Design, Implementation, and Evaluation of a Client Characterization Driven Web Server

Balachander Krishnamurthy
AT&T Labs–Research
Florham Park, NJ, USA
bala@research.att.com

Craig E. Wills
Worcester Polytechnic Institute
Worcester, MA USA
cew@cs.wpi.edu

Yin Zhang
AT&T Labs–Research
Florham Park, NJ, USA
yzhang@research.att.com

Kashi Vishwanath
Duke University
Durham, NC USA
kvv@cs.duke.edu

ABSTRACT

In earlier work we proposed a way for a Web server to detect connectivity information about clients accessing it in order to take tailored actions for a client request. This paper describes the design, implementation, and evaluation of such a working system. A Web site has a strong incentive to reduce the 'time-to-glass' to retain users who may otherwise lose interest and leave the site. We have performed a measurement study from multiple client sites around the world with various levels of connectivity to the Internet communicating with modified Apache Web servers under our control. The results show that clients can be classified in a correct and stable manner and that user-perceived latency can be reduced via tailored actions. Our measurements show that classification and determination of server actions are done without significant overhead on the Web server. We explore a variety of modified actions ranging from selecting a lower quality version of the resource to altering the manner of content delivery. By studying numerous performance related factors in a single unified framework and examining both individual actions as well as combination of actions, our modified Web server implementation shows the efficacy of various server actions.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; C.5 [**Computer Systems Organization**]: Computer System Implementation—*Servers*; D.0 [**Software**]: General

General Terms

Design, experimentation, measurement, performance.

Keywords

Web performance, Apache server, httperf, content delivery, client classification, server adaptation.

1. INTRODUCTION

User-perceived latency has a strong bearing on how long users stay at a Web site and the frequency with which they return to the site. A Web site that is trying to retain users has a strong incentive to reduce the "time to glass" (the delay between a user's request and the subsequent delivery and display) for a Web page. For Web sites that have a critical need to retain users beyond the first page there is a strong motivation to deliver the content quickly to the user. Other work has proposed enhanced admission control and scheduling policies at Web servers to give priority to some requests [5, 6]. However, the performance perceived by a Web client to a server may be due to low bandwidth, high latency, network congestion or the delay at intermediaries between the client and server, which are not under the control of the server. In spite of these potential impediments, the server has a strong incentive to deliver the most suitable content as quickly as possible to the user.

This work takes a new approach for how a server should identify and react to clients receiving poor performance. It builds on earlier work where we presented a way to characterize Web client's connectivity information and proposed mitigating actions a server could take such as selecting a lower quality version of a resource for delivery or by altering the manner of content delivery [11]. In this follow up work we present the design and implementation of a working prototype, using Apache [2], that demonstrates the feasibility of a Web server classifying clients and potentially taking alternate actions in serving content to clients.

This work also builds on earlier research work that has examined Web performance from the viewpoint of individual improvements in reducing user-perceived latency or load on the network. The set of ideas includes compression and delta encoding [14], stability in network performance [4], examining impact of various protocol variations of HTTP [16, 10] and bundling resources [22]. What these works have in common is the use of a single idea to explore impact on Web performance. Each of these pieces of research differ in their evaluation environment in the sense that they use different methodologies, workloads, and validation techniques.

This work is distinguished because we use our prototype to not only evaluate online characterization of live requests, but also to evaluate the potential performance improvements for various actions in a unified framework. We also examine meaningful combinations of actions. We use a canonical set of container documents with various distributions of embedded objects in terms of number and size. This approach allows the results of our work to be applied by a wide variety of sites to test the potential performance improvement for clients that visit them.

We explore the impact of various potential performance improvements by an active measurement testbed consisting of clients with different connectivity sending requests to a small number of

WWW2003, May 20–24, 2003, Budapest, Hungary.
ACM 1-58113-680-3/03/0005.

prototype Web servers under our control. Having control over the Web server and content allows us to examine the different performance components in an automated fashion. By downloading the canonical container document set from clients with different connectivity capabilities, we can measure the actual improvement as a result of various server actions. We measure the latency reduction to the clients as a result of the server's different actions tailored to the class of the client as well as the overhead imposed on the server for carrying out the classification and altered action execution.

We examine three classes of server actions in this work for improving the response time for clients:

1. *Altering the Content*. Given a range of content variants, a server could choose a reduced version for poorer clients, by including fewer, if any, embedded objects or by including "thinner" variants of embedded images.

2. *Altering Manner of Delivery of Content*. The content can be compressed with a suitable compression algorithm. The server can also bundle the embedded objects into a single resource, which can be retrieved by clients to avoid multiple rounds of requests to fetch the objects. This technique can be combined with compression to reduce the size of the bundle.

3. *Altering Server Policies*. A server could choose to maintain longer persistent connections for poor clients.

The rest of this paper is organized as follows: Section 2 describes the system architecture and implementation of our prototype system for live characterization and alternate server action. The section describes the implementation of the classifier and the instrumentation done to the Apache server. Section 3 describes testing we did to measure the overhead of our changes to the server and the results we obtained. Section 4 describes the methodology that we used to evaluate our implementation and the live experiments that were carried out. Section 5 presents the result of our study. Section 6 discusses related work. We conclude with Section 7 by presenting a summary of our results and ongoing work.

2. SYSTEM ARCHITECTURE AND IMPLEMENTATION

The goal of this work is to build a working prototype system that can both do live characterization of client connectivity based on their requests as well as take an appropriate action for clients classified as poor. Our prototype has two pieces: a classifier that uses client requests to classify clients into different connectivity classes; and an active Web server component that takes an available action for a request from a poor or rich client. This section describes the overall design we used for the prototype system as well as the implementation details of the classifier and instrumented Apache [2] Web server. The architecture of our prototype is shown in Figure 1.

We use a separate daemon process for the classifier. We originally considered including the classifier as a routine within the Apache Web server, but rejected this approach to minimize our modifications to the Apache server and to design a solution that works with the multiple pre-forked process architecture of Apache.

As shown in Figure 1, input to the classifier process is a standard Apache log file, which is written to by each Apache server process and read by the classifier. Reading from the log file introduces some potential delay for the classifier to start classifying a client, but requires no modification of the server.

