Value-Based Web Caching

Sean C. Rhea
srhea@cs.berkeley.edu

Kevin Liang
asiavu3@uclink4.berkeley.edu

Eric Brewer
brewer@cs.berkeley.edu

Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720-1776

ABSTRACT

Despite traditional web caching techniques, redundant data is often transferred over HTTP links. These redundant transfers result from both *resource modification* and *aliasing*. Resource modification causes the data represented by a single URI to change; often, in transferring the new data, some old data is retransmitted. Aliasing, in contrast, occurs when the same data is named by multiple URIs, often in the context of dynamic or advertising content. Traditional web caching techniques index data by its name and thus often fail to recognize and take advantage of aliasing.

In this work we present Value-Based Web Caching, a technique that eliminates redundant data transfers due to both resource modification and aliasing using the same algorithm. This algorithm caches data based on its *value*, rather than its *name*. It is designed for use between a parent and child proxy over a low bandwidth link, and in the common case it requires no additional message round trips. The parent proxy stores a small amount of soft-state per client that it uses to eliminate redundant transfers. The additional computational requirements on the parent proxy are small, and there are virtually no additional computational or storage requirements on the child proxy. Finally, our algorithm allows the parent proxy to serve simultaneously as a traditional web cache and is orthogonal to other bandwidth-saving measures such as data compression. In our experiments, this algorithm yields a significant reduction in both bandwidth usage and user-perceived time-to-display versus traditional web caching.

Categories and Subject Descriptors

C.2.2 [**Computer-Communications Networks**]: Network Protocols—*Applications*; C.2.4 [**Computer-Communications Networks**]: Distributed Systems—*Client/server*

General Terms

Algorithms, Performance, Design, Experimentation, Security

Keywords

aliasing, caching, duplicate suppression, dynamic content, HTTP, Hypertext Transfer Protocol, privacy, proxy, redundant transfers, resource modification, scalability, World Wide Web, WWW

1. INTRODUCTION

With the widespread deployment of broadband, it is often assumed that bandwidth is becoming cheap for the common Internet
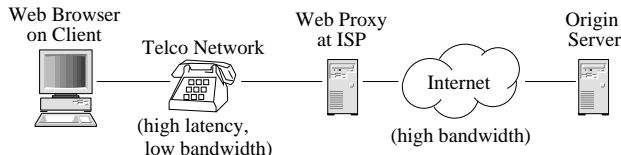
Figure 1: *Our target population.* **In this paper we look at reducing the user perceived time-to-display (TTD) of web pages served over a low bandwidth link, such as a telephone modem. Our techniques make use of computational and storage resources at the Internet service provider (ISP) to which a client is connected.**

user. Although this notion is true to an extent, a surprising number of users still connect to the Internet over 56 kbps modems. America Online, for example, has 33 million users who connect primarily via modems, and many other Internet service providers (ISPs) primarily support modem users. Moreover, the emerging deployment of universal wireless connectivity ushers in a new wave of users connecting over low-bandwidth links. In this paper we present a technique called Value-Based Web Caching (VBWC) that aims to mitigate the limitations of such connections.

We begin by assuming that our user has a very low-bandwidth (less than 80 kb/s) connection over a telephone or wireless network to an ISP, as illustrated in Figure 1. This service provider may run a web-proxy and/or cache on behalf of the user and is in turn connected to the Internet at large. In such a situation, the bandwidth through the telephone or wireless network is a fundamental limitation of the system; it is the largest contributor to client-perceived latency for many files. For example, in 1996 Mogul *et al.* [14] found that the average response size for a successful HTTP request was 7,882 bytes, which takes a little over a second to transmit over a modem; in contrast, the round-trip time between the client and server more often falls in the 100-300 ms range. Other traces show mean response sizes of over 21 kB [10].

In the Mogul *et al.* study, the authors also showed that much of the limited bandwidth available to clients was being wasted. Some of this waste is easy to eliminate: as many as 76.7% of the full-body responses in the studied trace were shortened by simple *gzip* compression, resulting in a total savings of 39.4% of the bytes transferred. Other bandwidth waste is more difficult to correct.

One example of bandwidth waste that is difficult to eliminate occurs when the data represented by a single Uniform Resource Identifier (URI) changes by small amounts over time. This phenomenon is called *resource modification*; it it tends to occur often with news sites such as *cnn.com*. Mogul *et al.* showed that 25–30% of successful, text-based responses were caused by resource modification, and they found that using delta encoding over the response

bodies eliminated around 50% of the data transferred. Our results confirm that when used between the requesting client and the origin server, delta encoding produces significant reductions in bandwidth consumed.

When an origin server does not compute deltas itself, a proxy may compute them on behalf of clients. Banga, Douglis, and Rabinovich proposed *optimistic deltas* [2], in which a web cache sends an old version of a given resource to a client over a low bandwidth link immediately after receiving a request for it. The cache then requests the current version of the resource from the origin server, and sends a delta to the client if necessary. To provide for the largest number of possible optimistic responses for dynamic web page content, it is desirable that the cache be able to transmit the response sent by the origin server for one client as the optimistic response to another. Unfortunately, since the cache does not know the semantic meaning of the page, this technique has the potential to introduce privacy concerns. Ensuring that no single client's personal information leaks into the response sent to another is a non-trivial task.

A second type of bandwidth waste occurs when two *distinct* URIs reference the same or similar data. This phenomenon is commonly termed *aliasing*; and can occur due to dynamic content, advertising, or web authoring tools. It was studied in 2002 by Kelly and Mogul [10], who studied the case in which one or more URIs represent *exactly* the same data. They found that 54% of all web transactions involved aliased payloads, and that aliased data accounted for 36% of all bytes transferred. The standard web caching model identifies cacheable units of data by the URIs that reference them; as such, it does not address the phenomenon of aliasing *at all*. Even if it is known in advance that two or more URIs share some data, there is no way to express that knowledge under the standard model.

In this paper, we present Value-Based Web Caching, a technique by which cached data is indexed by its value, rather than its name. Our algorithm has the following interesting properties.

- It detects and corrects excess bandwidth usage due to both resource modification and aliasing with the *same* algorithm.

- It is oblivious to data format; it requires no understanding of HTML syntax.

- It eliminates some of the privacy concerns associated with delta-encoding proxies; the client in our algorithm only receives data that was originally intended for it by an origin server.

- Although it is not strictly a stateless protocol, proxies store only soft-state; loss or inconsistency of this state results only in performance degradation, not a compromise of semantic transparency. Furthermore, it requires orders of magnitude less storage resources on the proxy than on the client, allowing a single proxy to support many clients.

- In the common case, it adds no round trips to the standard HTTP protocol, achieving performance comparable to conventional caching even in the absence of aliasing and resource modification.

The remainder of this paper is organized as follows: Section 2 presents the Value-Based Web Caching algorithm; it is followed by a performance evaluation in Section 3. Section 4 presents related work, and Section 5 concludes.

