Web Content Adaptation for Peak Loads
Trading Quality for Performance

by

Michael Gopshtein

Supervised by

Prof. Dror Feitelson

A Thesis Submitted In Partial Fulfillment Of The
Requirements For The Degree Of Masters Of Science

The Rachel and Selim Benin School of Computer Science and Engineering

The Hebrew University of Jerusalem

December 2009

Abstract

This work addresses a need of a web site to serve unpredictably high load volumes, as caused by a Slashdot effect or other exceptional event. We focus on content adaptation methods which reduce the quality of a page content, as a tradeoff for a boost in performance of the server, in terms of number of concurrent users it can successfully serve.

Based on estimation of the costs of each type of server's activity, defined as utilization of various hardware and system resources, we develop a strategy for effective optimization of the site. Main components of our optimizations are: reduction in the number of HTTP requests as result of eliminating some of embedded objects, and reduction in the number of transmitted bytes per page, which is achieved by compression of graphical and other media content.

We then build an experimental setup to validate the effectiveness of proposed optimization methods, where we compare the performance of Apache web server when serving the original page content versus the content in two levels of optimization. Results show a reduction of 91% in the average response time for a single transaction, and an increase of 340% in number of user transactions per second that could be successfully served.

Acknowledgments

I wish to express my gratitude to Prof. Dror Feitelson for his guidance, continuous support and high responsiveness throughout the project. His insights made me see many aspects of the work from a different perspective and to learn lots of new and interesting subjects.

I would also like to thank my family for their endurance and encouragement during the whole period of my studies.

Contents

1 Introduction

Success of many commercial as well as non-profit web sites depends on the number of visitors to the site and on ability of those users to complete their intended *transaction*, be it purchase in an online store, or reading an article on a news site. This goal can be in turn translated to ability of the users to perform the whole sequence of HTTP requests required for such a transaction. Failure of the user to get a proper response on one of the steps will put the whole transactions at risk and thus can be expressed in financial terms.

We are witnesses to real-world examples of efforts made by web site owners to be able to serve every single request. This includes various optimization techniques implemented in the web servers and hardware overprovisioning which aim to provide high availability and acceptable response times for cases of peak loads on the system.

1.1 Background

We should distinguish between the normal fluctuations in the average loads, which are function of time of the day, day of the week etc. (Fig. 1), and the irregular peaks in users' activity as result of unpredicted events. While periodic load changes of the first type are usually taken into account as part of sizing of hardware requirements, the latter pose higher challenges on the system.

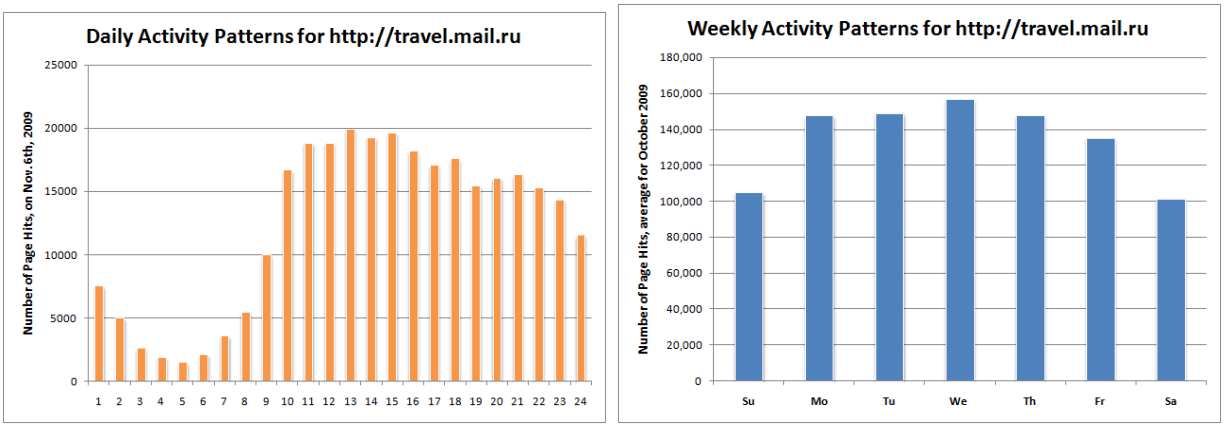


Figure 1: Periodical load distribution patterns for web site *travel.mail.ru*, based on *top100* statistics collected by Rambler.ru engine.

One rather extreme example of such an event is described in [12], and it shows the exponential increase in traffic for CNN.com as result of the 9/11 tragedy. The number of HTTP requests was doubled every 7 minutes, and grew from less than 85,000 hits/second to 229,000

hits/second in just 15 minutes. The description of the steps taken by CNN staff to continue to serve all incoming requests is interesting. Among those were extending the number of web servers from 10 to 52 in a short time interval and shutting down monitoring software to free additional resources for the web server processes. In addition, the content of the home page was shrinked, until the whole page consisted of only 1247 bytes of HTML, a logo and a small picture.

A more common cause for exceptionally high load peaks is the *Slashdot effect* [18]. It occurs when a popular web site posts a link to a smaller site, such that the number of users following that link largely exceeds the usual load on the smaller system, in some cases forcing it to become unavailable.

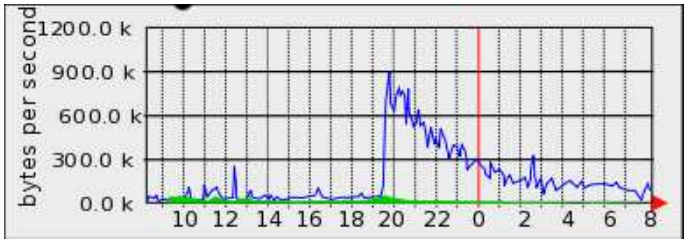


Figure 2: Slashdot Effect in Action
Graph from a web server statistics generator showing a moderate Slashdot effect in action (from Wikipedia).

1.2 Related Work

General Optimization Most of the research published so far on the subject of improving the performance of web sites focuses on functionally transparent optimization techniques, which generally allow the servers to cope with higher numbers of concurrent clients and to show lower response times, while preserving same level of content quality. Optimizations of this type usually aim to reduce the resources needed to serve the expected load, and are not used to handle peak loads. In particular, it would be always beneficial to implement such optimization methods in the areas where they are applicable.

Sounders [16] summarizes some of those techniques, including the following:

- Make Fewer HTTP Requests
  Combining multiple script files or multiple stylesheets into a single file. There are also methods for packing several small images to a single image file, and cropping original images from the large image on the client side, using one of available scripting technologies. As an example, consider Figure 3 which shows a query results page from the Google search engine. All graphical elements, highlighted by red frames, are downloaded by the browser as a single image file as shown in Figure 4.

- Add an Expires Header
  When a web browser encounters the same resource multiple times (e.g. an image, script file, or stylesheet), it should decide whether it can store the resource in local cache on the first occurence, and use the cached version later, as opposed to having it downloaded from the server each time. One of the instruments by which the server can affect caching algorithms of the browser is the "Expires" HTTP header, so making correct use of it for static resources can reduce the average number of HTTP requests per page made by the client.

- Gzip Components
  Gzip is the most popular compression method used for web resources in order to reduce network bandwidth, and it typically reduces the size of textual content by 70%. On the other hand, compression requires additional CPU cycles on the servers, so it may not improve performance of the server when the CPU is the bottleneck on the system.

- Make JavaScript and CSS External
  Both JavaScript code and stylesheets can be embedded inside HTML file, or be referenced from HTML as stand-alone components. There's advantage in having those resources as external files, as in such case they can be cached by the browser and be later reused for other pages on the same site, thus reducing the overall size of the data downloaded from the server by a single client over the course of a whole session.

- Configure ETags
  Entity tags (ETags) are part of the HTTP/1.1 specification, and their purpose is to tag HTTP resources with additional information (such as version number or checksum). When a web browser encounters a resource, which is already saved in a local cache, and it's unknown whether the resource is still up-to-date, it will send a HTTP request to the server, providing the ETag of that resource that was returned by the server in original response. The server can validate the status of the resource based on the ETag, and reply with 304 Not Modified response rather than sending the same resource over the network again, if the client already has the most recent version of the file.

In addition to this general set of rules of thumb which can be considered for any web site, all modern web servers can be tuned using various configuration parameters to the best performance based on specific hardware resources and functional requirements. The configuration includes parameterization of in-memory caching of resources by the server, operating system objects (e.g. number of processes and threads), network parameters (TCP connection timeouts), enabling various HTTP protocol features (compression, Keep-Alive connections) etc.

Dynamic Adaptation Obviously, making the system have enough capacity to handle exceptional peaks in the load would mean operating at very low utilization most of the time. A more suitable solution would be adaptation of the web server to load conditions in real time. One class of such solutions uses dynamic allocation of resources according to current

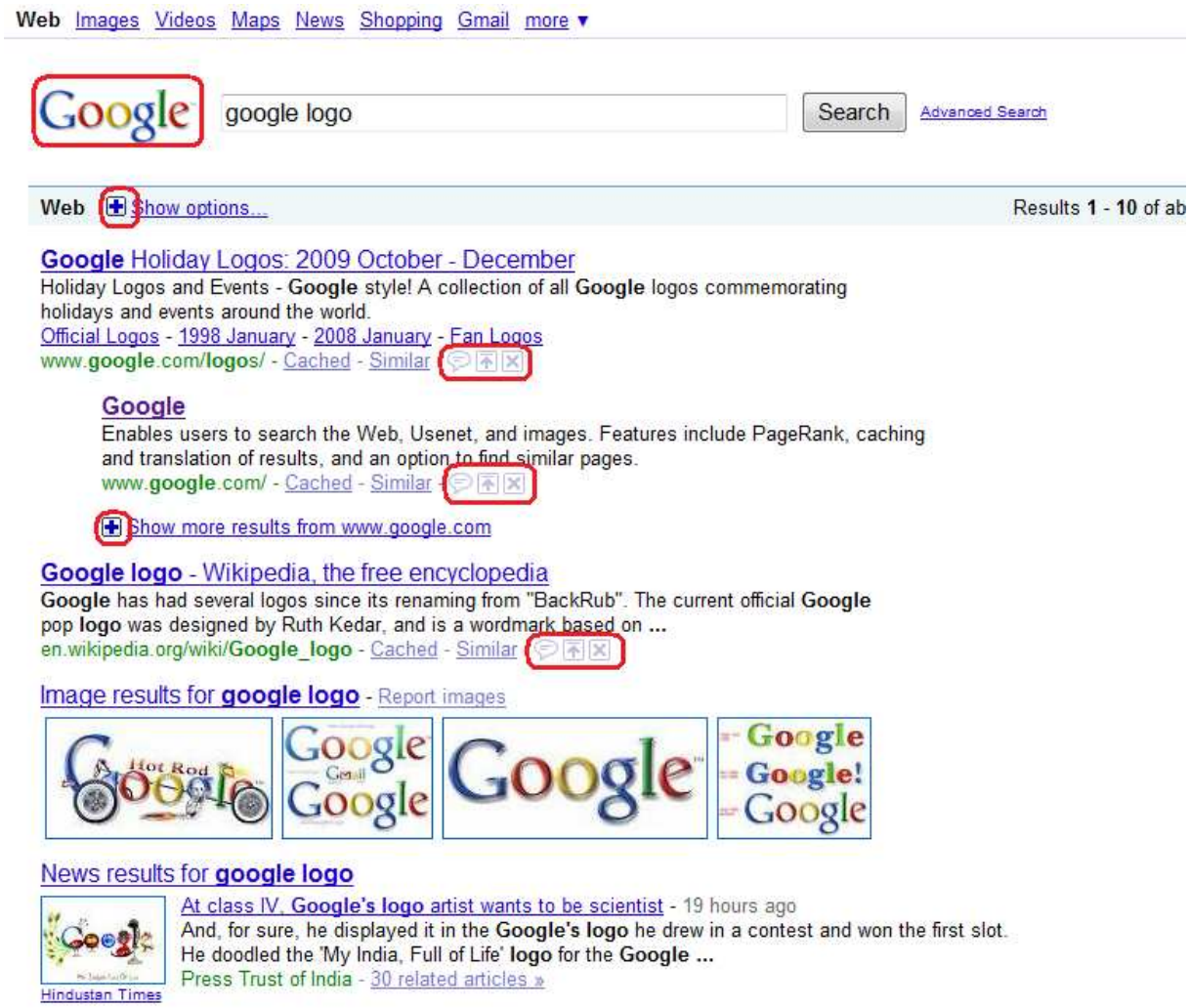


Figure 3: Google Search Page
Snapshot of search results page on Google (`http://www.google.com/search?source=ig&hl=en&rlz=&q=google+logo+&aq=f&oq=&aqi=g10`), as saved on Nov 14, 2009.

needs (and one such example would be cluster-based systems).

A different approach is demonstrated in [3], where the authors suggest implementing session-based admission control, which will not accept new user sessions while the server is already overloaded. The motivation is the will to avoid providing a bad service to all users, but rather serve as many users as possible with best quality of service (QoS). As a result the total number of successfully completed sessions (business processes) might be incremented. A clear drawback of such a solution is having some of the users rejected, with all negative implications of such behavior.

Figure 4: Google Logo Image
Single image file, containing multiple graphical elements on the Google search results page.

A number of articles present similar concepts to those discussed in the current work as they deal with content adaptation. The common feature of all methods in this category is the attempt to create a *lighter* version of the original web site which will allow serving a larger number of users.

A number of content adaptation techniques are presented in [1], such as *adaptation tags* inside HTML files, and image quality degradation by lossy compression. The authors emphasize the importance of the number of embedded objects on the overall site's performance, and suggest eliminating small cosmetic items from the page. Another technique is reduction in the number of internal links, thus making the users consume less content from the given web site. The authors further present a complete system for semi-automated content adaptation, which includes measuring server response time as a trigger to adaptation activation and a content adaptor which creates a copy of original directory tree while replacing the content with adapted versions.

Another work [13] also defines techniques for content adaptation and divides these into five categories:

1. Information Abstraction
   Compression of the data while preserving the most valuable information. Examples include text summarization, thumbnail generation for images and key-frame extraction for video.

2. Modality Transform
   Transforming the content from one mode to another, e.g. converting a video to set of images, speach-to-text and text-to-speach. The primary goal of this type of transforms is adaptation of the content to capabilities of the client device.

3. Data Transcoding
   Converting data to formats supported by client environment, such as GIF-to-JPEG or Postscript-to-PDF conversions.

4. Data Prioritization
   Dropping less important pieces of data under certain constraints. Requires the ability to distinguish between more and less important parts of the information.

5. Purpose Classification
   Removing redundant objects based on classification of their role in the web page.

1.3 Current Research

In this work we also follow the path of developing techniques for content adaptation on the way to creating an optimized version of an original web site. As a trade-off for improved performance, the quality of served content is lowered, as is the case with all content-adaptation approaches. Here we assume that many clients would prefer to receive a degraded version of the content rather that experiencing connection rejection or other failures.

We start with a detailed analysis of the *costs* of different types of HTTP requests, in terms of resources consumed by the web server to process them. In the same time we do a survey on a set of real web sites in order to estimate the volume of resources of each content type, both as number of individual HTTP requests and the total size of retrieved data.

Next we consider a number of techniques for content adaptation, and estimate the overall performance gain of each, based on the model of the costs as resulting from the initial analysis. In the proposed model, an optimized version of the site is created and maintained synchronized with new changes in the original files. The web server is monitored to detect a state when it gets overloaded, and in such case it is automatically switched to serve the optimized tree of the files. Once the load on the system gets back to the normal range, the server is switched back to the original version.

In the last part of the work we validate the results by applying the suggested methods on a sample web site, and measuring the impact of such adaptation on user-perceived performance of the web server.

The scope of current work is limited to static resources and results do not automatically apply to dynamically generated content, although some of the conclusions are useful in the general case.

2 Costs of Resources

In this part of the work we will estimate the system resources required to serve an HTTP request by a Web server. These quantities are highly dependent on the properties of the requested Web object, e.g. the size of returned data, computational resources for dynamic pages etc.
As result we will present a model of resource consumption of HTTP requests as a function of the request's parameters. The model will be used later in order to estimate the expected benefits of various content modifications, that are supposed to reduce resource usage.

2.1 Static Content

The first observation that we are going to make is that for static Web pages the most dominant parameter would be the total size of returned file, e.g. amount of resources required to serve a request for 5Kb image file is identical to that of required to serve a 5Kb HTML page. The major exception to this rule would be requests for textual files (HTML, CSS etc.) when a data compression algorithm is utilized by the server, and we will refer to those differences later.
The current work is limited to analysis of web sites serving static content only; as such most of resource costs will be given as a function of a size of returned file.

2.2 System Resources

The system resources that we should measure are:

1. Network bandwidth
2. CPU time
3. Disk utilization

In addition, we will measure response time. Although this quantity can't be classified as a system resource, it is important to include this parameter in the model for number or reasons:

- The response times experienced by the user largely contributes to perceived quality of the Web site;
- Response time is a measurement for the time a request spends on the server, and can be a basis for estimating number of parallel requests served by the server as a function of total number of requests per time unit.

2.3 Locality of Requests

Understanding the most common patterns of websites' usage is crucial to our ability to produce realistic estimates for consumption of resources by the web server. As the introduction to this work has stated, the goal is to handle cases of unexpectedly high peaks in number of users visiting the site; in cases when a site operates in reduced quality mode for a long time, it, probably, means that it should perform a more accurate sizing of the system.
We make an assumption that in those cases of peak loads, the requests will show a high locality - a large portion of requests will map to a small number of web pages, or files. The examples shown above support this assumption: visitors of CNN site at events on *September 11th* were interested in viewing the home page of the site only; when a peak is caused by a *Slashdot Effect*, the major part of the visitors follow an external link, leading to a specific page in the site.