The output of the classifier process is a mapping between client IP address and the classification (poor, normal or rich) for that

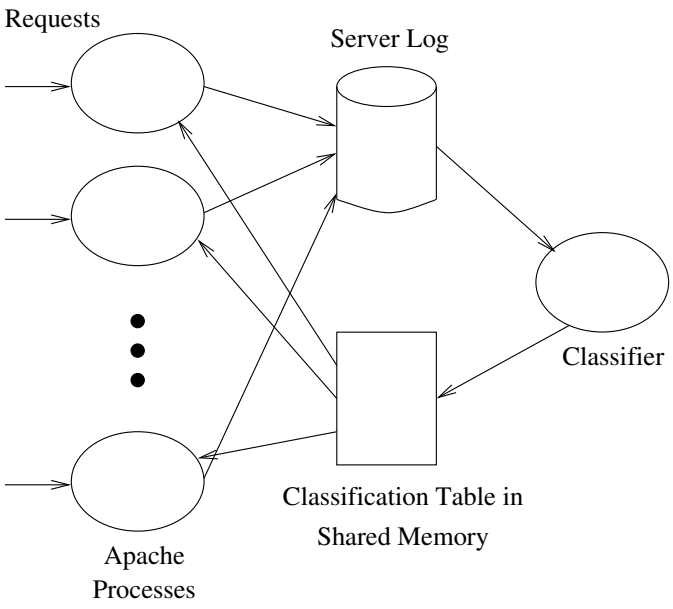

Figure 1: System Architecture

client. This classification is stored by the classifier in shared memory. This shared memory is available for reading by each of the Apache server processes. The classifier also determines the classification for network-aware clusters [9] of IP addresses, used to provide a coarser level of client categorization. Such a cluster categorization can then be used to categorize subsequent clients from that cluster for which a client-specific categorization is not available. Clustering is implemented using a fast longest prefix matching library. The cluster classifications are also stored in shared memory for use when per-client classification is not available.

Our architecture, with a separate classifier process, yields a clean implementation requiring only minor modifications to the Apache code. It also allows for the priority of the classifier to be reduced during heavy system load so that more CPU time can be devoted to the Web server processes, while still being able to access existing classification information in shared memory.

Another issue related to the system architecture is to define the set of pages that are used to classify a client, and the set of pages on which actions can be taken to alter content, manner of delivery or server policies. These sets may overlap. In the simplest case, all pages at a site constitute each set. However, it makes little sense to expend resources on infrequently accessed pages. Thus both sets of pages should be restricted to the more frequently requested pages at a site. Beyond popularity, pages in the first set should distinguish connectivity differences between clients. For example, pages with little content and few or no embedded objects may not be appropriate for use in classification. Pages in the second set should include those for which at least one action can significantly improve the response time for the page. Details of how these sets are specified and the available actions are given in Section 4.

2.1 Classifier Implementation

The classifier is written in C and works in a similar manner to the client characterization algorithm described in [11]. Prior to the classifier process running, an offline process is run to determine the most popular container URIs that account for the 70% of accesses to the server site. Standard server logs are used to gather this information. Only requests with 200 and 304 HTTP response codes

are considered. Container URIs with HTML content are identified using the URI string, although other site-specific knowledge could be incorporated. The set of popular container URIs constitute the set used for classification. This set can be updated as the dynamics of site access dictate.

When the classifier process starts up, it begins to read the log file for URI requests. It continues to read GET requests while they are available. If the classifier reaches the end of the log file while reading then it times out for a short period (currently a configurable parameter of 5 seconds) before it checks the log file again for fresh requests.

When a container URI is identified, the classifier checks if this URI is in the set for classification. If not then it is skipped for classification. If the URI is in the classification set then the classifier records the retrieval time for this base object for the given client. As subsequent requests from the same client are read for embedded objects (images, style sheets, javascript) of the base object, additional metrics are computed for client classification.

We used two metrics in classifying a client: the delay (in seconds) between the base object and the first embedded object and the delay between the base object and the last embedded object in a sequence. For each of these two metrics, we defined cumulative values E_{first} and E_{last} to represent long-term estimates for these two metrics for each client. To both minimize the amount of information stored for each client and to give greater weight to more recent history, we chose to use a exponentially weighted mean where the value for E_{first} (E_{last} is similarly defined) is given as:

$$E_{first} = \alpha E_{first} + (1 - \alpha) E_{measured}$$

where $E_{measured}$ is the current measurement for the delay between the base and first embedded object. In the prototype we used a configurable parameter of $\alpha = 0.7$.

After each page retrieval, a client was classified based on the values of E_{first} and E_{last}. In [11] we explored a variety of classification thresholds. In practice, these thresholds would be set based on content and client mix for a site. In the prototype we made the thresholds as configurable parameters, but for all tests we classified a client as poor

if $E_{first} > 3$ or $E_{last} > 5$ seconds

and a client as rich

if $E_{first} <= 1$ and $E_{last} <= 2$ seconds.

All clients not matching either criteria are classified as normal. The classifier begins classifying a client as soon as it sees the first request sequence from the client. As an aid to the Apache server in potentially taking actions for requests from previously unseen clients, the classifier process also classifies *clusters* of client IP addresses. The classifier does not classify pages on which an action has been taken.

2.2 Instrumented Apache Server

We used version 1.3.24 of the publicly available Apache Web server for our prototype. The changes to the Web server were concentrated in three files: `http_main.c`, `http_protocol.c` and `http_core.c`.

Changes made to Apache are in two categories. The first category are changes made at server start time—these steps need to be taken each time the server is restarted. The following steps are done in the function *REALMAIN()* in `http_main.c`.

1. Open the cluster library. Subsequent lookups of IP addresses for cluster membership are made via a simple library call.

2. Read in a config file with various server actions for tailorable content The entries look like:

```
/foo.html .gz .p_lc ; .r_mc
```

   The above entry states that a *gzip* compressed form of the resource `foo.html` as well as a version with less content is available for poor clients (`p_lc`). There is also an alternate version with more content (`r_mc`) available for rich clients. The total number of URIs for which alternate version exists is a strict subset of the popular URIs. No alternate content is served for unpopular URIs.

