DECEMBER 1997

WRL Research Report 97/4a

Potential benefits of delta encoding and data compression for HTTP (Corrected version)

Jeffrey C. Mogul
Fred Douglis
Anja Feldmann
Balachander Krishnamurthy

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There are two other research laboratories located in Palo Alto, the Network Systems Lab (NSL) and the Systems Research Center (SRC). Another Digital research group is located in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and technical notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`JOVE::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.pa.dec.com`
UUCP:	`decpa!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Reports and technical notes may also be accessed via the World Wide Web: `http://www.research.digital.com/wrl/home.html`.

Potential benefits of delta encoding and data compression for HTTP

Jeffrey C. Mogul

Digital Equipment Corporation Western Research Laboratory
mogul@wrl.dec.com

Fred Douglis
Anja Feldmann
Balachander Krishnamurthy

AT&T Labs -- Research
180 Park Avenue, Florham Park, New Jersey 07932-0971
{douglis,anja,bala}@research.att.com

December, 1997

Abstract

Caching in the World Wide Web currently follows a naive model, which assumes that resources are referenced many times between changes. The model also provides no way to update a cache entry if a resource does change, except by transferring the resource's entire new value. Several previous papers have proposed updating cache entries by transferring only the differences, or ''delta,'' between the cached entry and the current value.

In this paper, we make use of dynamic traces of the full contents of HTTP messages to quantify the potential benefits of delta-encoded responses. We show that delta encoding can provide remarkable improvements in response size and response delay for an important subset of HTTP content types. We also show the added benefit of data compression, and that the combination of delta encoding and data compression yields the best results.

We propose specific extensions to the HTTP protocol for delta encoding and data compression. These extensions are compatible with existing implementations and specifications, yet allow efficient use of a variety of encoding techniques.

This report is an expanded version of a paper in the *Proceedings of the ACM SIGCOMM '97 Conference*. It also contains corrections from the July, 1997 version of this report.

Table of Contents

List of Figures

List of Tables

1. Introduction

The World Wide Web is a distributed system, and so often benefits from caching to reduce retrieval delays. Retrieval of a Web resource (such as document, image, icon, or applet) over the Internet or other wide-area network usually takes enough time that the delay is over the human threshold of perception. Often, that delay is measured in seconds. Caching can often eliminate or significantly reduce retrieval delays.

Many Web resources change over time, so a practical caching approach must include a coherency mechanism, to avoid presenting stale information to the user. Originally, the Hypertext Transfer Protocol (HTTP) provided little support for caching, but under operational pressures, it quickly evolved to support a simple mechanism for maintaining cache coherency.

In HTTP/1.0 [3], the server may supply a ‘‘last-modified’’ timestamp with a response. If a client stores this response in a cache entry, and then later wishes to re-use the response, it may transmit a request message with an ‘‘if-modified-since’’ field containing that timestamp; this is known as a *conditional* retrieval. Upon receiving a conditional request, the server may either reply with a full response, or, if the resource has not changed, it may send an abbreviated reply, indicating that the client’s cache entry is still valid. HTTP/1.0 also includes a means for the server to indicate, via an ‘‘expires’’ timestamp, that a response will be valid until that time; if so, a client may use a cached copy of the response until that time, without first validating it using a conditional retrieval.

The proposed HTTP/1.1 specification [6] adds many new features to improve cache coherency and performance. However, it preserves the all-or-none model for responses to conditional retrievals: either the server indicates that the resource value has not changed at all, or it must transmit the entire current value.

Common sense suggests (and traces confirm), however, that even when a Web resource does change, the new instance is often substantially similar to the old one. If the difference (or *delta*) between the two instances could be sent to the client instead of the entire new instance, a client holding a cached copy of the old instance could apply the delta to construct the new version. In a world of finite bandwidth, the reduction in response size and delay could be significant.

One can think of deltas as a way to squeeze as much benefit as possible from client and proxy caches. Rather than treating an entire response as the ‘‘cache line,’’ with deltas we can treat arbitrary pieces of a cached response as the replaceable unit, and avoid transferring pieces that have not changed.

In this paper, we make use of dynamic traces of the full contents of HTTP messages to quantify the potential benefits of delta-encoded responses. Although previous papers [2, 9, 19] have proposed the use of delta encoding, ours is the first to use realistic traces to quantify the benefits. Our use of long traces from two different sites increases our confidence in the results.

We show that delta encoding can provide remarkable improvements in response-size and response-delay for an important subset of HTTP content types. We also show the added benefit of data compression, and that the combination of delta encoding and data compression yields the best results.

We propose specific extensions to the HTTP protocol for delta encoding and data compression. These extensions are compatible with existing implementations and specifications, yet allow efficient use of a variety of encoding techniques.

2. Related work

The idea of delta-encoding to reduce communication or storage costs is not new. For example, the MPEG-1 video compression standard transmits occasional still-image frames, but most of the frames sent are encoded (to oversimplify) as changes from an adjacent frame. The SCCS and RCS [17] systems for software version control represent intermediate versions as deltas; SCCS starts with an original version and encodes subsequent ones with forward deltas, whereas RCS encodes previous versions as reverse deltas from their successors. Jacobson's technique for compressing IP and TCP headers over slow links [11] uses a clever, highly specialized form of delta encoding.

In spite of this history, it appears to have taken several years before anyone thought of applying delta encoding to HTTP, perhaps because the development of HTTP caching has been somewhat haphazard. The first published suggestion for delta encoding appears to have been by Williams et al. in a paper about HTTP cache removal policies [19], but these authors did not elaborate on their design until later [18].

The possibility of compressing HTTP messages seems to have a longer history, going back at least to the early drafts of the HTTP/1.0 specification. However, until recently, it appears that nobody had attempted to quantify the potential benefits of loss-free compression, although the GloMop project [7] did explore the use of lossy compression. A study done at the World Wide Web Consortium reports on the benefits of compression in HTTP, but for only one example document [15]. Also, our traces suggest that few existing client implementations offer to accept compressed encodings of arbitrary responses (apparently, Lynx is the one exception). (Before the Web was an issue, Douglis [4] wrote generally about compression in distributed systems.)

The WebExpress project [9] appears to be the first published description of an implementation of delta encoding for HTTP (which they call ''differencing''). WebExpress is aimed specifically at wireless environments, and includes a number of orthogonal optimizations. Also, the WebExpress design does not propose changing the HTTP protocol itself, but rather uses a pair of interposed proxies to convert the HTTP message stream into an optimized form. The results reported for WebExpress differencing are impressive, but are limited to a few selected benchmarks.

Banga et al. [2] describe the use of *optimistic* deltas, in which a layer of interposed proxies on either end of a slow link collaborate to reduce latency. If the client-side proxy has a cached copy of a resource, the server-side proxy can simply send a delta. If only the server-side proxy has a cached copy, it may optimistically send its (possibly stale) copy to the client-side proxy, followed (if necessary) by a delta once the server-side proxy has validated its own cache entry with the origin server. This can improve latency by anticipating either that the resource has not changed at all, or that the changes are small. The use of optimistic deltas, unlike delta encoding, increases the number of bytes sent over the network. The optimistic delta paper, like the WebExpress paper, did not propose a change to the HTTP protocol itself, and reported results only for a small set of selected URLs.

We are also analyzing the same traces to study the rate of change of Web resources [5].

3. Motivation and methodology

Although two previous papers [2, 9] have shown that compression and delta encoding could improve HTTP performance for selected sets of resources, these did not analyze traces from ''live'' users to see if the benefits would apply in practice. Also, these two projects both assumed that existing HTTP clients and servers could not be modified, and so relied on interposing proxy systems at either end of the slowest link. This approach adds extra store-and-forward latency, and may not always be feasible, so we wanted to examine the benefits of end-to-end delta encoding and compression, as an extension to the HTTP protocol.

In this paper, we use a trace-based analysis to quantify the potential benefits from both proxy-based and end-to-end applications of compression and delta encoding. Both of these applications are supported by our proposed changes to HTTP. We also analyze the utility of these techniques for various different HTTP content-types (such as HTML, plain text, and image formats), and for several ways of grouping responses to HTTP queries. We look at several different algorithms for both delta encoding and data compression, and we examine the relative performance of high-level compression and modem-based compression algorithms.

We used two different traces in our study, made at Internet connection points for two large corporations. One of the traces was obtained by instrumenting a busy proxy; the other was made by capturing raw network packets and reconstructing the data stream. Both traces captured only references to Internet servers outside these corporations, and did not include any ''inbound'' requests. Because the two traces represent different protocol levels, time scales, user communities, and criteria for pre-filtering the trace, they give us several views of ''real life'' reference streams, although certainly not of all possible environments.

Since the raw traces include a lot of sensitive information, for reasons of privacy and security the authors of this paper were not able to share the traces with each other. That, and the use of different trace-collection methods, led us to do somewhat different analyses on the two trace sets.

3.1. Obtaining proxy traces

Some large user communities often gain access to the Web via a proxy server. Proxies are typically installed to provide shared caches, and to allow controlled Web access across a security firewall. A proxy is a convenient place to obtain a realistic trace of Web activity, especially if it has a large user community, because (unlike a passive monitor) it guarantees that all interesting activity can be traced without loss, regardless of the offered load. Using a proxy server, instead of a passive monitor, to gather traces also simplifies the task, since it eliminates the need to reconstruct data streams from TCP packets.

3.1.1. Tracing environment

We were able to collect traces at a proxy site that serves a large fraction of the clients on the internal network of Digital Equipment Corporation. Digital's network is isolated from the Internet by firewalls, and so all Internet access is mediated by proxy relays. This site, located in Palo Alto, California, and operated by Digital's Network Systems Laboratory, relayed more than a million HTTP requests each weekday. The proxy load was spread, more or less equally, across two AlphaStation 250 4/266 systems running Digital UNIX V3.2C.

To collect these traces, we modified version 3.0 of the CERN httpd code, which may be used as either a proxy or a server. We made minimal modifications, to reduce the risk of introducing bugs or significant performance effects. The modified proxy code traces a selected subset of the requests it receives:

- Only requests going to HTTP servers (i.e., not FTP or Gopher)
- Only those requests whose URL does *not* end in one of these extensions: ''.aif'', ''.aifc'', ''.aiff'', ''.au'', ''.avi'', ''.dl'', ''.exe'', ''.flc'', ''.fli'', ''.gif'', ''.gl'', ''.gz'', ''.ief'', ''.jpeg'', ''.jpg'', ''.mov'', ''.movie'', ''.mpe'', ''.mpeg'', ''.mpg'', ''.qt'', ''.snd'', ''.tiff'', ''.wav'', ''.xwd'', and ''.Z''. These URLs were omitted in order to reduce the size of the trace logs.

This pre-filtering considered only the URL in the request, not the HTTP Content-type in the response; therefore, many responses with unwanted content-types leaked through.

For each request that is traced, the proxy records in a disk file:

- Client and server IP addresses.
- Timestamps for various events in processing the request.
- The complete HTTP header and body of both the request and the response.

To allow one-pass generation of the trace logs, byte counts for the variable-sized fields (the HTTP headers and bodies) are written after the corresponding data. This means that the software which parses the logs must parse them in reverse order, but this is not especially difficult. Since the CERN proxy handles each request in a separate UNIX process, and these processes may terminate at unpredictable times, the log format includes special ''magic-number'' markers to allow the parsing software to ignore incomplete log entries.

This particular proxy installation was configured not to cache HTTP responses, for a variety of logistical reasons. This means that a number of the responses in the trace contained a full body (i.e., HTTP status code = 200) when, if the proxy had been operating as a cache, they might have instead been ''Not Modified'' responses with no body (i.e., HTTP status code = 304). The precise number of such responses would depend on the size of the proxy cache and its replacement policy. We still received many ''Not Modified'' responses, because most of the client hosts employ caches.

3.1.2. Trace duration

We collected traces for almost 45 hours, starting in the afternoon of Wednesday, December 4, 1996, and ending in the morning of December 6. During this period, the proxy site handled about 2771975 requests, 504736 of which resulted in complete trace records, and generated al-

most 9 GBytes of trace file data. (Many requests were omitted by the pre-filtering step, or because they were terminated by the requesting client.) While tracing was in progress, approximately 8078 distinct client hosts used the proxy site, which (including the untraced requests) forwarded almost 21 GBytes of response bodies, in addition to HTTP message headers (whose length is not shown in the standard proxy log format).

3.2. Obtaining packet-level traces

When a large user community is not constrained to use a proxy to reach the Internet, the option of instrumenting a proxy is not available. Instead, one can passively monitor the network segment connecting this community to the Internet, and reconstruct the data stream from the packets captured.

We collected a packet-level trace at the connection between the Internet and the network of AT&T Labs -- Research, in New Jersey. This trace represents a much smaller client population than the proxy trace. All packets between internal users and TCP port 80 (the default HTTP server port, used for more than 99.4% of the HTTP references seen at this site) on external servers were captured using *tcpdump* [14]. Packets between external users and the AT&T Labs -- Research Web server were not monitored. A negligible number of packets were lost due to buffer overruns. The raw packet traces were later processed offline, to generate an HTTP-level trace, as described in section 3.3.

Between Friday, November 8 and Monday, November 25, 1996, (17 days) we collected a total of 51,100,000 packets, corresponding to roughly 19 Gbytes of raw data. Unlike the proxy-based trace, this one was not pre-filtered to eliminate requests based on their content-type or URL extension.

3.3. Reassembly of the packet trace into an HTTP trace

The individual packets captured in the packet-level trace are not directly usable for our study; they must first be reassembled into individual TCP streams, after which the HTTP request response messages may be extracted.

Due to the huge amount of raw trace data (105 Gbytes), it was not feasible to process the entire trace as a whole. Instead, the raw trace was split into chunks (contiguous sub-sequences) of 3,100,000 packets (about 6 Gbytes), and each chunk was processed separately. Since an HTTP message might span the boundary between two chunks, each chunk overlaps with the previous chunk by 100,000 packets. This means that any given TCP stream should be present in its entirely in at least one chunk. Some streams might be fully or partially present in two chunks; we were able to eliminate duplicates by associating a timestamp with each stream. In subsequent processing, any HTTP message with a duplicate timestamp was ignored (with priority given to a full message over a partially reassembled message).

The first step in processing a chunk was to generate separate packet streams, such that all packets within a stream belong to packet paths between the same source IP address, source port number and destination IP address, destination port number. Because a client may reuse the same address tuple for a subsequent connection, the next step is to identify the individual TCP

connections within each packet path. Each TCP connection begins with an exchange of SYN packets and ends with an exchange of FIN packets, which when processed in the correct sequence, determine the extent of each connection.

Once the trace is divided into TCP connections, the packets within a connection must be converted into HTTP messages. TCP packets may be lost, reordered, corrupted, or duplicated, but by its design as a reliable stream protocol, TCP provides enough sequencing information in the packet headers for our software to reconstruct the actual data stream, except in the rare cases where our network monitor missed seeing a packet. (In these cases, we excluded the entire HTTP exchange from future analyses.)

Using these reassembled HTTP messages, the trace-processing software generates a set of files representing the body of each successful request and a log containing information about URLs, timestamps, and request and response headers. This log is fed into a Perl script that summarizes statistics and produces a trace format used as the input for later stages (see section 4.2). The trace format has one record per response, including the URL and the name of the file that stores the associated response body, as well as other fields (such as sizes and timestamps) necessary for our study.

In our traces we saw 1,322,463 requests, of which 26,591 (1.9%) had gaps, due to packet losses and the segmentation of the raw trace into chunks. Another 43,938 (3.3%) of the requests were detected as duplicates created by the overlaps between chunks. Both these sets were excluded from further analysis. To further restrict our analysis only to those references where the client received the complete HTTP response body, we included only those TCP streams for which we collected SYN and FIN packets from both client and server, or for which the size of the reassembled response body equaled the size specified in the Content-length field of the HTTP response. This left us with 1,075,209 usable responses (81% of the total).

4. Trace analysis software

Because the two traces were obtained using different techniques, we had to write two different systems to analyze them.