First request:
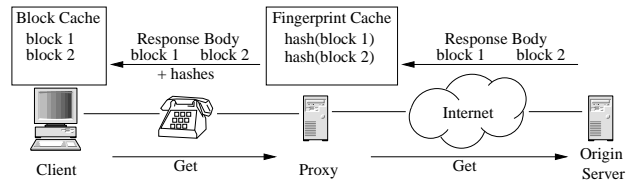
Subsequent request:
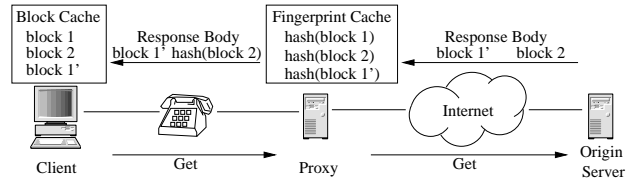
Figure 2: *Value-Based Web Caching.* **As the proxy receives each block from the origin server, it hashes it, stores the result in its cache, and forwards the block on to the client (upper figure). When a block is sent again as part of a different response, only the hash is retransmitted to the client (lower figure).**

2. THE VBWC ALGORITHM

In this section we present the Value-Based Web Caching algorithm; we begin with an overview before discussing the details.

2.1 Overview

Consider a web resource, such as the main page of a news web site. This page changes over time, and it contains references to other resources such as images and advertising content. In a traditional web cache, the data for each of these resources would be stored along with some freshness information and indexed under the URI by which it is named. The fundamental idea behind VBWC is to index cached data not only by its name, but by its value as well. To achieve this goal efficiently, we break the data for a resource into blocks of approximately 2 kB each, and name each block by its image under a secure hash function, such as MD5 [19]. This image is called the block's *digest*; by the properties of the hash, it is highly unlikely that two different blocks map to the same digest under the secure hash. A web cache using VBWC stores these blocks as its first-class objects. In order to also be able to find data by its name, a second table may be used to map resources to the blocks of which they are composed.

Indexing web resource data in this manner can result in better utilization of storage resources [1, 12]. For example, when aliasing occurs, the aliased data is stored only once. However, the real benefit of value-based caching comes when resources are transferred between caches. Figure 2 shows the basic algorithm. The first time a resource is requested by the client, the proxy fetches its contents from an origin server. These contents are broken into blocks and hashed. The proxy stores each block's digest, then it transmits both the digests and the blocks to the client. The client caches the blocks indexed by their digests and associated URIs. Later, the proxy can detect that a subsequent request contains an already transmitted block by checking for that block's digest in its cache. In such cases, only the digest is retransmitted to the client, which reassembles the original response using the data in its block cache.

Before delving into the details of this algorithm, let us consider its benefits. First, as shown in Figure 2, it does not matter whether

Before insert:
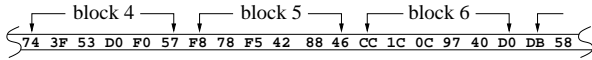
After insert:
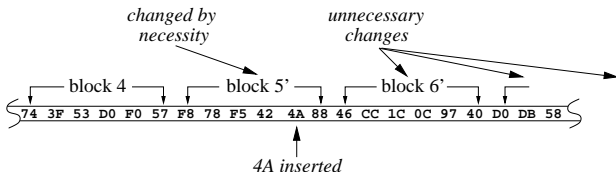
Figure 3: *The problem with fixed-sized blocks.* **After an insertion, not only is the block into which the new value was inserted changed, but all subsequent blocks are changed as well.**

Before insert:

After insert:

Figure 4: *Picking block boundaries using Rabin functions.* **An insert may change the block in which it occurs, split an existing block into two (shown here), or cause two existing blocks to be combined; the remainder of the blocks in a stream remain the same.**

the second request is for the same URI as the first request; all that is important is that the two responses contain identical regions. By naming blocks by their value, rather than the resources they are part of, we recognize and eliminate redundant data transfers due to both resource modification and aliasing with the same technique.

A second benefit of VBWC is the way in which it distributes load among the parties involved. As mentioned above, while the proxy may store the contents of blocks (as it would if it were also acting as a web cache, for instance), it is only required to store their digests. As such, the proxy only stores a few bytes (16 for MD5) for each block of several kilobytes stored by the client. A proxy may possess considerable storage resources; however, assuming it has only as much storage as a single client, this storage ratio allows a single proxy to support hundreds of clients.
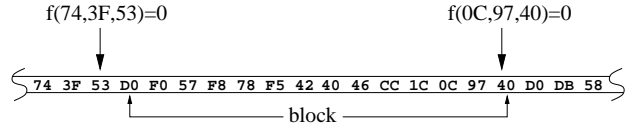
A third benefit of our algorithm is that the digests only need to be computed at the server; the client simply reads them from the data transmitted to it. In the case where the client is a low-power device such as a cellular phone or PDA, this computational savings could result in a significant reduction in latency. On the other hand, the Java implementation of our algorithm can process blocks at a rate of 7.25 MB/s on a 1 GHz Pentium III, a rate equivalent to the download bandwidth of over 1,000 modern modems; we thus expect the additional per-client computational load on the server to be small enough to allow it to scale to large numbers of clients.

A final benefit of Value-Based Web Caching is that it requires no understanding of the contents of a resource. For example, a delta-encoding proxy (such as that in used in the WebExpress project [8]) will generally make use of the responses that an origin server sends to many different clients in order to choose a base page for future use in delta compression. If this process is performed incorrectly, there is the possibility that one client's personal information may become part of the base page transmitted to other clients. Specifically, this information leakage is a result two interacting performance optimizations: using a single base page for multiple clients to save storage resources, and using delta-encoding on responses marked uncacheable by the origin server. In contrast, a client using VBWC only receives either the blocks sent to it by an origin server or the digests of those blocks.
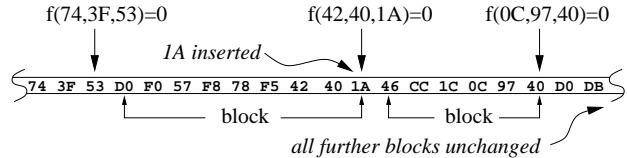
2.2 Choosing Block Boundaries

Above we mentioned that we break each response into blocks; in this section, we describe how we choose block boundaries. Naïvely, we could use a fixed block size, 2 kB for example. Figure 3 illustrates one problem with this approach: an insertion of a byte

into a block early in the file (into block 5 in the figure) offsets the boundaries of every block that follows. As a consequence, our algorithm would only notice common segments in two resources up to their first difference; common segments that occurred later in the resources would not be identified.