3. Initialize shared memory for reading in the classes corresponding to the IP addresses and the clusters. Shared memory is implemented using two libraries: *vmalloc* [19] and *cdt* [20]. *vmalloc* is a general purpose C library for allocating memory in regions with efficient algorithms for compact memory lay-out enabling applications to select such algorithms per region to tune memory usage. CDT provides a comprehensive set of container data types implemented on top of efficient data structures such as splay trees and adaptive hash tables. Experimental results [19] have shown that CDT containers outperform their counterparts in other similar packages including C++ STL containers. These libraries use the discipline and method library design enabling applications to compose them to provide high-performance in-memory containers as well as persistent ones.

The second category of modifications are for additional work that is needed for each request served by the server. The first modification is done in the function *read_request_line()* in `http_protocol.c`. The remaining four steps are in the *core_translate()* function of `http_core.c`.

1. Check if the requested URI is for a container document; this is done at time of reading the request line via the function *is_container()*. If so, then we record this in the data structure associated with this request (*request_rec*). If not, we ignore it since there is no altered content to be served.

2. If it is a container document, we locate the class for the client in the shared memory. If the class is available, we record it in the data structure associated with the request. Else, we identify the cluster of the client's IP address and if the cluster's class is available we record that in the request's data structure.

3. If the class is available (either via the IP or its cluster) and is not normal, then the server checks the possibility of modified content via the function *modify_uri()*.

4. The *modify_uri()* function looks up the various alternate versions available for a URI in the table initialized at server start time. A client expresses its willingness to accept particular actions through a `X-Server-Actions` HTTP header. Thus if the client includes the `X-Server-Actions:.gz` header, then the server checks if the `.gz` alternative is available to serve this URI. If found, the URI /foo.html is replaced in the data structure with the alternate URI (/foo.html.gz). The server serves the client this file and records the request in the log file as /foo.html.gz. This approach lets the classifier know that a tailored action was taken.

3. SERVER OVERHEAD

Our first step in evaluating the prototype system is to examine the amount of overhead introduced at the server for handling a

request. The mechanism should not introduce perceivable overhead for client requests, whether or not a server action is taken, nor should it reduce the request handling capacity of the server. For this evaluation we did *not* use the live clients described in Section 4, but used a controlled experiment as described below.

There were two categories of changes to the Apache server code: a set of changes that add to overhead at server start time and changes that result in overhead for each request handled. The overhead in the first category includes opening the cluster library, reading in the configuration file for the different URIs for which modified action are enabled, and initialization of the shared memory. The second category of overheads are on a per-request basis: these include a function to verify that the requested URI is a container document, looking up the class of the client in shared memory, looking up the cluster information in the shared memory after calculating the prefix, and altering the server action. Note that this is a per-request overhead and *not* per-connection. Finally, to measure the server overhead, we introduced the ability to handle a new header called `X-IP` to allow variations in IP addresses of clients in order to test the usefulness of clustering (this header was not used for subsequent live client tests).

We generated a test load with pages in different characteristics buckets drawn from the distribution in Table 2. Client IP addresses were chosen uniformly between 0.0.0.0 and 255.255.255.255 and specified via the `X-IP` header. The entire test resulted in over 140K GET requests. We ignored all but the first 50000 requests to ensure that the server has been warmed up. This is because we do not want to underestimate shared memory lookup costs when it is sparsely populated. Among these 50000 requests, many are not container objects and thus do not require classification. Using standard function call bracketing mechanisms we computed these overheads and subtracted the 5 usec overhead of the timing function calls themselves. The overhead is presented in Table 1. The average overhead is only 75 usec, which is well below user perception threshold. Note that there is considerable room for optimizing our code, especially in the functions *is_container()* and *modify_uri()*, which currently are responsible for around two thirds of the overhead. The standard deviation is only 18 usec, which is very small compared to the average overhead. Most of the variation is due to the function *is_container()*, which compares the URI sequentially with a list of suffixes (to determine if it is a container) and can thus return at different points of execution.

Table 1: Overhead on the Server

	Overhead (usec)		
Step	Mean	Med.	Stddev
Is URI a container document?	19.2	2	12.1
Class lookup in shared memory	12.5	9	5.8
Cluster related overhead			
Converting IP address	4.2	3	6.1
Looking up cluster	8.0	7	4.8
Cluster lookup in shared memory	2.5	2	4.9
Classification based on cluster	0.7	0	4.4
Server actions			
Modifying URI	25.5	25	6.1
Logging changed request	2.8	3	3.6
Total overhead	75.4	51	18.2

In addition to measuring the normal overhead, we also ran a stress test to ensure that the extra classification work and addition of shared memory etc., did not significantly reduce the ability of the server to handle a sudden increase in overload. The servers were installed on a 731 Mhz Pentium III (686-class CPU) PC running FreeBSD 4.4-STABLE with 2 GB memory with no other users and a uncongested link to the client machine.

We used *ab* (Apache Benchmark) [3], a benchmark program from the Apache distribution, to measure how many requests per second the server is capable of serving. In `httpd.conf` (configuration file of Apache), we set 3 parameters: MaxKeepAliveRequests to 0, i.e. infinity, KeepAliveTimeout to 300, i.e. 5 minutes, and MaxClients to 2048. We made 5 runs where each run consisted of sending 10,000 requests for a 716 byte file and logging the request handling time for the requests. Figure 2 shows the *average* overhead increase of the modified server over the regular Apache server across five separate runs (there was little difference between each run). The overhead shown is the relative increase in time to process the 10,000 requests in the modified versus regular Apache server. We increased the level of concurrency from 1 to 1000. As Figure 2 shows, the average increase in processing time for the modified server is around 11%, but showing steady signs of diminishing as the number of concurrent connections increases over 600. This result can be explained by the increased overhead of the regular Apache server when it uses a large number of Apache processes to handle the concurrent connections versus the relatively fixed cost per request due to server actions.

