Trading off Quality for Throughput Using Content Adaptation in Web Servers

Michael Gopshtein Dror G. Feitelson

School of Engineering and Computer Science, The Hebrew University, 91904 Jerusalem, Israel
mgopshtein@gmail.com, feit@cs.huji.ac.il

Abstract

A basic problem in managing web servers is capacity planning. A partial solution is to use content adaptation, where the system automatically trades off quality for throughput, e.g. by eliminating graphical decorations and adjusting page layout. We evaluate this approach based on a full implementation in Apache and increasing load patterns. The implementation uses two alternative versions of the files, and employs URL rewriting rules to select which version to use. Triggering a switch from one version to the other is done based on readily available load metrics. The experiments show that throughput can be increased by a factor of 2 to 4 at the price of minor to acceptable deterioration in graphical quality. Increasing throughput by an order of magnitude is also possible, but requires larger compromises. Nevertheless, this is still achievable without a real effect on content. Thus content adaptation is a viable tool, but may be insufficient by itself for handling huge surges in load such as flash crowds.

Categories and Subject Descriptors C.5.5 [*COMPUTER SYSTEM IMPLEMENTATION*]: servers; H.3.2 [*INFORMATION STORAGE AND RETRIEVAL*]: Information Storage—File organization

General Terms Design, Measurement, Performance

Keywords Web server, Overload, Throughput, Degraded service

1. Introduction

The success of many web sites depends on the number of visitors to the site, and on the ability of these users to complete their intended transaction, be it a purchase in an online store or reading an article on a news site. This goal is reflected by the 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 in one of the steps will put the whole transactions at risk and possibly lead to financial loss. web site owners therefore spend considerable effort to be able to serve every single request. This includes various optimization techniques implemented in the web servers, and hardware overprovisioning which aims to provide high availability and acceptable response times in cases of peak load on the system.

Load fluctuations imply that a large fraction of the available facilities are actually unused most of the time. This is partly due to the normal periodic fluctuations in the average load, which are a function of the time of day. But a larger problem is the phenomenon of "flash crowds", where a very large number of web surfers converge to one site and cause a surge in the load. A possible cause for such flash crowds is the *Slashdot effect* [16]. This 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 causing it to become unavailable.

Exceptionally high loads can also occur as a result of singular events. One rather extreme example is the traffic buildup at CNN.com during the 9/11 tragedy [9]. Measurements showed that traffic increased exponentially, and the number of HTTP requests was doubled every 7 minutes: it grew from less than 85,000 hits/second to 229,000 hits/second in just 15 minutes.

The description of the actions taken by CNN staff to continue to serve all incoming requests is illuminating. Among them were increasing 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 reduced, until the whole page consisted of only 1247 bytes of HTML, a logo, and a small picture.

The most common solution to the load fluctuation problem nowadays is to use a hosting service. The idea is that when many independent web sites are hosted together, it is unlikely that all of them would experience load surges at the same time. Thus the shared infrastructure can be partitioned in different ways based on momentary requirements, reducing the required overprovisioning. However, not all web sites use such shared hosting services, and even large sites may suffer from overload. For example, Facebook also reportedly have builtin levers allowing site administrators to disable peripheral features and focus on supporting core features when performance problems occur [13]. This is an example of Brewer's "DQ Principle", where large-scale services are managed by manipulating either the data-per-query or the queries-per-second [4].

Following these examples, our goal is to understand to what degree manipulations of the structure of a web page can be exploited to trade off quality for throughput. In previous work, we analyzed how the structure of a web page influences the work required to serve it [7]. For example, this showed that with normal caching the disk is not expected to be a bottleneck, and that reducing the number of requests made is more important than reducing the total size. We now use these findings to guide content adaptation actions similar to those performed by the administrators of CNN.com. Our main contribution is in the experimental quantification of the overheads involved and the performance benefits that may be obtained. While our experiments are limited to static pages, we note that the results nevertheless provide insights and data that are also applicable to the dynamic creation of web pages on the fly.