One solution to this problem was discovered by Manber in an earlier work [11]: choose block boundaries based on the value of the blocks rather than their position. Let f be a function mapping n one-byte inputs uniformly and randomly to the set $\{0, \dots, 2047\}$. In other words,

$$f : \overbrace{B \times B \times \cdots \times B}^{n \text{ times}} \to \{0, \dots, 2047\}$$

where B is the set of possible byte values. We can place a block boundary before byte i in a resource if the value of f on the n bytes proceeding byte i is 0. Since f is uniform and random, we expect to evaluate it on average 2048 times before finding a zero; thus the expected block size from this technique is 2 kB. In addition, we set a minimum and maximum block size to avoid choosing blocks that are too small or large. These limits are necessary primarily because in practice the function f is computationally easy to invert; without the limits, a malicious party could create documents consisting of many small or large blocks and thereby significantly skew the expected block size. Figure 4 shows an example of computing block boundaries in this manner for $n = 3$. There, $f(0C, 97, 40) = 0$, so the block in the upper figure ends on byte 40.

The benefit in picking block boundaries by the value of the underlying data is illustrated in the lower half of Figure 4, where the byte 1A has been inserted into the stream. In this example, $f(42, 40, 1A) = 0$, so this change introduces a new boundary, splitting the original block in two. However, since $n = 3$, the boundaries of all blocks three or more positions after the change are unaffected; in particular, the next block still starts with the byte value D0. In general, the insertion, modification, or deletion of any byte either changes only the block in which it occurs, splits that block into two, or combines that block with one of its neighbors. All other blocks in the stream are unaffected.

To implement f, Manber used Rabin functions [18], a decision we follow. Let b_i represent byte i in a stream and let p be a prime.

3

A Rabin function f is a function

$$
\begin{aligned}
f_i &= f(b_{i-n+1}, \ldots, b_i) \\
&\equiv b_{i-n+1}p^{(n-1)} + b_{i-n+2}p^{(n-2)} + \cdots + b_i \pmod{M}
\end{aligned}
$$

for some modulus M. Such functions are attractive because they can be computed iteratively:

$$
\begin{aligned}
f_{i+1} &= f(b_{i-n+2}, \ldots, b_{i+1}) \\
&\equiv (f_i - b_{i-n+1}p^{(n-1)}) \times p + b_{i+1} \pmod{M}
\end{aligned}
$$

Using this knowledge, we can compute f_{i+1} from f_i with only a subtraction, two multiplications, a division, and an add. Moreover, since there are only 256 possible values of b_{i-n+1}, the first multiplication can be computed efficiently via table lookup. As a result, we can compute f_i very efficiently for all i.

2.3   An Enhancement

Consider an HTTP client loading a web page through a proxy. The main page is stored on a server A, and it includes two embedded images, one stored on server B and the other on server C. The client opens a connection to the proxy and enqueues the request for the resource on A. Once it receives the response body, it parses it and discovers it must also retrieve the resources on B and C in order to display the page. To perform these retrievals, the client could enqueue the request for the resource on B followed by the request for the resource on C on the connection it already has open to the proxy, one after the other. However, because of the request-response semantics of the HTTP protocol, doing so would require the proxy to transmit the response for B before the response for C, regardless of which of them was available first. If server B was under heavy load or simply far away in latency from the proxy, a significant delay could result in which the data from C was available but could not be sent to the client, resulting in idle time on the low bandwidth link. This situation is commonly termed *head-of-line blocking*.

2.3.1   Allowing Multiple Connections

To reduce the occurrence of performance problems due to head-of-line blocking, most HTTP clients open several connections to a proxy at a given time. The Mozilla web browser, for instance, will open up to four connections to its parent proxy. For the same reason, we would like our child proxy running the VBWC algorithm to be able to open multiple connections to its parent. This decision introduces complications into the algorithm as follows. As discussed in the algorithm overview, the child proxy in our algorithm maintains a cache of previously transmitted blocks, and the parent proxy maintains a list of the digests of the blocks the child has already seen. Because the child only has finite storage resources available, it must eventually evict some blocks from its cache. Were the parent to later transmit a digest for a block the child had evicted, the child would have no way to reproduce the block's data. Thus there must be some way to keep the list of blocks on the parent consistent with the blocks actually cached by the child.

In the case where there is only one connection between the parent and the child, keeping the parent's list of blocks consistent with the child's block cache is simple. Both sides simply use some deterministic algorithm for choosing blocks to discard. For example, Spring and Wetherall [23] used a finite queue on either end. When the parent transmitted a block, it placed its digest on the head of the queue, and the child did the same on receiving a block. When the queue became full, the parent would remove a digest from its tail; the child would again do the same, but would also discard the block corresponding to the received digest. To decide whether to
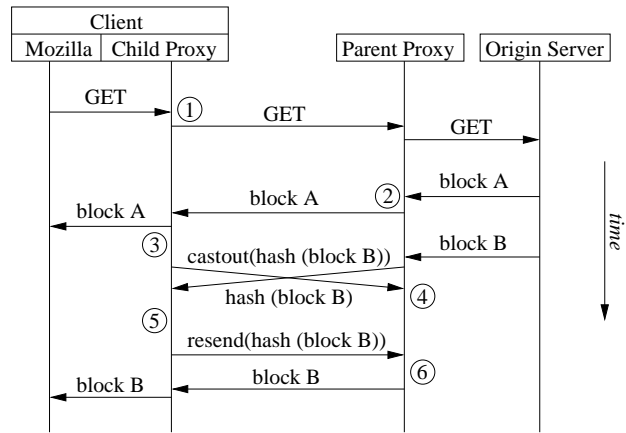
Figure 5: *The full algorithm.* **The web browser issues a GET request (1) that is forwarded all the way to the origin server. The server begins sending the response back, and it is broken up into blocks (2) at the parent proxy. On receipt of block A, the child proxy executes the LRU algorithm and chooses block B to discard. However, as it sends the castout message (3) to the parent proxy, the parent proxy is sending across only the digest of block B, assuming the child still has it cached (4). When the child receives the digest for B, it has already sent block A to the browser, so it cannot simply repeat the entire request. Instead, it requests block B by name from the parent proxy (5). Since the digest for block B was recently transmitted, it is still in the server's transmit buffer, so it is forwarded on to the child proxy, and in turn, to the client.**

transmit a whole block or only its digest, the parent need only examine the queue; if the digest of the block in question is already there, the block is cached on the child and only the digest must be transmitted. Unfortunately, with multiple connections between the parent and child, the order in which blocks are sent is no longer necessarily the same as the order in which they are received, so a more sophisticated algorithm must be used.

To allow for multiple connections between the child and parent proxies, we abandon the idea of using a deterministic function on either end of the link between them, and instead use the following optimistic algorithm. First, the server records the digests of the blocks that it has transmitted in the past, and assumes that unless it is told otherwise, the child still has the data for those blocks cached. The child, in turn, uses a clock algorithm to approximate LRU information about the blocks, discarding those least-recently used when storage becomes scarce. After discarding a block, the client includes its digest in the header of its next request to the parent, using a new "X-VBWC-Castout" header field. So long as the list of references to to the blocks contains some temporal locality, it is very unlikely that the parent will transmit the digest of a block that the child has just cast out—rather, the castout message will reach the server in time. If, however, this unfortunate case does occur, the child uses a request with a dummy URI to retrieve the data for the missing block. This entire process is illustrated in Figure 5.