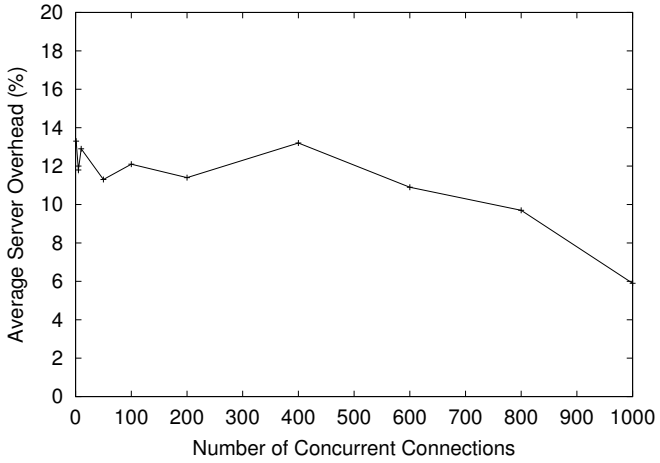

Figure 2: Stress Test Overhead

4. METHODOLOGY

After finding that the prototype system introduces little overhead for request handling, we went on to use the prototype to evaluate the two primary aspects of our approach.

The first aspect is the accuracy of the client classification mechanism in correctly identifying the classification state for a variety of clients. Unlike [11], the prototype allows us to evaluate a running system receiving live requests. Related to correctness of classification is the stability of the classification. We expect that the classification for a client should be relatively stable and not fluctuate wildly across the set of classification states during an interval of time.

The second aspect is the effectiveness of various server actions for reducing the response time. We would like to investigate the effect of different server actions for a variety of content and types of clients. This work extends that in [12] to combine client classification with server actions.

To evaluate these two aspects we created a testing environment for live testing of a variety of clients retrieving content from prototype servers deployed at different sites. Each prototype server contains the same content. The remainder of this section describes the content, clients and servers as well as the methodology used for testing.

4.1 Site Content

In order to run controlled experiments we want to create a site with content of known characteristics that can be used to investigate the correctness of client classification and the effectiveness of various server actions for reducing download time for clients classified as poor.

We focus on characterizing pages based on the amount of content of a page. We examine the number of bytes in the container object, the number of embedded objects and the total number of bytes for the embedded objects. We used recent proxy logs from a large manufacturing company with over 100,000 users, examined requests to the container object of a page by looking for HTML URLs, and selected the 1000 most popular pages. In April 2002 we downloaded each container object and embedded objects (frames, layers, cascading style sheets, javascript code and images) to determine the size of these objects. Objects referenced as a result of executing embedded javascript code were not considered. 641 URLs containing one or more embedded objects were successfully retrieved and using 33% and 67% percentile values we created a small, medium and large value range for each characteristic. The rationale for this approach is to examine the impact of server actions across the "space" of different content sizes.

Using these three ranges for each of the three characteristics defines a total of 27 "buckets" for the classification of an individual page. The cut off for container bytes in small, medium and large were less than 12K, less than 30K bytes, and more than 30K bytes respectively. Similarly, for embedded objects it was less than 7, 22, and more than 22 and for embedded bytes 20K, 55K, and more than 55K bytes.

Using these ranges we determined the percentage of pages that fell in each bucket. These are shown in Table 2. The table shows that 20% of these pages have a small number of container bytes, a small number of embedded objects and a small number of embedded bytes. 7% of the pages fall in the medium range for each characteristic and 14% fall in the large range for each characteristic.

Table 2: Percentage of Pages in Each Characteristic Bucket Based on Popular Pages from Proxy Log

	Embedded Bytes								
	Small			Medium			Large		
Embedded	Cont. Bytes			Cont. Bytes			Cont. Bytes		
Objects	S	M	L	S	M	L	S	M	L
Small	20	6	2	4	1	0	0	0	0
Medium	2	3	1	5	7	8	2	5	3
Large	0	0	0	1	2	4	1	8	14

We defined the ranges primarily to identify pages that spanned the space of all possible characteristics. We selected two representative pages from each bucket of the proxy log pages. In buckets containing many pages we tried to select two pages that were representative of characteristics within the bucket. In all, we selected 44 pages (not all buckets contained two pages) to cover the space of characteristics and downloaded them to a test site using *wget* [21].

Additional objects were created in preparation for testing the various server actions: compressed version of each container object using *gzip*, single bundled object with the embedded objects for each page, and a separate compressed bundle object.

4.2 Clients and Servers

We installed the prototype Apache server along with the test site content on relatively unloaded machines at three sites: a machine running Linux at att.com in New Jersey, U.S., a machine running Linux at wpi.edu in Massachusetts, U.S and a machine running FreeBSD at icir.org in California, U.S.

Just as we deliberately chose Web content for our test Web site to include a variety of characteristics, we located clients with different connectivity characteristics to the test sites. We tested from clients in five locations:

1. att: AT&T Labs–Research, New Jersey, USA,
2. de: Saarbruecken University in Germany,
3. cable: cable modem user in New Jersey,
4. modem: 56Kbps dialup modem user in New Jersey, and
5. uk: London, U.K. via a dedicated 56Kbps line.

Tests between clients and servers were run at different times of the day. We used measured round-trip time (RTT) and throughput to characterize the network connectivity characteristics between these clients and our test sites. The round trip time was determined using the average TCP connection setup time for the first object retrieval of each test. We computed the average throughput for retrieving each of the bundle objects in the test site. We used these objects because they contained a larger number of bytes. The round-trip times and throughput for each of our client/server pairs are shown in Figure 3. The standard deviation for the results shown is at most 20-30% of the mean, save for the RTT of the modem and uk clients. The RTT standard deviation for these clients is generally 60-110% of the average RTT to a server.