4.1. Proxy trace analysis software

We wrote software to parse the trace files and extract relevant HTTP header fields. The analysis software then groups the references by unique resource (URL), and to *instances* of a resource. We use the term *instance* to describe a snapshot in the lifetime of a resource. In our analyses, we group responses for a given URL into a single instance if the responses have identical last-modified timestamps and response body lengths. There may be one or more instances per resource, and one or more references per instance.

The interesting references, for the purpose of this paper, were those for which the response carried a full message body (i.e., HTTP status code = 200), since it is only meaningful to compute the difference between response bodies for just these references. Once the analysis program has grouped the references into instances, it then iterates through the references, looking for any full-body reference which follows a previous full-body reference to a different instance of the

same resource. (If two references involve the same instance, then presumably a caching proxy would have sent an If-Modified-Since request. The server then would have sent a ‘‘Not Modified’’ response, with status = 304 and no response body, rather than two identical responses.)

For each such pair of full-body responses for different instances of a resource, the analysis program computes a delta encoding for the second response, based on the first response. This is done using several different delta-encoding algorithms; the program then reports the size of the resulting response bodies for each of these algorithms.

The delta computation is done by extracting the relevant response bodies from the trace log files into temporary files, then invoking one of the delta-encoding algorithms on these files, and measuring the size of the output.

The delta-encoding algorithms that we applied include:

- *diff -e*: a fairly compact format generated by the UNIX ‘‘diff’’ command, for use as input to the ‘‘ed’’ text editor (rather than for direct use by humans).[1]
- compressed *diff -e*: the output of *diff -e*, but compressed using the *gzip* program.
- *vdelta*: this program inherently compresses its output [10].

We used *diff* to show how well a fairly naive, but easily available algorithm would perform. We also used *vdelta*, a more elaborate algorithm, because it was identified by Hunt et al. as the best overall delta algorithm, based on both output size and running time [10].

The UNIX *diff* program does not work on binary-format input files, so we restricted its application to responses whose Content-type field indicated a non-binary format; these included ‘‘text/html’’, ‘‘application/postscript’’, ‘‘text/plain’’, ‘‘application/x-javascript’’, and several other formats. *Vdelta* was used on all formats.

We ran our analysis software on an AlphaStation 600 5/333 with 640 MBytes of RAM. However, the program only used approximately 100 MBytes of virtual memory to analyze this set of traces. A typical analysis of the entire trace set took approximately 9 hours, but the analysis program has not been tuned or optimized, and the system spent a significant amount of time blocked on disk I/O.

4.2. Packet-level trace analysis software

We processed the individual response body files derived from the packet trace (see section 3.2) using a *Perl* script to compute the size of the deltas between pairs of sequentially adjacent full-body responses for the same URL, and the size of a compressed version of each full-body response. This analysis, like the proxy-trace analysis, used the *diff -e*, compressed *diff -e*, and *vdelta* algorithms to compute deltas.

[1]Because HTML files include lines of arbitrary length, and because the standard *ed* editor cannot handle long lines, actual application of this technique would require use of an improved version of *ed* [12].

While the proxy-based trace, by construction, omitted many of the binary-format responses in the reference stream, the packet-based trace included all content types. We classified these into ‘‘textual’’ and ‘‘non-textual’’ responses, using the URL extension, the Content-type HTTP response-header, or (as a last resort) by scanning the file using a variant of the UNIX *file* command.

5. Results of trace analysis

This section describes the results of our analysis of the proxy and packet-level traces.

5.1. Overall response statistics for the proxy trace

The 504736 complete records in the proxy trace represent the activity of 7411 distinct client hosts, accessing 22034 distinct servers, referencing 238663 distinct resources (URLs). Of these URLs, 100780 contained ‘‘?’’ and are classified as query URLs; these had 12004 unique prefixes (up to the first ‘‘?’’ character). The requests totalled 149 MBytes (mean = 311 bytes/message). The request headers totalled 146 MBytes (mean = 306 bytes), and the response headers totalled 81 MBytes (mean = 161 bytes). 377962 of the responses carried a full body, for a total of 2450 MB (mean = 6798 bytes); most of the other types of responses do not carry much (or any) information in their bodies. 17211 (3.4%) of the responses carried a status code of 304 (Not Modified).

Note that the mean response body size for all of the references handled by the proxy site (7773 bytes) is somewhat larger than the mean size of the response bodies captured in the traces. This is probably because the data types, especially images, that were filtered out of the trace based on URL extension tend to be somewhat larger than average.

5.2. Overall response statistics for the packet-level trace

The 1075209 usable records in the packet-level trace represent the activity of 465 clients, accessing 20956 servers, referencing 499608 distinct URLs. Of these URLs, 77112 instances (39628 distinct URLs) contained ‘‘?’’ and are classified as query URLs; these had 8054 unique prefixes (up to the first ‘‘?’’ character). 52670 of the instances (28872 distinct URLs) contained ‘‘cgi’’, and so are probably references to CGI scripts.

The mean request and response header sizes were 281 bytes and 173 bytes, respectively. 818142 of the responses carried a full body, for a total of 6104 MB of response bodies (mean = 7881 bytes for full-body responses). 145139 (13.5%) of the responses carried a status code of 304 (Not Modified). We omitted from our subsequent analyses 1144 full-body responses for which we did not have trustworthy timing data, leaving a total of 816998 fully-analyzed responses.

The mean response size for the packet-level trace is higher than that for the proxy trace, perhaps because the latter excludes binary-format responses, some of which tend to be large. The difference may also simply reflect the different user communities.

5.3. Characteristics of responses

Figure 5-1 shows cumulative distributions for total response sizes, and for the response-body size for full-body responses, for the proxy trace. Figure 5-2 shows the corresponding distributions for the packet-level trace. The median full-response body size was 3976 bytes for the proxy trace, and 3210 bytes for the packet-level traces, which implies that the packet-level trace showed larger variance in response size. Note that over 99% of the bytes carried in response bodies, in this trace, were carried in the status-200 responses; this is normal, since HTTP responses with other status codes either carry no body, or a very small one.

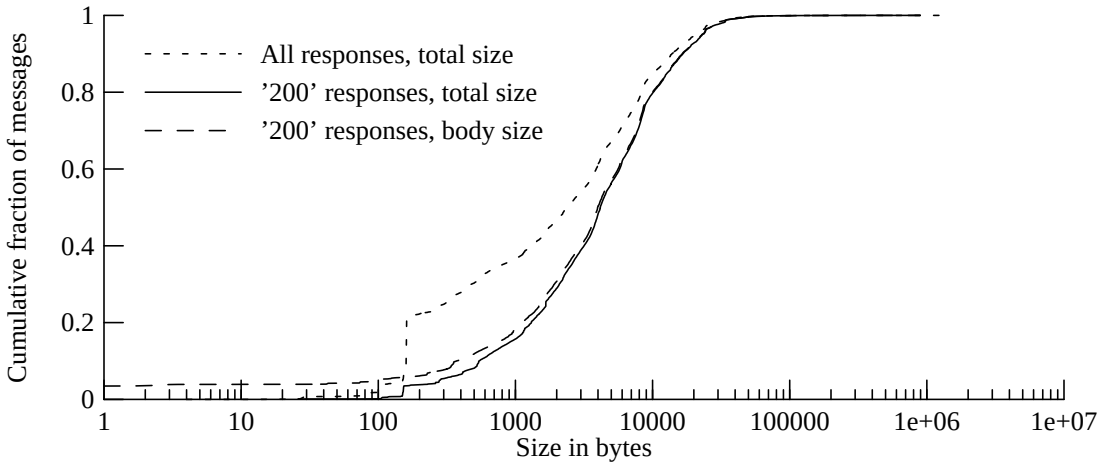

Figure 5-1: Cumulative distributions of response sizes (proxy trace)

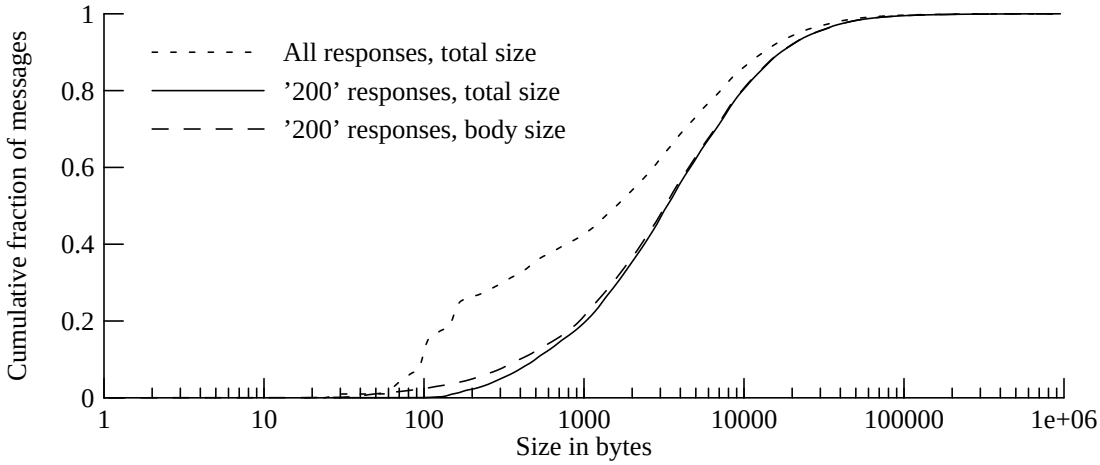

Figure 5-2: Cumulative distributions of response sizes (packet trace)

Delta encoding and/or caching are only useful when the reference stream includes at least two references to the same URL (for delta encoding), or two references to the same *(URL*, last-modified-date) instance (for caching). Figure 5-3 shows the cumulative distributions in the proxy trace of the number of references per URL, and per instance. Curves are shown both for all traced references, and for those references that resulted in a full-body response. We logged at least two full-body responses for more than half (57%) of the URLs in the trace, but only did so for 30% of the instances. In other words, resource values seem to change often enough that relatively few such values are seen twice, even for URLs that are referenced more than once. (An alternative explanation is that the values do not change, but the origin servers provide responses that do not allow caching.)

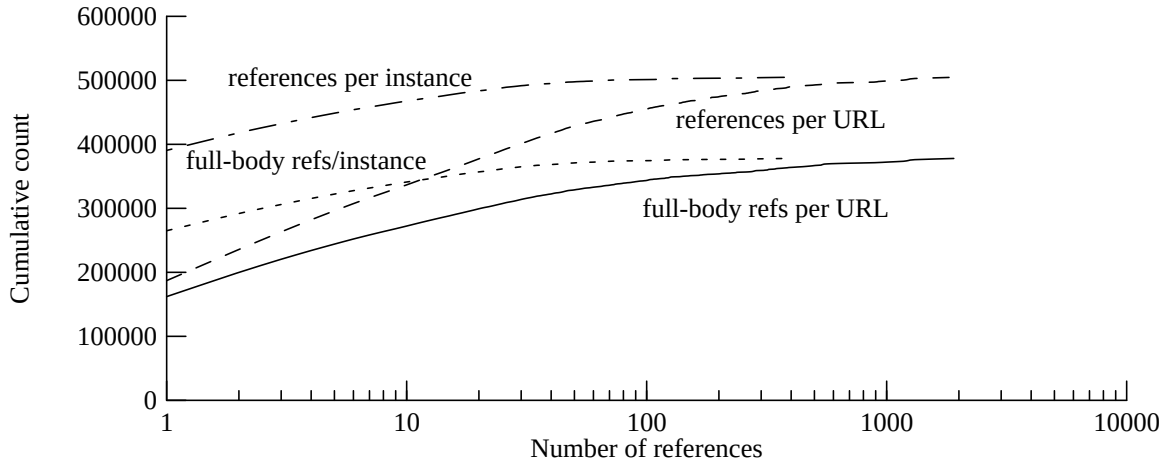

Figure 5-3: Cumulative distributions of reference counts (proxy trace)

5.4. Calculation of savings

We define a response as *delta-eligible* if the trace included at least one previous status-200 response for a different instance of the same resource. (We did not include any response that conveyed an instance identical to the previous response for the same URL, which probably would not have been received by a caching proxy.) In the proxy trace, 113356 of the 377962 status-200 responses (30.0%) were delta-eligible. In the packet-level trace, 83905 of the 816998 status-200 responses (10.3%) were delta-eligible. In the proxy trace, only 30% of the status-200 responses were excluded from consideration for being identical, compared to 32% for the packet-level trace.

We attribute much of the difference in the number of delta-eligible responses to the slower rate of change of image responses, which were mostly pre-filtered out of the proxy trace. In the packet-level trace, 66% of the status-200 responses were GIF or JPEG images, but only 3.0% of those responses were delta-eligible; in contrast, 19% of the status-200 HTML responses were delta-eligible. Some additional part of the discrepancy may be the result of the smaller client population in the packet-level traces, which might lead to fewer opportunities for sharing.

Our first analysis is based on the assumption that the deltas would be requested by the proxy, and applied at the proxy to responses in its cache; if this were only done at the individual clients, far fewer of the responses would be delta-eligible. In section 5.5.1, we analyze the per-client reference streams separately, as if the deltas were applied at the clients.

For each of the delta-eligible responses, we computed a delta using the *vdelta* program, based on the previous status-200 instance in the trace, and two compressed versions of the response, using *gzip* and *vdelta*. For those responses whose HTTP Content-type field indicated an ASCII text format (‘‘text/html’’, ‘‘text/plain’’, ‘‘application/postscript’’, and a few others), we also computed a delta using the UNIX *diff -e* command, and a compressed version of this delta, using *gzip*. 66413 (59%) of the delta-eligible responses in the proxy trace were text-format responses, as were 52361 (62%) of the delta-eligible responses in the packet-level trace.

For each response, and for each of the four computations, we measured the number of response-body bytes saved (if any). We also estimated the amount of retrieval time that would have been saved for that response, had the delta or compression technique been used. (We did not include the computational costs of encoding or decoding; see section 6 for those costs.)

Our estimate of the improvement in retrieval time is simplistic, but probably conservative. We estimated the transfer time for the response from the timestamps in our traces, and then multiplied that estimate by the fraction of bytes saved to obtain a prediction for the improved response transfer time. However, in the proxy traces it is not possible to separate the time to transmit the request from the time to receive the first part of the response, so our estimate of the original transfer time is high. We compensated for that by computing two estimates for the transfer time, one which is high (because it includes the request time) and one which is low (because it does not include either the request time, or the time for receiving the first bytes of the response). We multiplied the fraction of bytes saved by the latter (low) estimate, and then divided the result by the former (high) estimate, to arrive at our estimate of the fraction of time saved.

For the packet-level traces, we were able to partially validate this model. We measured the time it actually took to receive the packets including the first *N* bytes of an *M*-byte transfer, where *N* is the number of bytes that would have been seen if delta encoding or compression had been used. The results agree with our simpler model to within about 10%, but are still conservative (because we did not model the reduction in the size of the last data packet).