2. Related Work

Obvious approaches to reduce web server overload are to use caching and replication [14]. However, when this is not enough, content adaptation may be necessary. Most previous work on content adaptation emphasized functional transparency, where the appearance of the page does not change and therefore there is no loss of quality. For example, this is the case when images are compressed or unified into a single file (sprite) and then cropped for display. (see [15] and `http://www.websiteoptimization.com/` for surveys of such techniques.) Such optimizations should always be used to reduce infrastructure costs, and therefore cannot help when coping with overload. Other adaptations focus on improved accessibility. For example, this may be done by CSS-restyling to remove background shading and increase font sizes [6]. As this just changes the appearance of a page without changing the amount of information that is transmitted, this has no effect on server throughput.

To achieve a significant improvement in throughput one must use more drastic content adaptation and do so at the web server itself. The typical approach used is to create a *lighter version* of the original web site which will allow serving a larger number of users [1, 2, 12]. Techniques to achieve this include "adaptation tags" inserted manually into HTML files, image quality degradation by lossy compression, and elimination of small cosmetic items from the page. Indeed, in our previous work we also found that reducing the number of elements on a page is the most important optimization [7].

Similar methods for content adaptation have been developed in order to support mobile devices with limited bandwidth [11]. For example, it has been suggested that intermediary services adapt the content provided by a server to the context of the user, e.g by reducing image sizes [10]. While the applied adaptations are similar, the motivation in this case is reversed, as the bandwidth bottleneck is related to the user rather than to the server.

In this paper, we present results of a full implementation of content adaptation based on the Apache web server, and demonstrate that throughput can be increased by a factor of 2 to 4 at the price of minor to acceptable deterioration in quality. This improvement is much higher than that previously reported in [12], which was less than a factor of 2. It also improves on the results of [2], who achieved improvements of up to a factor of 7, but only for files of hundreds of kilobytes; for 64KB, their improvement was also less than a factor of 2. We also show that if more extreme deterioration is considered, the throughput improvement may reach a factor of 10.

3. Operation Modes

The web site optimization tool proposed in this work is composed of two parts: offline preparation of optimized versions of the original static files, as described in [7], and online monitoring of the web server's operation, leading to a switch between the normal and optimized versions of the content when required. In this section we focus on the indicators for performance of the web server, and how the transition between versions is performed.

3.1 Performance Indicators

Recall that our purpose is serving as many users as possible. Thus a suitable metric can be direct measurements of the number of HTTP requests per time unit and the server response times. The main problem with such metrics is that there is no immediate way to define thresholds on these values. For example, regarding the number of requests, we will need to know the maximal number of requests that the server can handle concurrently. This data is usually unavailable, and moreover, may depend on the specific requests. As for response time, this may be important for satisfying service-level agreements. But server-side response time may be hard to translate to user-perceived response time, due to networking effects. And in any case, it is again not clear where to place the threshold.

An alternative approach is to monitor system resources, especially utilization of hardware components, under the assumption that service degradation is always caused by overload on some resource. Naturally, different hardware and software configurations, coupled with different workloads, may cause different resources to become a bottleneck. However, since the total capacity of each hardware resource is known, it is 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 the additional advantage that we can then choose an optimization algorithm which reduces the use of the specific hardware resource that is overloaded. This is based on the results shown in [7], where some methods had a strong effect on CPU time consumption, while others affected the required network bandwidth.

In addition to monitoring hardware resources, most web servers make their internal performance indicators and "health" status available for administrators and external tools. One such example is the utilization of worker threads, which is the percentage of threads that are busy processing some user's request at a given moment, out of the total number of such threads configured on the server. This information can also be used as a trigger for optimized mode.

3.2 Switching Between 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. Different methods to switch between alternative versions of the content may be appropriate for different web servers. In our work the server chosen as a target platform is Apache HTTP Server 2.2, running on the Linux operating system.

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 the client. The simplest implementation is to perform the URL-to-file mapping is as follows:

- The web site is associated with a certain directory in the local file system, referred to as the "web root",
- The "file" portion of the URL is interpreted as a relative path inside the web root directory.

Like previous work on content adaptation [1], we assume that an optimized version of static web content is stored under a separate directory, which we will assume is called `opt`. Optimized files use the same relative paths as the original files have relative to the web root directory. For example, the optimized version of a logo originally stored at `images/logo.gif` will be stored at `opt/images/logo.gif`. Thus, to serve optimized content, it is enough to change the configuration so that the `opt` directory becomes the new web root.

However, we note that even static data can change over time, e.g. when new files are added, or existing files are deleted or modified. There may then be a certain delay between the creation or modification of the original data, and the generation of an optimized version reflecting the changes. The system should therefore be able to serve the original data if an optimized version does not exist or is out of date. This is not possible if the web root directory is simply switched.