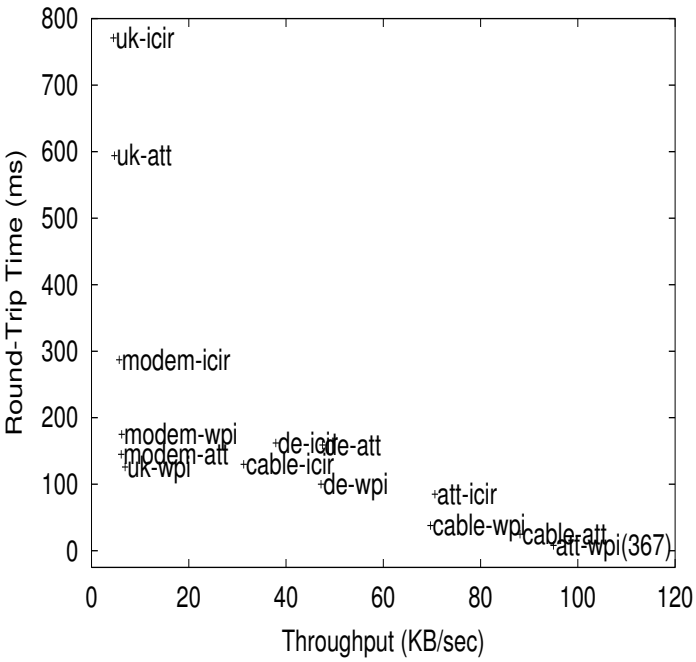

Figure 3: Client/Server Connectivity

4.3 Experiment

For client testing, we used *httperf* [15], to make automated retrievals to the prototype servers for testing of the various server actions. While retrieval using a real browser might be more realistic in measuring "time-to-glass," the use of *httperf* allows us to automate and control the retrieval under various conditions.

For testing, we wanted to generate requests with a similar mix of content as present for frequently accessed content in the original proxy logs of the large manufacturing company. We therefore used the relative distribution of pages in the buckets shown in Table 2 to guide which pages were requested from the prototype servers. We randomly generated a stream of 200 page requests with the given weightings for each type of page. This stream of requests was used by all client tests to each server.

Each client test was designed to both examine the classification state assigned by the prototype to a client and to measure the relative effectiveness of different server actions. Each page in the request stream was first sent to the server with an empty `X-Server-Actions` header indicating that the server should not take action on this page, but it should use this page for classification. As part of the response in serving the container object for the page, the server returns a header `X-Class` indicating whether the client is classified as poor, normal or rich. We use these results to determine the correctness and stability of the classification for the different client-server pairs.

The initial retrieval is done with *httperf* using up to four parallel HTTP/1.0 requests. We use this "para-1.0" measure as a baseline measure. We also study the relative effect of various server actions by making subsequent client requests and forcing particular actions to be taken based on the `X-Server-Actions` header.

We investigated two classes of server actions in our experiments. We examined four actions that change the manner in which content is delivered, but not the content itself:

1. compress—retrieve container object in compressed form and embedded objects (uncompressed) using up to four parallel HTTP/1.0 connections

2. serial-1.1—serialized requests are made for embedded objects using up to two persistent HTTP/1.1 connections

3. pipe-1.1—pipelined requests made for embedded objects using a single persistent HTTP/1.1 connection

4. bundle—retrieve a single bundle of embedded objects.

We also investigated three actions that reduced the amount of content served to the user:

1. baseonly—retrieve the container object but no embedded objects. This is intended to measure the potential response time savings for removing all embedded content.

2. halfobject—retrieve (top) half of the embedded objects using up to four parallel HTTP/1.0 connections; intended to examine the effect of removing some embedded objects from the container object.

3. halfres—retrieve a half resolution version of each embedded object using up to four parallel HTTP/1.0 connections; intended to examine the effect of creating thinner versions of each embedded object. The half resolution objects are generated using the `convert` program [8] with a sampling ratio of 50%.

Note that for the serial-1.1 and pipe-1.1 actions, our Apache prototype server currently does not change its policy based on client classification for when and how long a persistent connection is maintained. However, the prototype server does support persistent connections for all clients and we are thus able to test the effectiveness of these actions when used by the client. In practice, if a deployed server wanted to encourage (it cannot force) a client to use persistence then it would keep the network connection for the client open while indicating as part of the response to the client that the client should use persistent connections.

Many of the actions can be used in conjunction with each other. For example, we examined the impact of compressing the bundled object or compressing the container object and retrieving only it. We show the results for combinations of actions as appropriate.

Actions do add to the costs of the server. The server has to create and store thinner variants of some objects; it must generate or precompute and store compressed or bundled content. These costs however can be amortized across multiple requests.

5. RESULTS

We now present the results of our study. We begin by discussing the stability and correctness of our client classification mechanism and then discuss the effectiveness of various server actions.

5.1 Stability and Correctness of Client Classification

We used the methodology described in Section 4 to run tests from a number of clients to our test servers in September, 2002. Table 3 summarizes the client classification results from these tests. The correctness of this classification is compared with expectations based on the client/server connectivity results shown in Figure 3. The results in Figure 3 yield three client/server groups: 1) the relatively poor modem and uk clients, which have a low throughput, but variable RTT, to all servers; 2) the cable-icir pair and all pairs with the de client, which have a moderate throughput; and 3) the remaining pairs, which have a relatively high throughput.

In comparing these groups with Table 3, we see that no client expected to be poor is ever classified as rich, which is essential because it is highly undesirable to send richer content to a poor client, further degrading the already poor download performance. Similarly, no client with high throughput to a server is ever classified as poor. Overall, the classifier is able to give the classification consistent with expectations for most client/server pairs over 90% of the time. The only exceptions are the de-wpi and de-att pairs from the moderate throughput group. They have similar throughput but rather different client classification results. Further tests and inspection suggests that the server delay between serving the container and last embedded object for both pairs is close to the five-second threshold we use to classify poor clients. The boundary effects coupled with the different RTTs and their variation cause the classifier to show variation in the classification. In our experiments, we also classify based on requests for all pages in the test set. In practice, the administrator of a site would fine tune the threshold and set of pages used for classification to reflect the site content and client mix.

5.2 Server Actions for Poor Clients

In evaluating server actions, we only examined actions that would potentially improve performance for clients and did not test any actions where enhanced content is served to clients classified as rich. The following discussion focuses on the results for the clients classified as poor (uk and modem) in Table 3.

Table 3: Distribution of Client Classification Results

Pair	Poor	Normal	Rich
att-icir	-	11%	89%
att-wpi	-	-	100%
cable-att	-	11%	89%
cable-icir	10%	88%	2%
cable-wpi	-	12%	88%
de-att	73%	27%	-
de-icir	7%	92%	1%
de-wpi	24%	74%	2%
modem-att	99%	1%	-
modem-icir	99%	1%	-
modem-wpi	97%	3%	-
uk-att	100%	-	-
uk-icir	100%	-	-
uk-wpi	97%	3%	-

As shown in Figure 3, the connectivity between the two poor clients and the three servers is relatively consistent in terms of throughput, but shows variation in the round-trip time. In the following, we examine the results for these clients.