In order to validate the assumption of locality, we've analyzed web-logs, collected from web servers serving the official site of *World Cup 98* soccer games in France, which was created for a short time interval and was dedicated to that event. Figure 5 shows the percentage of HTTP requests to the most frequent files from the total number of requests to the site. The 15 most frequent components make 13.4% of the total requests during the normal load, compared to 24.2% during the peak hours (the most frequent components might be different in normal and peak loads). Starting from 30th most popular component, the graphs begin to be approximately parallel, which shows that frequencies of non-top pages do not significantly change on peak loads.

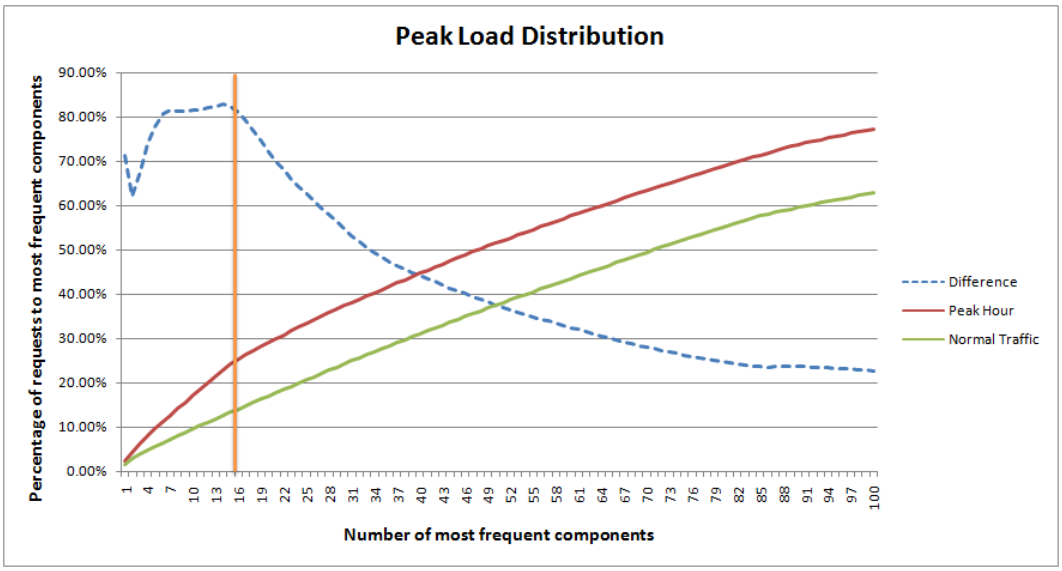


Figure 5: Peak Load Distribution
From the official site of the World Cup 98

The most noticeable implication of locality of requests is in areas where caching of response data is applicable, for which a high percentage of cache hits is expected, even with moderate cache sizes. We will use this conclusion in later sections, when estimating the requirements for different system resources.

2.4 Compression

Compression can be applied on the response content, in case that same compression algorithm is supported both by the client and the server. When applied to textual content, compression algorithms may reduce the size of the file by more than 75%. When compression is enabled on the web server, the data can be compressed on the fly for each request, or there can be a cache of compressed content, allowing reducing the CPU resources required for data compression. Caching of compressed content for static pages is available on IIS servers starting from version 6, and is enabled by default for static content in version 7.
The current work focuses on static content, under the assumption that the site is already optimized using methods which do not cause quality reduction, so we'll assume that for frequently accessed pages the content is already compressed in advance and is stored in the cache, and as was shown above, we consider cases with high ratio of cache hits. So we'll refer to compressed size as the *file size*, and can further ignore compression settings of the web server.

2.5 Response Time

Definition of Response Time Figure 6 shows a high-level overview of single HTTP request and its response. In this example the time the request is active on the server consists of.

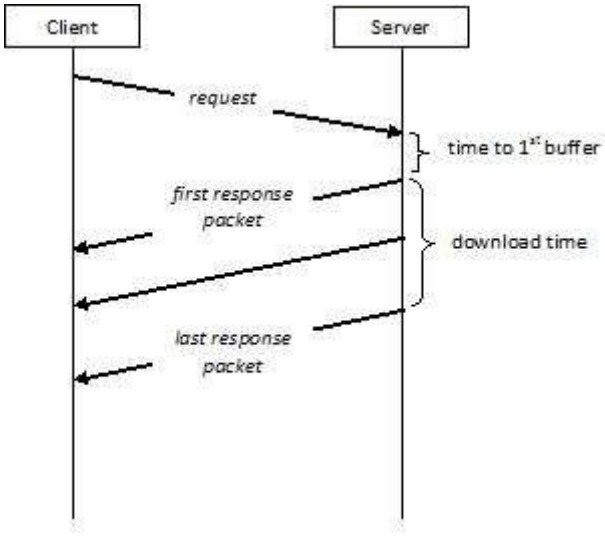


Figure 6: HTTP Request Overview

1. Time to 1^{st} buffer

2. Download time

The *time to 1st buffer* is, generally, a measure for a time it takes for the server to parse the request and to locate the response content (a *file* for static Web site). The *download time* is the time required to transfer the file back to the client, and (given certain server's state and properties) is a function of the size of the file being transferred, as well as of quality of network connection between the server and the client. As seen in the Figure, the download time can be effectively measured on the client side, assuming that the on average the latency between the server and the client does not significantly change in a course of a single response.

It's important to note, that the *perceived* response time may include TCP connection establishment time, as well as the time required for the browser to render the Web page. In addition, the client usually measures the total time taken to download a whole page, including all additional components.

Methodology In order to analyze web server response times, an automatic tool was created to perform the following tasks:

1. Select a *random* web site:
   This step is based on the "Random article" link available on Wikipedia (`http://en.wikipedia.org/wiki/Special:Random`). The title of the returned article is used as a search term in Google search engine. The first link is chosen from the results' set, and only the "host" part of the URL is taken, in order to make a request to the home page of the site.

2. Request the home page of selected site:
   The URL selected in the 1^{st} step is presented to Internet Explorer Web browser, which makes the request to the server, and it also performs all subsequent requests to additional resources required to render the page.

3. Statistical data is collected:
   The tool intercepts all system calls made by the browser to open connection to server and send and receive data. This information is aggregated to single HTTP requests and processed to measure the required parameters. The parameters collected for each request are:

   - URL
   - Domain ("Host" HTTP header)
   - Size ("Content-Length" HTTP header or size of chunked response)
   - Content type ("Content-Type" HTTP header)

- Download time (the time between the first and last successful "read" operations on the TCP socket)

4. Off-line analysis:
   The results of the previous step are stored in a database for further off-line analysis.

Results We used this procedure to collect data from 95 web sites.

Sizes of HTTP Components

Figure 7 shows the log-log complementary distribution (LLCD) of sizes of data returned by the server in response to measured HTTP requests. As it shows, the tail of the distributions approximately matches that of a Pareto distribution.

We can also see that about quarter of the total traffic bytes come from the largest files, with probability of 0.2%, this information can be used as a hint later in the work, as it shows that by eliminating a small number of largest files we can save large percentage of required bandwidth.

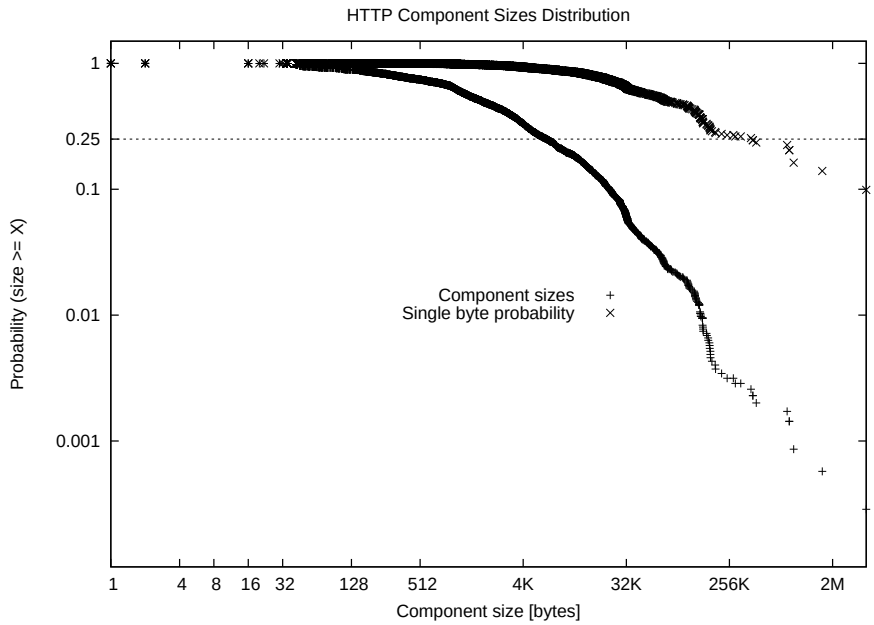


Figure 7: HTTP Component Sizes Distribution

Distribution of Requests by Content Type

The distribution of sizes of the components by the Content Type is shown in Figure 8. The percentage of the textual content (including HTML, XML, JavaScript, CSS and general Text) out of the total size of downloaded components is 44%, and the size of multi-media files - 56%.

Response Times

In order to see the response time as a function of component size, we have to choose only

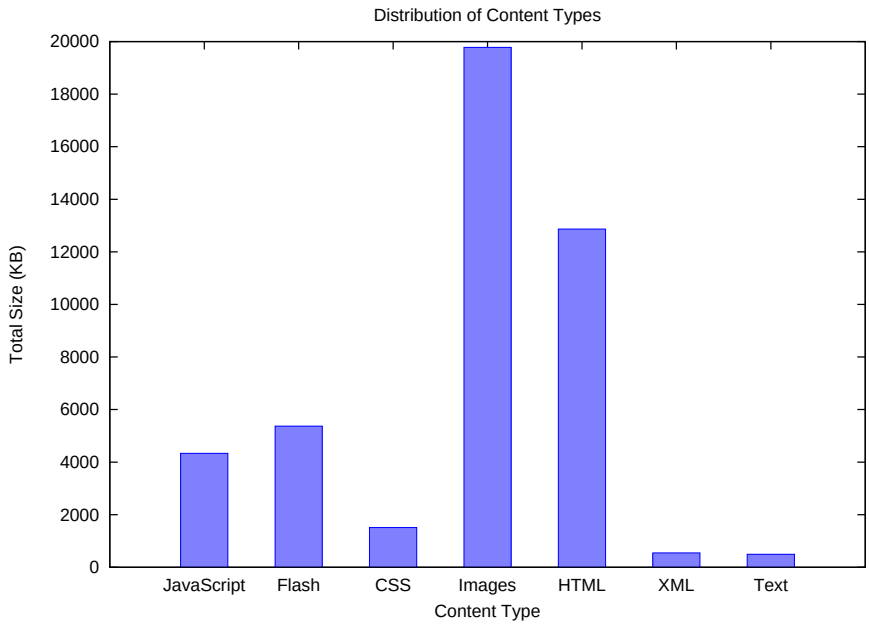


Figure 8: Distribution of Content Types

the requests made to the same Web server, otherwise the differences in network latency and bandwidth would affect the accuracy of the results. Figure 9 shows results for servers *i.usatoday.net* and *image.timeinc.net* accordingly (Note: for some small objects the time appears as 0 msec, it happens when all the data is received by the browser in single read system call).
The download times appear to have values which are multiple of 18 msec. The reason for this phenomena is unknown, we can assume it's related to scheduling mechanisms on Windows operating system. Nevertheless, we can see the trend - as expected, the download time is a linear function of the file size.

2.6 Bandwidth

As was shown in [14], the time required to route a single IP packet from source to destination is very little affected by the size of the packet, thus the more usable equivalent of content size would be the number of packets required to transfer it.
One immediate implication of this rule is that once the whole response can be sent in a single packet, the bandwidth consumed by this request (as well its response time) can't be optimized by reducing the size of the response.
In order to estimate the size of data which can be sent in the first response packet, we should take into account:

1. The total size of packet that can be transmitted

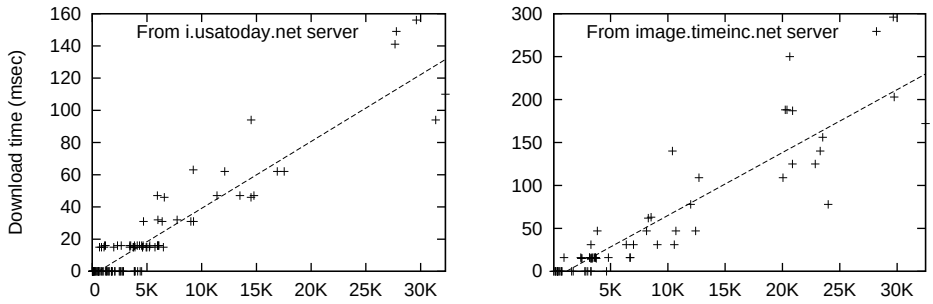


Figure 9: HTTP Component Download Times

2. The size of IP/TCP headers

3. The average size of HTTP response headers

Typically a single network packet contains up to 1470 bytes of HTTP data (1520 bytes total packet size, out of that 14 bytes for Ethernet header, 40 bytes for IP and TCP headers). In common implementations of TCP stacks special algorithms are being used to ensure that no IP fragmentation occurs.
Examples of HTTP header lengths:
from `www.cnn.com`, Apache: 280-325 bytes
from `www.top500.com`, Apache: 296-300 bytes

Finally, we can conclude that files under size of 920 bytes are expected to be downloaded in a single packet. The general formula for max file size transmitted on N packets would be 920 + (N-1)*1258 bytes.

According to data shown in Figure 7, 43% of total responses fit in single response packet and 24% - fit two in packets.

2.7 Disk Utilization

Concerning the disk utilization, the feature of locality of requests discussed above plays a very important role. If we consider the default configuration for IIS 6 server [17], it allocates

approximately half of available physical memory for caching, while a size of a cached file is limited by default to 256 KB. Assuming that a modern server is equipped with at least 4GB of memory, it can cache more than 8,000 files of maximal size, and on average the number of files larger than that size is 1/3%.

Based on this data we conclude that the disk utilization won't be a bottle neck for performance of the web server at peak loads, and it can be taken out of the model.

2.8 CPU Utilization

It's not possible to measure the CPU consumption in absolute units, therefore we are going to estimate the relative CPU times measured while serving requests of different sizes. We are going to break the overall CPU consumption to HTTP headers processing and file transfer portions.

Methodology As stated above, the type of the file being requested does not affect the amount of web server's resources required to serve it. In order to reach a more accurate model, there's a need to breakdown the CPU time into different components of a single HTTP request. The major parts of request processing by web server, when referring to static files, are:

1. Opening new TCP connection
2. Parsing request headers
3. Locating requested file
4. Network communications - receive request and transfer response

A number of different load types were applied on the web server. For each test case the total CPU time of all the server's processes was taken, and the difference in this time before and after the load is considered as the CPU time required by the server.
The server used for the tests was Apache 2.0 running on Red Hat Enterprise Linux 5.0. All *httpd* processes were monitored, and for each of them the stats were taken from */proc/PID/stat* pseudo-file - *utime* and *stime* indicating the number of jiffies that the process has been scheduled in user and kernel modes accordingly. Preliminary tests have shown that such measurements show consistent results with only a slight differences (up to 4%) between different tests with same load on the server.
The client used in all tests was *Internet Explorer 7.0*.

Test Cases All the CPU times are measured in number of *jiffies*, and we are mostly interested in relative cost of each of the load types.

1. Opening a new connection
The purpose of this test is estimating the cost of opening a new connection by a client. In final calculations we should take into account that usually the same TCP connection is reused by the browser to download multiple HTTP objects. In the test a new TCP connection was opened to the server, and closed immediately afterwards, repeated 10,000 times.
Results: total of 156 jiffies (68 in user mode, 88 in kernel mode).

2. Downloading a big file
This test comes to estimate the cost of processing time required for data transfer, when the same TCP connection is used to deliver large data volumes. In this test a single textual file of 1GB was downloaded by the client.
Results: total of 414 jiffies (11 in user mode, 403 in kernel mode).

3. Requesting a small file
In this case a small HTML file (73 bytes) was requested 10,000 times. The size of request and response HTTP headers was 315 and 253 bytes accordingly, so a total of approx. 6.1MB of traffic was transferred between the peers, and a new TCP connection was opened for each HTTP request.
Results: total of 602 jiffies (333 in user mode, 269 in kernel mode).

4. Requesting a medium file
This case is similar to previous one, but a larger files were downloaded - 10KB.
Results: total of 622 jiffies (345 in user mode, 277 in kernel mode).

5. Non-existing file
This final test case comes to measure the amount of CPU time needed to return "File not found" response (HTTP code 404). Server's response did also contain a short HTML content describing the error to the user (287 bytes). Size of request and response headers was 310 and 180 bytes accordingly.
Results: total of 585 jiffies (335 in user mode, 250 in kernel mode).

Table 1 summarizes the number of jiffies consumed in each test case, here a plus marks show which server-side processing was involved in each case.

Analysis From comparison of test cases (3) and (4) we can estimate the amount of CPU required to process the request and to deliver the file content. We should first eliminate the cost of opening new connections, by subtracting the CPU time calculated in test (1). Conclusion: total CPU time required for 10,000 requests for 10K files: 466 jiffies, 446 of them (96%) - to parse and process the request, 20 (4%) - to transfer 10K of cached data.

If we take into account the CPU time required to open a TCP connection, we will get the following results: HTTP request for a 10KB file, which opens a new connection, requires

	Connection establishment	Parsing and processing	Transmission	Total jiffies
New connection	+			156
Large file		+	+	414
Small file	+	+	+	602
Medium file	+	+	+	622
Non-existing file	+	+		585

Table 1: Summary of CPU Consumption in Test Cases

25% of CPU time for connection initiation, 72% to parse and process request, remaining 3% - to transfer the response.