A final modification is necessary to finish the optimistic algorithm. Before, the server only cached the digests of blocks it transmitted, not their values. If the child were to re-request a block as described above, the server would not be able to reproduce the data for it. To prevent this problem, we add to the server a queue of the last few blocks transmitted, which we call the *transmit buffer*. The size of this queue is determined by the bandwidth-delay product of the link between the parent and its child, plus some additional space to handle unexpected delays. For example, if the parent is on

4

a link with a one-way latency of l seconds and a bandwidth β, a queue size of $s\beta$ will allow the child $s - l$ seconds from the time it receives a unrecognized digest to send a retrieval request before that block is discarded from the transmit buffer. If the parent proxy is also acting as a web cache with an LRU eviction policy, the storage used for the transmit buffer can be shared with that used for general web caching.

2.3.2 Discussion

The transmit buffer in our algorithm might seem unnecessary; after all, the server can always retrieve a resource from the relevant origin server in order to re-acquire the data for a particular block. This is not the case, however, and the reasoning why it is not is somewhat non-obvious, so we present it here.

In building a web proxy, one possible source of lost performance is called a *store-and-forward delay*. Such a delay occurs when a web proxy must store an entire response from an origin server (or another proxy) before forwarding that response on to its own clients. Our algorithm does not suffer from such delays (except as necessary to gather all of the data from a given block), and this feature is an important one for its performance. Consider a single HTML page with a reference to a single image early in the page. If the page is transmitted without store-and-forward delays, the requesting client sees the reference early in the page's transmission and can begin fetching the corresponding image through another connection.[1] Otherwise, it must wait until the entire file is available before it has the opportunity to see the reference, so the image retrieval begins much later. Since the bandwidth of the modem is low and its latency is long, it is crucial that the child proxy forward each block of a response to the client as quickly as possible, in order to minimize the time until the client sends subsequent requests for embedded objects.

Unfortunately, eliminating store-and-forward delays conflicts with the notion of an optimistic algorithm as follows. First, modern web sites are rich with dynamic content, both due to advertising and due to extensive per-user and time-dependent customization. Two requests for the same resource, even when issued at almost the same time, often return slightly different results. If the child proxy in our algorithm were to receive a digest for a block that it did not have cached, the parent proxy might not be able to retrieve the contents of that block via a subsequent request to the associated origin server. If the child proxy has already forwarded some of the response to the client program, it cannot begin transmitting a different response instead.

We would like our algorithm to provide *semantic transparency*; that is, any response a client receives through our pair of proxies should be identical to some response it could have received directly from an origin server. Under normal circumstances, this notion means that the response received is one that the server sent at a given point in time, not the mix of several different responses it has sent in the past. Of course, it is always possible for a server to fail during the transmission of a response, in which case a client must by necessity see a truncated response. By increasing the size of the transmit buffer in our algorithm, we can provide the same apparent consistency. Under almost all circumstances, the child proxy is able to re-request a missing block before it leaves the transmit buffer. In the extremely rare case that it does not, and the block cannot be recovered, the connection to the client may be purposefully severed. By always using either a "Content-Length" header or chunked transfer encoding (rather than indicating the end of a response through a "Connection: close" header), we can also en-

---

[1]The importance of starting the retrieval of embedded objects as early as possible was first pointed out by Nielson *et al.* [16].

sure that in this rare case the client program is able to detect the error and display a message to the user. The user can then manually reload the page. In summary, connections may be dropped even without our pair of proxies, and by manipulating the size of the transmit buffer, we can drive the probability of additional drops due to unavailable blocks arbitrarily low.

2.3.3 On Statelessness and Soft-State

An important difference between our algorithm and conventional web caching is that conventional web caches are stateless with respect to their clients. As pointed out by Sandberg *et al.* [21], stateless protocols greatly simplify crash recovery in network protocols; in stateless protocols, a server that has crashed and recovered appears identical to a slow server from a client's point of view. Furthermore, a stateless protocol prevents a server from keeping per-client state, eliminating a potential storage and consistency burden. For these reasons a stateless protocol is generally preferred over a stateful one. We justify our use of a stateful protocol as follows: it outperforms a stateless one, especially over high-latency links, and the state it stores is used only for performance, not correctness. We discuss these two points in detail below.

In a delta-encoding protocol such as RFC 3229 [13], a client submits information about earlier responses of which it is aware with each subsequent request for the same resource. If the proxy serving the client is also aware of one of those responses, it can compute a delta and send it to the client. However, this technique cannot eliminate redundant transfers due to aliasing; by definition, an aliased response contains redundant data from resources other than that requested. Unless a client were to transmit information about earlier responses for *all* resources to the proxy, some aliasing could be missed. Kelly and Mogul presented an algorithm that uses an extra round trip to catch aliasing [10], but over high-latency modem lines this additional round trip could introduce significant performance reductions. By storing a small amount of state per client on the proxy, our algorithm avoids extra round trips in the common case.

Moreover, while our protocol is not stateless, it utilizes only soft state to achieve its performance gains. A protocol is termed *soft-state* if the state stored is used only for performance, not correctness. If the proxy in our algorithm loses information about which blocks the child has cached, it will result only in the redundant transfer of data. If the parent thinks the child is caching data that it is not (either due to corruption on the parent or data loss on the child), the child will request the data be resent as described in Section 2.3.1. If a sufficient number of such resends occur, the child proxy could proactively instruct its parent to throw out all information about the contents of the child's cache through a simple extension to our protocol. Such an extension would be particularly valuable if a child were periodically to switch proxies. In conclusion, we believe the use of per-client state in our protocol is justified by the combination of its performance benefits and its absence of an effect on correctness.

3. PERFORMANCE EVALUATION

In this section we describe a detailed evaluation of the ability of VBWC to reduce bandwidth consumed and—more crucially—reduce user-perceived time-to-display (TTD). In this evaluation we concentrate on the domain in which the elimination of redundant transfers is likely to be fruitful. An evaluation based on full traces of user activity is left for future work. We start by describing our methodology and experimental setup, then present our results.

