko, Markku Kojo, Mika Liljeberg, and Kimmo Raatikainen. Mowgli: Improvements for internet applications using slow wireless links. In *Proceedings of PIMRC*, pages 1038–1042, Helsinki, Sep 1997. IEEE.

[2] Gaurav Banga, Fred Douglis, and Micheal Rabinovich. Optimistic deltas for www latency reduction. *USENIX*, 1997.

[3] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of www client-based traces. Technical Report BU-CS-95-010, Department of Computer Science, Boston University, July 1995.

[4] Adam Dingle and Tomas Partl. Web cache coherence. *Fifth International World Wide Web Conference*, May 1997.

[5] Bradley M. Duska, David Marwood, and Micheal J. Feeley. The measured access characteristics of world-wide-web client proxy caches. *USENIX Symposium on Internet Technologies and Systems*, Dec 1997.

[6] Armando Fox and Eric Brewer. Reducing www latency and bandwidth requirements by real-time distillation. In *Fifth International World Wide Web Conference*, May 1996.

[7] James Griggioen and Randy Appleton. The design, implementation, and evaluation of a predictive caching file system. Technical Report CS-264-96, Department of Computer Science, University of Kentucky, Lexington, KY, June 1996.

[8] James Gwertzman and Margo Seltzer. World-wide web cache consistency. *Proceedings of the USENIX Technical Conference*, 1996.

[9] Barron C. Housel and David B. Lindquist. Webexpress: A system for optimizing web browsing in a wireless environment. *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, pages 108–116, Nov 1996.

[10] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: an empirical analysis. *ACM Transactions on software Engineering and Methodlogy*, 7(2):192–214, Apr 1998.

[11] Thomas M. Kroeger, Darrell D.E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. *USENIX Symposium on Internet Technologies and Systems*, DEC 1997.

[12] Jeffery C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *Proceedings of the ACM SIGCOMM*, pages 181–194, 1997.

[13] Venkata N. Padmanabhan and Jeffery C. Mogul. Using predictive prefetching to improve world wide web latency. In *Computer Communication Review*. ACM, July 1996.

[14] Duane Wessels and K. Claffy. ICP and the Squid Web Cache. *IEEE Journal on Selected Areas in Communication*, 16(3):345–357, April 1998.

[15] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transaction of Information Theory*, IT–23(3):337–343, May 1977.

[16] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transaction of Information Theory*, IT–24(3):530–536, Sep 1978.