According to [15], the average number of requests per connection is 27.6, taking this into account we will get the following CPU time distribution for 10KB file requests: 1% to establish TCP connection, 95% for parsing and processing, 4% to transfer the response content.

The CPU consumption for transfer of a file generally depends on its size - the smaller it is, the lesser is the percentage of CPU cycles required for network transfer out of total cycles used to process the request. We have shown that for 10KB file the parsing required 24 times more CPU time than the transfer, so by extrapolating this result, we conclude that in order for the network transfer to take half of the total CPU consumption, the size of a file should be 240KB (24 times 10KB). The analysis of HTTP component sizes from *random* websites has shown that the probability of a component to be larger than 10KB is less than 20%, and to be larger than 240KB - only 0.3%.

We see that a large fraction of the time is spent on parsing and processing, and only a small part on transfer. Therefore, when CPU utilization is the bottleneck, we should look for a ways to reduce number of HTTP requests made to the server, as the expected benefit of this approach is significantly larger than that of reducing the size of individual components.

3 Optimization Methods

In this part of the work we present different methods for optimizing Web pages in terms of server resources required to handle all related HTTP requests. Using the results from the previous section, we will estimate the expected impact of each optimization on the utilization of each resource type. All optimizations will be classified by expected reduction in quality of the web page as perceived by the end user.
Each optimization method is applicable to certain types of page components. In order to perform calculation of the over-all effect of suggested optimizations, we will consider a *representative* distribution of component types in the page.

3.1 General Approach

Considering the results presented in the previous section, we can conclude that the following resources can be potentially optimized:

1. Network bandwidth
2. CPU

For each component of the original web page, the following optimization approaches are possible:

1. Reduce the size of returned file
2. Eliminate HTTP request to the component

Reducing Size For small and medium file sizes we expect to find a positive effect of such optimization on network bandwidth requirements only, but not significant reduction in CPU utilization, as most of processing resources are consumed by the server to parse the request and locate the file to be sent as response to the client.
There's a much higher motivation to reduce the size of large files. As shown in 7, large components, although being rare, are responsible for a large portion of the total traffic, hence we are interested in reducing their size as much as we can. When transmitting a large file, the CPU consumption also represents a considerable overhead.

Eliminating HTTP requests By reducing the number of HTTP requests made to the server, we can significantly reduce the CPU time consumption, as was shown in the previous section. Depending on the size of eliminated component, total network bandwidth will be also reduced, as in case with size reduction.
Although this work makes an assumption that the websites are generally being optimized using techniques that are transparent to the users (do not reduce the quality of web page), tests have shown that optimization on number of HTTP requests is rarely performed. We would expect that some effort will be applied to consolidate multiple HTTP requests into

a single one, e.g. by combining a number of images into a larger image, and cropping the relevant areas on the client side using one of the scripting technologies. The only site that has performed such optimization was `www.google.com`, as was mentioned in Introduction (Figure 4).
This observation leads us to conclusion, that by removing visual elements from the web page, we eventually reduce the number of HTTP request made by the client while rendering the page.

3.2 Perceived Quality

The optimization methods presented in this work make, in general, a trade-off between quality of the web page, as perceived by the users, and performance of the web server. In order to be able to measure the impact of certain changes on the *quality* of the site, we have first to identify the main features by which the user judges the quality of the site.

One way to define a user-perceived quality is in terms of responsiveness of the site [2]. This, in turn, can be broken down to server's response times for HTTP requests for components of a same page, network latency and client-side presentation time. Obviously, optimizations which are discussed in current work, do not affect network connection quality, as such they should not increase the network time. As optimization is aimed to improve the performance of the server, the HTTP response times are expected to be reduced. Considering reduced number of HTTP requests for optimized version of the page, the total page download time, which is the time interval between beginning of the first HTTP request to the completion of the last, will decrease. We also propose that the total time to render the page on the client side will decline too, as the number of embedded graphical objects will be smaller than in original page. Based on this we can conclude that from the "response time" perspective, the user-perceived quality will only increase when applying optimization methods as suggested here.

There are another dimensions to the "quality" of a web site, which are summarized in [11]:

1. Accuracy
   Who is the author of the page, what is the purpose of the content and is the person qualified to write on given subject?

2. Authority
   Who has published the document, what is the domain of the URL of the site? Is the publisher a known institution or does he have an expertise in the field?

3. Objectivity
   What were the objectives of the author? Does the author express any personal opinions?

4. Currency
   When was the page originally produced and when was it updated for the last time? Are the links provided on the page still valid?

5. Coverage
   Is complimentary information available in form of links to additional materials on the same theme? Is the information cited correctly?

All of the aspects listed above refer to the textual content of the page, which is not altered by suggested optimization techniques, and we can deduce that the quality, as defined by those aspects, is preserved.

Another important class of web page quality indicators is related to design of the page, including usability, graphic elements, aesthetics etc. Figure 10 shows the building blocks of a visual design [10]. Optimizations discussed in the current work do preserve a general layout of a web page, link and text formatting, but introduce various changes to non-textual content elements, from elimination of some of them to reduction the quality of the others. The subject of visual quality is rather subjective, so for us the guideline was to try and make the optimized version look as close as possible to the original page. Later in the work we show some examples of both acceptable and inacceptable optimizations.

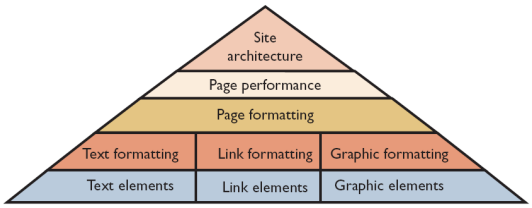


Figure 10: Web Site Structure
Text, link and graphic elements are the building blocks of a Web interface.

3.3 Images

Image files make up for about a half of total website's content size, thus becoming a prime target for optimization. In order to select the best optimization method for each image, we have first to classify all images according to their *purpose*. Based on previous works on the subject, and observation of different websites, we could identify the following classes of images [9]:

Story
Along with related textual information such image forms a central part of the web page, being an illustration for an article, or in some cases becoming the main content.

Preview
Has a similar purpose as previous item, but is used with references to a different content, e.g link to other article in same site.

Commercial
Used in advertisement or links to sponsors of the site.

Host
A photo of an author, can be used both in the main content and in links to other articles, blogs etc.

Logo
The logo of the website, usually containing publisher's trade mark.

Decoration
Such images can be used as elements of website's design, e.g. drawing a frame, serving as *bullets* for list of items, tagging areas of the page.

Text
Replaces parts of HTML, containing the same text in graphically enhanced format, is usually used for short text sequences, such as titles, menu items.

Figures 11 and 12 show examples of different image classes.

Classification Tool In order to build a database of images found on different web sites, a special tool was created, with the following functionality:

1. Open a home page of a *random* website (the algorithm is explained in section 2.5);
2. Find and highlight the first image - which can be represented in the HTML code as IMG tag, or as a background of other tags;
3. The user clicks on one of shortcut keys to select one of the classification types, which describes the current image most accurately;
4. The class of the image, as well as the following parameters, are stored in the database:
   - HTML tag
   - image source URL
   - ALT property of the image (if present)
   - image file format (GIF, PNG, BMP, JPEG)
   - number of colors (for GIF images only)
   - URL of the link associated with image (if any)
   - size of the image in the browser after rendering

Figure 11: Images Classification: www.cs.huji.ac.il
(a) Story, (b) Logo, (c) Preview, (d) Decoration

- size of the original image file
- number of times this image appears on the page

5. Same process is repeated for all images on the website, and next website is chosen.

Optimization Strategy Overall 959 images from 30 sites were analyzed. 50% of them were classified as *Decoration*, 18% as *Preview*, 11% as *Commercial*, 21% - other types, see Figure 13.

Considering these results and semantics of each image type, we can define the strategy we would like to apply on images while optimizing a website.

Decoration and *Preview* images are the best candidates to be removed from the page, as it will not affect the main content, and expected number of such images constitute about 70% of total number of images on the page.

At the same time we would like to preserve all *Navigation* images, as the functionality of the page depends on them. The same is true regarding *Text* images, although given appropriate tools, we can replace those images with a text box containing the same information, but embedded in HTML code and thus not requiring any additional HTTP requests to be

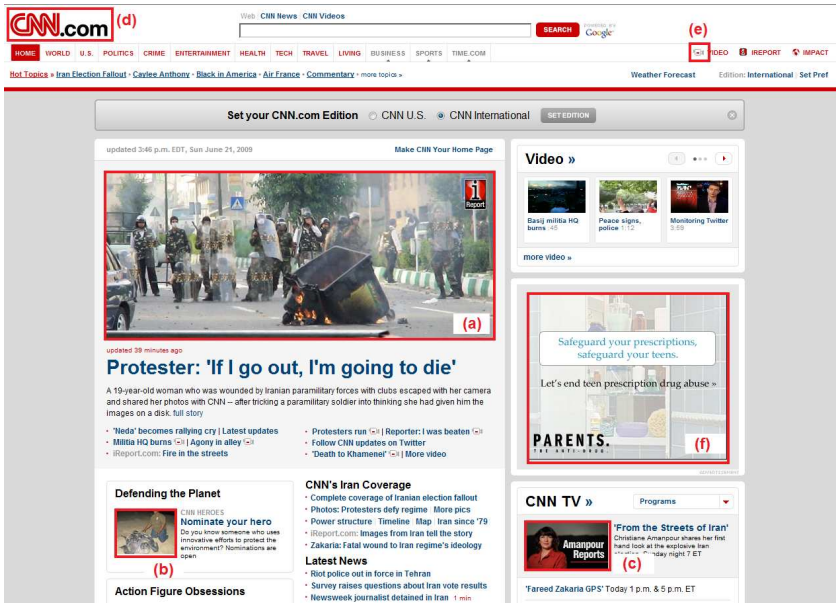


Figure 12: Images Classification: www.cnn.com
(a) Story, (b) Preview, (c) Host, (d) Logo, (e) Decoration, (f) Commercial

made.

We would also like to assure that *Logo* and *Story* images are not being removed from the page because of their semantic importance.

There's an open question regarding *Commercial* content. On one hand, those images are not an integral part of the content of the website, but on the other - profitability of the site may depend on them. We may leave this decision to the administrator of the website, having it as a configurable parameter in the process of optimization.

Classification Rules In this section we are going to define a set of rules which can be used to identify certain classes of images based on their observable properties. We will start from defining and validating some single-parameter rules, and will finally conclude a combined definition that produces the best results.

1. Most Decoration images are encoded in *GIF* format.
   Results: overall there were 523 GIF images, 312 (60%) of them are Decorations, covering 66% of all Decorations, but producing 40% of false positives.

2. Only Decoration images appear in HTML tags other than IMG.
   Results: total number of 312 such images was detected, 266 (82%) are Decorations (56% of all Decoration images).

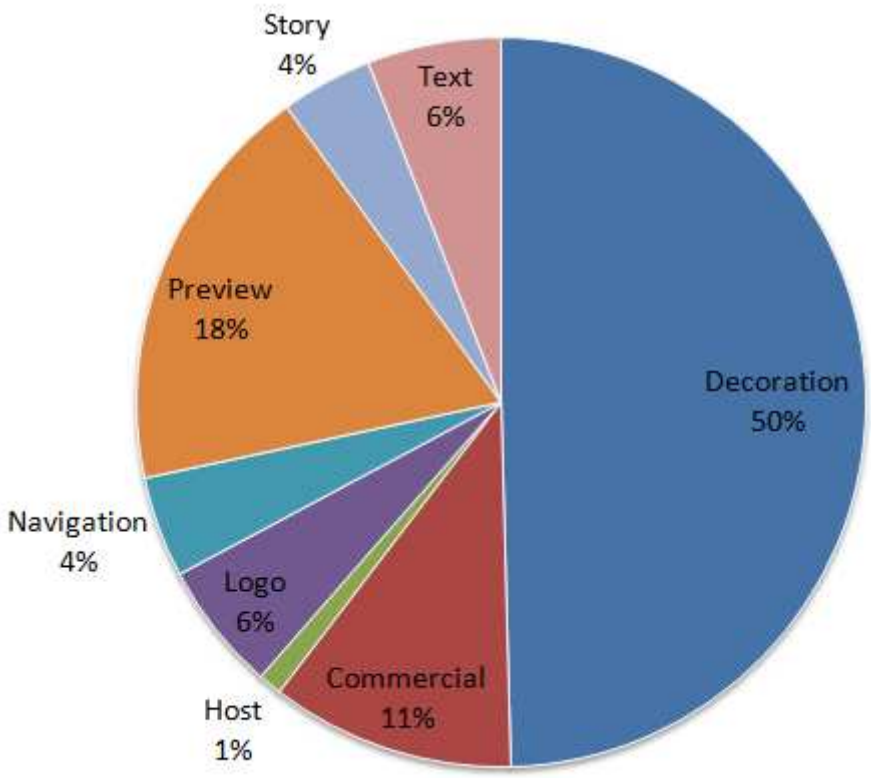


Figure 13: Image Types Distribution

3. Same Decoration image can appear more than once on same page.
   Results: 148 images had multiple copies on the same page, 139 (94%) out of them were Decorations (29% of all Decorations).

4. Tests for presence of ALT property for images, or for presence of a link associated with the image did not produce meaningful results.

The next set of rules is related to geometrical properties of images: pixel width and height of original image file and of the area covered by the image in rendered page. We will pay a special interest to aspect ratio between the width and the height, and to how this measurement changes from original file to its final manifestation on the web page. Figures 14 and 15 show the distribution of image sizes as a function of image type.

As mentioned above, another aspect of image size is the ratio by which the image is compacted or stretched on the HTML page relative to pixel size of the original file. Figure 16 shows how the size of the image changes when rendered inside HTML page, both horizontally and vertically, for different image classes. The size of the "bubble" is proportional to the number of images falling into a given bucket: the central square refers to the images whose size is the same both in original file and in the HTML page, next cell on the right includes

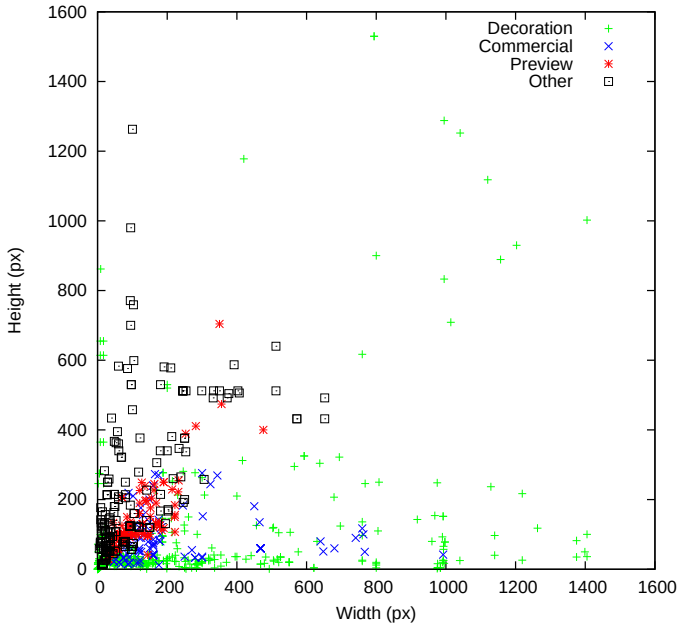


Figure 14: Image Sizes Distribution

images that were enlarged horizontally by the factor of up to 1.5, and so on. For the images falling into the main diagonal the aspect ratio between the width and the height is preserved.

As we can estimate based on the charts, most of the images that are placed in the squares far from the main diagonal are Decorations. Another noticeable feature is the large number of Preview images, which are compressed inside the HTML page, while preserving the original aspect ratio. This can be explained by the fact that in some cases the same image file is used in a full size as a Story image in one page, and as a Preview image in another. It also shows that a large number of preview images could be reduced in the size without any reduction of the quality of the hosting web page.

Based on those observations, we can make additional assumptions regarding expected geometrical properties of different image types, and validate them against the images' database.

1. Decoration images tend to be small.
   We checked the number of images having pixel size of original file (width * height) less than 100px. Results: 162 such images found, 156 (96%) are Decorations (making 33% of all Decorations).

2. We expect the value of "file aspect ratio" / "HTML aspect ratio" to be in the range of 2/3 - 3/2 for most of the images, except for Decorations, for which the variance can be higher.
   Result: found 235 images with large differences between aspect ratios in original file compared to that of rendered inside HTML page, 210 (89%) out of them are decorations

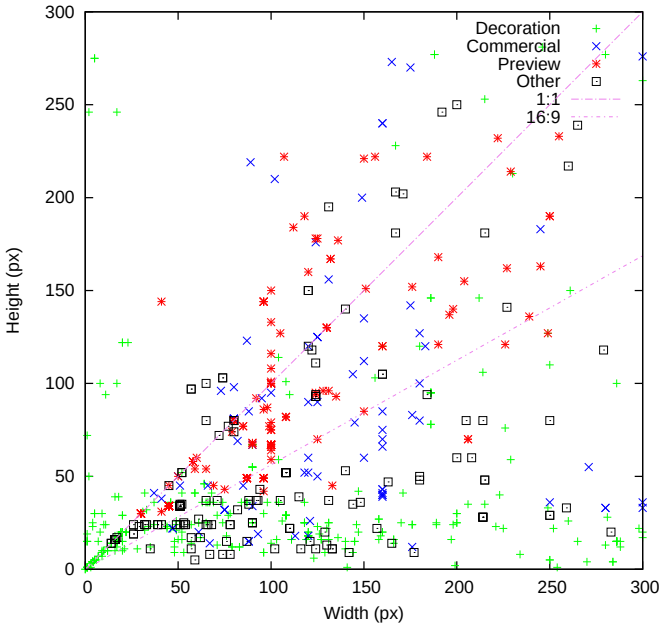


Figure 15: Image Sizes Distribution (zoom)

(44% of all Decorations). Other image types with non-standard aspect ratios included text and commercial images.