Bandwidth Reduction with VBWC
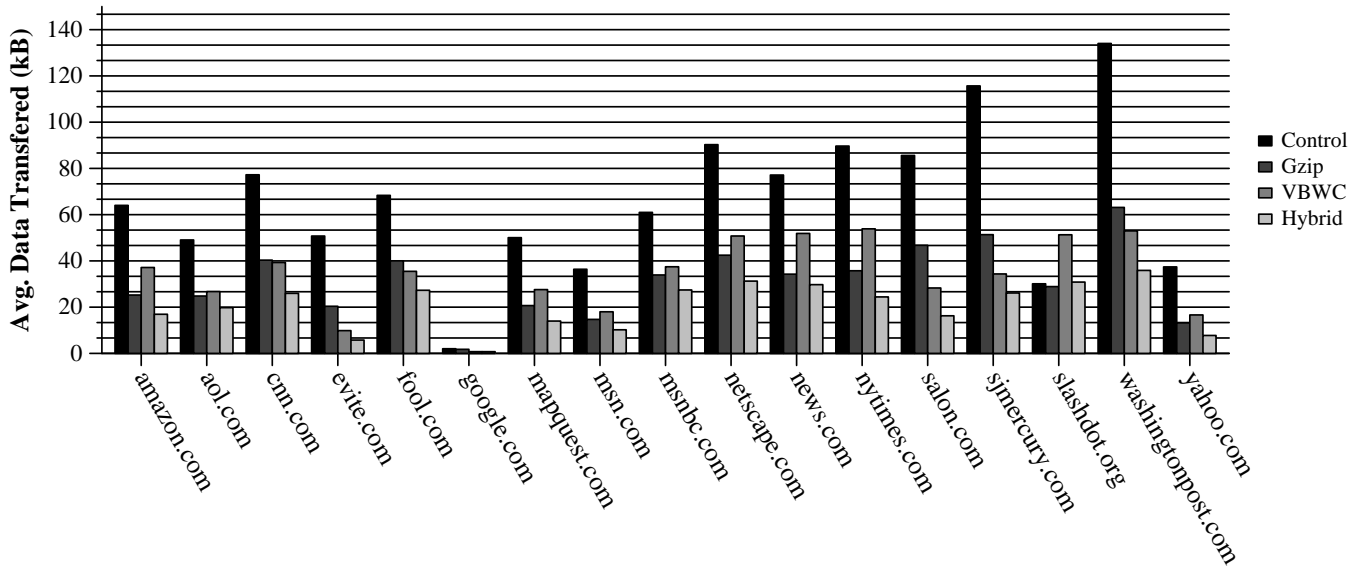
Figure 7: *Mean bytes transferred per download.* **This graph shows the mean bytes transferred across the modem to the client as a function of the algorithm used. In all but one case, the Hybrid algorithm outperforms all three others. Some origin servers return compressed response bodies (using the "Content-Encoding: gzip" header); the Control experiment performs about as well as the Gzip algorithm in such cases (e.g.,** *slashdot.org* **).**
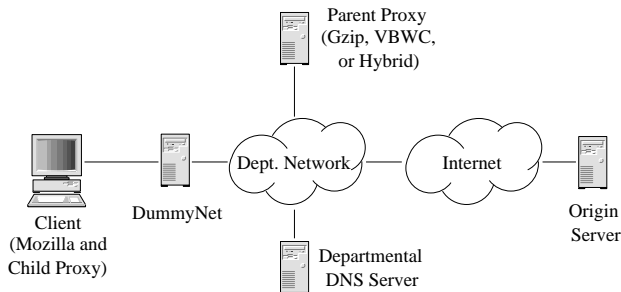
Figure 6: *Our experimental setup.* **Our instrumented version of Mozilla and our Java implementation of the child proxy run on the client machine, which talks to all other machines on the Internet through a FreeBSD machine using DummyNet to simulate a modem. The parent proxy (running either the Gzip, VBWC, or Hybrid algorithm) runs on the Internet side of this fake modem.**

3.1 Experimental Methodology and Setup

In order to test the ability of our algorithm to reduced the user-perceived TTD of common web pages, we built the following test suite. First, we instrumented Mozilla version 1.0.1 to read URIs from a local TCP port. After reading each URI, our instrumented browser loads the resource and sends the total load time back over the socket. This time corresponds to the time from when a user of an uninstrumented version of the program types a URI into the URI field on the toolbar until the Mozilla icon stops spinning and the status bar displays the message, "Done. (x seconds)".

To this web browser we added a Perl script that takes as input a list of URIs, then loads each one through an instrumented browser running without a proxy and a second one running through a proxy using the VBWC algorithm. We simulate a modem using a FreeBSD machine running DummyNet [20], which adds latency and bandwidth delays to all traffic passing through it. In our exper-

iments, we configured DummyNet to provide 56 kb/s downstream and 33 kb/s upstream bandwidth, with a 75 ms delay in either direction. These parameters mimic the observed behavior of modern modems. The use of DummyNet also allows us to monitor the total number of bytes transferred across the simulated modem in each direction, and the Perl script records this number after each load. After each iteration through the list, the script sleeps for twenty minutes before repeating the test, resulting in each URI being loaded through the algorithm and control approximately every thirty minutes. This experimental configuration is illustrated in Figure 6.

In our experiments, Mozilla and the child proxy run on a 750 MHz Pentium III with 1 GB of RAM, while the parent proxy runs on a 1 GHz Pentium III with 1.5 GB of RAM (as noted below, however, we limit our caches to a small fraction of the total available memory). The machine running DummyNet is an 866 MHz Pentium III with 1 GB of RAM. Both the child and parent machines were otherwise unloaded during our tests; the DummyNet machine was not completely isolated, but saw only light loads.

Mozilla consists of over 1.8 million lines of C++ source code. Rather than familiarize ourselves with the full extent of this code base necessary to add the VBWC algorithm to it, we implemented our algorithm in Java atop the Staged Event-Driven Architecture (SEDA) [26]. Implementing the algorithm in a separate proxy has the advantage of making it browser neutral (we have also run it with Internet Explorer), but adds some latency to local interactions. Worse, it forces us to utilize two separate web caches; one internal to Mozilla and one in the proxy itself. As such, in the control experiments Mozilla runs with a 10 MB memory cache, while in the VBWC experiments Mozilla and the proxy each have a 5 MB cache. We leave quantifying the advantages of integrating these two caches as future work, but it is clear that the caches will have some overlapping data, putting our algorithm at some disadvantage. Our results are somewhat pessimistic in this sense. The parent proxy in our tests is also implemented in Java atop SEDA. Altogether, the parent and child proxy are made up of about 6,000 lines of Java
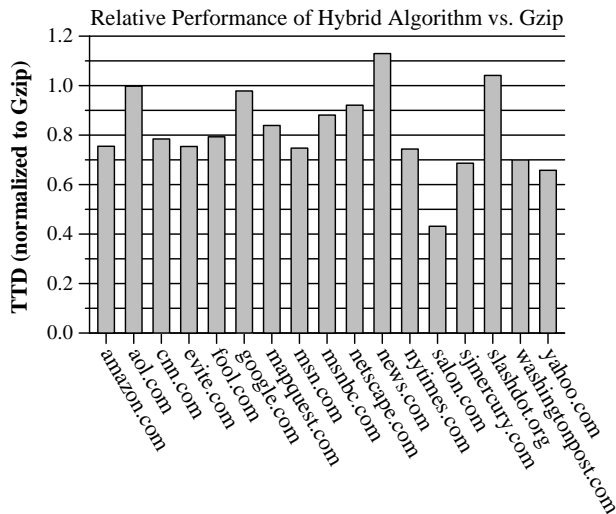
Figure 8: *TTD of Hybrid vs. Gzip.* **This graph shows the median time-to-display (TTD) of Value-Based Web Caching with compression (called the "Hybrid" algorithm), versus compression alone (called the "Gzip" algorithm). For most web sites, the Hybrid algorithm gives a significant improvement.**