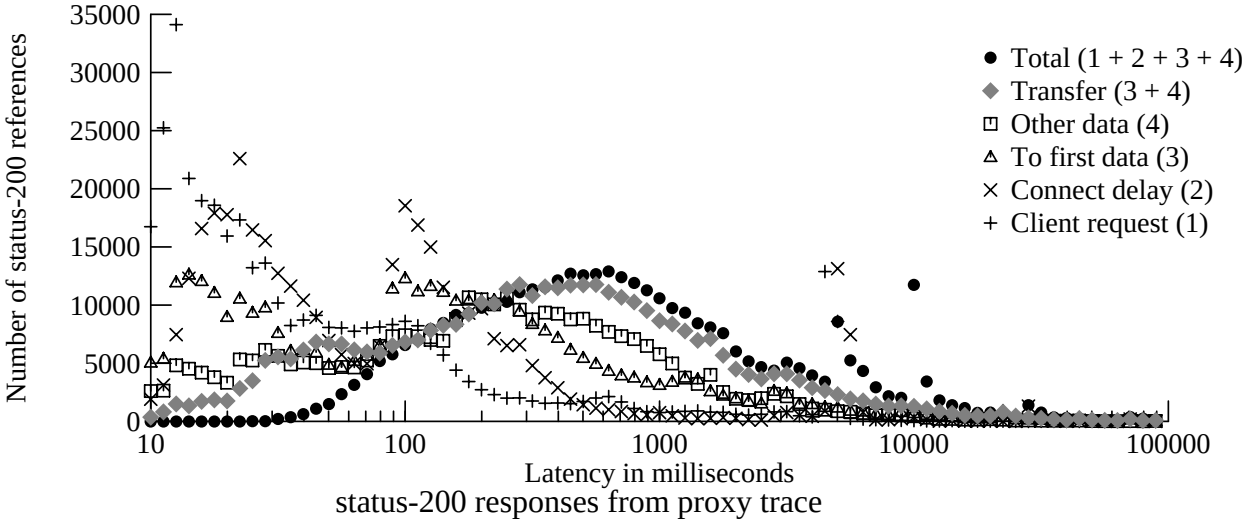

Figure 5-4: Distribution of latencies for various phases of retrieval (proxy trace)

Figure 5-4 shows the distribution of latencies for the important steps in the retrieval of full-body (status-200) responses from the proxy trace. The four steps measured are: (1) the time for the proxy to read and parse the client's request, (2) the time to connect to the server (including any DNS lookup cost), (3) the time to forward the request and to receive the first bytes of response (i.e., the first read() system call), and (4) the time to receive the rest of the response, if any. (The spikes at 5000 msec may represent a scheduling anomaly in the proxy software; the spike at 10000 msec simply represents the sum of two 5000-msec delays.) We used the sum of steps 3 and 4 as the high estimate for transfer time, and step 4 by itself as the low estimate.

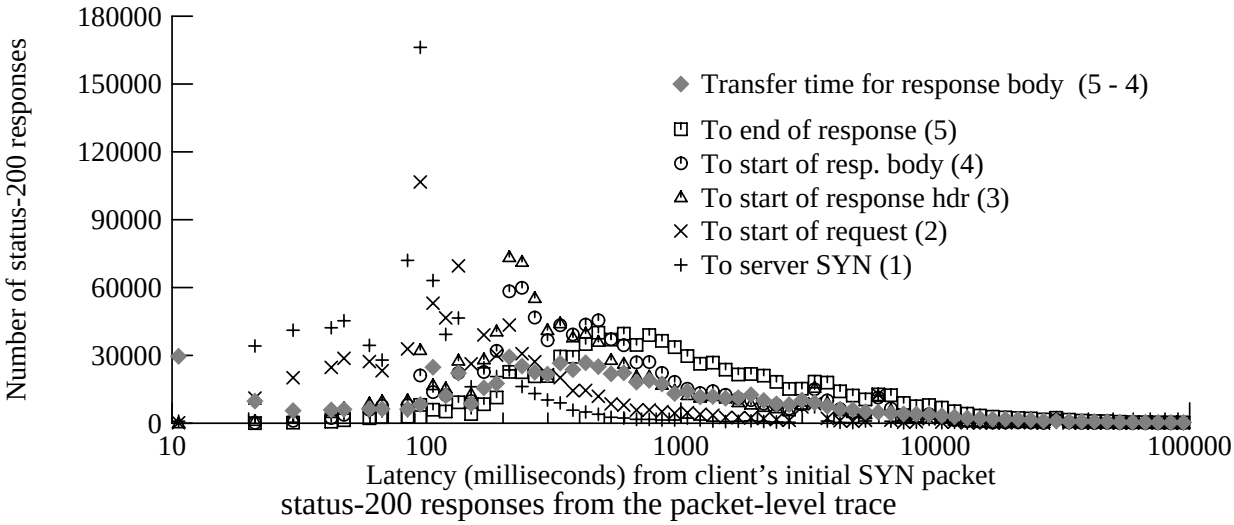

Figure 5-5: Distribution of cumulative latencies to various phases (packet-level trace)

Figure 5-5 shows a similar view of the packet-level trace. The individual steps are somewhat different (the packet-level trace exposes finer detail), and the latencies are all measured from the start of the connection (the client's SYN packet). The steps are (1) arrival of the server's SYN, (2) first packet of the HTTP request, (3) first packet of the response header, (4) first packet of the response body, and (5) end of the response. The figure also shows the transfer time for the response body, which is similar to (but smaller than) the transfer-time estimate used in figure 5-4.

5.5. Net savings due to deltas and compression

Tables 5-1 and 5-2 show (for the proxy trace) how many of the responses were improved, and by how much. Table 5-1 shows the results relative to just the delta-eligible responses; Table 5-2 shows the same results, but expressed as a fraction of all full-body responses. Because these account for more than 99% of the response-body bytes in the traces, this is also nearly equivalent to the overall improvement for all traced responses.

In tables 5-1 and 5-2, the rows labeled ‘‘unchanged’’ shows how many delta-eligible responses would have resulted in a zero-length delta. (An ‘‘unchanged’’ response is delta-eligible because its last-modified time has changed, although its body has not.) The rows labelled ‘‘diff -e’’, ‘‘diff -e | gzip’’, and ‘‘vdelta’’ show the delta-encoding results only for those responses where there is at least some difference between a delta-eligible response and the previous instance. Two other lines show the results if the unchanged responses are included. The rows labelled ‘‘vdelta compress’’ and ‘‘gzip compress’’ show the results for compressing the responses, without using any delta encoding. The final row shows the overall improvement (not including unchanged responses), assuming that the server uses whichever of these algorithms minimizes each response.

It is encouraging that, out of all of the full-body responses, tables 5-1 and 5-2 show that that 22% of the response-body bytes could be saved by using *vdelta* to do delta encoding. This implies that the use of delta encoding would provide significant benefits for textual content-types. It is remarkable that 77% of the response-body bytes could be saved for delta-eligible responses;

Computation	Improved references		MBytes saved		Retrieval time saved (seconds)	
unchanged responses	25339	(22.4%)	145	(20.8%)	11697	(7.3%)
diff -e	37806	(33.4%)	215	(30.8%)	23400	(14.6%)
diff -e (inc. *unchanged*)	63145	(55.7%)	361	(51.6%)	35098	(21.9%)
diff -e | gzip	39800	(35.1%)	264	(37.7%)	32331	(20.1%)
vdelta	86825	(76.6%)	394	(56.2%)	47647	(29.7%)
vdelta (inc. *unchanged*)	112164	(98.9%)	539	(77.0%)	59344	(37.0%)
vdelta compress	75414	(66.5%)	207	(29.6%)	27285	(17.0%)
gzip compress	73142	(64.5%)	237	(33.8%)	31567	(19.7%)
best algorithm above	112198	(99.0%)	541	(77.2%)	59490	(37.1%)

N = 113356, 701 MBytes total, 160551 seconds total

Table 5-1: Improvements assuming deltas are applied at a proxy (proxy trace, relative to all delta-eligible responses)

Computation	Improved references		MBytes saved		Retrieval time saved (seconds)	
unchanged responses	25339	(6.7%)	145	(6.0%)	11697	(2.1%)
diff -e	37806	(10.0%)	215	(8.8%)	23400	(4.2%)
diff -e (inc. *unchanged*)	63145	(16.7%)	361	(14.8%)	35098	(6.3%)
diff -e | gzip	39800	(10.5%)	264	(10.8%)	32331	(5.8%)
vdelta	86825	(23.0%)	394	(16.1%)	47647	(8.5%)
vdelta (inc. *unchanged*)	112164	(29.7%)	539	(22.0%)	59344	(10.6%)
vdelta compress	302739	(80.1%)	832	(34.0%)	104092	(18.7%)
gzip compress	289914	(76.7%)	965	(39.4%)	124045	(22.3%)
best algorithm above	340845	(90.2%)	1270	(51.9%)	152086	(27.3%)

N = 377962, 2462 MBytes total, 557373 seconds total

Table 5-2: Improvements assuming deltas are applied at a proxy (proxy trace, relative to all status-200 responses)

that is, in those cases where the recipient already has a cached copy of a prior instance. And while it appears that the potential savings in transmission time is smaller than the savings in response bytes, the response-time calculation is quite conservative (as noted earlier).

For the 88017 delta-eligible responses where the delta was not zero-length, *vdelta* gave the best result 92% of the time. *diff -e* without compression and with compression each was best for about 2% of the cases, and simply compressing the response with *gzip* worked best in 2% of the cases. Just over 1% of the delta-eligible responses were best left alone. The *vdelta* approach

clearly works best, but just using *diff -e* would save 52% of the response-body bytes for delta-eligible responses. That is, more than half of the bytes in ‘‘new’’ responses are easily shown to be the same as in their predecessors.

Computation	**Improved references**		**MBytes saved**		**Retrieval time saved (seconds)**	
unchanged responses	6332	(7.5%)	8	(1.2%)	1459	(0.8%)
diff -e	49681	(59.2%)	242	(38.2%)	56485	(30.2%)
diff -e (inc. unchanged)	59744	(71.2%)	292	(46.2%)	57943	(30.9%)
diff -e | gzip	50467	(60.1%)	280	(44.2%)	70487	(37.6%)
vdelta	73483	(87.6%)	467	(73.8%)	100073	(53.4%)
vdelta (inc. unchanged)	83546	(99.6%)	517	(81.7%)	101532	(54.2%)
vdelta compress	76257	(90.9%)	250	(39.5%)	52424	(28.0%)
gzip compress	72819	(86.8%)	277	(43.8%)	59402	(31.7%)

N = 83905, 633 MBytes total, 187303 seconds total

Table 5-3: Improvements assuming deltas are applied at a proxy (packet-level trace, relative to all delta-eligible responses)

Computation	**Improved references**		**MBytes saved**		**Retrieval time saved (seconds)**	
unchanged responses	6332	(0.8%)	8	(0.1%)	1459	(0.1%)
diff -e	49681	(6.1%)	242	(3.9%)	56485	(2.8%)
diff -e (inc. unchanged)	59744	(7.3%)	292	(4.7%)	57943	(2.8%)
diff -e | gzip	50467	(6.2%)	280	(4.5%)	70487	(3.4%)
vdelta	73483	(9.0%)	467	(7.5%)	100073	(4.9%)
vdelta (inc. unchanged)	83546	(10.2%)	517	(8.4%)	101532	(4.9%)
vdelta compress	597469	(73.1%)	1099	(17.8%)	250822	(12.2%)
gzip compress	604797	(74.0%)	1274	(20.6%)	294036	(14.3%)

N = 816998, 6193 MBytes, 2053027 seconds

Table 5-4: Improvements assuming deltas are applied at a proxy (packet-level trace, relative to all status-200 responses)

Tables 5-3 and 5-4 show, for the responses in the packet-level trace, how much improvement would be available using deltas if one introduced a proxy at the point where the trace was made. The results in table 5-3 and 5-4 are somewhat different from those in table 5-1 and 5-2 for several reasons. The packet-level trace included a larger set of non-textual content types, which leads to a reduction in the effectiveness of delta encoding and compression (see section 5.8).

Because the packet-level trace analysis uses a somewhat more accurate (and so less conservative) model for the savings in transfer time, similar reductions in the number of bytes transferred lead to different reductions in transfer time.

Taken together, the results in tables 5-1, 5-2, 5-3, and 5-4 imply that if delta encoding is possible, then it is usually the best way to transmit a changed response. If delta encoding is not possible, such as the first retrieval of a resource in a reference stream, then data compression is usually valuable.

5.5.1. Analysis assuming client-applied deltas

Computation	Improved references		MBytes saved		Retrieval time saved (seconds)	
unchanged responses	16417	(27.6%)	67	(22.8%)	6175	(5.9%)
diff -e	23072	(38.7%)	126	(42.9%)	15475	(14.7%)
diff -e (inc. *unchanged*)	39489	(66.3%)	194	(65.7%)	21650	(20.6%)
diff -e | gzip	24424	(41.0%)	157	(53.3%)	22326	(21.3%)
vdelta	42223	(70.9%)	195	(66.0%)	31047	(29.6%)
vdelta (inc. *unchanged*)	58640	(98.5%)	262	(88.8%)	37223	(35.4%)

N = 59550, 296 Mbytes, 105020 seconds

Table 5-5: Improvements assuming deltas are applied at individual clients (proxy trace, relative to delta-eligible responses)

Computation	Improved references		MBytes saved		Retrieval time saved (seconds)	
unchanged responses	16417	(4.3%)	67	(2.8%)	6175	(1.1%)
diff -e	23072	(6.1%)	126	(5.2%)	15475	(2.8%)
diff -e (inc. *unchanged*)	39489	(10.4%)	194	(7.9%)	21650	(3.9%)
diff -e | gzip	24424	(6.5%)	157	(6.4%)	22326	(4.0%)
vdelta	42223	(11.2%)	195	(8.0%)	31047	(5.6%)
vdelta (inc. *unchanged*)	58640	(15.5%)	262	(10.7%)	37223	(6.7%)

N = 377962, 2450 MBytes, 557373 seconds

Table 5-6: Improvements assuming deltas are applied at individual clients (proxy trace, relative to all status-200 responses)

Tables 5-5 and 5-6 show (for the proxy trace) what the results would be if the deltas were applied individually by each client of the proxy, rather than by the proxy itself. For delta-eligible responses, client-applied deltas perform about as well as proxy-applied deltas. However, a much smaller fraction of the responses are delta-eligible at the individual clients (19% instead

of 30%), and so the overall improvement from delta encoding is also much smaller. In other words, the utility of delta encoding depends somewhat on the large, shared cache that a proxy would provide. Alternatively, a reference stream longer than our two-day trace might show a larger fraction of per-client delta-eligible responses.

5.6. Distribution of savings

Tables 5-5 and 5-6 report mean values for improvements in the number of bytes saved, and the amount of time saved. One would not expect delta encoding to provide the same improvement for every delta-eligible response. In some cases, especially for small responses or major changes, delta encoding can save only a small fraction of the bytes. In other cases, such as a small change in a large response, delta encoding can save most of the response bytes. Figure 5-6 shows the distribution of the fraction of response bytes saved, for all delta-eligible responses in the proxy trace. (Note that the vertical axis is a log scale.)