Conclusion: we can define the following rule to identify Decoration images:
Decoration can be identified by any of following characteristics:

- Appears more than once on same page;
- Pixel size of image file is less that 100px;
- The aspect ratio is changing by more than 2/3 when comparing that of original image with aspect ratio of image as it appears in HTML page.

When applying this criteria on the image set, we achieve the following results:
Total images passing the test: 322. Out of those 289 (90%) are Decorations (constituting 61% of all Decoration images), 12 are Text (21% of all Text images), 11 - Commercial (11%), 4 - Preview (2%), 3 - Navigation (7%), 2 - Logo (4%) and 1 - Story (3%).

As shown in Figure 16, Preview images can be identified by the following factors:

- Aspect ratio of the image is preserved in the web page;
- The image is compacted by a factor of 2 to 10 times.

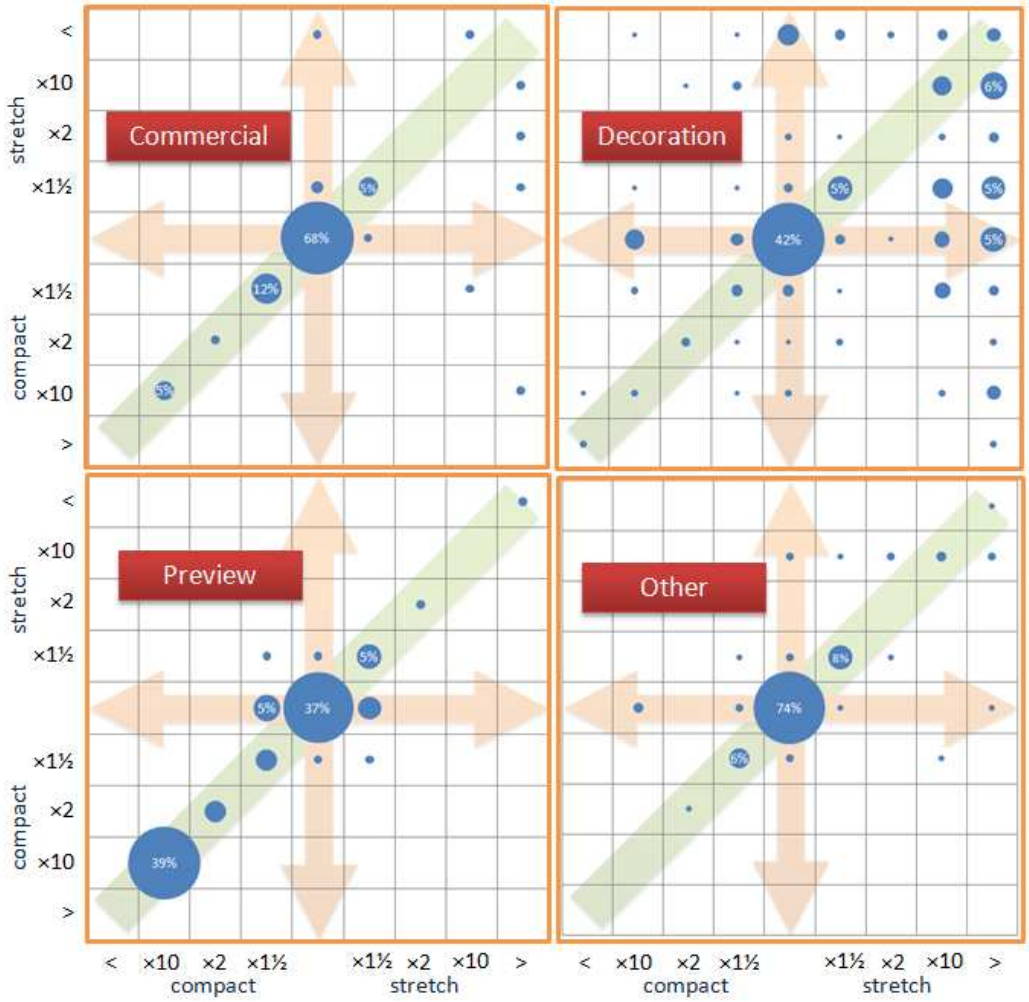


Figure 16: Image Sizes: File vs HTML

3.4 Auxiliary Content

Style Sheets Web style sheets are a form of separation of presentation and content for web design, where the style is defined in an external stylesheet file using a language such as CSS or XSL.

In order to estimate the importance of style sheets for a quality of a web site, we took a sample page, removed all style sheet files and compared to original version of the page. Please take a look at Figures 17 and 18. In this example removal of style sheets caused the whole structure of the page to be lost.

We can conclude that style sheets may play important part in defining the layout and format of a web page, and as such our optimization can't eliminate those files.

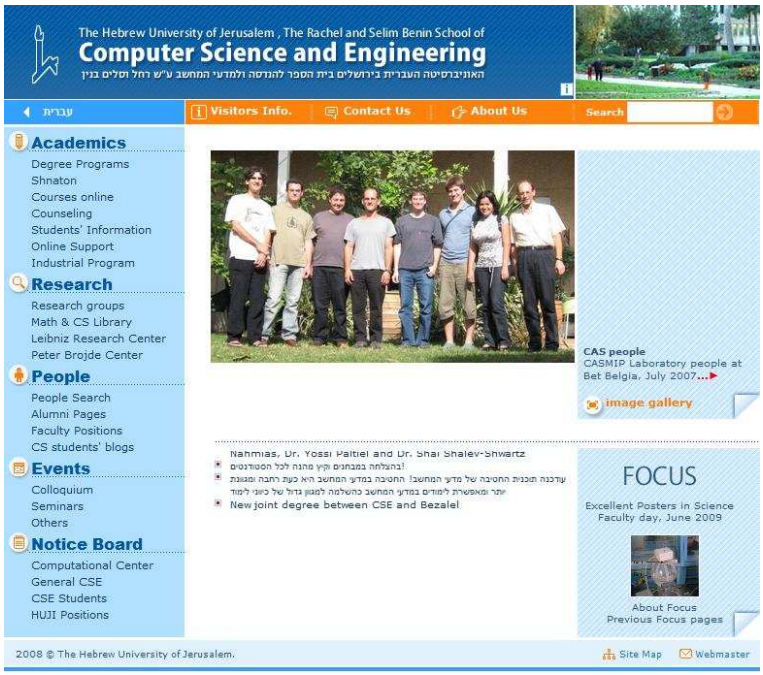


Figure 17: Original Page (www.cs.huji.ac.il)

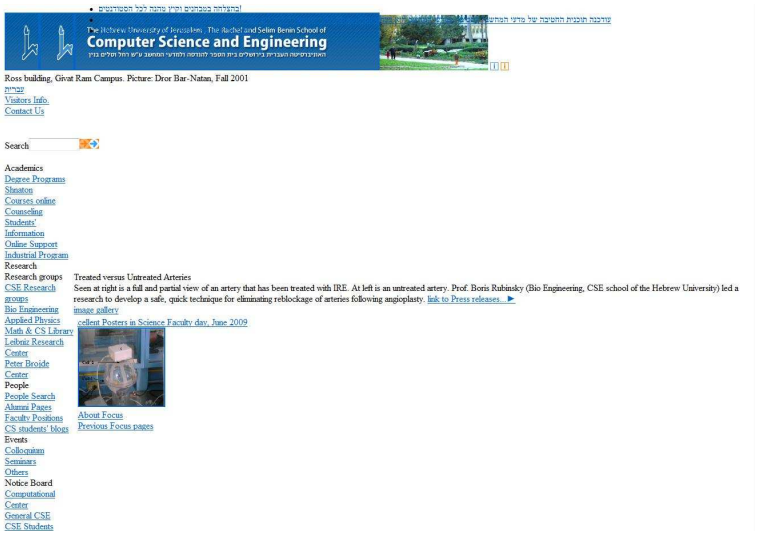


Figure 18: Without Style Sheets

Java Script JavaScript is a scripting language used to enable development of enhanced user interfaces and dynamic websites. In the context of the current work, JavaScript can be employed for making dynamic HTTP requests in response to certain client-side events, e.g. loading new image when mouse pointer is moved over graphical element. In some of these cases we can prefer to disable such JavaScript functionality in order to reduce number of

HTTP requests when running in optimized mode. However, the analysis of JavaScript code is beyond the scope of current work, and no JavaScript optimizations will be proposed.

General Web format supports embedding auxiliary content discussed above inside the HTML file itself, in addition to having a reference to a separate file. According to general optimization guidelines, as presented in Introduction section, it's recommended keeping JavaScript and CSS files as external resources. But in our case, when reducing the number of HTTP requests is the highest-priority goal, we should consider embedding all auxiliary content inside an optimized version of HTML page, even if it would enlarge the size of the file.

The principle of locality should be also taken into consideration. The main advantage (from performance perspective) of having CSS and JavaScript code as separate files is in the ability of web browsers to reuse same static files for number of pages viewed by the client, as this content is usually static and can be effectively cached. In cases of a peak load, the average number of pages in a single user's session is usually smaller than that of a regular browsing of a site, thus the rule of separation CSS and JavaScript content from HTML is less applicable in such scenario.

3.5 Textual Content

HTML Hyper Text Markup Language (HTML) is the predominant markup language for web pages [19]. The first type of HTML-related optimization is compacting HTML code, e.g. by removing comments from the text, trimming spaces etc. The second type, which is more interesting in the context of the current work, is rewriting source pages in a way that certain parts of the page are eliminated. Web pages are often created in a modular layout [7], where the content is arranged inside vertical and horizontal shapes, as shown in Figure 19. Moreover, in a typical website there are a number of predefined layouts, and each page follows one of them, e.g. locating a navigation pane on the left column of each page.

When generating an optimized version of the site, some of those layout modules can be eliminated from all relevant pages, such that all components of those areas are effectively removed and the total number of HTTP requests required to render an optimized page is reduced.

Automatic semantic classification of different areas of a web page is out of the scope of the current work. Instead, we can suggest a framework for website designers to tag areas on the web page according to their importance. It can be achieved by defining special keywords that can be added to HTML code inside comment tags, and will mark the beginning and the end of each area.

However, modern web development tools put the feasibility of such solution to question, because in many cases designers are not required to type HTML Markup manually, but it's

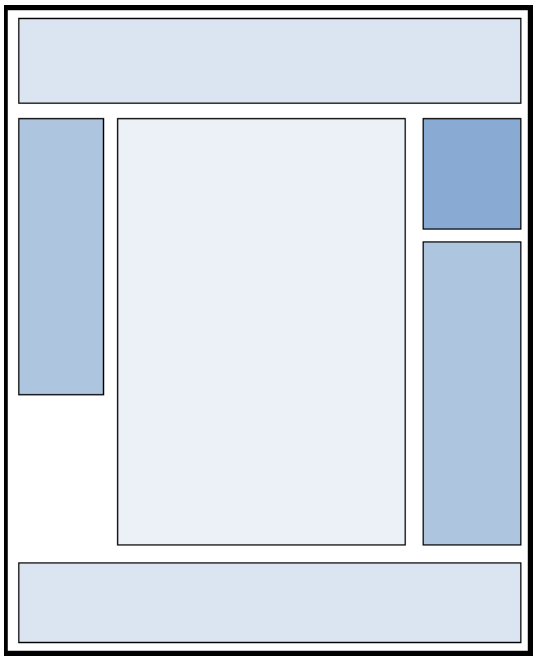


Figure 19: Blueprint of a Modular Layout

rather generated automatically by a visual tool. On the other hand, when the layout is defined in a centralized HTML file, and the content reside on different HTML pages which are loaded as frames in the main page, it would be much easier to control the presence of whole frames in the web page.

In addition, for the cases where the web site is not designed with a use of frames, an automated tool can base its optimization on eliminating certain tags (with all internal structure) based on tag name and ID. For example, in case of `www.top500.org` site we can configure the tool to hide DIV tags having "sidebanners" as an ID, and the whole right-side pane containing advertisement images will be removed in all pages. The structure of `www.cnn.com` site is too controlled by DIV tags, having meaningful IDs and class names, making it possible to define appropriate optimization rules. The advantage of this method, compared to insertion of special marks into HTML file, is in its non-intrusiveness - there's no need to modify original pages or to introduce any changes to a regular workflow of updates to the content of the site.

3.6 Summary

The following list summarizes the major potential areas for optimization:

- Elimination of Decoration images
  Decorations constitute roughly half of all images in a page, and can be identified with a high accuracy by automatic tool.

- Resizing large images
  Large image files can be compressed to create a lower quality version of reduced file size.

- Consolidating textual content
  It's possible to embed JavaScript files and Style-Sheets into HTML code, thus reducing the total number of components of a page.

- Hiding whole blocks in page layout
  Based on relatively simple manual configuration, whole blocks can be removed from original pages, including all contained resources.

4 Operation Modes

Conceptually the web site optimization tool proposed in this work operates in two modes: offline preparation of modified versions of original static files and online monitoring of web server's "health" and performing the switch between normal and optimized versions of the content when required.

Rules for web site optimizations were generally developed in the previous section, this section is about to describe what are the indicators for performance of the web server and how the transition between predefined operation modes is performed.

4.1 Performance Indicators

In this section we discuss different types of performance monitors which should trigger the transition to optimized content mode, and back to normal operation.

Recalling that the original purpose of this work is serving as many users as possible, one approach can be measuring parameters like "Number of HTTP request" per "Time unit", server response times etc directly. The main complexity of a such solution comes from the fact that there's no immediate way to define thresholds on those values. For number of requests this will require the knowledge of the maximal number of requests the server can handle concurrently. This data is usually unavailable. Moreover, HTTP requests may vary in processing resources on server side. Neither can we use the average response time values, as we expect the peak load traffic to show high locality of requests, leading to large differences of average response times during the normal operation mode and the peak load. Same is true for response time metrics.

The second approach would be monitoring system resources, especially utilization of hardware components, making an assumption that degradation in response times is always caused by overloading on some of the resources. For each hardware platform, web server software and web site properties there can be a different set of resources being a bottleneck. The total capacity of each hardware resource is known, so it's always possible to define thresholds in terms of percentage of utilization of each resource, CPU time being the most obvious example.

Monitoring hardware resources has an additional advantage, and it is the fact that we can choose an optimization algorithm which reduces the use of the specific hardware resource being currently overloaded. As shown in section 3.1, some methods can make the highest effect on CPU time consumption, others on required network bandwidth etc.

Most web servers make their internal performance indicators and *health* status available for administrators and for external tools. One such example is utilization of worker threads, which is the percentage of threads being busy with processing some user's request at a given

moment, out of total number of such threads configured on the server. This information can be also used as a trigger for optimized mode.

4.2 Operation Modes

When the system detects the need to boost the performance of the web server, as described in the previous section, it should configure the server to serve optimized content of lower quality.

There can be different approaches to solution for various web server types, and we don't expect a single method to be the best fit for all vendors. For current project the server chosen as a target platform is Apache HTTP Server 2.2, running on Linux operating system.

In this section we assume that an optimized version of static web content was created in advance. Moreover, we expect the location of optimized files to be as following:

- A separate folder F will be created to contain the optimized data;
- Optimized versions of the content will be located in same path relative to F as the original file was relative to the root folder of the web site;
- Even static data can change over time - new files are being added, existing files are getting deleted or modified. There will be a certain delay between the moment the original data was altered, and the moment the system detects the change and modifies optimized files' tree accordingly. Such discrepancies should be considered and handled by proposed solution.

When an HTTP request is processed by a web server, the URL is resolved to a filename, and, in case of static files, the content of this file is sent back to client in response. In simple configuration the mapping is defined by the following rules:

- The web site is associated with a certain folder in local file system, referred by the term "web root";
- The "file" portion of the URL is interpreted as a relative path inside the web root folder;

In addition to this basic functionality, advanced URL-to-file mapping instructions can be configured for *mod_rewrite* rule-based rewriting engine, as described in the module's documentation [4]. We can add rules that will redirect all traffic to the web site to optimized files' folder, in case that the following conditions are met:

1. The web server is currently running in optimized mode. As a flag we can use certain environment variable (global or internal for Apache server), or can check for existence of special file in known location; both methods are supported by the rules engine (documented at [5]).

2. The optimized version of requested file already exists, and is up to date. As mentioned above, there's certain amount of time it takes for changes in web site files to propagate to the optimized version, and in case that the original file was modified after creation of optimized version, we'll have to serve the original file to the client.

One of the advantages of the proposed method of content redirection is the fact that the configuration files of the web server are modified only once, when the system is integrated. After this initial change the server continues operating in its regular mode, and all further activations of optimized content are triggered by external components and are made effective immediately.

Performance Validation There was a need to validate that the overhead of rewrite rules is negligible, and we can use the method described above safely. The test was designed in the following way:

1. A local copy of web site `www.adagio.com` (online tea shop) was created using HTTrack [6] tool. The tool was configured with all default settings with addition of the following parameters:
   - max search depth was limited to 3 clicks;
   - forcing all images to be copied too.
2. Stored copy of the web site was configured as local site on Apache web server.
3. In order to emulate simultaneous traffic from multiple users, series of HTTP requests were recorded as LoadRuner [8] script.
4. An "optimized" version was created by adding a symbolic link to the web site root folder, this way all the files remained the same both in regular and optimized versions.
5. Based on recorded LoadRunner script a load was generated on Apache web servers, having half of *Virtual Users* connecting to original web site, and the second half - to optimized version, using mod_rewrite rules, as explained above. Average time to execute all requests in single instance of the script was measured.

Please see histogram at Figure 20. It shows that average response times of the server are the same for both normal version of the web site, and the one having mod_rewrite rules activated.