	TTD (s)		Percent
Web site	Gzip	Hybrid	Improvement
amazon.com	6.65	5.02	24.5
aol.com	4.31	4.30	0.2
cnn.com	7.28	5.71	21.6
evite.com	3.13	2.36	24.6
fool.com	7.28	5.77	20.7
google.com	0.70	0.68	2.2
mapquest.com	6.27	5.26	16.1
msn.com	2.89	2.16	25.3
msnbc.com	6.89	6.07	11.9
netscape.com	12.33	11.36	7.9
news.com	6.02	6.80	-13.0
nytimes.com	6.32	4.70	25.6
salon.com	7.63	3.29	56.8
sjmercury.com	9.28	6.37	31.4
slashdot.org	4.61	4.80	-4.1
washingtonpost.com	10.32	7.22	30.0
yahoo.com	2.60	1.71	34.2

Table 1: *TTD of Hybrid vs. Gzip.* **This table shows the median TTD of the Hybrid and Gzip algorithms as represented in Figure 8, as well as the percent improvement the Hybrid algorithm achieves over Gzip.**

code, including an HTTP parser and a specialized string-processing library.

To compare VBWC against something other than just Mozilla itself, we implemented a second parent-child proxy pair that simply compressed each response from any web server using *gzip* before sending it over the modem. We do not compress any response that contains a "Content-Type" header starting with "image", since in our workload these are all GIF and JPEG files that are already compressed. Also, Mozilla includes an "Accept-Encoding: gzip" header in all its requests, and some origin servers take advantage of this, responding with "Content-Encoding: gzip" in their response headers. We do not try to further compress such responses.

Finally, we implemented a hybrid algorithm that uses VBWC but compresses each new block before sending it to the client. Non-image responses that are compressed by the origin server are uncompressed by the hybrid algorithm before they are broken into blocks; the resulting blocks are recompressed before being transmitted to the client.

Many previous studies in this field have measured bandwidth saved as a performance metric for algorithms similar to our own. Instead, we chose to instrument a full web browser. As we will show below, simple bandwidth savings do not necessarily guarantee an equivalent improvement in TTD, the main component of the quality of a user's experience.

3.2 Experimental Results

In this section we examine the results of our experiments, which ran from approximately 7 PM on November 12, 2002 until approximately 5:30 PM on November 14, 2002.

3.2.1 The Costs of Optimism

For our first result, we analyze the costs of the parent proxy's digest cache being only loosely synchronized with the child proxy's block cache. In all of our experiments, the transmit buffer on the parent was limited to 100 kB of block data. During our experiments, the parent in our algorithm transmitted a total of 17,768 digests without their corresponding data to the client; for 42 of these,

the child had already cast out the blocks' data from its cache, a 0.24% miss rate. Since the cost of a miss is only an additional round-trip time over the modem, we feel that this low miss rate clearly justifies our use of an optimistic algorithm.

3.2.2 Bandwidth Savings with VBWC

Next, we examine the bandwidth savings provided by our algorithm. Figure 7 shows the average kilobytes transferred per web page as a function of the algorithm used. In most cases, the combination of VBWC and *gzip* (called the Hybrid algorithm in our figures) outperforms all three other algorithms. However, the simple Gzip algorithm performs quite well. The HTTP 1.1 standard allows for such compression to start at the origin server when requested by a client (using the "Accept-Encoding" header field), but in our experience, very few servers take advantage of this portion of the standard. In the cases where they do (such as *slashdot.org*), the control case is much closer in bandwidth usage to the Gzip case.

In cases where the Hybrid algorithm underperforms Gzip, it is often due to the granularity of resource modification on those sites. For example, the spacing between differences on successive versions of *slashdot.org* is under 2 kB, and there are many small differences. Because of this, over the course of our experiments, only 24% of the main *slashdot.org* page sent from the parent to the child was sent as the digests of previously transmitted blocks. In contrast, 64% of the main *nytimes.com* page was sent as the digests of previously transmitted blocks. Using a smaller average block size would presumably mitigate the effects of small changes, at the cost of more overhead. In theory, one could also choose the average block size dynamically in response to past performance, but we have not yet investigated this technique.

3.2.3 TTD Reduction with VBWC

Figure 8 shows the median TTD of the Hybrid algorithm for various web sites, normalized against the TTD of the Gzip algorithm. We use medians instead of means when reporting TTD numbers because web server response time distributions show very long tails, especially on loaded web servers like those used in our experiments. In contrast, server load does not affect the sizes of responses. The relative performance of the control case and VBWC

URIs	2,881
Modified URIs	48
Unique payloads	3,639
Aliased payloads	97
(URI, payload) pairs	4,223
Unique blocks	12,328
Aliased blocks	510
Transactions	7,502
w/ modified URIs	1,979
w/ aliased payloads	922
w/ aliased blocks	1,345
Payload sizes (bytes)	
Range (min–max)	1–85,463
Median	2,927
Mean	14,909
Sum	54,252,472
Sum of aliased	586,887
Block sizes (bytes)	
Range (min–max)	1–8,192
Median	1,588
Mean	2,236
Sum	27,565,326
Sum of aliased	962,677
Transfer sizes (bytes)	
Median	1,115
Mean	8,604
Sum	64,545,462
Sum of aliased	1,748,601

Table 2: Statistics for our workload.

without *gzip* are less interesting—they roughly follow the bandwidth differences shown in Figure 7. Table 1 shows the actual times for each point in Figure 8, as well as the percent improvement the Hybrid algorithm achieves over Gzip for each web site. The benefits of VBWC are clear from the table and graph. For one web site, the Hybrid algorithm achieves a 56.8% improvement in TTD; for three others, it achieves at least a 30% improvement; and for another six it achieves at least a 20% improvement. It performs worse that compression alone in only two cases. We are quite satisfied with the performance of the algorithm in general.

3.2.4 Workload Analysis

As a final result, we characterized our workload so as to compare it to previous traces. To do so, we instrumented our VBWC code to print the name of each block it receives, the block's size, and the URI with which that block is associated. For every occurrence of a particular block, we noted whether the block had been seen earlier in the workload in the payload for the same or another URI. The transfer of blocks seen in the payloads of earlier transactions on the same URI could be eliminated through sophisticated name-based techniques such as delta encoding. The transfer of blocks seen only in the payloads of earlier transactions on *different* URIs, however, can only be eliminated through value-based caching. In this analysis, we found that a total of 34.0 MB of the 61.6 MB transferred could be eliminated by name-based caching. (Note that choosing blocks differently could improve this number; our blocking algorithm is sub-optimal for small deltas.) In contrast, 35.3 MB of the transferred data could be saved by value-based caching, leaving 1.3 MB that could be saved only by value-based caching. This potential 57% savings is over the transactions performed by Mozilla, which is already maintaining a 5 MB conventional cache of its own.