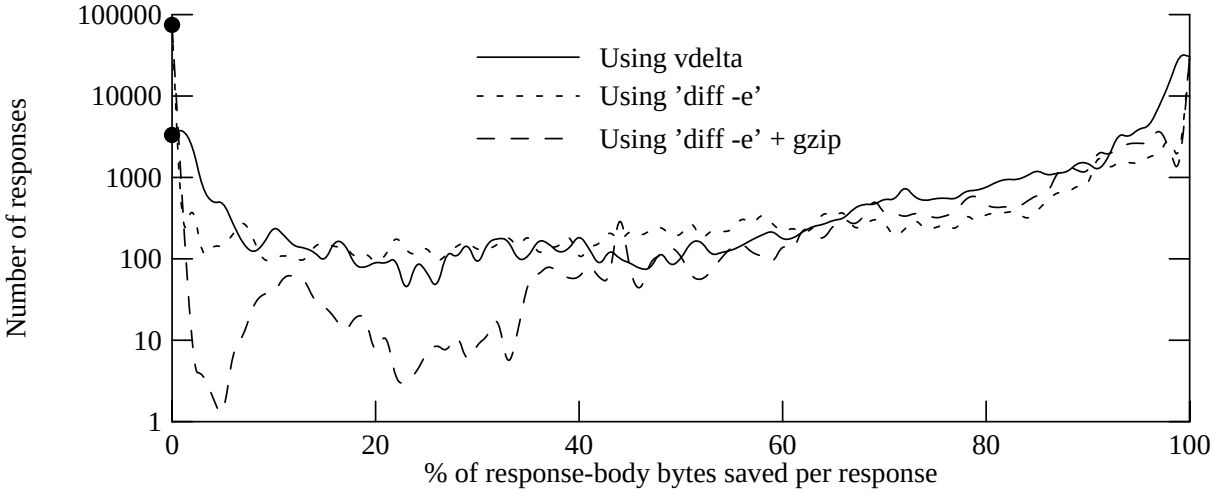

Figure 5-6: Distribution of response-body bytes saved for delta-eligible responses (proxy trace)

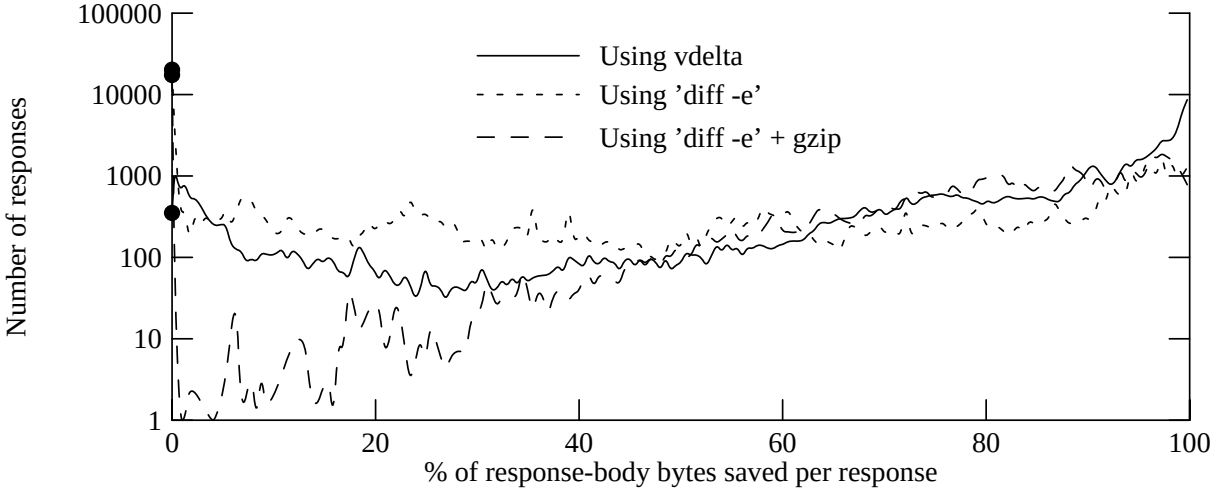

Figure 5-7: Distribution of response-body bytes saved for delta-eligible responses (packet trace)

Although delta encoding saves few or no bytes for many of the delta-eligible responses, the bimodal distribution in figure 5-6 suggests that when delta encoding does work at all, it saves

most of the bytes of a response. In fact, for delta-eligible responses in the proxy trace, the median number of bytes saved per response by delta encoding using *vdelta* is 2177 bytes (compared to a mean of 4994 bytes). For half of the delta-eligible responses, *vdelta* saved at least 96% of the response-body bytes (this includes cases where the size of the delta is zero, because the response value was unchanged). This is encouraging, since it implies that the small overhead of the extra HTTP protocol headers required to support delta encoding will not eat up most of the benefit.

Figure 5-7 shows the corresponding distribution for the packet trace. Since this trace covers all content-types, including images, the distribution differs somewhat from that in figure 5-6, but in general, both traces produce similar distributions.

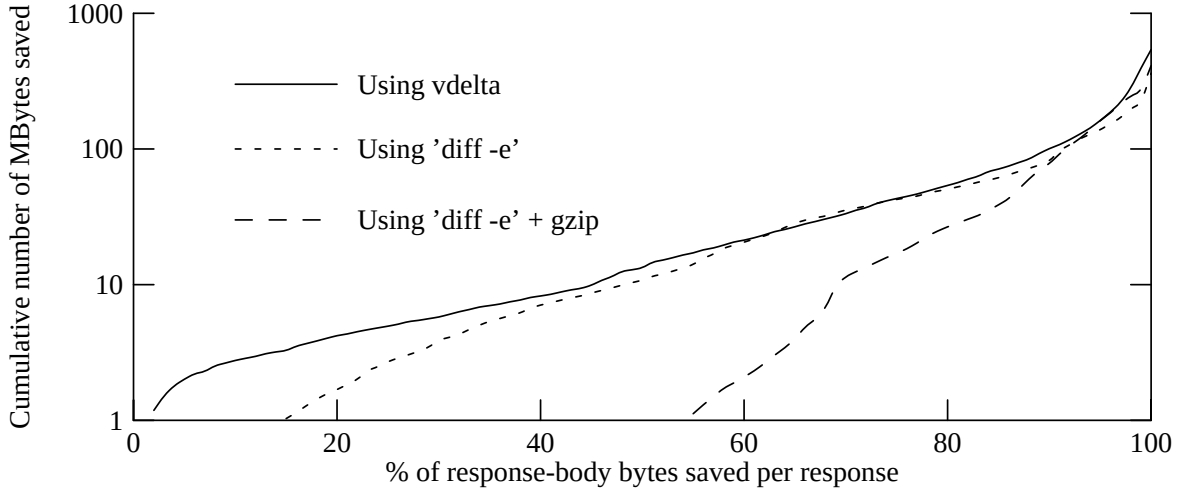

Figure 5-8: Weighted distribution of response-body bytes saved for delta-eligible responses (proxy trace)

Figure 5-8 shows the same data as in figure 5-6, except that instead of showing the number of responses improved, the vertical (logarithmic) axis shows the cumulative number of bytes saved. Essentially all of the savings comes from responses where the delta-encoded representation is less than half the size of the original representation. For example, using the *vdelta* algorithm, 13 Mbytes in total are saved from responses where delta encoding saves 50% of the original response body or less, but 527 Mbytes in total are saved from responses where delta encoding saves more than 50% of the original body size. This suggests that, from the point of view of network loading, it probably is not worth sending a delta-encoded response unless it is much smaller than the original.

Another way to look at the distribution of the results is to look at the means, standard deviations, and medians of various values. Table 5-7 shows these values for all delta-eligible responses in the traces, both for the original set of responses, and after the application of *vdelta* (wherever it decreases the response body size, and including ‘‘unchanged’’ responses where delta-encoding eliminates all of the response body bytes).

For the delta-eligible responses in the proxy trace, the mean and median savings in bytes transferred are both significant fractions of the original values. However, if one assumes that deltas are applied at the individual clients, this reduces the savings, perhaps because the set of delta-eligible responses available to an individual client has somewhat different characteristics than the set available to a proxy.

Context	Metric	Original N	Original mean	Original median	Mean savings	Median savings
Proxy trace, deltas applied at proxy	bytes	113356	6485 (8428)	3830	4994 (7572)	2177
Proxy trace, deltas applied at proxy	msec.	113356	1416 (6611)	309	524 (3349)	74
Proxy trace, deltas applied at clients	bytes	59550	5280 (8322)	2534	4624 (7787)	1934
Proxy trace, deltas applied at clients	msec.	59550	1764 (7400)	386	625 (3899)	91
Packet-level trace, deltas applied at proxy	bytes	83905	7913 (14349)	3652	5949 (11833)	2676
Packet-level trace, deltas applied at proxy	msec.	83905	2232 (9923)	610	1213 (8165)	129

Standard deviations are shown in parentheses

Table 5-7: Mean and median values for savings from *vdelta* encoding, for all delta-eligible responses

For the original set of delta-eligible responses, the proxy trace shows a large standard deviation in the retrieval time, several times the mean value. The discussion of figure 5-4 suggests some explanations.

The results for the packet-level trace are similar, although because that trace includes images, the standard deviation of the response size is larger. This may explain why the packet-level trace also shows a larger standard deviation for retrieval time, as well.

Table 5-8 shows the same measurements as table 5-7, except that the sample sets are reduced to include only those delta-eligible responses actually made shorter by *vdelta*. The results in tables 5-7 and 5-8 are quite similar, because fewer than 1% of the delta-eligible responses are excluded from table 5-8.

Table 5-9 shows the means, standards deviation, and medians for all status-200 responses, both for the original responses, and after application of *gzip* compression (wherever it decreases the response body size).

Table 5-10 shows the same measurements as table 5-9, except that the sample sets are reduced to include only those responses actually made shorter by *gzip*.

Context	Metric	Original N	Original mean	Original median	Mean savings	Median savings
Proxy trace, deltas applied at proxy	bytes	112164	6546 (8449)	3872	5047 (7594)	2259
Proxy trace, deltas applied at proxy	msec.	112164	1401 (6540)	308	529 (3367)	78
Proxy trace, deltas applied at clients	bytes	58640	5280 (8322)	2617	4695 (7825)	2006
Proxy trace, deltas applied at clients	msec.	58640	1739 (7316)	385	635 (3928)	97
Packet-level trace, deltas applied at proxy	bytes	80008	8190 (14514)	3812	6238 (12043)	2903
Packet-level trace, deltas applied at proxy	msec.	80008	2300 (10120)	640	1272 (8357)	161

Standard deviations are shown in parentheses

Table 5-8: Mean and median values for savings from *vdelta* encoding, for delta-eligible responses improved by *vdelta*

Context	Metric	Original N	Original mean	Original median	Mean savings	Median savings
Proxy trace	bytes	377962	6797 (10212)	3973	2964 (5912)	944
Proxy trace	msec.	377962	1475 (7273)	369	386 (5795)	28
Packet-level trace	bytes	817000	7948 (20468)	3258	1571 (7336)	170
Packet-level trace	msec.	817000	2513 (11731)	680	358 (4416)	9

Standard deviations are shown in parentheses

Table 5-9: Mean and median values for savings from *gzip* compression, for all status-200 responses

For the compression results in tables 5-9 and 5-10, the ratio of median savings to median original size is much smaller than for the delta-encoding results in tables 5-7 and 5-8. A similar, but less pronounced, relationship holds for the mean savings. While compression can improve a much larger fraction of the responses in our traces than delta-encoding can, in many cases the

Context	Metric	Original N	Original mean	Original median	Mean savings	Median savings
Proxy trace	bytes	308115	7573 (10900)	4719	3636 (6359)	1706
Proxy trace	msec.	308115	1683 (7853)	475	474 (6415)	68
Packet-level trace	bytes	594734	9997 (23335)	4590	2158 (8524)	346
Packet-level trace	msec.	594734	2995 (13230)	840	492 (5169)	30

Standard deviations are shown in parentheses

Table 5-10: Mean and median values for savings from *gzip* compression, for status-200 responses improved by *gzip*

savings from compression are relatively small, while when delta-encoding is applicable, it often saves most of the bytes in a response (see figure 5-6).

5.7. Time intervals of delta-eligible responses

The use of deltas for a resource implies that both the client and server must store information that might otherwise not be needed. The client must store, in its cache, one or more older instances of the resource, even if these would otherwise be replaced by the cache-management policy. The server must store either one or more pre-computed deltas, or one or more obsolete instances of the resources, from which deltas are computed.

How long must such extra storage be used before it pays off? That is, how long, after a response is sent, is the next delta-eligible response sent for the same resource?

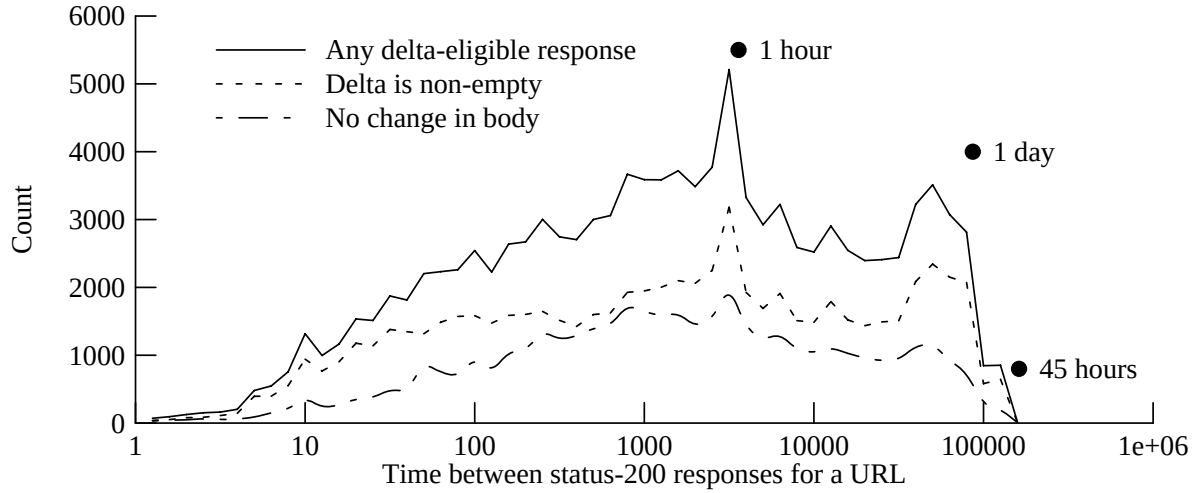

Figure 5-9: Time intervals for delta-eligible responses (proxy trace)

Figure 5-9 shows the results of a simple and preliminary study of this question. The horizontal axis shows, on a log scale, the number of seconds between pairs of non-identical (and therefore delta-eligible) status-200 responses in the proxy trace. The vertical axis shows the number of such responses seen within a given interval since its predecessor. The solid curve shows the

distribution for all delta-eligible responses; the dotted curve shows the distribution for those delta-eligible responses where the delta is non-empty (i.e., where the response body actually changes). The dashed curve shows the distribution for responses where the body itself did not change at all.

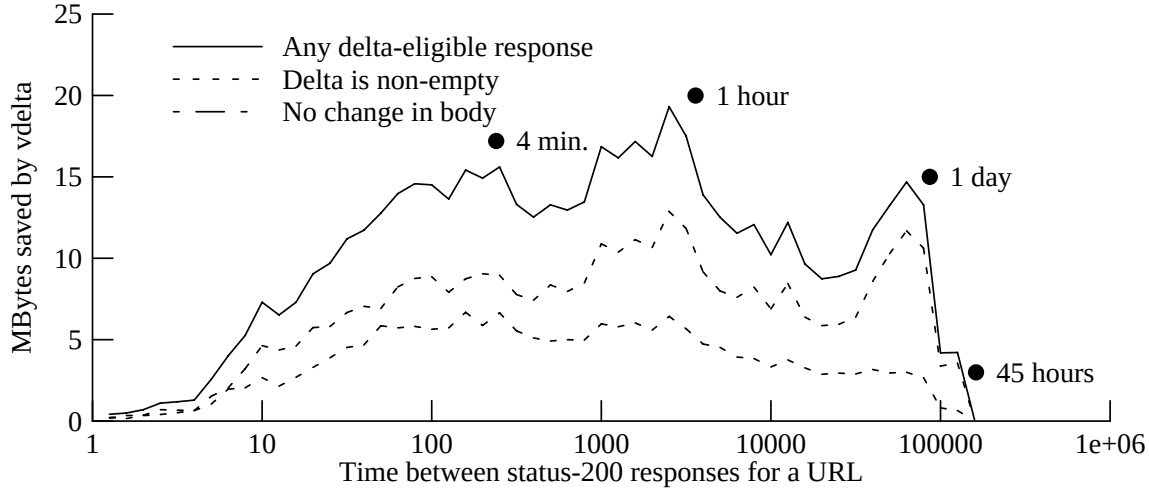

Figure 5-10: Time intervals for delta-eligible responses (proxy trace), weighted by number of bytes saved by delta encoding using *vdelta*

Figure 5-10 shows the same distributions, except that the vertical axis shows the number of bytes saved by using *vdelta* to produce delta encodings.

The distributions all show strong peaks at 3600 seconds (1 hour). This suggests that a lot of the delta-eligible responses are generated by ‘‘push’’ applications, which involve periodic updates of information such as stock quotes, news headlines, etc. Another peak appears at about 12 hours, which might also be caused by push applications.

Because the proxy trace lasted only 45 hours, the distributions in figure 5-9 obviously cannot include any intervals longer than that. Generally, one would expect that the distribution might include an artificially low number of intervals whose duration is more than half the length of the entire trace. Therefore, we would need to examine the time intervals from a much longer trace to determine if, as implied by the figure, most of the benefit from delta-encoding comes from relatively short intervals.

However, 90% of the density in the distribution is concentrated at intervals below about 14 hours (50,400 seconds). Even if one ignores the strong peaks at 1 hour and 12 hours, the bulk of the intervals shown in figure 5-9 are only a few hours long. The distributions in figure 5-10, weighted by the number of bytes saved, are even more skewed to relatively short durations. This suggests that if clients and servers can cache obsolete responses for just a few hours, they should obtain most of the available benefit from delta-encoding.