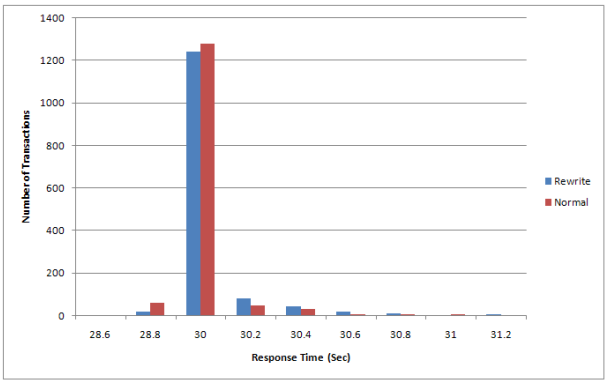


Figure 20: Performance Overhead of mod_rewrite Rules

5 Experimental Validation

In this part of the work we are going to validate the methods proposed in the previous sections on a real environment, and measure the effects of proposed optimization methods.

5.1 Environment

Current section describes in detail the environment that was used for all performance tests.

Web Server We used Apache Web Server, version 2.2, running on Linux OS - Fedora, release 11 (Kernel 2.6.30.8). Hardware components:

- Model: Dell OptiPlex GX260
- Memory: 500 MB
- Processor: Intel Pentium 4, 2.4GHz

Apache Web Server is running with all default configuration parameters, except for support for Keep-Alive connections, which was turned on.

LoadRunner The load on the web server was generated by HP LoadRunner tool [8], version 9.50

The "Controller" is the central component of LoadRunner, which orchestrates the generation of the load by "Load Generators" and collects statistical data which is finally analyzed with "Analysis" tool.

Controller was installed on Microsoft Windows Vista OS (Enterprise edition, version 6.0.6001). Hardware components:

- Model: HP Compaq 8510w
- Memory: 4 GB (3 GB accessible to OS)
- Processor: Intel Core 2 Duo CPU T7500, 2.20 GHz, 2 cores

The Controller machine was also used as one of the Load Generators.

In addition to Controller there were three Load Generators:

- two identical to the web server, with Microsoft Windows Server 2008 OS (Standard edition, version 6.0.6001).
- additional machine with Intel Pentium 4 Processor, 3.20 Ghz with 2 GB RAM, running Microsoft Windows Vista OS (Ultimate edition, version 6.0.6000).

Network All computers are connected on 100 Mb/sec Ethernet LAN via Linksys WRT54G router.

5.2 Web Site

Load tests were performed on local copy of site `http://www.top500.org/`. The site was captured using HTTrack tool [6], with all default parameters, except for:

- The depth of the search (when following links) was limited to 3.
- The instructions in "robots.txt" file were set to be ignored.
  This file is a standard way for a web site to instruct automatic tools, mostly search engines, to avoid accessing certain content, but was not relevant for our use.

In all pages there was a JavaScript code which was intended to randomize the order of appearance of advertisement images, by injecting external images in static HTML content. Those functions were removed from all pages, and replaced with static images of the same type.

LoadRunner Script All performance tests described below are based on LoadRunner scripts, which define a sequence of HTTP requests to be submitted to the server. In the common use case of LoadRunner the script is automatically created ("recorded") based on the pages opened by the user in web browser during a recording session. The type of the script which is being used is HTTP-based, which means that each HTTP request appears as a command in the script (as opposed to recording HTML pages only, and having LoadRunner request embedded objects automatically). The main advantage of HTTP-based scripts is better performance of Load Generators, as while replaying the script, there's no need to parse HTML content of responses.

Other script parameters include:

- Simulation of browser's cache
  If multiple HTTP requests to same URL appear in the script, that request would be downloaded only once, and later it will be assumed that the file is already in client's cache.
- Connection management
  To emulate real browser's behavior, two connections to the server will be opened by each user, and HTTP requests for components of same HTML page will be downloaded simultaneously on these connections.
- Transaction status
  Whole sequence of HTTP instructions (required to view a single HTML page or number of pages) is defined as "Transaction". Transaction response time is measured and

reported by LoadRunner. The transaction may end with either Success of Failure status. A failure can be caused by any error, such as new connection being refused by the server, timeout while waiting for server's response etc.

When the script is being used for stress testing the server, a certain number of "Virtual Users" is started, and each user repeats the same set of instructions multiple times. Additional parameters are defined per scenario, such as the "pacing" between iterations (which defines in what time interval should a virtual user start a new iteration after finishing the previous one). The actual configuration of each scenario is described for each test case.

5.3 Performance Tests

Optimized Script In order to validate optimization methods proposed by this work there's a need to create an optimized version of the web site, which differs from original version by the following:

- Some images are removed from the HTML file
- Large images are compressed
- Blocks from HTML files are deleted (eliminating all embedded objects in those blocks)

Alternatively, we have created a modified version of LoadRunner script, where requests of all objects targeted to be removed from HTML file are deleted from the script manually. This way the script is downloading same files as would a browser when directed to optimized version of the site, although the content of those files is not changed or compressed.

As result the difference between original and optimized scripts are:

- Number of HTTP requests: In original script - 81, in optimized - 43 (reduced by 47%)
- Total size of downloaded files: In original script - 702KB, in optimized - 570KB (reduced by 22%)

In addition to requested URLs, Load Runner does also record *Think Time* steps, which represent the time it took a person to read the page before clicking on the next link. Normally a Think time is recorded between web pages, and does not separate components of the same page, as the latter are downloaded automatically by the browser. When creating an optimized version of the script, we've left the Think time steps intact in their original locations, assuming that it takes approximately the same time to read a stripped version of the page, as it takes for original content.

Scenario Properties Performance tests are organized by LoadRunner in *scenarios*, which define the number of active virtual users (vusers) at each point of time, and run-time settings for each vuser.

The vusers start running in groups, for each such group we define a start time (relative to the beginning of the test), number of users in group and the ramp-up rate, in number of users per time unit.

The Run-time settings define the following parameters:

- *Think Time*
  The Think time recorded in the script reflects the time intervals between subsequent pages' requests. When replaying the script, we can choose whether to ignore the Think time (in which case a page starts downloading as soon as previous page was fully retrieved), or to replay the Think time - as recorded, or using random values in certain range of percents of the original times.

- *Pacing*
  This parameter defines at which rate each vuser starts a new transaction, and the options are:

  i As soon as previous transaction ends;

  ii After certain amount of time after the end of the previous transaction, with option to give a range of possible values, and the actual time is selected randomly from that range;

  iii It constant time intervals, regardless of duration of previous transaction (given that it's lower that the defined interval). In this case too there's an option to randomize the time intervals in specified range.

Default Apache Settings The first test was conducted on the web server with all default settings, as it was first installed. The parameters of the loads generated by Load Runner were as following:

Groups of virtual users (Figure 21):

- At the beginning of the test, 200 users, ramp-up of 1 user every 2 seconds;
- At 0:30 hours, 200 users, ramp-up of 1 user every 5 seconds;
- At 1:00 hours, 200 users, 1 user every 12 seconds.

Run-time Settings:

- Think Time: ignored;

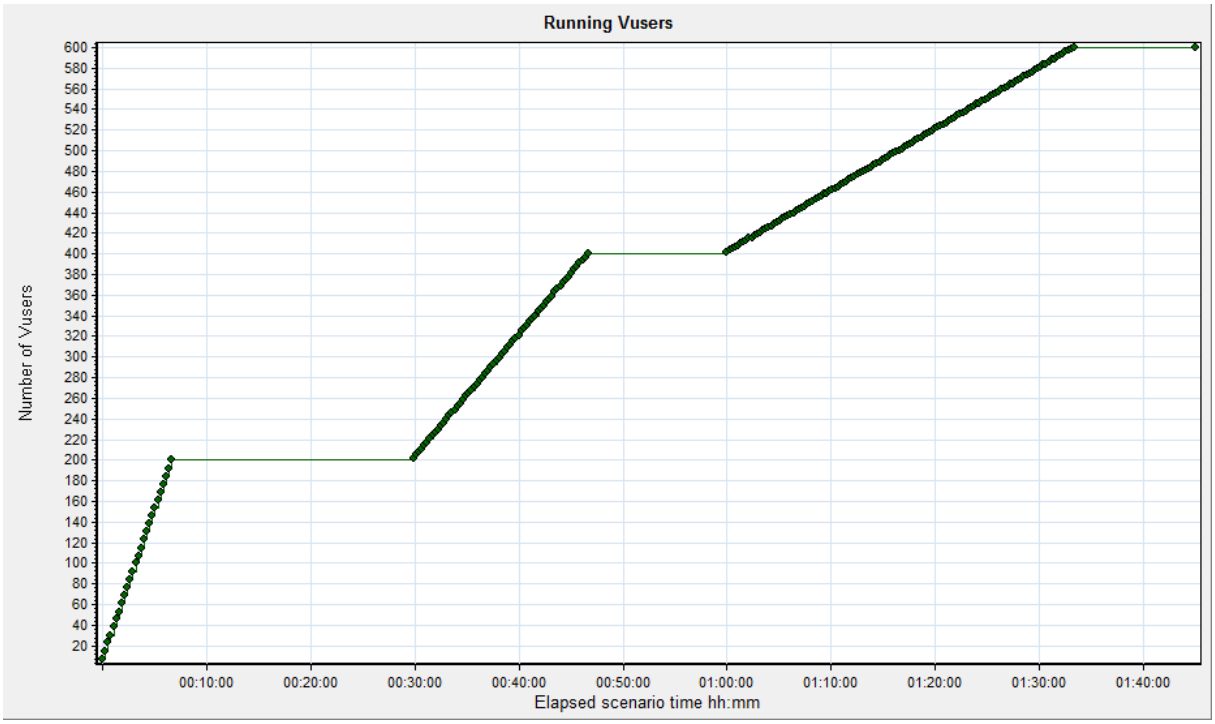


Figure 21: Number of Running Users
Default Apache Settings test: number of running virtual users.

- Pacing: predefined intervals between transactions, random between 15 and 25 seconds.

Results

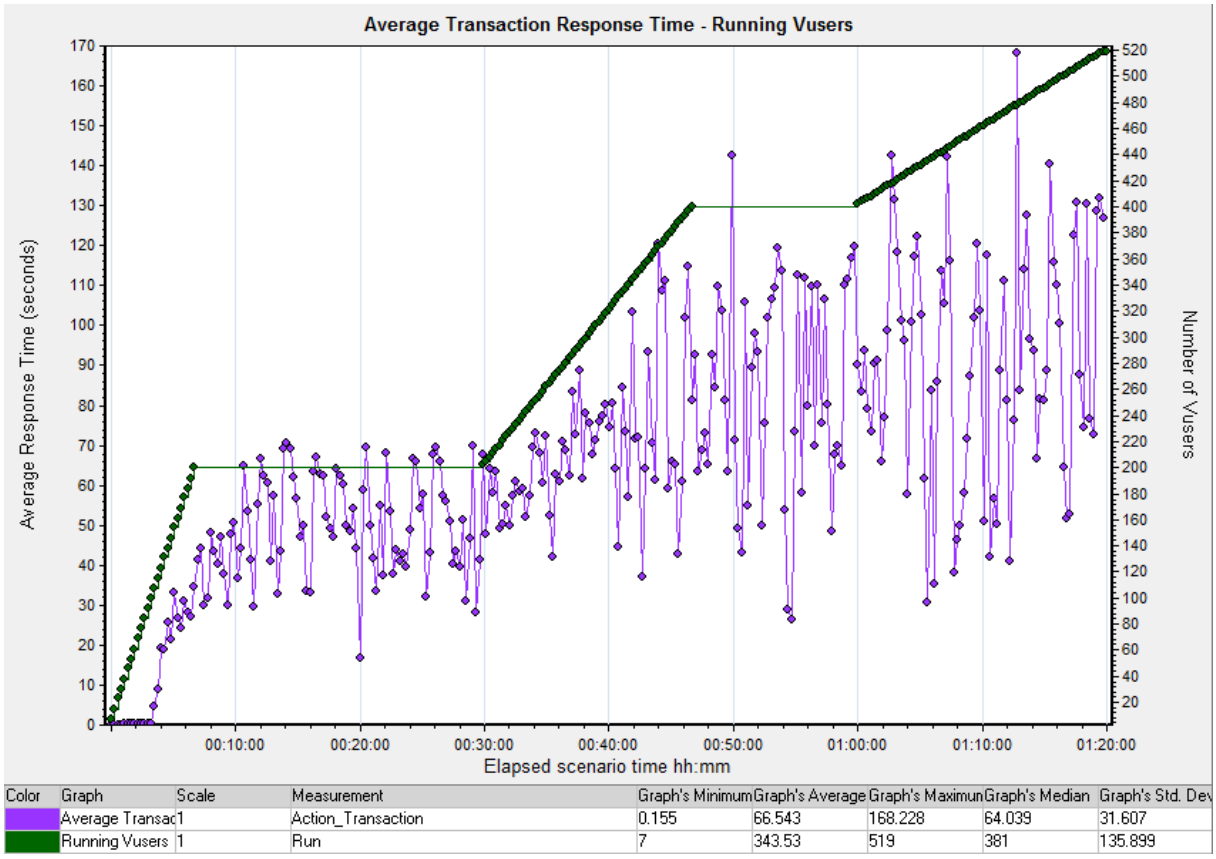
The results below are based on first 1:20 hours of the scenario. The average Transaction Response Time (excluding the Think time) is 66.5 seconds for regular script, and 31.9 seconds for optimized version (Figures 22 and 23). 12,709 transactions were successfully completed by vusers executing the regular script, compared to 27,530 by those executing optimized version of the script. The number of transactions completed with errors was 4,510 and 7,955 respectively.

These results show a 52% reduction in average transaction response time, and 106% increase in total number of completed transactions during same time interval.

Discussion

As stated above, the test was performed on default Apache web server configuration, and, as it appears, the default configuration has some of optimization features turned off:

- Keep-Alive Connections
  By default the server does not enable Keep-Alive feature, as result a new TCP connection is opened for each HTTP request. With such configuration reduction in number of HTTP requests has even larger effect than in case of long-living connections. This



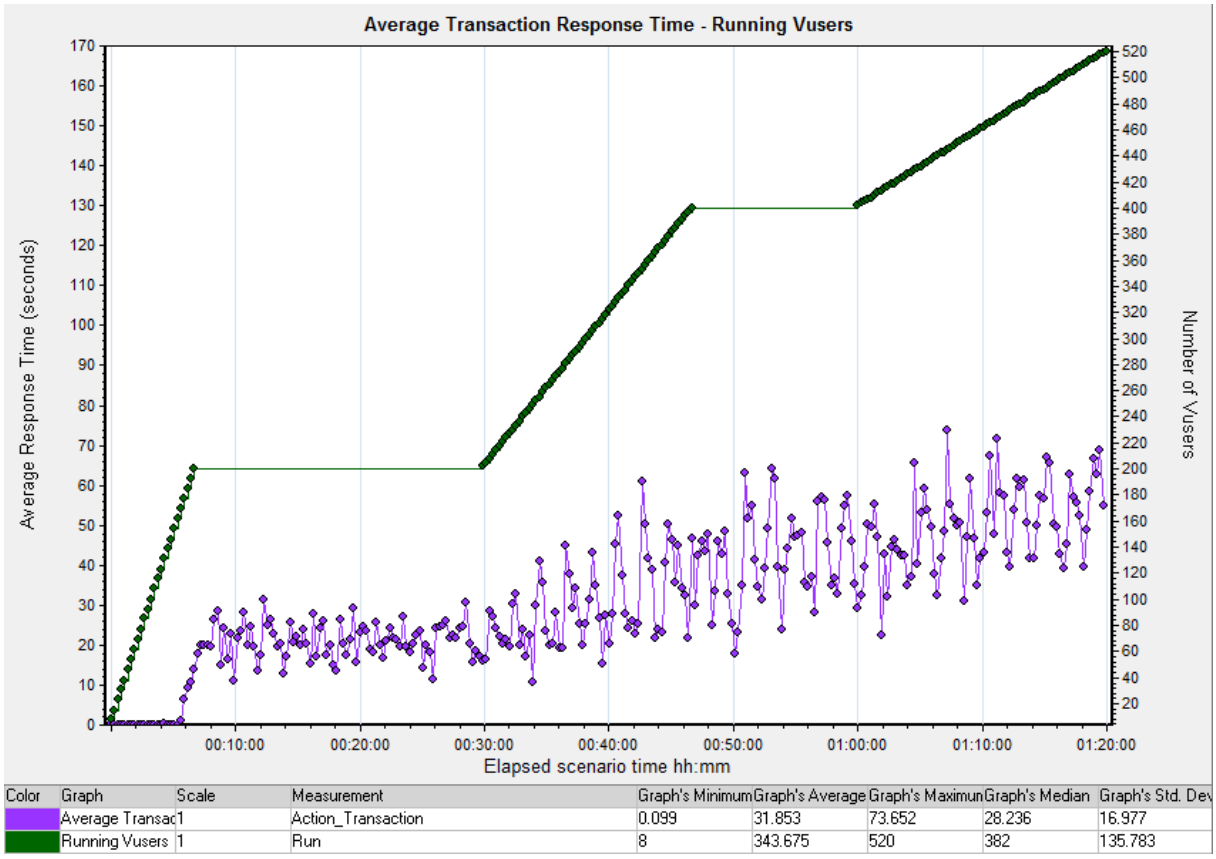
Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev
	Average Transac	1	Action_Transaction	0.155	66.543	168.228	64.039	31.607
	Running Vusers	1	Run	7	343.53	519	381	135.899

Figure 22: Average Transaction Response Time (regular)
Default Apache Settings test: average transaction response time for regular script, correlated to number of running users.

can partially explain the large difference between regular and optimized versions of the script, which exceeds the expected results for number of completed transactions.