Table 4 shows the results for the uk-wpi client/server pair, which has the lowest latency among all low throughput client/server pairs. The table shows the results of actions for 8 of the 27 buckets of content mix shown in Table 2. These buckets represent 71% of the actual pages for the respective data sets. Space limitations prevent showing results for all 27 buckets. The top row in the table shows the average time in seconds to retrieve a page with the given mix of content. The remaining actions are divided into lossless, which do not change the content served to the client, and lossy, which change the content served.

The results show that the time to retrieve a page with large container page, a large number of embedded objects and a large number of embedded bytes is 21.08 seconds. Subsequent lines in the table show the relative percentage improvement if the various actions are taken. For emphasis on significant differences, server actions that yield greater than 20% improvement are highlighted. For example, using a compressed version of the container document for this bucket saves 22% of the 21.08 seconds. Actions yielding performance degradation are shown with a 0. None of the server actions we considered should degrade performance, but the *measured* relative percentage improvement for actions yielding only a small performance benefit can be negative due to variation in the network conditions. The cases of negative improvement were generally less than 10% or 0.25 seconds.

Overall the results show that compression of the container object has a significant effect for buckets with larger container objects, but the lossless actions of pipelining and bundling do not show at least 20% improvement. The combination of first bundling then compressing the embedded objects also leads to a more significant improvement in download time, especially for pages with a large number of embedded objects. The improvement of compressed bundling over bundling alone suggests that the overhead of HTTP response headers has some significance for clients with low throughput and relatively low latency. The action of serving only the container object yields a significant cost savings for all content mixes. Serving only half of the embedded objects also has a significant effect, but trying to reduce the size of embedded objects while still serving the same number of objects has little positive effect, largely because many of the objects are already small.

Table 5 shows the results for the uk-icir pair, which has low throughput and high latency. The significance of compressing the container page is reduced as the latency grows, while pipelining and bundling yield more significant improvement than compression. The lossy actions of reducing or eliminating the number of embedded objects yields significant performance gains, but again reducing the size of embedded objects does not.

The results for modem client to the AT&T server are shown in Table 6 and are representative of results from this client to the other servers. Despite similar RTT and throughput values as the uk-wpi pair in Table 4, the modem-att pair shows differences in the effects of some actions. Compression is much less effective for this client, which is not surprising because this test was performed on a machine running Win2K, which enables software compression by default. Hence, it is not as beneficial for the server to compress the container object. The results do show similar benefits for pipelining and bundling as in Table 4. In contrast to Table 4, the use of reduced quality images does improve the response time for this client. This is again due to the use of software compression, which effectively increases the fraction of bytes in the embedded images by reducing the the size of the container document and the request/response headers. As a result, reducing the quality of embedded images yields more significant improvement than without compression.

5.3 Server Actions for Normal and Rich Clients

While the server would not take a mitigating action for clients classified as normal or rich in Table 3, we did collect results on server action effectiveness as part of our experiments. We show the results for a medium and high throughput client/server pair because if a server supports lossless actions for poor clients, it could use the actions for other clients as well.

Table 7 shows results for the de-att client/server pair, which has medium throughput in the results shown in Figure 3. This client also has a number of accesses where it is classified as poor. The relative improvement for compression is not significant, but the impact of pipelining and bundling have a significant effect. These results indicate that for better connected clients, the amount of content is less important than the reducing the impact of of requests. The tone of the results is similar in Table 8 for the cable-wpi client/server pair, which also has a high throughput, but a lower latency. The other high throughput client/server pairs shown in Figure 3 yield similar results.

5.4 Server Action Discussion

The results show that the lossy action of removing embedded objects is the only action that has a significant effect in all cases. Simply reducing the quality of embedded objects without reducing the number does not yield a significant improvement under virtually all circumstances except for the modem client. This result emphasizes the need for client classification as a server site might want to improve response time for its poor clients, but does not want to unnecessarily reduce the quality for its other clients.

The lossless actions, which could be potentially applied to any type of client, are less consistent in their usefulness to reduce response time. Compression is an attractive action to take because most clients are already capable of handling it and prior work [14] has shown that decompression costs are insignificant. However, we did not find it had a significant effect on reducing response time for well-connected clients. In addition, some poor clients, such as our modem client, may already have compression enabled, which reduces the effect by the server. The effectiveness is also reduced for poor clients as the latency between the client and server increases.

Table 4: uk-wpi—Para-1.0 Retrieval Time and Percentage Improvements of Actions

	Container Bytes-Embedded Objects-Embedded Bytes							
Action	S-S-S	S-M-M	M-M-M	M-M-L	M-L-L	L-M-M	L-L-M	L-L-L
para-1.0	2.05s	5.95s	8.63s	12.70s	17.99s	11.72s	12.64s	21.08s
pipe-1.1	10	19	13	6	7	10	15	6
compress	15	5	**28**	**20**	13	**41**	**40**	**22**
bundle	0	9	7	7	18	5	15	15
bundle.gz	0	**26**	19	**20**	**28**	14	**22**	**24**
baseonly	**68**	**89**	**74**	**82**	**87**	**65**	**67**	**80**
halfobject	**48**	**40**	**23**	**33**	**27**	**20**	**24**	**25**
halfres	6	8	8	0	15	6	5	13

Table 5: uk-icir—Para-1.0 Retrieval Time and Percentage Improvements of Actions

	Container Bytes-Embedded Objects-Embedded Bytes							
Action	S-S-S	S-M-M	M-M-M	M-M-L	M-L-L	L-M-M	L-L-M	L-L-L
para-1.0	6.75s	13.64s	16.61s	20.61s	32.41s	20.28s	26.41s	36.08s
pipe-1.1	**28**	**40**	**30**	18	**31**	**24**	**40**	**27**
compress	17	5	17	15	6	**26**	**22**	13
bundle	0	**23**	13	8	**31**	11	**32**	**28**
bundle.gz	6	**32**	**20**	17	**38**	17	**37**	**35**
baseonly	**71**	**85**	**72**	**77**	**85**	**65**	**73**	**80**
halfobject	**37**	**32**	**23**	**36**	**33**	18	**31**	**29**
halfres	**22**	0	0	0	4	0	0	3