We did not do exactly the same analysis of the packet-level trace, but a similar analysis of that trace [5] provides the probability density of the intervals between successive accesses to a given URL. This shows a generally similar distribution to that in figure 5-9.

5.8. Influence of content-type on coding effectiveness

The output size of delta-encoding and data-compression algorithms depends on the nature of the input [10, 13], and so, in the case of HTTP, on the content-type of a response. The effectiveness of delta encoding also depends on the amount by which the two versions differ, which might also vary with content-type. We subdivided the packet-level traces by content-type and analyzed each subset independently, to see how important these dependencies are in practice.

Content-type	References	MBytes	Mean size	Median size	Mean time	Median time
All status-200	817000 (100%)	6193	7948	3258	2513	680
image/gif	434277 (53%)	2221	5362	2161	1896	590
text/html	184634 (23%)	1271	7216	3951	2821	720
image/jpeg	106022 (13%)	1513	14963	6834	4099	1260
application/ octet-stream	75780 (9%)	803	11113	4785	2533	660
text/plain	6988 (1%)	67	10057	3055	2799	601
image/*other*	2328 (0%)	9	4113	1408	1165	280
application/ x-msnwebqt	401 (0%)	0	344	302	235	120
application/ *other*	3789 (0%)	146	40482	748	8357	384
video/*	225 (0%)	88	409032	359012	58148	26075
text/*other*	45 (0%)	0	3701	2754	1509	471
other or unknown	2509 (0%)	75	31404	4224	6073	950

Sizes are in bytes, except as noted; times are in msec.

Table 5-11: Breakdown of status-200 responses by content-type (packet-level trace)

Table 5-11 shows the distribution of status-200 responses by content-type. A significant majority of the responses (both by count and by bytes transferred) are images, either GIF or JPEG.

Table 5-12 shows the distribution of delta-eligible responses by content-type. Although about 19% of the status-200 HTML responses are delta-eligible, only 3% of the status-200 GIF responses are.

Table 5-13 shows what fraction of the delta-eligible responses had bodies that were entirely unchanged from the previous instance. This might happen because the two requests came from separate clients, or because the server was unable to determine that an ‘‘If-Modified-Since’’ request in fact refers to an unmodified resource, or because while the resource body was not modified, some important part of the response headers did change. The table also shows other type-specific differences in the data; for example, ‘‘text/html’’ responses change more often than

Content-type	References	% of same-type status-200 responses	MBytes	% of same-type status-200 Mbytes	Mean size (bytes)
All delta-eligible	83905	10.3%	633.1	10.2%	7913
text/html	35066	19.0%	282.3	22.2%	8440
application/octet-stream	31536	41.6%	237.5	29.6%	7898
image/gif	14162	3.3%	80.9	3.6%	5992
image/jpeg	2058	1.9%	24.9	1.6%	12711
text/plain	479	6.9%	1.5	2.2%	3335
application/x-msnwebqt	256	63.8%	0.1	100.0%	386
application/*other*	143	3.8%	2.3	1.6%	16842
image/*other*	83	3.6%	0.1	4.4%	855
other or unknown	122	4.9%	3.5	4.7%	30165

Table 5-12: Breakdown of delta-eligible responses by content-type (packet-level trace)

Content-type	Delta-eligible responses	MBytes	Total time	Responses unchanged	Bytes unchanged	Time wasted
All delta-eligible	83905	633	187303	12.0%	7.9%	0.9%
text/html	35066	282	96128	4.2%	3.2%	1.2%
application/octet-stream	31536	238	61743	0.1%	0.0%	0.0%
image/gif	14162	81	22415	52.1%	36.4%	1.6%
image/jpeg	2058	25	4844	50.9%	41.5%	1.8%
text/plain	479	2	422	19.8%	17.9%	6.9%
application/x-msnwebqt	401	0	94	0%	0%	0%
application/*other*	143	2	473	25.2%	20.3%	3.3%
image/*other*	83	0	115	3.6%	11.5%	0.1%
other or unknown	170	4	1045	32.9%	30.4%	6.1%

Table 5-13: Summary of unchanged response bodies by content-type (packet-level trace)

‘‘text/plain’’ responses, but the ‘‘text/plain’’ responses that remain unchanged are smaller than the ‘‘text/plain’’ responses that do change. The last column shows a conservative estimate for the amount of time wasted in the transmission of unchanged responses.

Table 5-14 shows the delta-encoding effectiveness, broken down by content-type, for *vdelta*. This table also shows a dependency on content-type; for example, delta encoding of changed responses seems to be more effective for ‘‘text/html’’ resources than for ‘‘application/octet-stream’’ resources. (Most ‘‘octet-stream’’ resources seem to be associated with the PointCast application [16].) Somewhat surprisingly, the *vdelta* algorithm improved more than two thirds of

Content-type	Delta-eligible Responses	MBytes	Total time	Responses improved	Bytes saved	Time saved
All content-types	83905	633	187303	95.1%	75.0%	54.2%
text/html	35066	282	96128	100.0%	93.6%	61.6%
application/octet-stream	31536	238	61743	100.0%	83.6%	63.0%
image/gif	14162	81	22415	71.3%	7.8%	8.6%
image/jpeg	2058	25	4844	99.8%	8.9%	8.9%
text/plain	479	2	422	99.6%	84.1%	32.2%
application/x-msnwebqt	256	0	65	100.0%	80.2%	0.5%
application/*other*	143	2	473	89.5%	15.8%	11.0%
image/*other*	83	0	115	100.0%	84.0%	13.0%
other or unknown	122	4	1099	99.2%	50.9%	73.1%

Table 5-14: Summary of savings by content-type for delta-encoding with *vdelta*, (all delta-eligible responses in packet-level trace)

the ‘‘image/gif’’ and ‘‘image/jpeg’’ responses, albeit not reducing the byte-counts by very much (both these image formats are already compressed). We suspect that the savings may come from eliding redundant header information in these formats.

The apparent scarcity of delta-eligible images greatly reduces the utility of delta encoding when it is viewed in the context of the entire reference stream. However, we believe that in many bandwidth-constrained contexts, many users avoid the use of images, which suggests that delta encoding would be especially applicable in these contexts.

Table 5-15 shows the effectiveness of compression, using the *gzip* program, broken down by content-type. Although a majority of the responses overall were improved by compression, for some content-types compression was much less effective. It is not surprising that ‘‘image/gif’’ and ‘‘image/jpeg’’ responses could not be compressed much, since these formats are already compressed when generated. The ‘‘application/x-msnwebqt’’ responses (used in a stock-quote application) compressed nicely, but doing so would not save much transfer time at all, because the responses are already quite short.

5.9. Effect of clustering query URLs

A significant fraction of the URLs seen in the proxy trace (42% of the URLs referenced) contained a ‘‘?’’ character, and so probably reflect a query operation (for example, a request for a stock quote). By convention, responses for such URLs are uncachable, since the response might change between references (HTTP/1.1, however, provides explicit means to mark such responses as cachable, if appropriate). In this trace, 23% of the status-200 responses were for query URLs. (There are fewer status-200 responses for query URLs than distinct query URLs in the trace, because many of these requests yield a status-302 response, a redirection to a different URL.)

Content-type	Responses	MBytes	Total time	Responses improved	Bytes saved	Time saved
All status-200	816998	6193	2053027	72.8%	19.8%	14.3%
image/gif	434277	2221	823214	55.7%	4.6%	3.0%
text/html	184634	1271	520817	99.7%	68.8%	41.5%
image/jpeg	106022	1513	434607	99.1%	2.8%	2.5%
application/octet-stream	75780	803	191968	66.0%	10.3%	12.2%
text/plain	6988	67	19561	95.2%	55.6%	30.1%
image/*other*	2328	9	2713	98.6%	47.1%	27.4%
application/*other*	3789	146	31665	59.9%	28.8%	13.1%
application/x-msnwebqt	401	0	94	99.5%	56.3%	0.4%
video/*	225	88	13083	93.3%	12.6%	11.0%
text/*other*	45	0	68	100.0%	71.7%	38.1%
other or unknown	2509	75	15236	77.0%	35.0%	33.3%

Table 5-15: Summary of *gzip* compression savings by content-type (all status-200 responses in packet-level trace)

Housel and Lindquist [9], in their paper on WebExpress, point out that in many cases, the individual responses to different queries with the same ‘‘URL prefix’’ (that is, the prefix of the URL before the ‘‘?’’ character) are often similar enough to make delta encoding effective. Since users frequently make numerous different queries using the same URL prefix, it might be much more effective to compute deltas between different queries for a given URL prefix, rather than simply between different queries using an identical URL. Banga et al. [2] make a similar observation. We will refer to this technique as ‘‘clustering’’ of different query URLs with a common prefix. (Such clustering is done implicitly for POST requests, since POST requests carry message bodies, and so the response to a POST may depend on input other than the URL.)

The WebExpress paper did not report on the frequency of such clustering in realistic traces. We found, for the proxy trace, that the 100780 distinct query URLs could be clustered using just 12004 prefix URLs. Further, of the 86191 status-200 responses for query URLs, only 28186 (33%) were delta-eligible if the entire URL was used, but 76298 (89%) were delta-eligible if only the prefix had to match.

Tables 5-16 and 5-17 show that, for the proxy trace, clustering not only finds more cases where deltas are possible, but also provides significantly more reduction in bytes transferred and in response times. In fact, a comparison of tables 5-17 and 5-2 shows that when queries are clustered, delta encoding improves query response transfer efficiency more than it does for responses in general. (We note, however, that because most query responses are generated on the fly, and are somewhat shorter on average than other responses, the query processing overhead at the server may dominate any savings in transfer time.)

Computation	Improved References		MBytes saved		Retrieval time saved	
unchanged	9285	(10.8%)	12	(3.2%)	1575	(1.1%)
diff -e	4925	(5.7%)	27	(6.8%)	3437	(2.4%)
diff -e | gzip	5112	(5.9%)	34	(8.8%)	5226	(3.7%)
vdelta	18876	(21.9%)	61	(15.3%)	12217	(8.7%)

N = 86191, 419 MBytes, 141076 seconds

Table 5-16: Improvements relative to all status-200 responses to queries (no clustering, proxy trace)

Computation	Improved References		MBytes saved		Retrieval time saved	
unchanged	14044	(16.3%)	6	(1.6%)	1145	(0.8%)
diff -e	38890	(45.1%)	97	(24.4%)	9800	(6.9%)
diff -e | gzip	40438	(46.9%)	226	(56.6%)	18015	(12.8%)
vdelta	60711	(70.4%)	262	(65.6%)	24817	(17.6%)
diff -e (inc. *unchanged*)	52934	(61.4%)	103	(25.9%)	10946	(7.8%)
vdelta (inc. *unchanged*)	74755	(86.7%)	268	(67.2%)	25962	(18.4%)

N = 86191, 419 MBytes, 141076 seconds

Table 5-17: Improvements when clustering queries (all status-200 responses to queries, proxy trace)

6. Including the cost of end-host processing

The time savings calculation described in section 5.4 omits any latency for creating and applying deltas, or for compressing and decompressing responses. Since these operations are not without cost, in this section we quantify the cost of these operations for several typical hardware platforms. We chose three systems: a 50 MHz 80486 (running BSD/OS, SPECint92 = 30), which would now be considered very slow; a 90 MHz Pentium (running Linux, SPECint95 = 2.88); and a 400 MHz AlphaStation 500 (running Digital UNIX V3.2G), SPECint95 = 12.3). The 90 MHz Pentium might be typical for a home user, and the 400 MHz AlphaStation is typical of a high-end workstation, but by no means the fastest one available.

Tables 6-1, 6-2, and 6-3 show the results, which were computed from 10 trials on files (or, for deltas, pairs of instances) taken from the packet-level trace. For the delta experiments, we used 65 pairs of text files and 87 pairs of non-text files; for the compression experiments, we used 685 text files and 346 non-text files. The files were chosen to be representative of the entire set of responses. (We sorted the responses in order of size, and chose every *n*th entry to select 1% of the pairs, and 0.1% of the single-instance responses.) We express the results in terms of the throughput (in KBytes/sec) for each processing step, and for the sequential combination of the server-side and client-side processing steps. (Deltas created by *diff* are applied using the *ed*

	Text		Non-text	
Computation	**Mean**	**Std. dev.**	**Mean**	**Std. dev.**
diff -e	72	57	∅	∅
ed	90	64	∅	∅
both steps above	40	30	∅	∅
diff -e | gzip	34	29	∅	∅
gunzip | ed	15	14	∅	∅
both steps above	10	9	∅	∅
vdelta	63	46	89	61
vupdate	100	97	177	176
both steps above	38	29	56	40
gzip	73	43	57	32
gunzip	145	124	139	110
both steps above	47	31	40	24
vdelta (compress)	102	57	86	46
vupdate (decomp)	181	155	250	264
both steps above	64	41	61	39

Values are in Kbytes/sec., based on elapsed times ∅: not applicable

Table 6-1: Compression and delta encoding rates for 50 Mhz 80486 (BSD/OS 2.1)

program; deltas and compressed output created by *vdelta* are fed to the *vupdate* program.) For deltas, the throughput is calculated based on the average size of the two input files.

We also show the standard deviations of these values. The deviations are large because there is a large fixed overhead for each operation that does not depend on the size of the input, and so throughputs for the larger files are much larger than the means. Much of this fixed overhead is the cost of starting a new process for each computation (which ranges from 15 to 34 msec. on the systems tested). However, since several of the delta and compression algorithms already exist as library functions, an implementation could easily avoid this overhead[2]. The last three lines in tables 6-2 and 6-3 show measurements of a library version of the *vdelta* and *vupdate* algorithms on two of the tested platforms. The results of these tests suggest that simply eliminating the use of a separate process reduces overheads by an order of magnitude. Although the Alpha's performance for the non-library versions of *vdelta* and *vupdate* are poor, relative to the much slower

[2]The existing versions of the ''*diff -e*'' command generates output that is not entirely compatible with the *ed* command. *ed* requires one additional line in its input stream, which is normally generated by running another UNIX command. This adds significant overhead on some versions of UNIX, and since there is a simple, efficient fix for this problem, our measurements do not include the execution of this additional command.

	Text		Non-text	
Computation	**Mean**	**Std. dev.**	**Mean**	**Std. dev.**
diff -e	137	135	∅	∅
ed	102	94	∅	∅
both steps above	56	52	∅	∅
diff -e | gzip	91	83	∅	∅
gunzip | ed	38	34	∅	∅
both steps above	27	24	∅	∅
vdelta	183	160	193	133
vupdate	272	302	460	571
both steps above	106	102	122	90
gzip	101	79	106	78
gunzip	200	221	219	216
both steps above	64	54	70	57
vdelta (compress)	134	106	130	100
vupdate (decomp)	173	197	260	395
both steps above	73	65	80	74
vdelta *(library)*	925	782	1580	1661
vupdate *(library)*	1774	1172	3541	7829
both steps above	556	386	900	875

Values are in Kbytes/sec., based on elapsed times ∅: not applicable
Values larger than 10^3 might not have more than one significant digit

Table 6-2: Compression and delta encoding rates for 90 MHz Pentium (Linux 2.0.0)

Pentium, the results for the library version of *vdelta* imply that the Alpha's poor performance on the non-library code is due to some aspect of the operating system, not the CPU.

We did not make an attempt to include these costs when calculating the potential net savings in section 5.5, because (1) we have no idea of the actual performance of the end systems represented in the trace, (2) some of the computation could be done in parallel with data transfer, since all of the algorithms operate on streams of bytes (3) it would not be always necessary to produce the delta-encoded or compressed response ''on-line''; these could be precomputed or cached at the server, and (4) historical trends in processor performance promise to quickly reduce these costs.