- Caching
  In-memory caching is also disabled in default configuration, as result the server reads the content from the disk on each HTTP request. This manifests both in longer response times and in increased locking between working threads of the server, while these compete for disk access.

- Working Processes
  Apache allows configuring maximal numbers of server processes, and maximal number of worker threads per process. In addition the configuration defines the maximal allowed number of idle threads, before some of them are closed. All those properties have low values in default configuration (as described in detail in configuration of the following tests), in result users may have to wait for one of the threads to become idle, or may force creation of new process or thread, which is a costly operation.



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev
	Average Transac	1	Action_Transaction	0.099	31.853	73.652	28.236	16.977
	Running Vusers	1	Run	8	343.675	520	382	135.783

Figure 23: Average Transaction Response Time (optimized)
Default Apache Settings test: average transaction response time for optimized script, correlated to number of running users.

In the following tests we will modify the configuration of Apache server and continue comparing original and modified versions of the script on more optimized environments.

Optimized Apache Settings In this test we've used the same regular and optimized scripts, but have performed the following configuration changes to Apache web server:

- Keep-Alive Connections
  The feature was enabled by adding configuration line "KeepAlive On"

- Caching
  In-memory caching was enabled by uncommenting section for "mod_mem_cache" module, while leaving all default values.

- Working Processes
  Number of processes allowed to be started was increased to 1500 using "MaxClients" directive.

Each "virtual user" thread in the Load Generators was scheduled to execute the script once in every 20 seconds (on average, randomized in the range of [15, 25] seconds), but in case that certain transaction does not finish within that time, the next transaction is delayed. Thus we can see the load being generated as an "open system" while the response times are low, and as a "closed system" once response times start overflowing the configured time intervals between subsequent transactions.

Results

The average transaction response times, as function of number of running Vusers (script execution threads) is shown in Figures 24 and 25. During the first 35-40 minutes both versions of the script were executing at stable transaction times, the regular version running at 270 msec per transaction, while the optimized - 138 msec, which shows a reduction of 49% in average response time.

Starting from certain point of time, which is different for regular and optimized versions, we see a sharp change and the average response time begins growing rapidly, and stabilizes (at much higher value) only when the load on the system becomes constant. We assume that such behavior pattern is caused by exhaust of some of system resources, such as number of connections or processes currently defined for a web server.

Figures 26 and 27 show the number of completed transactions per second, as a function of number of active vusers' threads. The regular script was completing 16 transactions/second at the peak, and addition of more virtual users could not increase the throughput, while optimized version of the script reached 20 transactions/second.

Discussion

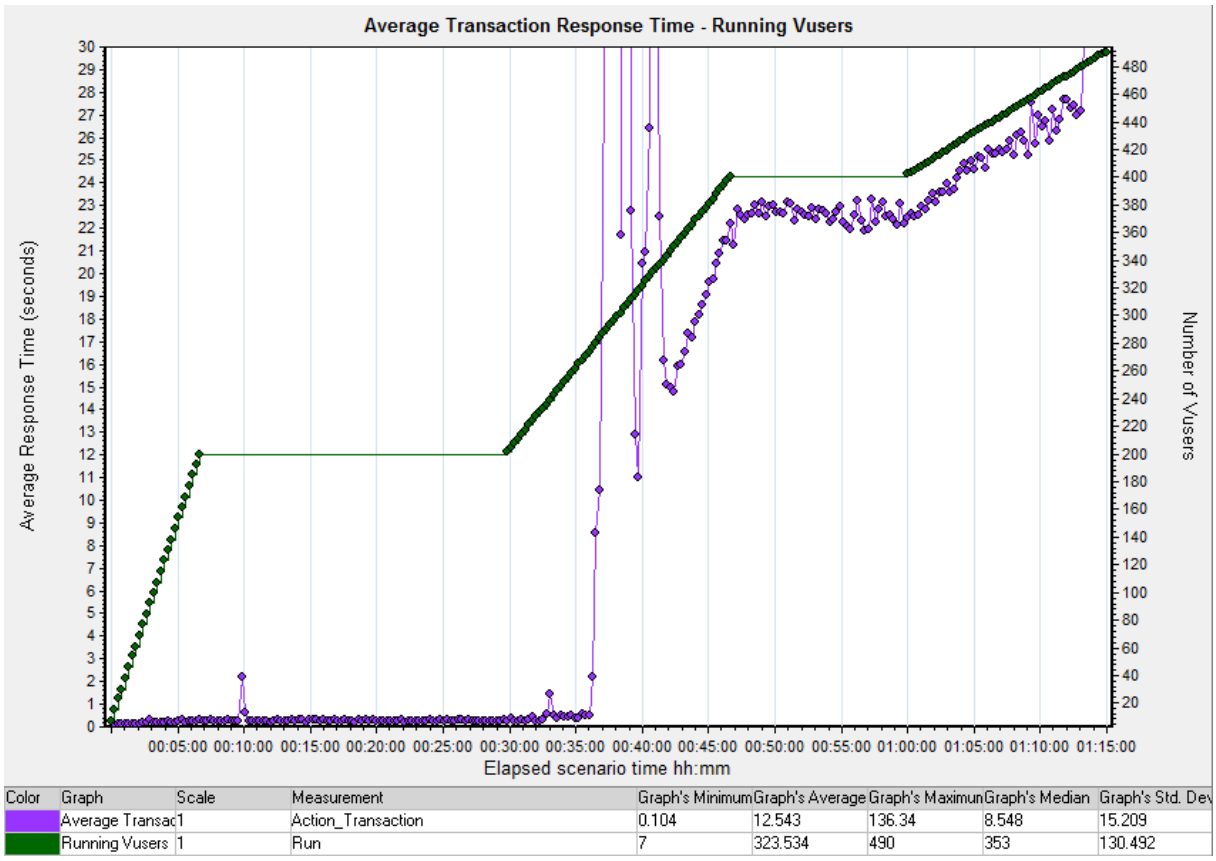
These results support the proposition that the number of HTTP requests is an important factor for utilization of the web server. In the optimized version of the script the number of HTTP requests is reduced by 47%, while the total size of downloaded content is reduced by 22% only, and still the optimized script shows improvement of response times by almost a factor of two.

As described above, current test had a property of converging to a "closed system" once the load on the server was high enough. In the next test we are going to overcome this limitation and define a more realistic scenario, in which the number of new users trying to access the site does not depend on current performance of the server.

5.4 Slashdot Effect Simulation

While in the previous section we described performance tests aimed to compare the behavior of the server on regular and optimized content, current section describes a series of tests which put as a goal to conduct a simulation of slashdot effect in a way that:

1. We create an "open system", where new users are arriving to the web site at certain rate, regardless the current performance of the server;



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev
	Average Transac	1	Action_Transaction	0.104	12.543	136.34	8.548	15.209
	Running Vusers	1	Run	7	323.534	490	353	130.492

Figure 24: Average Transaction Response Time (regular)
Enhanced Apache Settings test: average transaction response time for regular script, correlated to number of running users.

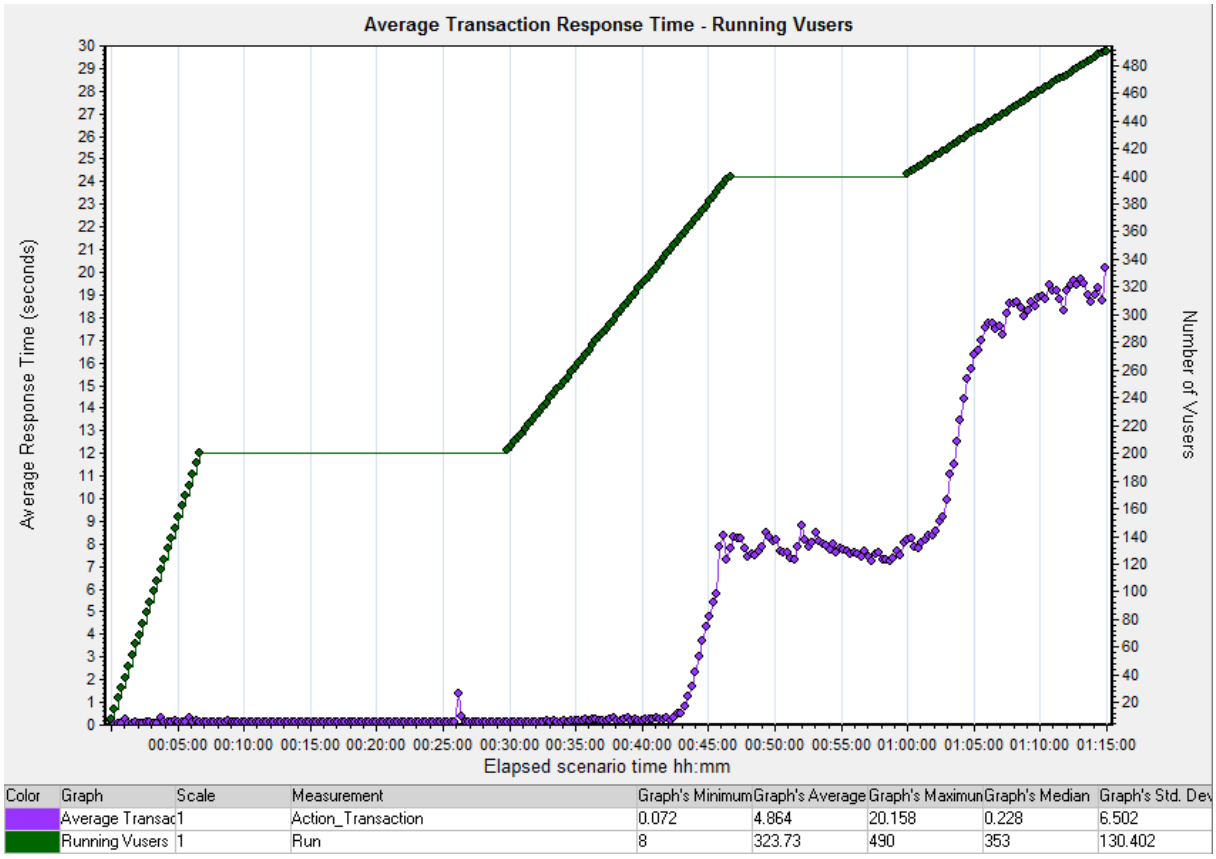
2. Test the system assuming "extreme" optimization scheme, which would be more aggressive in eliminating components of a page than one presented in the previous section.

Transaction Types The simulation is going to be repeated using three versions of LoadRunner scripts:

- Regular
- Moderate optimization
- Extreme optimization

Regular

A slashdot effect, in its usual form, is caused by a link to the web site being posted on another, much more popular, site, such that a large number of readers of the popular site follow that link. The workload generated in this use case is expected to show high localization property. In particular, we expect a large volume of users to be requesting a single page



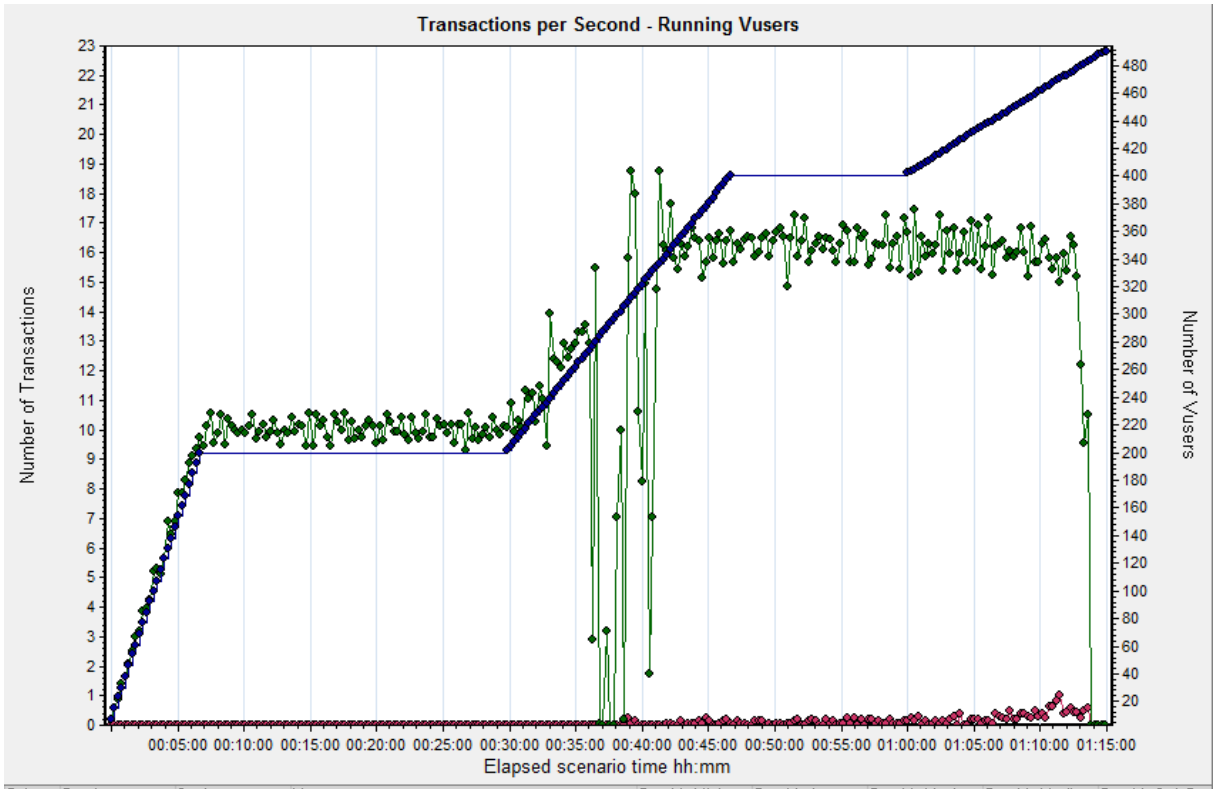
Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev
	Average Transac	1	Action_Transaction	0.072	4.864	20.158	0.228	6.502
	Running Vusers	1	Run	8	323.73	490	353	130.402

Figure 25: Average Transaction Response Time (optimized)
Enhanced Apache Settings test: average transaction response time for optimized script, correlated to number of running users.

from the site (including all its embedded components), and not necessary the home page, but rather one of internal pages being pointed by the link.

For this purpose the LoadRunner script records a request to a single article on a local copy of `www.top500.org` site. Overall the script opens 2 TCP connections, makes 59 HTTP requests and downloads total of 322 KB of content.

Moderate optimization
An optimized version of the script was based on the regular script, and was created by eliminating request to decoration images and advertisements. In addition, two images (the logo and a "story" image) were compressed by reducing the quality of the image in JPEG lossy format. As result the script was opening 2 TCP connections, issuing 29 HTTP requests, and the total size of the content was 188 KB.



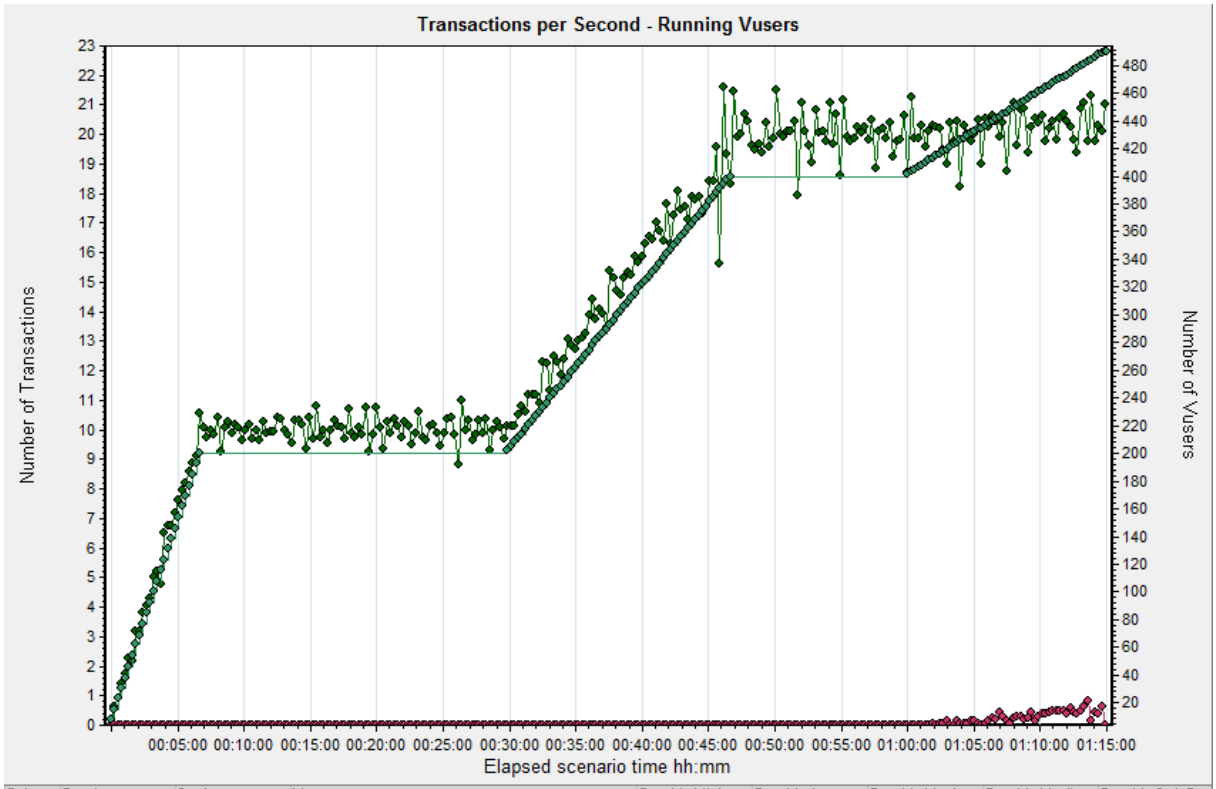
Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev
	Transactions per	1	Action_Transaction:Fail	0	0.073	1	0	0.143
	Transactions per	1	Action_Transaction:Pass	0	11.996	18.75	11.25	4.59
	Running Vusers	1	Run	7	323.534	490	353	130.492

Figure 26: Completed Transactions Rate (regular)
Enhanced Apache Settings test: number of transactions per second for regular script.