The desired functionality can be achieved by using Apache's `mod_rewrite` rule-based rewriting engine, which supports the definition of advanced URL-to-file

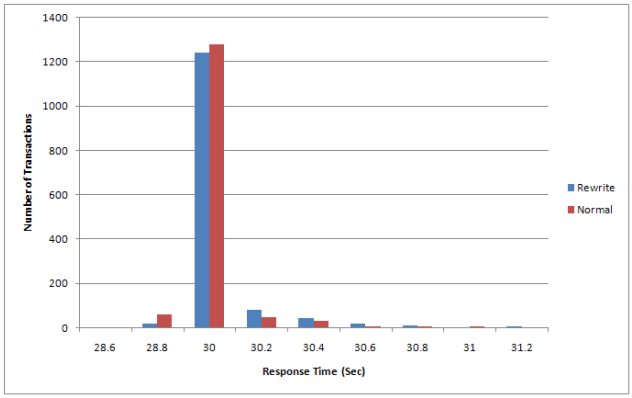

Figure 1. Performance Overhead of mod_rewrite Rules.

mapping instructions [3]. In particular, we can add rules that will redirect requests to the optimized version only provided that the following conditions are met:

1. The web server is currently running in optimized mode. In order to flag the operating mode one can use an environment variable (global or internal to the Apache server), or the existence of a special file in a known location; both methods are supported by the rules engine.
2. The optimized version of the requested file already exists, and is up to date.

One of the advantages of this method for content redirection is 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.

A possible problem with using URL rewriting is the overhead involved. We therefore measured the performance of an Apache web server with and without URL rewriting. The test was designed in the following way:

1. A local copy of the web site `www.adagio.com` (an online tea shop) was created using the HTTrack tool (`http://www.httrack.com/`). The tool was configured with all default settings, with the addition of the following parameters:
   - Maximal search depth was limited to 3 clicks, and
   - All images were copied too.
2. The stored copy of the web site was configured as the local site of an Apache web server.
3. In order to emulate simultaneous traffic from multiple users, a series of HTTP requests were recorded as a LoadRunner script [8].
4. An "optimized" version was created by adding a symbolic link to the web site root directory. Thus all the files remained the same both in the regular and the optimized versions.
5. Based on the recorded LoadRunner script a load was generated on the web servers, with half of the virtual users connecting to the original web site, and the other half to the optimized version, using the mod_rewrite rules as explained above. The average time to execute all requests in a single instance of the script was measured.

The results are shown in Figure 1. This shows that the distributions of response times of the server are practically the same for both the normal version of the web site, and the one having mod_rewrite rules activated. We therefore find that the overhead of rewrite rules is negligible, so we can use the method described above safely.

4. Experimental Validation

In this section we validate the methods proposed in the previous sections by constructing a full implementation, and measuring the effects of the proposed optimization methods in a realistic environment.

4.1 Experimental Setup

The environment that was used for all performance tests was as follows.

We used an Apache web server, version 2.2, running on a Linux operating system (Fedora release 11, Kernel 2.6.30.8). The hardware base was a Dell OptiPlex GX260, with an Intel Pentium 4 processor running at 2.4GHz, and 500 MB of memory. The web server ran with all default configuration parameters, except as noted below in specific experiments.

The load on the web server was generated by the HP LoadRunner tool [8], version 9.50. The *Controller* is the central component of LoadRunner. It orchestrates the generation of the load by *Load Generators*, and collects statistical data which is finally analyzed with the *Analysis* tool. The controller was installed on an HP Compaq 8510w, with an Intel Core 2 Duo CPU T7500,

running at 2.2 GHz, and 4 GB of memory. This machine was running Microsoft Windows Vista OS (Enterprise edition, version 6.0.6001). This machine was also used as one of the Load Generators. In addition there were three other Load Generators:

- Two identical to the web server, with Microsoft Windows Server 2008 OS (Standard edition, version 6.0.6001).
- An additional machine with an Intel Pentium 4 Processor at 3.2 Ghz, and 2 GB RAM, running Microsoft Windows Vista OS (Ultimate edition, version 6.0.6000).

All computers are connected by a 100 Mb/sec Ethernet LAN via a Linksys WRT54G router.