Overall, the above numbers indicate that there is a good deal of bandwidth to be saved using named-based caching, while there is less clear benefit to value-based caching if named-based caching with delta encoding is already being performed. To qualify the potential benefits of our algorithm on a more general trace, we analyzed our workload in the style of Kelly and Mogul [10]. They call a transaction a pair (U, P) where U is a request URI and P is a reply data payload. They say that a reply payload is *aliased* if there exist two or more transactions $(U, P), (U', P)$ where $U \neq U'$. They say that a URI is *modified* if there exist two or more transactions $(U, P), (U, P')$ where $P \neq P'$. We extend their nomenclature as follows. We say that a payload P is a sequence of blocks $\{B_1, B_2, \ldots, B_n\}$. We say that a block B is *aliased* if there exist two or more transactions $(U, \{\ldots, B, \ldots\}), (U', \{\ldots, B, \ldots\})$ with $U \neq U'$.

Table 2 shows that 12% of transactions in our workload have aliased payloads, as compared to the 54% that Kelly and Mogul observed in their WebTV trace. Furthermore, we found that aliased payloads account for only 3% of the bytes transferred, as opposed to 36% in the WebTV trace. These results indicate that our chosen workload demonstrates a comparatively small opportunity for value-based caching to reduce redundant transfers due to aliasing. As such, a clear avenue for future research is to test our algorithm on the WebTV trace. Finally, we note that Table 2 shows 18% of transactions contain aliased blocks, an additional 6% over the number that contain completely aliased payloads, indicating the importance of looking for aliasing at the block level.

4. RELATED WORK

There are a number of unique features to our algorithm and our experimental approach, but there are also a number of prior studies that touched upon many of the same ideas. We review them here.

The first work on delta encoding in the web that we know of was in the context of the WebExpress project [8], which aimed to improve the end-user experience for wireless use of the web. They noted that the responses to POST transactions often shared similar responses, and utilized delta encoding to speed up transaction processing applications with web interfaces. Deltas were computed with standard differencing algorithms. Mogul *et al.* performed a trace-driven study to determine the potential benefits of both delta encoding and data compression for HTTP transactions [14].

An innovative technique for delta encoding, called *optimistic deltas*, was introduced by Banga, Douglis, and Rabinovich [2]. In their scheme, a web cache sends an old version of a given resource to a client over a low bandwidth link immediately after receiving a request for it. It then requests the current version of the resource from the origin server, and sends a delta to the client if necessary. This approach assumes there exists enough idle time during which the origin server is being contacted to send the original response over the low bandwidth link, and the authors perform some analysis to show that such time exists. In any case, their algorithm is capable of aborting an optimistic transfer early as soon as the cache receives a more up-to-date response and decides the transfer is no longer profitable. We believe that optimistic deltas are effectively orthogonal to our technique, although we have yet to try to combine the two. Ionescu's thesis [9] describes a less aggressive approach to delta encoding.

Several studies of the nature of dynamic content on the web were performed by Wills and Mikhailov, who demonstrated that much of the response data in dynamic content is actually quite static. For example, they found that two requests with different cookies in their headers often resulted in the exact same response, or possibly responses that differed only in their included advertising content [27].

Later, they found a 75% reuse rate for the bytes of response bodies from popular e-commerce sites [28].

Several studies of which we are aware looked at the rate of change of individual web sites. Brewington and Cybenko studied the rate of change of web pages in order to predict how often search engines must re-index the web [3]; Padmanabhan and Qiu studied the rates and nature of change on the *msnbc.com* web site [17]. The latter study found a median inter-modification time of files on the server of around three hours. Looking back at our Figure 7 this result is intriguing, as we found a much higher rate of change in the actual server response bodies (recall that we sampled the site approximately every 30 minutes).

Studies of aliasing came later to the web research community. The earliest of these that we are aware of was by Douglis *et al.* [5], who studied a number of aspects of web use, including resource modification. They noted that 18% of full body responses were aliased in one trace (although the term "aliasing" was not yet used). Interestingly, they did not consider this aliasing to be a useful target of caching algorithms since most of these responses were for "uncacheable" responses. We note that caching data by its value as we have done in this study does not suffer from such a limitation.

Alias detection through content digests has been used in the past to prevent storing redundant copies of data. The Inktomi Traffic Server [12] and work by Bahn *et al.* [1] both used this approach. The main advantage of this technique is that it allows a server to scale well in the number of clients; as opposed to simply compressing cache entries, the use of digests allows common elements in the responses to distinct clients to be stored only once.

The HTTP Distribution and Replication Protocol [25] was designed to efficiently mirror content on the web and is similar to the Unix *rsync* utility [24]. It uses digests, called *content identifiers*, to detect aliasing and avoid transferring redundant data. The protocol also supports delta encoding in what they call *differential transfers*.

Santos and Wetherall presented the earliest work of which we are aware that used digests to directly suppress redundant transfers in networks by using a child and parent proxy on either end of a low bandwidth connection [22]. Spring and Wetherall added the idea of using Rabin functions to choose block boundaries [23]. We have used both of these ideas in our work. In contrast to ours, both these algorithms are targeted at the network packet layer. Spring and Wetherall note that their caching technique introduces the problem that the caches can become unsynchronized due to packet loss, and they note that more sophisticated algorithms for caching might be used. In our optimistic algorithm we address both of these problems: we allow castout decisions to be driven completely by the client, where the most information about the actual usage patterns of data is available, and we provide recovery mechanisms to deal with inconsistencies. Like the Santos and Wetherall algorithm, we use MD5 hashes rather than chains of fingerprints to name blocks. This technique removes from the server the burden of storing all of the response bodies that the client stores; for a large client population this can be a significant savings. Finally, Spring and Wetherall found only a 15% improvement in bandwidth usage using *gzip*; we find this result curious and believe it might have occurred because they were compressing packets individually, whereas the *gzip* algorithm is more efficient with larger block sizes.

Rabin fingerprinting was used earlier by Manber [11] to find similar files within a file system. Muthitacharoen, Chen, and Mazières combined his techniques with those of Spring and Wetherall to build a network file system for use over low bandwidth links [15]. In their system, before writing new data to the server, a client first sends only the digest of each new block; the server then asks for the data corresponding to any digests it does not recognize. In web parlance, their technique is successful because a great deal of file system writes are due to both resource modification and aliasing. For example, they note that a file and the backup generated for it by many popular text editors share much of the same data, but have different names. Their application is not suited to the optimistic techniques we present here, and thus their protocol requires an extra round trip. The addition of this extra round trip over a high-latency link is acceptable if the potential savings is large; we believe the small size of the average web resource precludes the use of such round trips, however.

Kelly and Mogul performed a detailed study of aliasing on the web using traces from the WebTV network and the Compaq corporate network [10]. We recommend their related work section as an excellent introduction to the space. In that work, they found a very high percentage of transactions included aliased responses (54%) versus the percentage whose responses contained redundant data due to resource modification (10%). Our workload is more limited; this limitation may explain the disparity in our observed results. Nevertheless, since aliased payloads are often incompressible, such a distribution would probably improve our relative performance versus *gzip*. Kelly and Mogul also proposed the basics of a scheme to take advantage of aliasing over low bandwidth links. They index caches by both URI and digest. In a response, the parent proxy in their scheme first sends the digests for every block to the child proxy. Then, the server may optimistically start sending the resource's data and accept cancel requests from the child, or the child may send explicit requests for the data corresponding to unrecognized digests. We are not aware of an implementation or performance results thereof for this algorithm.