However, we make several observations. First, the throughputs for almost all of the computations (except, on the slowest machine, for ''gunzip | ed'') are faster than a Basic-rate ISDN line (128 Kbits/sec, or 16KBytes/sec), and the library implementations of *vdelta* and *vupdate* computations are significantly faster than the throughput of a T1 line (1.544 Mbits/sec, or 193

	Text		Non-text	
Computation	**Mean**	**Std. dev.**	**Mean**	**Std. dev.**
diff -e	407	305	∅	∅
ed	1295	1420	∅	∅
both steps above	282	210	∅	∅
diff -e | gzip	153	136	∅	∅
gunzip | ed	472	408	∅	∅
both steps above	114	99	∅	∅
vdelta	150	332	188	227
vupdate	332	529	341	406
both steps above	101	134	117	136
gzip	252	152	189	139
gunzip	413	564	375	407
both steps above	148	103	122	100
vdelta (compress)	122	117	133	119
vupdate (decomp)	156	87	126	137
both steps above	66	47	63	61
vdelta *(library)*	2640	1880	3713	3461
vupdate *(library)*	5190	5325	7939	10648
both steps above	1606	1209	2246	2339

Values are in Kbytes/sec., based on elapsed times ∅: not applicable
Values larger than 10^3 might not have more than one significant digit

Table 6-3: Compression and delta encoding rates for 400 MHz AlphaStation 500 (Digital UNIX 3.2G)

KBytes/sec.) This suggests that delta encoding and compression would certainly be useful for users of dialup lines (confirming [2]) and T1 lines, would probably be useful for sites with multiple hosts sharing one T3 line, and might not be useful over broadband networks (at current levels of CPU performance).

Second, computation speed often scales with CPU performance, but not always. For example, the cost of using *ed* to apply a delta appears to depend on factors other than CPU speed. Generally, *vdelta* seems to be the most time-efficient algorithm for both delta encoding and compression, except sometimes when compared against ‘‘*diff -e*’’ (which produces much larger deltas).

Finally, the cost of applying a delta or decompressing a response is lower than the cost of creating the delta or compressed response (except for some uses of *ed*), for a given CPU. This is

encouraging, because the more expensive response-creation step is also the step more amenable to caching or precomputation.

6.1. What about modem-based compression?

Many users now connect to the Internet via a modem; in fact, most of the slowest links, and hence the ones most likely to benefit from data compression, are modem-based. Modern modems perform some data compression of their own, which could reduce the benefit of end-to-end (HTTP-based) compression. However, we believe that a program which can see the entire input file, and which has available a moderate amount of RAM, should be able to compress HTML files more effectively than a modem can.

We conducted a simple experiment to test this, transferring both plain-text and compressed versions of several HTML files via FTP over both 10 MBit/sec Ethernet LAN and modem connections. URLs for these files are listed in table 6-4; our measurements used local copies, made in January, 1997, of resources named by these URLs.

Table 6-5 shows the measurements. The modems involved were communicating at 28,800 bps, and used the V.42bis compression algorithm (a form of the Lempel-Ziv-Welch algorithm; *gzip* uses the Lempel-Ziv algorithm). We used FTP instead of HTTP for a number of reasons, including the lack of caching or rendering in FTP clients The retrieved files were written to disk at the client (a 75 MHz Intel 486 with Windows 95).

File	URL
A	http://www.w3.org/pub/WWW/Protocols/
B	http://www.w3.org/pub/WWW/
C	http://www.specbench.org/osg/cpu95/results/results.html
D	http://www.specbench.org/osg/cpu95/results/rint95.html

Table 6-4: URLs used in modem experiments

Table 6-5 shows that while the modem compression algorithms do work, and the use of high-level compression algorithms reduce the link-level bit rate, the overall transfer time for a given file is shorter with high-level compression than with modem compression. For example, the achieved transfer rate for file C using only modem compression was 55.3 Kbps (over a nominal 28.8 Kbps link), while the transfer rate for the vdelta-compressed version of the same file was only 16.3 Kbps. But, ignoring the costs of compression and decompression at the client and server, the overall transfer time for the file was 67% shorter when using high-level compression.

We found that although *vdelta* provided greater savings for large files (C and D), for the smaller files (A and B) the *gzip* algorithm apparently provides better results. It might be useful for an HTTP server generating compressed responses to choose the compression algorithm based on both the document size and the characteristics of the network path, although it could be difficult to discover if the path involves a compressing modem. In any case, using high-level compression seems almost always faster than relying on modem compression, particularly for large files.

	Size (bytes)			LAN transfer (seconds)			Modem transfer (seconds)			
File	**HTML**	**gzip**	**vdelta**	**HTML**	**gzip**	**vdelta**	**HTML**	**gzip**	**vdelta**	**saved w/*vdelta***
A	17545	6177	7997	0.17 (0.05)	0.12 (0.04)	0.10 (0.00)	6.6 (0.52)	4.7 (0.58)	5.8 (0.54)	0.8 sec (13%)
B	6017	2033	2650	0.10 (0.00)	0.10 (0.00)	0.10 (0.00)	2.2 (0.20)	1.8 (0.30)	2.0 (0.42)	0.2 sec (12%)
C	374144	39200	35212	1.22 (0.04)	0.22 (0.04)	0.20 (0.00)	66.4 (0.17)	25.1 (3.07)	21.8 (2.95)	44.6 sec (67%)
D	97125	10223	8933	0.38 (0.04)	0.12 (0.04)	0.12 (0.04)	17.7 (0.12)	6.9 (1.04)	6.1 (0.99)	11.6 sec (66%)

Times are the mean of at least 7 trials; standard deviations shown in parentheses

Table 6-5: Effect of modem-based compression on transfer time

We measured the time required to perform compression and decompression, on two systems of widely differing speed. Table 6-6 shows the costs when using the slowest system available for these tests (the 50 MHz 80486 running BSD/OS). Table 6-7 shows the costs when using the fastest system we had available (the 400 MHz AlphaStation 500 running Digital UNIX). Even when the costs of compression and decompression are included, when using the 28,800 bps modem the overall transfer time using high-level compression is still faster than the transfer time using only modem compression.

For LAN transfers, the cost of compression on the slower (50 MHz 80486) system exceeds the benefit for all but the largest file. However, the faster (400 MHz AlphaStation) system performs compression fast enough for it to be a net benefit, except for the smallest file (where we observed no change in LAN transfer time). Also, note that, since decompression requires less computation than compression, if the cost of compression can be amortized over several retrievals, even the slow system can decompress files fast enough for this to improve the overall transfer costs of files A, C, and D.

The measurements described in tables 6-6 and 6-7 used the non-library versions of the compression software (see section 6); each trial required the invocation of a separate UNIX command. With the compression and decompression algorithms integrated into the server and client software, we would expect much lower overheads for processing small responses, and so the break-even response size for end-to-end compression might be much smaller than implied by these measurements.

7. Extending HTTP to support deltas

Based on our analysis of traces, we believe that the use of deltas to update invalid HTTP cache entries could provide significant performance advantages in many cases. But in order to be feasible, a delta mechanism must impose overheads significantly smaller than the potential benefits, and it must be reasonably compatible with the existing HTTP design. In this section,

File	gzip compress	gunzip decompress	gzip total	vdelta compress	vdelta decompress	vdelta total
A	0.151	0.065	**0.216**	0.121	0.042	**0.163**
B	0.061	0.031	**0.092**	0.044	0.020	**0.064**
C	2.028	0.319	**2.347**	1.438	0.020	**1.458**
D	0.481	0.100	**0.581**	0.290	0.020	**0.310**

Mean times, in seconds, of 10 trials

Table 6-6: Compression and decompression times for files in tables 6-4 and 6-5 using 50 Mhz 80486 (BSD/OS 2.1)

File	gzip compress	gunzip decompress	gzip total	vdelta compress	vdelta decompress	vdelta total
A	0.020 (0.010)	0.007 (0.0046)	**0.027**	0.012 (0.006)	0.011 (0.0070)	**0.023**
B	0.011 (0.007)	0.006 (0.0049)	**0.017**	0.009 (0.003)	0.007 (0.0046)	**0.016**
C	0.249 (0.019)	0.026 (0.008)	**0.275**	0.106 (0.009)	0.028 (0.0204)	**0.134**
D	0.061 (0.007)	0.012 (0.006)	**0.073**	0.026 (0.008)	0.010 (0.0)	**0.036**

Mean times, in seconds, of 10 trials; standard deviations are shown in parentheses

Table 6-7: Compression and decompression times for files in tables 6-4 and 6-5 using 400 MHz AlphaStation 500 (Digital UNIX 3.2G)

we sketch how HTTP might be extended to include a delta mechanism. This is not meant as a formal proposal to extend the HTTP standard, but rather as an indication of one possible way to do so. (In particular, we do not propose protocol extensions to support query clustering, as described in section 5.9.)

We assume the use of HTTP/1.1 [6], which (while not yet widely deployed) provides much better control over caching mechanisms than do previous versions of HTTP.

7.1. Background: an overview of HTTP cache validation

When a client has a response in its cache, and wishes to ensure that this cache entry is current, HTTP/1.1 allows the client to do a ‘‘conditional GET’’, using one of two forms of ‘‘cache validators.’’ In the traditional form, available in both HTTP/1.0 and in HTTP/1.1, the client may use the ‘‘If-Modified-Since’’ request-header to present to the server the ‘‘Last-Modified’’ time-stamp (if any) that the server provided with the response. If the server’s timestamp for the resource has not changed, it may send a response with a status code of 304 (Not Modified), which does not transmit the body of the resource. If the timestamp has changed, the server would normally send a response with a status code of 200 (OK), which carries a complete copy of the resource, and a new Last-Modified timestamp.

This timestamp-based approach is prone to error because of the lack of timestamp resolution: if a resource changes twice during one second, the change might not be detectable. Therefore,

HTTP/1.1 also allows the server to provide an ‘‘entity tag’’ with a response. An entity tag is an opaque string, constructed by the server according to its own needs; the protocol specification imposes a bare minimum of requirements on entity tags. (In particular, the entity tag must change if the value of the resource changes.) In this case, the client may validate its cache entry by sending its conditional request using the ‘‘If-None-Match’’ request-header, presenting the entity tag associated with the cached response. (The protocol defines several other ways to transmit entity tags, for certain specialized kinds of requests.) If the presented entity tag matches the server’s current tag for the resource, the server should send a 304 (Not Modified) response. Otherwise, the server should send a 200 (OK) response, along with a complete copy of the resource.

In the existing HTTP protocol, a client sending a conditional request can expect either of two responses:

- status = 200 (OK), with a full copy of the resource, because the server’s copy of the resource is presumably different from the client’s cached copy.
- status = 304 (Not Modified), with no body, because the server’s copy of the resource is presumably the same as the client’s cached copy.

Informally, one could think of these as ‘‘deltas’’ of 100% and 0% of the resource, respectively. Note that these deltas are relative to a specific cached response. That is, a client cannot request a delta without specifying, somehow, which two instances of a resource are being differenced. The ‘‘new’’ instance is implicitly the current instance that the server would return for an unconditional request, and the ‘‘old’’ instance is the one that is currently in the client’s cache. The cache validator (last-modified time or entity tag) is what is used to communicate to the server the identity of the old instance.

7.2. Requesting the transmission of deltas

In order to support the transmission of actual deltas, the HTTP protocol would need to provide these features:

1. A way to mark a request as conditional.
2. A way to specify the old instance, to which the delta will be applied by the client.
3. A way to indicate that the client is able to apply a specific form of delta.
4. A way to mark a response as being delta-encoded in a particular format.

The first two features are already provided by HTTP: the presence of a conditional request-header (such as ‘‘If-Modified-Since’’ or ‘‘If-None-Match’’) marks a request as conditional, and the value of that header uniquely specifies the old instance (ignoring the problem of last-modified timestamp granularity)[3].

We defer discussion of the fourth feature, until section 7.4.

[3]It might be safe to allow the use of a last-modified timestamp when this timestamp is ‘‘strong,’’ as defined in the HTTP/1.1 specification [6].

The third feature, a way for the client to indicate that it is able to apply deltas (aside from the trivial 0% and 100% deltas), can be accomplished by transmitting a list of acceptable delta-encoding algorithms in a request-header field. The presence of this list in a conditional request indicates that the client is able to apply delta-encoded cache updates.

The ‘‘Accept-Encoding’’ request-header field defined by HTTP/1.1 provides a means for the client to indicate which *content-codings* it will accept. This header, already used to declare what compression encodings are acceptable to a client, also provides the necessary syntax to communicate a list of acceptable delta-encoding algorithms. (The original specification [6] of this header field was not suitable for this purpose, because it failed to prevent a server from sending a delta-encoding that would not be intelligible to the client.)

For example, a client might send this request:

```
GET /foo.html HTTP/1.1
If-None-Match: "123xyz"
Accept-Encoding: diff-e, vdelta, gzip
```

The meaning of this request is that:

- The client wants to obtain the current value of `/foo.html`.
- It already has a cached response for that resource, whose entity tag is ‘‘123xyz’’.
- It is willing to accept delta-encoded updates using either of two formats, ‘‘diff-e’’ (i.e., output from the UNIX ‘‘diff -e’’ command), and ‘‘vdelta’’.
- It is willing to accept responses that have been compressed using ‘‘gzip,’’ whether or not these are delta-encoded.

If, in this example, the server’s current entity tag for the resource is still ‘‘123xyz’’, then it should simply return a 304 (Not Modified) response, as would an existing server.

If the entity tag has changed, presumably but not necessarily because of a modification of the resource, the server could instead compute the delta between the instance whose entity tag was ‘‘123xyz’’ and the current instance.

We defer discussion of what the server needs to store, in order to compute deltas, until section 7.5.

We note that if a client indicates it is willing to accept deltas, but the server does not support this form of content-coding, the HTTP/1.1 specification for ‘‘Accept-encoding’’ allows the server to simply ignore this. Such a server acts as if the client had not requested a delta-encoded response: it generates a status-200 response.

7.3. Choice of delta algorithm

The server would not be required to transmit a delta-encoded response. For example, the result might be larger than the current size of the resource. The server might not be able to compute a delta for this type of resource (e.g., a compressed binary format). The server might not have sufficient CPU cycles for the delta computation. The server might not support any of the delta formats supported by the client. Or, finally, the network bandwidth might be high

enough that the delay involved in computing the delta is not worth the delay avoided by sending a smaller response.

Given that the server does want to compute a delta, and the set of encodings its supports has more than one encoding in common with the set offered by the client, which encoding should it use? We believe that there are a number of possible approaches. For example, if CPU cycles are plentiful and network bandwidth is scarce, the server might compute each of the possible encodings and then send the smallest result. Or the server might use heuristics to choose an encoding algorithm, based on things such as the type of the resource, the current size of the resource, and the expected amount of change between instances of the resource.

Note also that it may pay to cache the deltas internally to the server, if a resource is typically requested by several different delta-capable clients between modifications. In this case, the cost of computing a delta may be amortized over many responses, and so the server might use a more expensive computation.

7.4. Transmission of deltas

When a server transmits a delta-encoded response, it must identify it as such, and must also indicate which encoding format is used. HTTP/1.0 provides the ‘‘Content-encoding’’ header to mark responses that have been encoded with one or more algorithms. It might be possible to extend the list of supported content-encodings to include delta algorithms.

However, a simplistic application of this approach would cause serious problems if the response flows through an intermediate (proxy) cache that is not cognizant of the delta mechanism. Because the Internet is full of HTTP/1.0 caches, which might never be entirely replaced, and because the HTTP specifications insist that message recipients ignore any header field that they do not understand, a non-delta-capable proxy cache that receives a delta-encoded response might store that response, and might later return it to a non-delta-capable client that has made a request for the same resource. This naive client would believe that it has received a valid copy of the entire resource, with predictably unpleasant results.

Instead, we propose that delta-encoded responses be identified as such using a new HTTP status code; for specificity in the discussion that follows, we will assume the use of the (currently unassigned) code of 266 for this purpose. There is some precedent for this approach: the HTTP/1.1 specification introduces the 206 (Partial Content) status code, for the transmission of sub-ranges of a resource. Existing proxies apparently forward responses with unknown status codes, and do not attempt to cache them.[4]

Given this, a delta-encoded response differs from a standard response in three ways:

1. It carries a status code of 266 (Delta).

[4]An alternative to using a new status code would be to use the ‘‘Expires’’ header to prevent HTTP/1.0 caches from storing the response, then use ‘‘Cache-control: max-age’’ (defined in HTTP/1.1) to allow more modern caches to store delta-encoded responses. This adds many bytes to the response headers, and so would reduce the effectiveness of delta encoding.