Table 6: modem-att—Para-1.0 Retrieval Time and Percentage Improvements of Actions

	Container Bytes-Embedded Objects-Embedded Bytes							
Action	S-S-S	S-M-M	M-M-M	M-M-L	M-L-L	L-M-M	L-L-M	L-L-L
para-1.0	2.63s	7.84s	9.25s	13.88s	20.40s	10.80s	12.43s	21.95s
pipe-1.1	9	0	0	3	10	0	9	9
compress	6	7	14	6	1	15	8	2
bundle	0	19	16	7	17	13	**26**	16
bundle.gz	0	**23**	**20**	13	**20**	16	**29**	19
baseonly	**70**	**90**	**80**	**86**	**91**	**70**	**74**	**85**
halfobject	**52**	**41**	**36**	**50**	**34**	**29**	**36**	**30**
halfres	**30**	**27**	**22**	**29**	**30**	18	16	**28**

Table 7: de-att—Para-1.0 Retrieval Time and Percentage Improvements of Actions

	Container Bytes-Embedded Objects-Embedded Bytes							
Action	S-S-S	S-M-M	M-M-M	M-M-L	M-L-L	L-M-M	L-L-M	L-L-L
para-1.0	2.47s	8.18s	8.60s	7.49s	11.09s	9.44s	10.19s	11.93s
pipe-1.1	**48**	**66**	**56**	**35**	**39**	**59**	**46**	**42**
compress	0	0	0	0	0	10	0	8
bundle	**28**	**75**	**74**	**70**	**76**	**70**	**77**	**73**
bundle.gz	**32**	**71**	**70**	**70**	**79**	**67**	**80**	**75**
baseonly	**82**	**94**	**90**	**88**	**92**	**89**	**90**	**91**
halfobject	**48**	**32**	**33**	**48**	**30**	**35**	13	**32**
halfres	0	13	11	0	6	19	0	12

Table 8: cable-wpi—Para-1.0 Retrieval Time and Percentage Improvements of Actions

	Container Bytes-Embedded Objects-Embedded Bytes							
Action	S-S-S	S-M-M	M-M-M	M-M-L	M-L-L	L-M-M	L-L-M	L-L-L
para-1.0	0.58s	1.58s	1.81s	2.27s	3.57s	2.05s	2.58s	3.80s
pipe-1.1	**38**	**60**	**52**	**50**	**59**	**47**	**47**	**56**
compress	0	0	0	10	9	1	10	14
bundle	**29**	**56**	**47**	**47**	**61**	**42**	**56**	**58**
bundle.gz	**33**	**61**	**52**	**51**	**65**	**46**	**59**	**61**
baseonly	**72**	**89**	**82**	**86**	**91**	**77**	**82**	**87**
halfobject	**53**	**21**	18	**34**	**31**	16	**38**	**29**
halfres	4	0	0	2	6	0	3	6

Bundling of content shows a significant effect for better-connected clients and when the latency is large for poor clients. Combining it with compression is useful for these poor clients, but it does not show much additional benefit for better-connected clients.

The other lossless action we studied was server support for persistent connections, both with serialized and pipelined requests. Results for persistent connections with serialized requests were not shown in the tables because it did not show significant performance improvement under a wide variety of client/content conditions. However, pipelining does have a significant effect for clients with a high throughput or RTT. This can be an effective action for clients with a high delay to a server, but the server can only control the persistence of the connection, it cannot force the client to actually pipeline the requests.

6. RELATED WORK

Different approaches have been investigated to improve response time for users. One approach has been to investigate alternate policies to mark network packets for improved interactive network application performance [17]. This approach seeks to improve performance for all Web clients, rather than for clients of a specific Web server. Work in [1] seeks to adapt the content served to users based on server load. As a server becomes loaded it begins to degrade the content served to lower-priority clients. Another approach is for Web servers to take into account user expectations in scheduling requests [5]. Subsequent studies have proposed alternate admission control and server scheduling policies [7, 6] for improved response time of their clients. In contrast to these approaches, our work includes a working server prototype and examines a broader set of server actions that could be taken in response to poor client performance.

The concept of dynamically altering multimedia content in a Web page depending on network path characteristics was first reported in a United States patent [13]. In this proposed scheme, a Web server would monitor a variety of network characteristics (such as round trip time, packet loss) between itself and the client and correspondingly adjust the quality of the content returned to the client. However, this patent deals exclusively with altering the content or its delivery. It does not cover the range of other server actions that are part of our approach. Additionally, the network aware clustering we use to construct a coarse grouping of clients has been shown to be superior to the "nearness" approach discussed in [13]. We are not aware of any published research on the idea proposed in this patent.

Rather than let the server estimate a client's characteristics, other approaches can be used. Explicit client specification of network connectivity is used in many multimedia players. The SPAND system uses an approach where a group of clients estimate cost to retrieve resources and share it amongst themselves [18]. If available, these approaches are useful, but a client may not know connectivity information or this may information may change over time.

7. CONCLUSIONS AND ONGOING WORK

In this work we present the results of evaluating a modified Web server that is capable of classifying clients online, delivering modified server actions, and measuring the latency reduction to different clients all on a common platform. We can also evaluate the cumulative effect of two or more actions. Administrators of high volume Web sites can benefit from our results by examining their content mix to see how different actions will benefit their clients.

The overhead imposed on the server as a result of classifying clients and taking alternate actions is small: on average 75 microseconds and thus not noticeable to an end user. Even under overloaded conditions our server's degradation is reasonable. Note that, under overloaded conditions, site administrators can simply turn off classification to avoid incurring any costs. However, the alternate action available for poor clients can be used for all clients, if the server is overloaded. For example, `www.cnn.com` site resorted to a simpler, text-oriented home page on September 11th, 2001 in order to serve more clients.

Results from our classification of clients shows that classifications largely match the expected values based on our measurements of the client connectivity. The results are relatively stable over the lifetime of a client test, although the mechanism can adapt over a longer period of time if a client's connectivity does vary. These results are particularly encouraging because we used all page requests for classification and used the same classification thresholds for all tests. Classification accuracy can only be improved for individual Web sites through appropriate selection of thresholds and pages to use for classification.