Finally, another body of work addresses the aliasing problem in a different way. Chan and Woo [4] use similarity in URIs to find cached resources from the same origin server related to a given request, then use a delta encoding algorithm to compute a response based on one or more cached items. Like our algorithm, theirs uses specialized child and parent proxies, but they do not specify a mechanism for keeping these in sync. In a similar approach, Douglis, Iyengar, and Vo [7] use a fingerprinting algorithm in the style of Manber as well as URI similarity to identify related documents. In contrast to our algorithm, they use an extra round trip to coordinate the client and server. Because they use delta encoding algorithms rather than a block-based algorithm, these algorithms have the potential to save more bandwidth, as their deltas can be at a finer granularity. The addition of an extra round trip in the latter protocol eliminates the need to maintain per-client state on the server, but it may prevent the algorithm from providing an overall reduction in TTD over high-latency links. Douglis and Iyengar [6] performed a study of several bodies of data, including web resources, to quantify the ability of algorithms to recognize similarities in resources between which no relationship is known *a priori*.

5. CONCLUSIONS AND FUTURE WORK

We have presented Value-Based Web Caching, a new technique for eliminating redundant transfers of data over HTTP links. Our algorithm detects and eliminates redundant transfers due to both resource modification and aliasing. In the common case, our algorithm adds no extra round trips to a normal HTTP transaction, and it does not require any understanding of the response bodies transferred to achieve performance improvements.

We have used a detailed performance study to compare our algorithm against simple *gzip* compression, and found that it improves user-perceived time-to-display (TTD) up to 56.8%. On 58% of the web sites we studied, our algorithm achieved at least a 20% TTD improvement. Ours is the first study of which we are aware to

present performance numbers for this class of algorithm using a full featured web browser and to report TTD improvements.

There are three areas in which we would like to continue this work. First, an important step in quantifying the performance of our algorithm is the use of trace-driven simulations. Our current workload was designed to test our algorithm in the areas where we felt it would be most useful. As such, our results do not guarantee that users will see a *net* improvement in TTD by using our algorithm. Furthermore, comparing our workload with the WebTV trace used by Kelly and Mogul seems to indicate that there are other opportunities to take advantage of aliasing that are not captured by in our work to date. As such, a trace-driven simulation of our algorithm would likely provide valuable additional insight into its behavior.

Second, our current implementation uses a child proxy that is separate from the Mozilla web browser. This architecture made the implementation of our algorithm easy, but limits its performance. It hides some of the reference stream from the child proxy, degrading the quality of the information fed to the LRU algorithm with which castout decisions are made. Moreover, by having two separate caches, our effective cache size is smaller. Often, Mozilla's internal cache and the cache in the child proxy contain the same data; for fairness of evaluation, we have limited their combined size to the size of Mozilla's cache in the control case. An integrated cache should produce strictly better performance results than we have presented here.

Finally, we would like quantify the scalability of the parent proxy. As we argued in Section 2, there are good reasons to believe that the parent proxy should be able to support many simultaneous clients. It needs several orders of magnitude less storage resources than any one client, and the throughput of our block recognition and digesting algorithm is sufficient to support over 1,000 clients on a modest processor. Nonetheless, the scalability of the server plays an important role in the economic feasibility of deploying our algorithm within ISPs, so we feel it is important to quantify.

6. REFERENCES

[1] Hyokyung Bahn, Hyunsook Lee, Sam H. Noh, Sang Lyul Min, and Kern Koh. Replica-aware caching for web proxies. *Computer Communications*, 25(3):183–188, February 2002.

[2] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *Proc. of the USENIX Annual Technical Conf.*, 1997.

[3] Brian E. Brewington and George Cybenko. How dynamic is the web? In *Proc. of the 9th Intl. WWW Conf.*, 2000.

[4] Mun Choon Chan and Thomas Y. C. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proc. of IEEE INFOCOM*, March 1999.

[5] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey C. Mogul. Rate of change and other metrics: a live study of the World Wide Web. In *Proc. of the USENIX Symp. on Internet Technologies and Systems*, 1997.

[6] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. To appear in *Proc. of USENIX Annual Technical Conference*, June 2003.

[7] Fred Douglis, Arun Iyengar, and Kiem-Phong Vo. Dynamic suppression of similarity in the web: a case for deployable detection mechanisms. Technical Report RC22514, IBM Research, July 2002.

[8] Barron C. Housel and David B. Lindquist. WebExpress: a system for optimizing web browsing in a wireless environment. In *Proc. of ACM MobiCom*, 1996.

[9] Mihut D. Ionescu. xProxy: A transparent caching and delta transfer system for web objects. Master's thesis, University of California at Berkeley, December 2000.

[10] Terence Kelly and Jeffrey Mogul. Aliasing on the World Wide Web: Prevalence and performance implications. In *Proc. of the 11th Intl. WWW Conf.*, 2002.

[11] U. Manber. Finding similar files in a large file system. In *Proc. of the USENIX Winter Technical Conf.*, 1994.

[12] P. Mattis, J. Plevyak, M. Haines, A. Beguelin, B. Totty, and D. Gourley. U.S. Patent #6,292,880: "Alias-free content-indexed object cache", September 2001.

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

[14] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proc. of ACM SIGCOMM*, 1997.

[15] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. of ACM SOSP*, 2001.

[16] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proc. of ACM SIGCOMM*, 1997.

[17] Venkata N. Padmanabhan and Lili Qiu. The content and access dynamics of a busy web site: Findings and implications. In *Proc. of ACM SIGCOMM*, 2000.

[18] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[19] R.L. Rivest. The MD5 message digest algorithm. April 1992.

[20] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[21] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proc. of USENIX Summer Technical Conf.*, 1985.

[22] Jonathan Santos and David Wetherall. Increasing effective link bandwidth by suppressing replicated data. In *Proc. of USENIX Annual Technical Conference*, June 1998.

[23] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. of ACM SIGCOMM*, 2000.

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

[25] Arthur van Hoff, John Giannandrea, Mark Hapner, Steve Carter, and Milo Medin. The HTTP distribution and replication protocol. Technical Report NOTE-DRP, World Wide Web Consortium, August 1997.

[26] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. of ACM SOSP*, October 2001.

[27] Craig E. Wills and Mikhail Mikhailov. Examining the cacheability of user-requested web resources. In *Proc. of the 4th Intl. Web Caching Workshop*, 1999.

[28] Craig E. Wills and Mikhail Mikhailov. Studying the impact of more complete server information on web caching. In *Proc. of the 5th Web Caching and Content Delivery Workshop*, 2000.