2. It carries a ‘‘Content-encoding’’ header, indicating which delta encoding is used in this response.
3. Its body is a delta-encoding of the current instance, rather than a full copy of the instance.

For example, a response to the request given in section 7.2 might look like:

```
HTTP/1.1 266 Delta
ETag: "489uhw"
Content-Encoding: vdelta
```

(when showing examples of HTTP messages, we show only some of the required header fields, and we do not show the response body).

7.5. Management of base instances

If the time between modifications of a resource is less than the typical eviction time for responses in client caches, this means that the ‘‘old instance’’ indicated in a client’s conditional request might not refer to the most recent prior instance. This raises the question of how many old instances of a resource should be maintained by the server, if any.

There are many possible options; for example:

- The server might not store any old instances, and so would never respond with a delta.
- The server might only store the most recent prior instance; requests attempting to validate this instance could be answered with a delta, but requests attempting to validate older instances would be answered with a full copy of the resource.
- The server might store all prior instances, allowing it to provide a delta response for any client request.
- The server might store only a subset of the prior instances, which we call ‘‘base instances.’’

The server might not have to store prior instances explicitly. It might, instead, store just the deltas between specific base instances and subsequent instances (or the inverse deltas between base instances and prior instances). This approach might be integrated with a cache of computed deltas.

None of these approaches necessarily requires additional protocol support. However, suppose that a server administrator wants to store only a subset of the prior instances, but would like the server to be able to respond using deltas as often as possible.

We identify two additional protocol changes to help solve this problem, although neither of them is fully elaborated here.

The first approach uses an existing feature of the ‘‘If-None-Match’’ header, its ability to carry more than one entity-tag. This feature was included in HTTP/1.1 to support efficient caching of multiple variants of a resource, but it is not restricted to that use.

Suppose that a client has kept more than one instance of a resource in its cache. That is, not only does it keep the most recent instance, but it also holds onto copies of one or more prior,

invalid instances. (Alternatively, it might retain sufficient delta or inverse-delta information to reconstruct older instances.) In this case, it could use its conditional request to tell the server about all of the instances it could apply a delta to. For example, the client might send:

```
GET /foo.html HTTP/1.1
If-None-Match: "123xyz", "337pey", "489uhw"
Accept-Encoding: diff-e, vdelta
```

to indicate that it has three instances of this resource in its cache. If the server is able to generate a delta from any of these prior versions, it can select the appropriate base version, compute the delta, and return the result to the client.

In this case, however, the server must also tell the client which base instance to use, and so we need to define a response-header for this purpose. For example, the server might reply:

```
HTTP/1.1 266 Delta
ETag: "1acl059"
Content-Encoding: vdelta
Delta-base: "337pey"
```

This response tells the client to apply the delta, using the *vdelta* encoding, to the cached response with entity tag ‘‘337pey’’, and to associate the entity tag ‘‘1acl059’’ with the result. (It may be desirable to include a Delta-base value in every delta-encoded response, even when the client supplies only one entity tag.)

Of course, if the server has retained more than one of the prior instances identified by the client, this could complicate the problem of choosing the optimal delta to return, since now the server has a choice not only of the delta algorithm, but also of the base instance to use.

We also believe that it might be useful to allow the server to tell the client, in advance of a conditional request, which instances of a resource are useful base instances for deltas. That is, if the server intends to retain certain instances and not others, it could label the responses that transmit the retained instances. This would help the client manage its cache, since it would not have to retain all prior versions on the possibility that only some of them might be useful later. The label would be a ‘‘hint’’ to the client, not a promise that the server will indefinitely retain an instance.

Such labeling could be done through the use of yet another response header, but it might be more efficient to simply add a directive to the existing ‘‘Cache-control’’ header. For example, in response to an unconditional request, the server might send:

```
HTTP/1.1 200 OK
ETag: "337pey"
Cache-control: retain
```

to tell a delta-capable client to retain this version. The ‘‘retain’’ directive could also appear in a delta response:

```
HTTP/1.1 266 Delta
ETag: "1acl059"
Cache-control: retain
Content-Encoding: vdelta
Delta-base: "337pey"
```

In practice, the ‘‘Cache-control’’ response-header field might already be present, so the cost (in bytes) of sending this directive might be smaller than this example implies.

7.6. Deltas and intermediate caches

Although we have designed the delta-encoded responses so that they will not be stored by naive proxy caches, if a proxy does understand the delta mechanism, it might be beneficial for it to participate in sending and receiving deltas.

A proxy could participate in several independent ways:

- In addition to forwarding a delta-encoded response, it might store it, and then use it to reply to a subsequent request with a compatible ‘‘If-None-Match’’ field (i.e., one that is either a superset of the corresponding field of the request that first elicited the response, or one that includes the ‘‘Delta-base’’ value in the cached response), and with a compatible ‘‘Content-encoding’’ field (one that includes the actual delta-encoding used in the response.) Of course, such uses are subject to all of the other HTTP rules concerning the validity of cache entries.
- In addition to forwarding a delta-encoded response, it might apply the delta to the appropriate entry in its own cache, which could then be used for later responses (even from non-delta-capable clients).
- When it receives a conditional request from a delta-capable client, and it has a complete copy of an up-to-date (‘‘fresh,’’ in HTTP/1.1 terminology) response in its cache, it could generate a delta locally and return it to the requesting client. (Or it might decide that it would be more efficient to return the entire response from its cache, rather than forwarding the delta request over a busy network.)
- When it receives a request from a non-delta-capable client, it might convert this into a delta request before forwarding it to the server, and then (after applying a resulting delta response to one of its own cache entries) it would return a full-body response to the client.

All of these techniques increase proxy software complexity, and might increase proxy storage or CPU requirements. However, if applied carefully, they should help to reduce the latencies seen by end users, and load on the network. Generally, CPU speed and disk costs are improving faster than network latencies, so we expect to see increasing value available from complex proxy implementations.

Because it is possible for a proxy to store a delta-encoded response and then return it for a subsequent request, we believe that the server must always specify the delta-encoding method in a delta-encoded response, even if the client has only offered one possibly encoding algorithm. Otherwise, it may be impossible to know if the response is compatible with the delta-capable request from a different client.

7.7. Quantifying the protocol overhead

The proposed protocol changes increase the size of the HTTP message headers slightly. In the simplest case, the request headers for conditional requests (i.e., those for which the client already has a cache entry) would be about 24 bytes longer, or about 9% of the observed mean request

size[5]. Because a client must have an existing cache entry to use as a base for a delta-encoded response, it would never send ‘‘Accept-encoding: vdelta’’ for unconditional requests. Also, unless it supports a delta-encoding suitable for use with images, the client would presumably omit this request-header field for image content-types. Therefore, the mean increase in request header size would be much less than 9%.

Delta-encoded responses would carry slightly longer headers (about 24 bytes in the simplest case), but this would be insignificant compared to the reduction in body size: about 1% of the median reduction shown in table 5-8. The header size for non-delta-encoded responses would not change.

7.8. Ensuring data integrity

When a recipient reassembles a complete HTTP response from several individual messages, it might be necessary to check the integrity of the complete response. For example, the client’s cache might be corrupt, or the implementation of delta-encoding (either at client or server) might have a bug.

HTTP/1.1 includes mechanisms for ensuring the integrity of individual messages. A message may include a ‘‘Content-MD5’’ response header, which provides an MD5 message digest of the body of the message (but not the headers). The Digest Authentication mechanism [8] provides a similar message-digest function, except that it includes certain header fields. Neither of these mechanisms makes any provision for covering a set of data transmitted over several messages, as would be the case for the result of applying a delta-encoded response.

It might therefore be necessary to define a new message header, ‘‘Delta-MD5’’, which carries an MD5 message digest of the final reassembled response value, rather than of the delta value. One might still want to use the Digest Authentication mechanism, or something stronger, to protect delta messages against tampering.

7.9. Implementation experience

To verify the feasibility of these HTTP extensions, we implemented a somewhat simplified version of the extended protocol.

For the server, we started with Apache (version 1.2b7). Apache is the most widely used server on the Internet, and is freely available in source form. The original code is roughly 30,000 lines of C code. Our changes added roughly 400 lines. 100 of these lines were changes to Apache source and header files, and the rest were in a library to interface between Apache and several algorithms for delta-encoding and compression. The *vdelta* algorithm itself is implemented in about 1900 lines of C code; we used external programs for *gzip* and *diff*.

[5]This might be significantly reduced by using a proposed ‘‘sticky header’’ mechanism: when multiple HTTP messages are sent on a single TCP connection, headers that would be repeated verbatim in each message could, in principle, be suppressed.

Because our changes were small, localized, and triggered only when specific HTTP headers were present in incoming requests, there was no appreciable difference in the performance of the server when the new features were not used. We have not gathered comparative performance data on the various differencing and compression algorithms.

We are currently working on a client implementation of these HTTP extensions. This work is based on the ‘‘bimodal proxy’’ software described by Banga et al. [2].

8. Future work

We have not been able to explore all aspects of delta encoding in this study. Here we briefly discuss several issues that could be addressed using a trace-based analysis. Of course, the most important proof of the delta-encoding design would be to implement it and measure its utility in practice, but because many variations of the basic design are feasible, additional trace-based studies might be necessary to establish the most effective protocol design. (The previous studies [2, 9] did implementations, but using a double-proxy-based approach that adds store-and-forward delays.)

We also note that all of our analyses would benefit from a more accurate model for the transfer time. This might include more precise measurements in the traces, a model of the slow-start state of the TCP stream, and perhaps even a model of network congestion.

8.1. Delta algorithms for images

A significant fraction of the responses in our traces (or logged but not traced by the proxy), and an even larger fraction of the response body bytes, were of content-type ‘‘image/*’’ (i.e., GIF, JPEG, or other image formats). Delta-eligible image responses are relatively rare, but if these could be converted to small deltas, that would still save bandwidth. While *vdelta* appears capable of extracting deltas from some pairs of these image files, it performs much worse than it does on text files. We also have evidence that *vdelta* does poorly on images generated by cameras, such as the popular ‘‘WebCam’’ sites, many of which are updated at frequent intervals. MPEG compression of video streams relies on efficient deltas between frames, so we have some hopes for a practical image-delta algorithm.

8.2. Effect of cache size on effectiveness of deltas

Our trace analyses assumed that a client (or proxy) could use any previously traced response as the base instance for a delta. Although in many cases the two responses involved appear close together in the trace, in some cases the interval might be quite large. This implies that, in order to obtain the full benefits of delta encoding shown in our analyses, the client or proxy might have to retain many GBytes of cached responses. If so, this would clearly be infeasible for most clients.

It would be fairly simple to analyze the traces using a maximum time-window (e.g., 1 hour or 24 hours) rather than looking all the way back to the beginning of the trace when searching for a base instance. By plotting the average improvement as a function of the time-window length,

one could see how this parameter affects performance. It might be somewhat harder to model the effect of a limited cache size. The preliminary analysis of time intervals in section 5.7 suggests that, in reality, the storage requirements might be quite modest.

8.3. Deltas between non-contiguous responses

Our analyses of delta-eligible responses looked only at the most recent status-200 response preceding the one for which a delta was computed. This policy simplifies the analysis, and would also simplify both the client and server implementations, since it limits the number of previous instances that must be stored at each end.

It is possible, however, that reductions in the delta sizes might be possible by computing deltas between the current instance and several previous instances, and then sending the shortest. The complexity and space and time overheads of this policy are significant, but the policy would not be hard to support in the protocol design (see section 7.5). We could modify our trace analysis tools to evaluate the best-case savings of such policies.

8.4. Avoiding the cost of creating deltas

The response-time benefits of delta encoding are tempered by the costs of creating and applying deltas. However, as shown in section 6, the cost of creating a delta is usually much larger than the cost of applying it.

Fortunately, it may be possible to avoid or hide the cost of creating deltas, in many cases. Whenever a server receives several requests that would be answered with the same delta-encoded responses, it could avoid the computation cost of delta-creation by simply caching the delta. We could estimate the savings from this technique by counting the number of status-304 (Not Modified) and unchanged responses for a given URL, following a delta-eligible response for that URL in the trace. (The estimate would be conservative, unless the trace included the server's entire reference stream.)

Even when a delta is used only once, it may be possible for the server to hide the cost of creating it by precomputing and caching the delta when the resource is actually changed, rather than waiting for a request to arrive. While this might substantially increase the CPU and disk load at the server (because it would probably result in the creation of many deltas that will never be used), it should reduce the latency seen by the client, especially when the original files are large. Many studies have shown that Web server loads are periodic and bursty at many time scales (e.g., [1]). If the server sometimes has background cycles to spare, why not spend them to precompute some deltas?

8.5. Decision procedures for using deltas or compression

While our results show that deltas and compression improve overall performance, for any given request the server's decision to use delta encoding, compression, or simply to send the unmodified resource value may not be a trivial one. It would not make much sense for the server to spend a lot more time deciding which approach to use than it would take to transfer the un-

modified value. The decision might depend on the size and type of the file, the network bandwidth to the client, perhaps the presence of a compressing modem on that path (see section 6.1), and perhaps the past history of the resource. We believe that a simple decision algorithm would be useful, but we do not yet know how it should work.

9. Summary and conclusions

Previous studies have described how delta encoding and compression could be useful. In this study, we quantified the utility based on traces of actual Web users. We found that, using the best known delta algorithm, for the proxy trace 77% of the delta-eligible response-body bytes and 22% of all response-body bytes could have been saved; at least 37% of the transfer time for delta-eligible responses and 11% of the total transfer time could have been avoided. For the packet-level trace, we showed even more savings for delta-eligible responses (82% of response-body bytes), although the overall improvement (8% of response-body bytes) was much less impressive. We confirmed that data compression can significantly reduce bytes transferred and transfer time, for some content-types. We showed that the added overheads for encoding and decoding are reasonable, and support for deltas would add minimal complexity to the HTTP protocol. We conclude that delta encoding should be used when possible, and compression should be used otherwise.

The goal for a well-designed distributed system should be to take maximal advantage of caches, and to transmit the minimum number of bits required by information theory, given acceptable processing costs. delta encoding and compression together will help meet these goals.

Acknowledgments

We would like to thank Kiem-Phong Vo for discussions relating to *vdelta*, Glenn Fowler for discussions regarding *diff -e* update problems, Guy Jacobson for discussions regarding *vdelta* compression, and Manolis Tsangaris and Steven Bellovin for discussions relating to modem compression. We would also like to thank Kathy Richardson and Jason Wold, for help in obtaining traces; Gaurav Banga and Deborah Wallach, for proofreading; and the SIGCOMM ’97 reviewers, for valuable suggestions.

References

[1] Martin F. Arlitt and Carey L. Williamson. *Web Server Workload Characterization: The Search for Invariants (Extended Version)*. DISCUS Working Paper 96-3, Dept. of Computer Science, University of Saskatchewan, March, 1996.
ftp://ftp.cs.usask.ca/pub/discus/paper.96-3.ps.Z.

[2] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proc. 1997 USENIX Technical Conference*, pages 289-303. Anaheim, CA, January, 1997.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol -- HTTP/1.0*. RFC 1945, HTTP Working Group, May, 1996.

[4] Fred Douglis. On the Role of Compression in Distributed Systems. In *Proc. Fifth ACM SIGOPS European Workshop*. Mont St.-Michel, France, September, 1992. Also appears in *Operating Systems Review* 27(2):88-93, April, 1993.

[5] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of Change and Other Metrics: a Live Study of the World Wide Web. In *Proc. Symposium on Internet Technologies and Systems*. USENIX, Monterey, CA, December, 1997. To appear.

[6] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, and Tim Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2068, HTTP Working Group, January, 1997.

[7] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to Network and Client Variation via On-Demand Dynamic Transcoding. In *Proc. ASPLOS VII*, pages 160-170. Cambridge, MA, October, 1996.

[8] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, L. Stewart. *An Extension to HTTP: Digest Access Authentication*. RFC 2069, HTTP Working Group, January, 1997.

[9] Barron C. Housel and David B. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. In *Proc. 2nd Annual Intl. Conf. on Mobile Computing and Networking*, pages 108-116. ACM, Rye, New York, November, 1996. http://www.networking.ibm.com/art/artwewp.htm.