The load tests were performed on a local copy of the Top500 site `http://www.top500.org/`. The copy was captured using the HTTrack tool, with all default parameters, except for:

- The depth of the search (when following links) was limited to 3. This is sufficient as most of our experiments only accessed a page or two from the site, as is most common in flash crowd scenarios.
- The instructions in the `robots.txt` file were ignored, as this was not relevant for our use.

All pages of this site include JavaScript code which is intended to randomize the order of appearance of advertisement images, by injecting external images into static HTML content. These functions were removed from all pages, and replaced with static images of the same type.

To run a performance test, the LoadRunner controller implements a scenario that specifies how many Virtual Users (vusers) to create and when. These Virtual Users are simulated by the LoadRunner Load Generators. The activity of each Virtual User is defined as a sequence of instructions listed in a script file, which are repeated multiple times.

Instructions may be either individual HTTP requests or complete HTML requests, in which case embedded objects are also requested automatically. We used HTTP requests because this leads to better performance of Load Generators, as there is no need to parse HTML content of responses when replaying the script during a test. The HTTP requests are generated based on recording the activity during an actual interactive recording session. Sets of requests can be designated as a "concurrent group", meaning that they should be fetched together, because they actually constitute the elements of a single page.

In addition, it is possible to insert think times between requests in the script. Since we typically only consider a single page request, think times are not needed. Another parameter specifies the *pacing* of repeated iterations of the whole script. The options for this parameter are:

1. Repeat as soon as previous iteration ends.
2. Repeat after a certain amount of time from the *end* of the previous iteration, selected at random from a range.
3. Repeat after a certain amount of time from the *beginning* of the previous iteration, selected at random from a range. This can lead to executing the script at a given average rate, regardless of the time needed for each execution. However, if the previous iteration takes too much time, the new one is delayed.

Our use of this parameter is explained below in the different scenarios.

Other settable parameters of the LoadRunner script include:

- Simulation of the browser's cache. If multiple HTTP requests to the same URL appear in the script, the request is actually sent only once.
- Connection management. To emulate real browser behavior, two connections to the server are opened by each user, and concurrent groups of HTTP requests are downloaded simultaneously on both of them.

The analysis of performance results relate to transactions, which can be defined as any subset of instructions in the script. In our experiments the whole script was always considered a single transaction (and in most cases this corresponds to downloading a single page). In addition, transactions may end with either *success* or *failure* status. A failure can be caused by any an error, such as a new connection being refused by the server, by a timeout while waiting for server's response, etc.

4.2 Basic Optimization Performance Tests

In order to evaluate the overall performance implications of the suggested optimizations, and in particular the potential increase in throughput (i.e. the ability to serve more clients under loaded conditions), we need to

create optimized versions of the web site. In this section we present results for basic optimizations and different server configurations. In the next section we consider a more extreme optimization, and a workload scenario that simulates a flash crowd.

4.2.1 Workload Scenario

Given that our load is generated by LoadRunner, we can emulate an optimized site by modifying the LoadRunner script rather than modifying the site itself. To do so, we simply delete certain requests from the LoadRunner script, specifically those representing decoration images and blocks of HTML that are to be removed. The modified script then downloads the same files as a browser directed to the optimized version of the site would, albeit the content of these files is not changed or compressed. Using this approach on a manually-adapted version of the Top500 site leads to the following changes:

- The number of HTTP requests is reduced from 81 in the original script to 43 in the optimized one (reduced by 47%).
- The total size of downloaded files is reduced from 702KB in the original script to 570KB in the optimized one (reduced by 22%).

In order to investigate the effect of load and how much the server can support, we chose to create a scenario in which the load on the system — as reflected by the number of users — grows with time. This is done in three phases, enabling the system to stabilize each time before the load continues to grow. Moreover, at the higher loads the rate of growth is reduced. These considerations led to the following profile:

- In phase one, starting at the beginning of the test, 200 vusers are created at a rate of one new user every 2 seconds.
- In phase two, starting at 30 minutes into the test, an additional 200 users are added at a rate of one user every 5 seconds.
- Finally, in phase three, starting at 1 hour into the test, the last 200 users are added at a rate of one user every 12 seconds.

This profile of increasing users is shown in Figures 2 and 3. The pacing parameter was set to use random intervals in the range between 15 and 25 seconds between iterations of the script for each vuser.