Client classification is important because only the lossy server actions of reducing or eliminating embedded objects were found to be significantly reduce download time in all cases. The lossless actions of pipelining and bundling yielded significant performance improvements for poor clients with long latencies and better-connected clients, but both of these actions require support from clients. Compression, more widely supported by current browsers, was most effective for poor clients with relatively shorter latencies or all poor clients when combined with bundling. However, compression is a default option for some client operating systems, so its effect for those clients is reduced.

While the reduction of time to glass will vary with server action and the resource being downloaded, a server can carry out a few simple actions to establish benchmarks of potential reduction. For example, if the choice of a specific compression algorithm on

certain resources requested often reduces the size by a significant amount, it would be worth considering using it. Earlier work [14] has shown that this is feasible. Similarly, for each of the other server actions, one could establish basic reduction thresholds and test it with clients of differing connectivity. Such actions need to be done only for a few resources for a few different levels of connectivity and can be applied broadly.

As part of ongoing work, we are looking at other possible server actions that matter for repeat accesses for a pages (such as delta encoding) and policies regarding cacheability of objects. Finally, our testing methodology did not allow us to test the usefulness of client clustering in the classification process. In our earlier work [11], we found this technique to be particularly useful for some sites. We would like to create a test for clustering where we have multiple live clients with different IP addresses in the same cluster.

Acknowledgments

The authors would like to thank those who supplied us data for the project in the form of proxy logs without which such research would be impossible. In addition, we thank all those who have given us access to their machines to conduct our experiments. Finally, we wish to thank the anonymous reviewers for their comments on an earlier version of this paper.

8. REFERENCES

[1] T. F. Abdelzaher and N. Bhatti. Web Server QoS Management by Adaptive Content Delivery. In *Proceedings of the International Workshop on Quality of Service*, London, England, June 1999. http://www.eecs.umich.edu/~zaher/iwqos99.ps.

[2] Apache Software Foundation. http://www.apache.org.

[3] Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/docs-2.0/programs/ab.html.

[4] H. Balakrishnan, M. Stemm, S. Seshan, and R. H. Katz. Analyzing Stability in Wide-Area Network Performance. In *Measurement and Modeling of Computer Systems*, pages 2–12, 1997. http://www.cs.cmu.edu/~srini/Papers/publications/1997.sigmetric/sigmetrics97.pdf.

[5] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-Perceived Quality into Web Server Design. In *Proceedings of the ninth International World Wide Web Conference*, Amsterdam, The Netherlands, May 2000. http://www9.org/w9cdrom/92/92.html.

[6] J. Carlstrom and R. Rom. Application-aware Admission Control and Scheduling in Web Servers. In *Proceedings of the IEEE Infocom 2002 Conference*, New York City, June 2002. IEEE. http://www.ieee-infocom.org/2002/papers/560.pdf.

[7] X. Chen, P. Mohapatra, and H. Chen. An Admission Control Scheme for Predictable Server Response Time for Web Accesses. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, May 2001. http://www.cs.ucdavis.edu/~prasant/pubs/conf/www10.ps.

[8] `convert`. http://www.imagemagick.org/www/convert.html.

[9] B. Krishnamurthy and J. Wang. On Network-aware Clustering of Web Clients. In *Proceedings of ACM Sigcomm*, August 2000. http://www.research.att.com/~bala/papers/sigcomm2k.ps.

[10] B. Krishnamurthy and C. E. Wills. Analyzing Factors that Influence End-to-End Web Performance. In *Proceedings of the Ninth World Wide Web Conference*, Amsterdam, The Netherlands, May 2000. http://www.research.att.com/~bala/papers/www9.html.

[11] B. Krishnamurthy and C. E. Wills. Improving Web Performance by Client Characterization Driven Server Adaptation. In *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, USA, May 2002. http://www.research.att.com/~bala/papers/lac.ps.

[12] B. Krishnamurthy, C. E. Wills, and Y. Zhang. Preliminary Measurements on the Effect of Server Adaptation for Web Content Delivery. In *Proceedings of the Internet Measurement Workshop. Short abstract*, Nov. 2002. http://www.research.att.com/~bala/papers/spinach-sa.ps.

[13] J. C. Mogul and L. S. Brakmo. Method for dynamically adjusting multimedia content of a web page by a server in accordance to network path characteristics between client and server, June 2001. United States Patent 6,243,761.

[14] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proc. ACM SIGCOMM*, Aug. 1997. http://www.research.att.com/~bala/papers/sigcomm97.ps.gz.

[15] D. Mosberger and T. Jin. httperf—A Tool for Measuring Web Server Performance. In *Proceedings of WISP '98*, Madison, Wisconsin, USA, June 1998. http://www.hpl.hp.com/personal/David_Mosberger/httperf.

[16] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of the ACM SIGCOMM '97 Conference*. ACM, Sept. 1997. http://www.acm.org/sigcomm/sigcomm97/papers/p102.html.

[17] W. Noureddine and F. Tobagi. Improving the Performance of Interactive TCP Applications Using Service Differentiation. In *Proceedings of the IEEE Infocom 2002 Conference*, New York City, June 2002. IEEE. http://www.ieee-infocom.org/2002/papers/354.pdf.

[18] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared Passive Network Performance Discovery. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, USA, Dec. 1997. http://www-2.cs.cmu.edu/~srini/Papers/publications/1997.USITS/usits97.ps.

[19] K.-P. Vo. Vmalloc: A General and Efficient Memory Allocator. *Software: Practice and Experience*, 26:1–18, 1996. http://www.research.att.com/sw/tools/vmalloc.

[20] K.-P. Vo. CDT: A Container Data Type Library. *Software: Practice and Experience*, 27:1177–1197, 1997. http://www.research.att.com/sw/tools/cdt.

[21] `wget`. http://www.gnu.org/software/wget/wget.html.

[22] C. E. Wills, M. Mikhailov, and H. Shang. N for the Price of 1: Bundling Web Objects for More Efficient Content Delivery. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, May 2001. http://www.cs.wpi.edu/~cew/papers/www01.pdf.