[10] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. An Empirical Study of Delta Algorithms. In *IEEE Soft. Config. and Maint. Workshop*. 1996.

[11] Van Jacobson. *Compressing TCP/IP Headers for Low-Speed Serial Links*. RFC 1144, Network Working Group, February, 1990.

[12] Glenn Fowler, David Korn, Stephen North, Herman Rao, and Kiem-Phong Vo. Libraries and File System Architecture. In Balachander Krishnamurthy (editor), *Practical Reusable UNIX Software*, chapter 2. John Wiley & Sons, New York, 1995.

[13] Debra A. Lelewer and Daniel S. Hirschberg. Data Compression. *ACM Computing* Surveys 19(3):261-296, 1987.

[14] Steven McCanne and Van Jacobson. An Efficient, Extensible, and Portable Network Monitor. Work in progress.

[15] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *Proc. SIGCOMM '97*. Cannes, France, September, 1997.

[16] PointCast Corporation. What is the PointCast Network? http://www.pointcast.com/. 1997

[17] W. Tichy. RCS - A System For Version Control. *Software - Practice and Experience* 15(7):637-654, July, 1985.

[18] Stephen Williams. Personal communication. June, 1996. http://ei.cs.vt.edu/~williams/DIFF/prelim.html.

[19] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox . Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. SIGCOMM '96*, pages 293-305. Stanford, CA, August, 1996.

WRL Research Reports

‘‘Titan System Manual.’’ **Michael J. K. Nielsen.** WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’ **David W. Wall.** WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’ **William R. Hamburgen.** WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’ **David W. Wall and Michael L. Powell.** WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’ **Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.** WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’ **Christopher A. Kent, Jeffrey C. Mogul.** WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’ **Christopher A. Kent.** WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’ **David W. Wall.** WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’ **Paul J. Asente.** WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’ **Brian K. Reid.** WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’ **Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.** WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’ **Jeremy Dion.** WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’ **Joel F. Bartlett.** WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’ **Jeffrey C. Mogul.** WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’ **David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.** WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’ **Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.** WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’ **Joel F. Bartlett.** WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’ **Silvio Turrini.** WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’ **William R. Hamburgen.** WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’ **Jeffrey C. Mogul.** WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’ **V. Srinivasan and Jeffrey C. Mogul.** WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’ **Norman P. Jouppi and David W. Wall.** WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’ **Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.** WRL Research Report 89/8, July 1989.

''Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.'' **Norman P. Jouppi.** WRL Research Report 89/9, July 1989.

''Integration and Packaging Plateaus of Processor Performance.'' **Norman P. Jouppi.** WRL Research Report 89/10, July 1989.

''A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.'' **Norman P. Jouppi and Jeffrey Y. F. Tang.** WRL Research Report 89/11, July 1989.

''The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.'' **Norman P. Jouppi.** WRL Research Report 89/13, July 1989.

''Long Address Traces from RISC Machines: Generation and Analysis.'' **Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.** WRL Research Report 89/14, September 1989.

''Link-Time Code Modification.'' **David W. Wall.** WRL Research Report 89/17, September 1989.

''Noise Issues in the ECL Circuit Family.'' **Jeffrey Y.F. Tang and J. Leon Yang.** WRL Research Report 90/1, January 1990.

''Efficient Generation of Test Patterns Using Boolean Satisfiablilty.'' **Tracy Larrabee.** WRL Research Report 90/2, February 1990.

''Two Papers on Test Pattern Generation.'' **Tracy Larrabee.** WRL Research Report 90/3, March 1990.

''Virtual Memory vs. The File System.'' **Michael N. Nelson.** WRL Research Report 90/4, March 1990.

''Efficient Use of Workstations for Passive Monitoring of Local Area Networks.'' **Jeffrey C. Mogul.** WRL Research Report 90/5, July 1990.

''A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.'' **John S. Fitch.** WRL Research Report 90/6, July 1990.

''1990 DECWRL/Livermore Magic Release.'' **Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.** WRL Research Report 90/7, September 1990.

''Pool Boiling Enhancement Techniques for Water at Low Pressure.'' **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Research Report 90/9, December 1990.

''Writing Fast X Servers for Dumb Color Frame Buffers.'' **Joel McCormack.** WRL Research Report 91/1, February 1991.

''A Simulation Based Study of TLB Performance.'' **J. Bradley Chen, Anita Borg, Norman P. Jouppi.** WRL Research Report 91/2, November 1991.

''Analysis of Power Supply Networks in VLSI Circuits.'' **Don Stark.** WRL Research Report 91/3, April 1991.

''TurboChannel T1 Adapter.'' **David Boggs.** WRL Research Report 91/4, April 1991.

''Procedure Merging with Instruction Caches.'' **Scott McFarling.** WRL Research Report 91/5, March 1991.

''Don't Fidget with Widgets, Draw!.'' **Joel Bartlett.** WRL Research Report 91/6, May 1991.

''Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.'' **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Research Report 91/7, June 1991.

''Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.'' **G. May Yip.** WRL Research Report 91/8, June 1991.

''Interleaved Fin Thermal Connectors for Multichip Modules.'' **William R. Hamburgen.** WRL Research Report 91/9, August 1991.

''Experience with a Software-defined Machine Architecture.'' **David W. Wall.** WRL Research Report 91/10, August 1991.

''Network Locality at the Scale of Processes.'' **Jeffrey C. Mogul.** WRL Research Report 91/11, November 1991.

''Cache Write Policies and Performance.'' **Norman P. Jouppi.** WRL Research Report 91/12, December 1991.

''Packaging a 150 W Bipolar ECL Microprocessor.'' **William R. Hamburgen, John S. Fitch.** WRL Research Report 92/1, March 1992.

''Observing TCP Dynamics in Real Networks.'' **Jeffrey C. Mogul.** WRL Research Report 92/2, April 1992.

''Systems for Late Code Modification.'' **David W. Wall.** WRL Research Report 92/3, May 1992.

''Piecewise Linear Models for Switch-Level Simulation.'' **Russell Kao.** WRL Research Report 92/5, September 1992.

''A Practical System for Intermodule Code Optimization at Link-Time.'' **Amitabh Srivastava and David W. Wall.** WRL Research Report 92/6, December 1992.

''A Smart Frame Buffer.'' **Joel McCormack & Bob McNamara.** WRL Research Report 93/1, January 1993.

''Recovery in Spritely NFS.'' **Jeffrey C. Mogul.** WRL Research Report 93/2, June 1993.

''Tradeoffs in Two-Level On-Chip Caching.'' **Norman P. Jouppi & Steven J.E. Wilton.** WRL Research Report 93/3, October 1993.

''Unreachable Procedures in Object-oriented Programing.'' **Amitabh Srivastava.** WRL Research Report 93/4, August 1993.

''An Enhanced Access and Cycle Time Model for On-Chip Caches.'' **Steven J.E. Wilton and Norman P. Jouppi.** WRL Research Report 93/5, July 1994.

''Limits of Instruction-Level Parallelism.'' **David W. Wall.** WRL Research Report 93/6, November 1993.

''Fluoroelastomer Pressure Pad Design for Microelectronic Applications.'' **Alberto Makino, William R. Hamburgen, John S. Fitch.** WRL Research Report 93/7, November 1993.

''A 300MHz 115W 32b Bipolar ECL Microprocessor.'' **Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.** WRL Research Report 93/8, December 1993.

''Link-Time Optimization of Address Calculation on a 64-bit Architecture.'' **Amitabh Srivastava, David W. Wall.** WRL Research Report 94/1, February 1994.

''ATOM: A System for Building Customized Program Analysis Tools.'' **Amitabh Srivastava, Alan Eustace.** WRL Research Report 94/2, March 1994.

''Complexity/Performance Tradeoffs with Non-Blocking Loads.'' **Keith I. Farkas, Norman P. Jouppi.** WRL Research Report 94/3, March 1994.

''A Better Update Policy.'' **Jeffrey C. Mogul.** WRL Research Report 94/4, April 1994.

''Boolean Matching for Full-Custom ECL Gates.'' **Robert N. Mayo, Herve Touati.** WRL Research Report 94/5, April 1994.

''Software Methods for System Address Tracing: Implementation and Validation.'' **J. Bradley Chen, David W. Wall, and Anita Borg.** WRL Research Report 94/6, September 1994.

''Performance Implications of Multiple Pointer Sizes.'' **Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.** WRL Research Report 94/7, December 1994.

''How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.'' **Keith I. Farkas, Norman P. Jouppi, and Paul Chow.** WRL Research Report 94/8, December 1994.

''Drip: A Schematic Drawing Interpreter.'' **Ramsey W. Haddad.** WRL Research Report 95/1, March 1995.

''Recursive Layout Generation.'' **Louis M. Monier, Jeremy Dion.** WRL Research Report 95/2, March 1995.

''Contour: A Tile-based Gridless Router.'' **Jeremy Dion, Louis M. Monier.** WRL Research Report 95/3, March 1995.

''The Case for Persistent-Connection HTTP.'' **Jeffrey C. Mogul.** WRL Research Report 95/4, May 1995.

''Network Behavior of a Busy Web Server and its Clients.'' **Jeffrey C. Mogul.** WRL Research Report 95/5, October 1995.

''The Predictability of Branches in Libraries.'' **Brad Calder, Dirk Grunwald, and Amitabh** Srivastava. WRL Research Report 95/6, October 1995.

''Shared Memory Consistency Models: A Tutorial.'' **Sarita V. Adve, Kourosh Gharachorloo.** WRL Research Report 95/7, September 1995.

''Eliminating Receive Livelock in an Interrupt-driven Kernel.'' **Jeffrey C. Mogul and K. K. Ramakrishnan.** WRL Research Report 95/8, December 1995.

''Memory Consistency Models for Shared-Memory Multiprocessors.'' **Kourosh Gharachorloo.** WRL Research Report 95/9, December 1995.

''Register File Design Considerations in Dynamically Scheduled Processors.'' **Keith I. Farkas, Norman P. Jouppi, Paul Chow.** WRL Research Report 95/10, November 1995.

''Optimization in Permutation Spaces.'' **Silvio** Turrini. WRL Research Report 96/1, November 1996.

''Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory.'' **Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath.** WRL Research Report 96/2, November 1996.

''Efficient Procedure Mapping using Cache Line Coloring.'' **Amir H. Hashemi, David R. Kaeli, and Brad Calder.** WRL Research Report 96/3, October 1996.

''Optimizations and Placement with the Genetic Workbench.'' **Silvio Turrini.** WRL Research Report 96/4, November 1996.

''Memory-system Design Considerations for Dynamically-scheduled Processors.'' **Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic.** WRL Research Report 97/1, February 1997.

''Performance of the Shasta Distributed Shared Memory Protocol.'' **Daniel J. Scales and Kourosh Gharachorloo.** WRL Research Report 97/2, February 1997.

''Fine-Grain Software Distributed Shared Memory on SMP Clusters.'' **Daniel J. Scales, Kourosh Gharachorloo, and Anshu Aggarwal.** WRL Research Report 97/3, February 1997.

''Potential benefits of delta encoding and data compression for HTTP.'' **Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy.** WRL Research Report 97/4, July 1997.

WRL Technical Notes

''TCP/IP PrintServer: Print Server Protocol.'' **Brian K. Reid and Christopher A. Kent.** WRL Technical Note TN-4, September 1988.

''TCP/IP PrintServer: Server Architecture and Implementation.'' **Christopher A. Kent.** WRL Technical Note TN-7, November 1988.

''Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.'' **Joel McCormack.** WRL Technical Note TN-9, September 1989.

''Why Aren't Operating Systems Getting Faster As Fast As Hardware?.'' **John Ousterhout.** WRL Technical Note TN-11, October 1989.

''Mostly-Copying Garbage Collection Picks Up Generations and C++.'' **Joel F. Bartlett.** WRL Technical Note TN-12, October 1989.

''Characterization of Organic Illumination Systems.'' **Bill Hamburgen, Jeff Mogul, Brian Reid, Alan Eustace, Richard Swan, Mary Jo Doherty, and Joel Bartlett.** WRL Technical Note TN-13, April 1989.

''Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.'' **Norman P. Jouppi.** WRL Technical Note TN-14, March 1990.

''Limits of Instruction-Level Parallelism.'' **David W. Wall.** WRL Technical Note TN-15, December 1990.

''The Effect of Context Switches on Cache Performance.'' **Jeffrey C. Mogul and Anita Borg.** WRL Technical Note TN-16, December 1990.

''MTOOL: A Method For Detecting Memory Bottlenecks.'' **Aaron Goldberg and John** Hennessy. WRL Technical Note TN-17, December 1990.

''Predicting Program Behavior Using Real or Estimated Profiles.'' **David W. Wall.** WRL Technical Note TN-18, December 1990.

''Cache Replacement with Dynamic Exclusion.'' **Scott McFarling.** WRL Technical Note TN-22, November 1991.

''Boiling Binary Mixtures at Subatmospheric Pressures.'' **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Technical Note TN-23, January 1992.

''A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach.'' **John S. Fitch.** WRL Technical Note TN-24, January 1992.

''TurboChannel Versatec Adapter.'' **David Boggs.** WRL Technical Note TN-26, January 1992.

''A Recovery Protocol For Spritely NFS.'' **Jeffrey C. Mogul.** WRL Technical Note TN-27, April 1992.

''Electrical Evaluation Of The BIPS-0 Package.'' **Patrick D. Boyle.** WRL Technical Note TN-29, July 1992.

''Transparent Controls for Interactive Graphics.'' **Joel F. Bartlett.** WRL Technical Note TN-30, July 1992.

''Design Tools for BIPS-0.'' **Jeremy Dion & Louis Monier.** WRL Technical Note TN-32, December 1992.

''Link-Time Optimization of Address Calculation on a 64-Bit Architecture.'' **Amitabh Srivastava and David W. Wall.** WRL Technical Note TN-35, June 1993.

''Combining Branch Predictors.'' **Scott McFarling.** WRL Technical Note TN-36, June 1993.

''Boolean Matching for Full-Custom ECL Gates.'' **Robert N. Mayo and Herve Touati.** WRL Technical Note TN-37, June 1993.

''Piecewise Linear Models for Rsim.'' **Russell Kao, Mark Horowitz.** WRL Technical Note TN-40, December 1993.

''Speculative Execution and Instruction-Level Parallelism.'' **David W. Wall.** WRL Technical Note TN-42, March 1994.

‘‘Ramonamap - An Example of Graphical Groupware.’’ **Joel F. Bartlett.** WRL Technical Note TN-43, December 1994.

‘‘ATOM: A Flexible Interface for Building High Performance Program Analysis Tools.’’ **Alan Eustace and Amitabh Srivastava.** WRL Technical Note TN-44, July 1994.

‘‘Circuit and Process Directions for Low-Voltage Swing Submicron BiCMOS.’’ **Norman P. Jouppi, Suresh Menon, and Stefanos Sidiropoulos.** WRL Technical Note TN-45, March 1994.

‘‘Experience with a Wireless World Wide Web Client.’’ **Joel F. Bartlett.** WRL Technical Note TN-46, March 1995.

‘‘I/O Component Characterization for I/O Cache Designs.’’ **Kathy J. Richardson.** WRL Technical Note TN-47, April 1995.

‘‘Attribute caches.’’ **Kathy J. Richardson, Michael J. Flynn.** WRL Technical Note TN-48, April 1995.

‘‘Operating Systems Support for Busy Internet Servers.’’ **Jeffrey C. Mogul.** WRL Technical Note TN-49, May 1995.

‘‘The Predictability of Libraries.’’ **Brad Calder, Dirk Grunwald, Amitabh Srivastava.** WRL Technical Note TN-50, July 1995.

‘‘Simultaneous Multithreading: A Platform for Next-generation Processors.’’ **Susan J. Eggers, Joel Emer, Henry M. Levy, Jack L. Lo, Rebecca Stamm and Dean M. Tullsen.** WRL Technical Note TN-52, March 1997.

WRL Research Reports and Technical Notes are available on the World Wide Web, from `http://www.research.digital.com/wrl/techreports/index.html`.