Extreme optimization

In the last version of the script we have created the most optimized version of the script, by applying the following techniques:

- Removed all images except for:
  - Logo (compressed from 30 KB GIF to 3 KB JPEG);
  - "Story" image (JPEG, compressed from 7.4 KB to 2.4 KB).
- Removed all JavaScript references.
- Removed all references to stylesheet files, but copied the content of some of CSS files to the body of HTML file, in order to preserve the basic formatting of the page. As result the size of HTML page grew from 37.9 KB to 55.2 KB.
- The script was modified so it makes all requests on single TCP connection. It's possible to make the browser reuse the same connection for all requests by injecting a JavaScript



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev
	Transactions per	1	Action_Transaction:Fail	0	0.047	0.813	0	0.131
	Transactions per	1	Action_Transaction:Pass	0.188	14.422	21.625	14.563	5.38
	Running Vusers	1	Run	8	323.73	490	353	130.402

Figure 27: Completed Transactions Rate (optimized)
Enhanced Apache Settings test: number of transactions per second for optimized script.

code which will request all relevant images synchronously, after having the HTML page completely downloaded.

See Figures 28 and 29 for visual comparison of regular and extreme optimized versions of the page.

LoadRunner Scenario In order to simulate an "open system" with LoadRunner tool, following configuration was required:

- Connection establishment time was limited to 10 seconds;
- Download time for each HTTP request was limited to 12 seconds;
- Overall transaction response time was limited to 25 seconds. In the case that the transaction could not be completed within this time interval, it was aborted and marked as "failed". This behavior matches a real-life case of a user waiting for a web page, and aborting the download it it takes too long;

Figure 28: Article from www.top500.org - original
Original version of article from `www.top500.org` site.

- Each virtual user's thread was configured to start a new transaction every 40 seconds (randomized in the range [37, 43] seconds). Considering the limit on total transaction time mentioned above we are assured that each new iteration starts within the requested interval, and is not affected by server's response times;
- Number of such user threads was incremented in steps of 500-1000 users followed by a constant load for three minutes. At the peak the number of threads reached 5,000;
- At the peak load 5,000 user threads were generating a load of 125 transaction attempts per second (each thread initiating a transaction once in 40 seconds).

Results Figures 30, 31 and 32 show the success rate of attempted transactions for regular, optimized and extremely optimized scripts accordingly. Note: in "extremely optimized" graph there's a small number of failed transactions, even at low load on the system, and those are, most probably, results of client errors (in LoadRunner software or the hosting

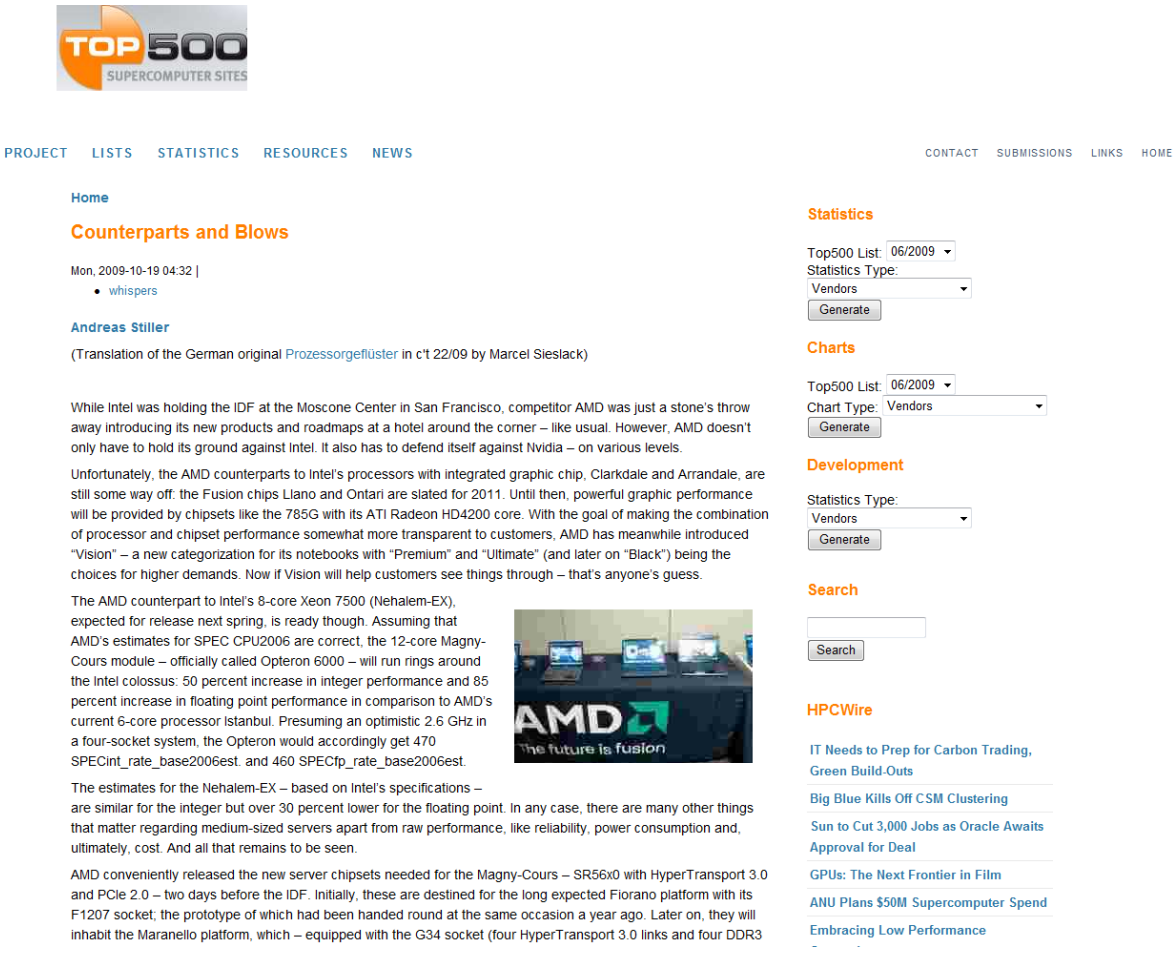


Figure 29: Article from www.top500.org - optimized
Optimized version of article from `www.top500.org` site.

operating system), as all errors occurred on same Load Generator machine, but not on the others.

Figure 33 summarizes the data for success rate of transactions and provides a comparison between different versions of the script. We can make the following observations:

1. The regular version of the site successfully serves a load up to 23 transactions per second, optimized version reaches 35 transactions per second, and extremely optimized - more than 100 transactions per second.

2. Starting from 45 transactions per second the regular version shows better success rate than the optimized one. We assume that it is caused by the settings of connection establishment timeout (10 seconds), which makes an effect of "admission control", allowing fewer simultaneous active transactions in a regular scenario compared to optimized version. And for both versions of the site the success rate is lower than 50%

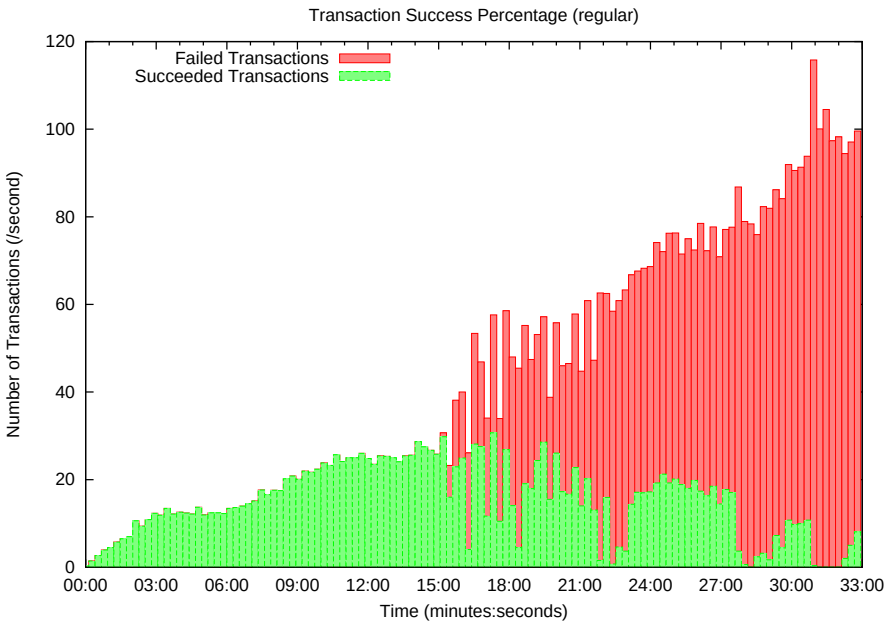


Figure 30: Transaction Success Rate (regular script)

at such workload.

Figure 34 shows the average transaction response times (of succeeded transactions), comparing different versions of the script (Figure 35 shows same information, zooming on low response times). While the load on the server is relatively low, the average response times are:

- Regular script: 145 msec.;
- Optimized script: 46 msec. (reduced by 68%);
- Extremely optimized: 13 msec. (reduced by 91%).

Regular script reaches a value of 1 second at load of 25 transactions per second, optimized - at 37 transactions per second, while extremely optimized - at load of 110 transactions per second.

5.5 Switching Between Modes

The goal of this test is to validate the automatic mechanism for switching the web server between normal and optimized modes.

Performance Indicators The algorithm which controls the running mode of the server is based on the following performance counters:

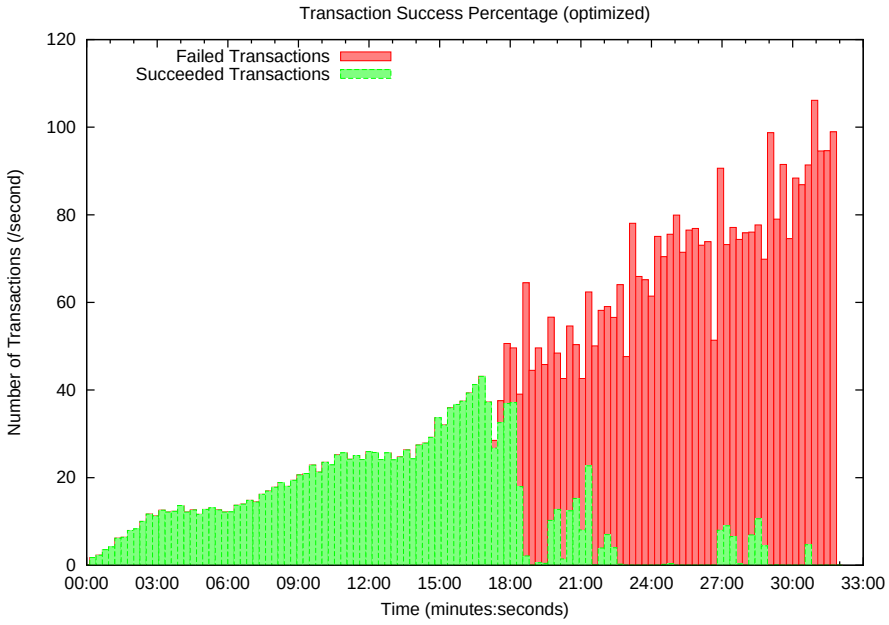


Figure 31: Transaction Success Rate (optimized script)

1. CPU Utilization;
2. Number of incoming TCP connections per second;
3. Status of running processes of Apache Web Server.

The CPU utilization was collected using "sar" utility, available as an optional package for Linux distributions. It collects certain cumulative activity counters from the operating system, and performs required calculations in order to generate the requested data. Command "*sar -u 5 1*" is used to show the average CPU utilization during the last five seconds.

The total accumulated number of incoming TCP connections can be found in "*/proc/net/snmp*" virtual file, which is available in a default installation of Fedora Linux, and is reported under the name *Tcp: PassiveOpens*. It's possible to calculate the average number of new TCP connections per second based on two values, when taken with certain time interval (in our scripts an interval of 5 seconds was used).

Apache Web Server includes *mod_status* module, which, once enabled, allows collecting various statistics regarding the current state of the server, by accessing `http://server/server-status` page (`http://server/server-status?auto` page provides the same information in machine-readable format). One of the parameters available on the status page is the current state of each process of the web server, our script uses this data to collect the total number of *Idle* processes.

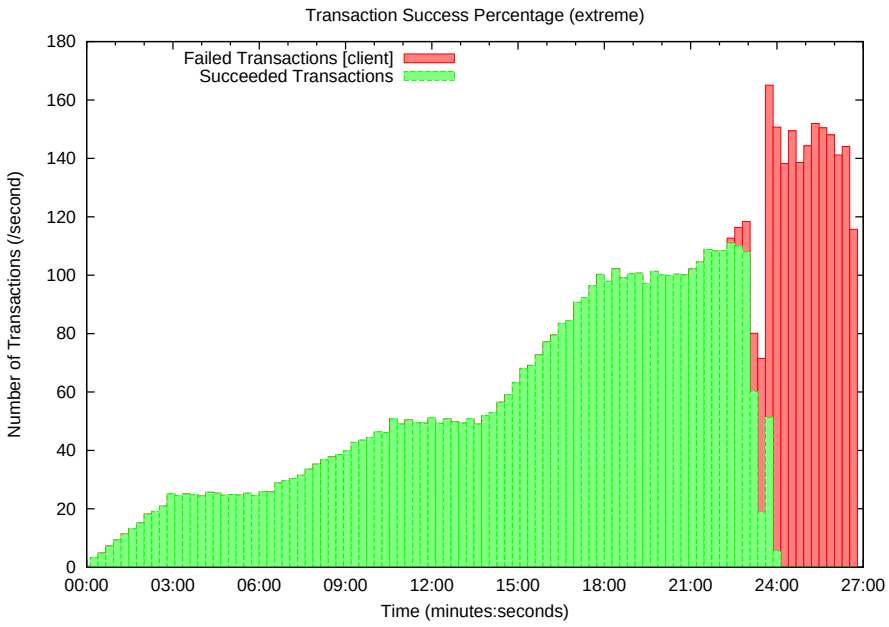


Figure 32: Transaction Success Rate (extremely optimized script)

Server Modes The optimized version of the site is activated by redirecting all requests to alternative file tree, in which the modified content is being stored. The following configuration was added to *httpd.conf* configuration file:

```
RewriteEngine On
RewriteCond /var/www/html/opt.do -f
RewriteRule ^/top500(.*) /var/www/html/opt/top500$1
```

The first line turns the "*mod_rewrite*" module on. The second line is the condition, which should be satisfied in order to the actual path-rewriting rule to be activated; in this case we want to test for existence of */var/www/html/opt.do* file. The last line redirects all requests for "top500" site to */var/www/html/opt/top500* folder, in which the optimized versions of the original files are located.

Once the Apache server is started with new configuration, we can instantly switch to optimized mode by creating */var/www/html/opt.do* file, and can switch back to normal mode by deleting that file.

We should also pay a special attention to configuration of the caching algorithm used by the server. Apache server supports two major modes of caching: URL-based and path-based. URL-based mode is not appropriate for our purpose, as once a URL is cached, any subsequent requests to that URL would bypass all rewriting rules, and the same version of the file will be served until this cache entry expires. The URL-based mode can be used in

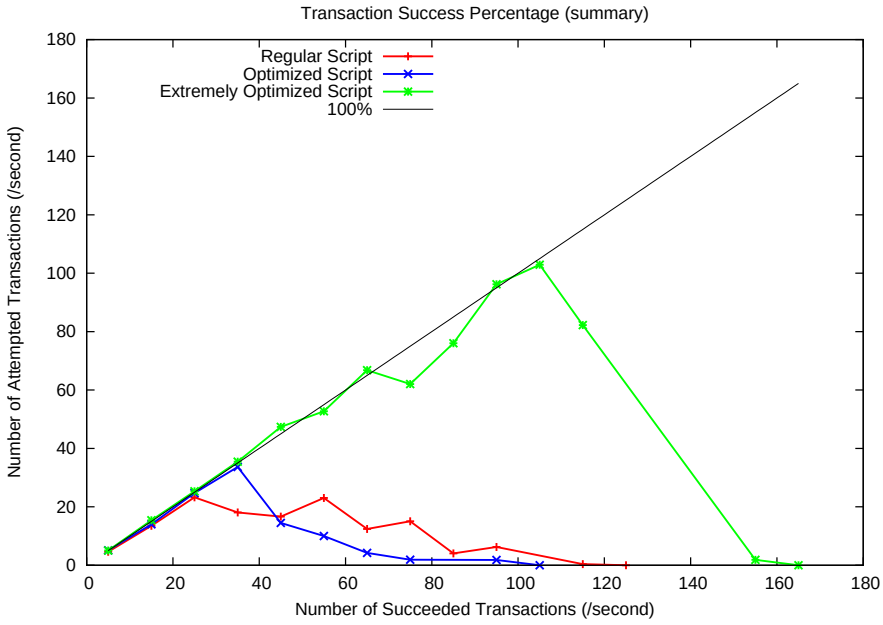


Figure 33: Transaction Success Rate (summary)

our context only if the cache can be cleared as part of the transition between normal and optimized modes. In our tests we have used path-based caching scheme, in which the files are cached in the memory according to their physical location in the file system.

Switching Algorithm We have created a rather simple Perl script which controls the operation mode of the web server, and initiates a change of the mode when a need is detected. The script runs in an infinite loop, which performs the following set of commands every five seconds:

1. Collects performance measurements, as explained above.
2. When in the *Normal* mode, will switch to *Optimized* mode if the CPU utilization exceeds a threshold of 85%, by writing */var/www/html/opt.do* file.
3. While still in the *Normal* mode (and the CPU utilization is below the threshold of 85%), watches the throughput of incoming TCP connections per second, and keeps the highest value as an indicator for a "safe" load on the server.
4. When in the *Optimized* mode, will switch back to *Normal* when all of the following conditions are met three times in a row:
   - CPU utilization is below 85%;
   - There's at least one idle server process;

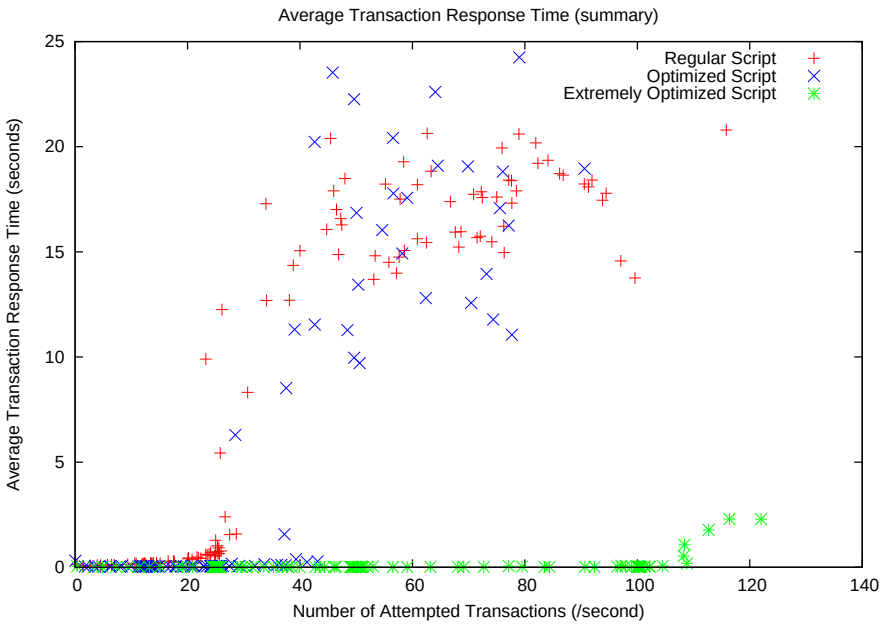


Figure 34: Average Transaction Response Time (summary)

- The rate of incoming TCP connections is lower that the maximal "safe" rate which was stored while running in the *Normal* mode.

5. *Normal* mode is activated by deleting previously created */var/www/html/opt.do* file. In the same time the script reduces the "safe" value of number of connections per second by 30%, and it can be incremented again later.

The script uses the number of concurrent connections as a measure for a number of active users in the system. We can't rely on the number of requests per second, as the number of components in a single page is different for normal and optimized versions of the site, thus the average number of requests per transaction is also different.

The number of TCP connections per user is changing too. While with the regular site each client usually opens two simultaneous connections to the server, we assume that a single connection will be opened for an optimized version. This behavior is taken into account, and the number of connections per second is normalized when comparing connection rates of *Normal* and *Optimized* modes.

LoadRunner Script The script always starts a transaction from requesting the same HTML page - one of the articles available on `top500` site. It then searches for certain string in the content, in order to identify which version of the page was returned by the server. According to the version of the page, the script would send HTTP requests for additional components of the page, this way the set of requests generated by LoadRunner is identical

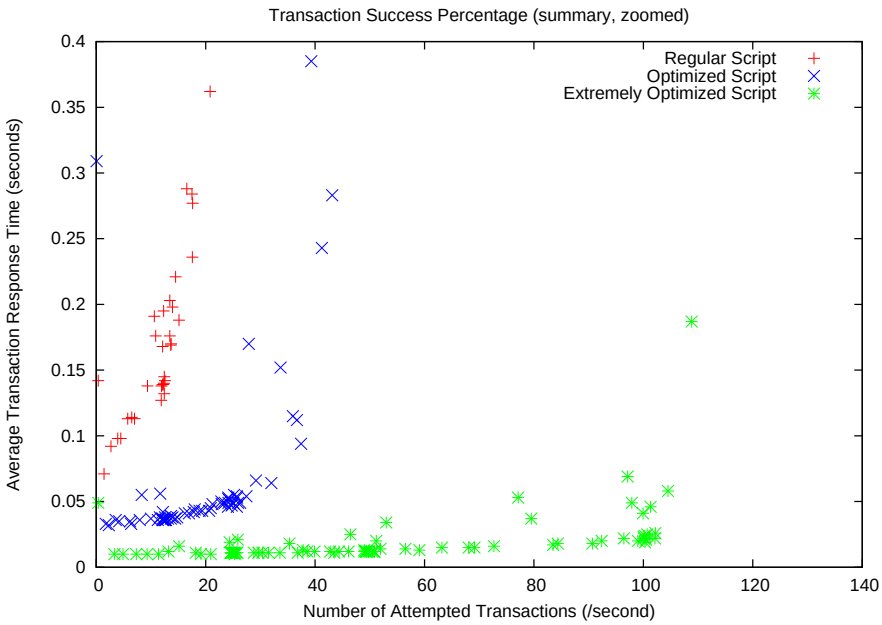


Figure 35: Average Transaction Response Time (summary, zoom)

to that of a real web browser, and depends on the mode in which the server is currently running.

Results Figure 36 shows the load on the server. In the first 10 minutes the users were arriving in a constant rate of 15 transactions per second on average. At the time of 10:00 minutes the load started increasing sharply, and has reached the rate of 85 transactions per second within 1:45 minutes. The high load remained for 10 minutes, and then has dropped back to the initial rate, in which it has remained for the rest of the test.

The average CPU utilization is shown in Figure 37, along with an indicator for server's running mode (0 for normal mode, 1 for optimized). As the graph shows, at the time of 10:16 minutes the server has already started running in the optimized mode, 16 seconds after the beginning of the peak load, triggered by high CPU utilization (the spike in the CPU time at time 10:16 had a value of 86%, but the graph shows a lower value of 75% as result of calculation of time buckets). We can also learn the following properties from the CPU utilization graph:

- While serving the optimized version during the peak load, the CPU utilization of the server was lower than that of the original version under a regular load (25% compared to 50%). This indicates that it might be possible to process even higher load peaks, from CPU utilization perspective.
- There are spikes in the CPU utilization both at the beginning and at the end of the load peaks, we assume that this is a result of creation and destruction of a large number

of server processes in a course of a short time interval.

The number of transactions per seconds started to decline at the time of 21:30 minutes, and the server has switched back to the normal version of the site at 22:45. We can also see that at the time of 25:00 minutes the server has entered the optimized mode again for a short time period, as a result of a short spike in the CPU utilization value.

The average transaction response time is shown in Figure 38. As in the previous tests, the average response times for optimized version are lower than response times of an unmodified page.

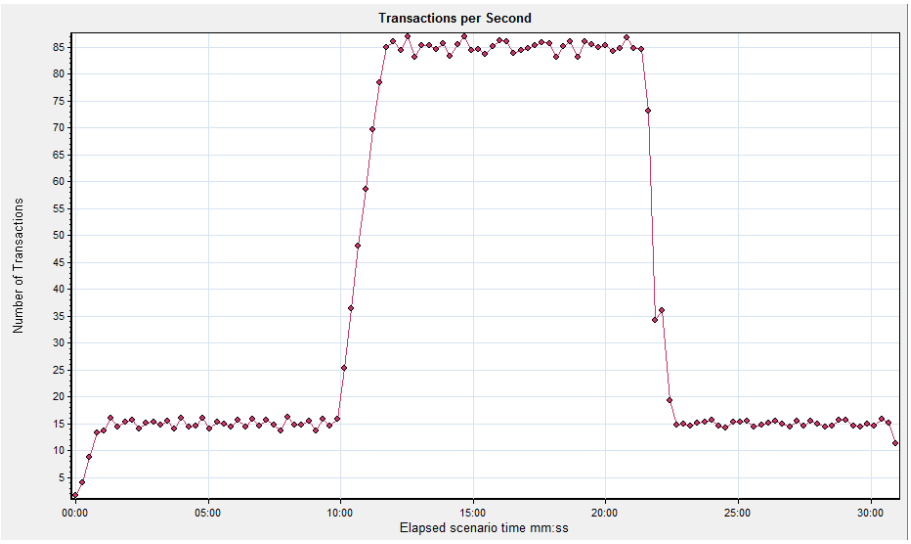


Figure 36: Average Number of Transactions per Second

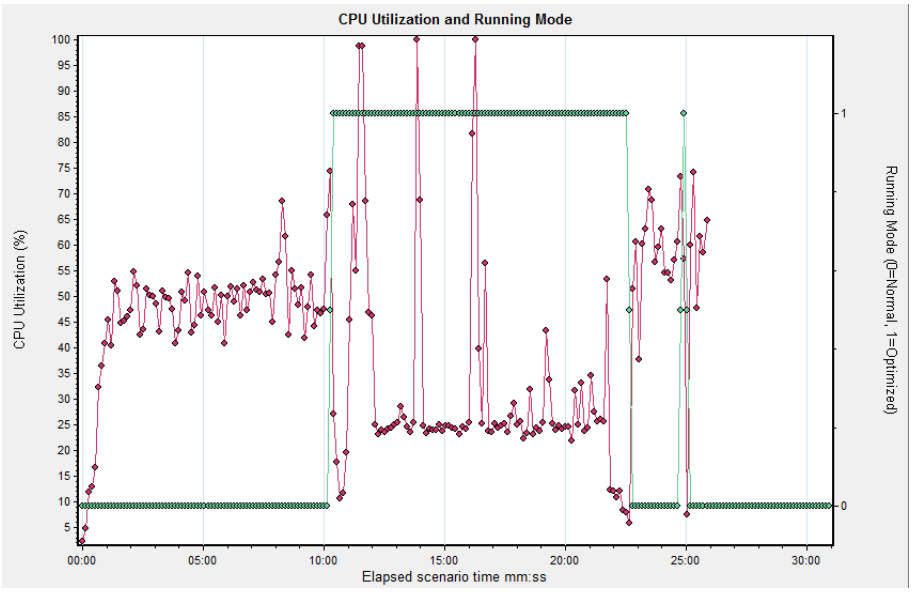


Figure 37: CPU Utilization and Running Mode

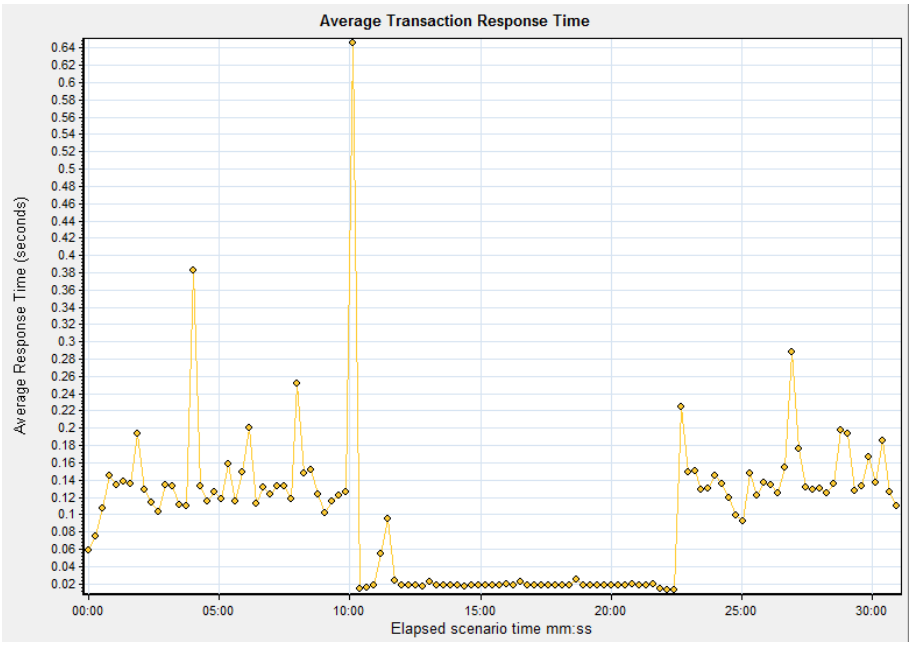


Figure 38: Average Transaction Response Time

6 Summary

In the current work we have addressed a challenge that is faced by many web sites, which is the ability to cope with exceptionally high peaks in the load. Our solution is based on an assumption that it is preferable to serve a degraded content to many users, rather than serving a full version to a smaller number of clients.

We have proposed a model for content adaptation, which reduces the perceived quality of the site in some of the aspects, as a price for better performance of the server. Optimization methods developed in this work focus on reduction in the number of HTTP requests required to present the content in the browser, and reduction in the number of transmitted bytes per page as result of compression of graphical and other media content.

We propose a set of rules for automated classification of the images in the page according to their semantic role, and based on that it's made possible to eliminate most of decoration images from the page. Other types of images can be removed too, or compressed to a smaller file sizes. For a textual content, the optimization techniques include embedding JavaScript code and CSS files in the body of HTML, and rewriting HTML code such as whole blocks are removed from the page, based on manual configuration.

Comparison of the performance of Apache Web Server when serving regular and optimized versions of the site has shown a reduction of 91% in the average response time for a single transaction, and an increase of 340% in number of user transactions per second that could be successfully served.

We also suggest a framework for integration of the solution into existing web server deployment. The framework is divided into two major modules: the first creates an optimized version of the web site in a separate folder (and maintains that copy in sync with the changes in the original files), and the second monitors the performance of the web server, and switches the server from normal to optimized mode when a need arises, by redirecting all HTTP requests to optimized files' tree.

The implementation of automatic optimizer was left out of the scope of current work, and all the tests were conducted with manually created versions of the site, while following optimization rules we've developed. We have implemented a module which monitors Apache server (CPU utilization, status of server's processes, number of concurrent connections) and automatically switches between normal and optimized versions. In the test scenario the system has switched to optimized mode 16 seconds after the beginning of the peak load, and has returned back to normal mode 1:15 minutes after the peak was over.

6.1 Future Work

Our research was limited to static web sites only. Although some of the concepts might be applicable for dynamic sites too, different techniques may provide better results for dynamic content. There're additional aspects to the quality of dynamic web sites - currency of the data and speed of updates. Potential optimization might, for example, focus on degrading some of those dynamic properties, and benefit from reduced processing power requirements.

Image classification algorithm presented in this work is rather a basic one, and more sophisticated solutions can be found to improve it's accuracy, and as result - effectiveness of automatic optimization techniques.

In our work the optimization of HTML content is based on manual rules only. It might be possible to automatically identify blocks in a web page that can be removed as a whole, based on the position of the block in the layout of the page, or by using other methods for semantic classification of different areas in the page.

References

[1] Tarek Abdelzaher and Nina Bhatti. Web content adaptation to improve server overload behavior. In *WWW8 / Computer Networks*, pages 1563–1577, 1999.

[2] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. *Comput. Netw.*, 33(1-6):1–16, 2000.

[3] L. Cherkasova and P. Phaal. Peak load management for commercial web servers using adaptive session-based admission control. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9*, page 9022, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Apache Foundation. Web server documentation: mod_rewrite. Website, 2009. `http://httpd.apache.org/docs/2.2/rewrite/`.

[5] Apache Foundation. Web server documentation: Rewritecond. Website, 2009. `http://httpd.apache.org/docs/2.2/mod/mod_rewrite.html#rewritecond`.

[6] Free Software Foundation. Httrack website copier. Website, 2009. `http://www.httrack.com`.

[7] Luis Francisco-Revilla and Jeff Crow. Interpreting the layout of web pages. In *HT '09: Proceedings of the 20th ACM conference on Hypertext and hypermedia*, pages 157–166, New York, NY, USA, 2009. ACM.

[8] L.P. Hewlett-Packard Development Company. HP LoadRunner software. Website, 2009. `https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17%5E8_4000_100__`.

[9] A. Hu, J. Bagga. Categorizing images in web documents. *IEEE MULTIMEDIA*, 11(1):22–31, 2004. ISSN 1070-986X.

[10] Melody Y. Ivory and Marti A. Hearst. Improving web site design. *IEEE Internet Computing*, 6(2):56–63, 2002.

[11] J. Kapoun. Teaching Undergrads Web Evaluation: A Guide for Library Instruction. *College & Research Libraries News*, 59(7):522–3, 1998.

[12] William LeFebvre. CNN.com: Facing a world crisis. In *LISA*. USENIX, 2001.

[13] Wei-Ying Ma, Ilja Bedner, Grace Chang, Allan Kuchinsky, and HongJiang Zhang. Framework for adaptive content delivery in heterogeneous network environments. In *Multimedia Computing and Networking*, pages 86–100. SPIE, 2000. Vol. 3969.

[14] Konstantina Papagiannaki, Sue Moon, Chuck Fraleigh, Patrick Thiran, and Christophe Diot. Measurement and analysis of single-hop delay on an IP backbone network. In *IEEE Journal on Selected Areas in Communications*, page 2003, 2003.

[15] Ning Sun Rema Hariharan. Workload characterization of SPECweb2005. http://spec.it.miami.edu/workshops/2006/papers/02_Workload_char_SPECweb2005_Final.pdf, 2006. Sun Microsystems. Presentation at SPECworkshop.

[16] Steve Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, 2008.

[17] Microsoft TechNet. Global registry entries (IIS 6.0). Website, 2009. `http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/60a90c91-a8d0-43b6-89db-a431d0ea0cb4.mspx?mfr=true`.

[18] Wikipedia. Html. Website, 2009. `http://en.wikipedia.org/w/index.php?title=Slashdot_effect&oldid=314706748`.

[19] Wikipedia. Html. Website, 2009. `http://en.wikipedia.org/w/index.php?title=HTML&oldid=312368589`.