Table 1. Summary of results for default Apache configuration.

parameter	*normal*	*optimized*
avg. resp. time	66.5 sec	31.9 sec
successful	12,709	27,530
failed	4,510	7,955

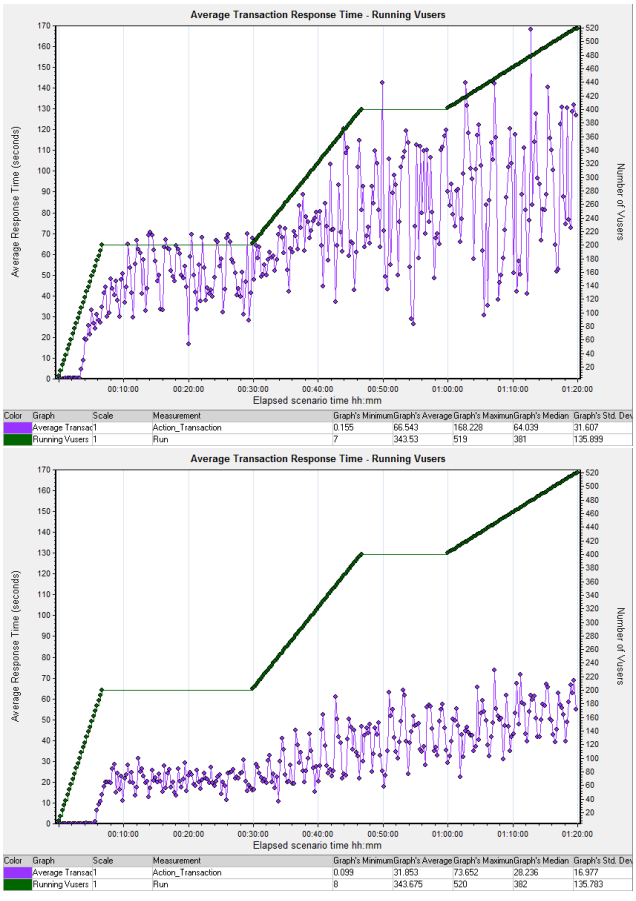

Figure 2. *Average response times for default Apache configuration, in relation to the number of vusers. Top: normal script. Bottom: script reflecting optimized site.*

4.2.2 Default Apache Settings

The first test was conducted on the web server with all the default settings, as it was first installed. The results in Table 1 are based on the first 1:20 hours of the scenario. These results show a 52% reduction in average transaction response time, and 106% increase in total number of completed transactions during the same time interval. This makes sense given that the number of HTTP requests per transaction was nearly reduced to half; it also demonstrates that the total volume of data is less important than the number of requests. Detailed

response times are shown in Figure 2. This shows that with the normal site response times start to grow at less than 4 minutes (around 100 vusers), and with higher loads become erratic in the range of 30 seconds to 2 minutes. With the optimized version they only start to grow at about 6 minutes (around 160 vusers), and even under high loads remain in the rang of 30–60 seconds.

Note that in both cases the system does not seem to become saturated. This is a result of how LoadRunner's Load generators operate. Recall that vusers are scheduled to execute the script once every 20 seconds on average. But if the previous iteration did not terminate yet, the next one 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 successive iterations.

4.2.3 Enhanced Apache Settings

The default Apache setting are very conservative. We therefore conducted another set of tests with the following enhancements to the configuration.

- Keep-Alive connections. By default the server does not enable the Keep-Alive feature, and as a result a new TCP connection is opened for each HTTP request. With such a configuration any reduction in the number of HTTP requests has a very large effect, which might cause our results to be overly optimistic. To be more realistic, we activated this feature by adding the configuration line `KeepAlive On`. `MaxKeepAliveRequests` was left at the default value of 1000, and `KeepAliveTimeout` at 10 seconds.
- Caching. In-memory caching is also disabled in the default configuration. As a result the server reads the content from the disk on each HTTP request. This leads to longer response times and increased lock contention between the server's worker threads. We therefore enabled in-memory caching by uncommenting the section for the `mod_mem_cache` module, while leaving all its default values. The default cache size is 4 MB.
- Working Processes. Apache allows configuring the maximal numbers of server processes, and the 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 these parameters have low values in the default configuration, so requests may be delayed waiting for one of the threads to become available or for the creation of a new one. We increased the number of processes allowed to 1500 using the `MaxClients` directive.

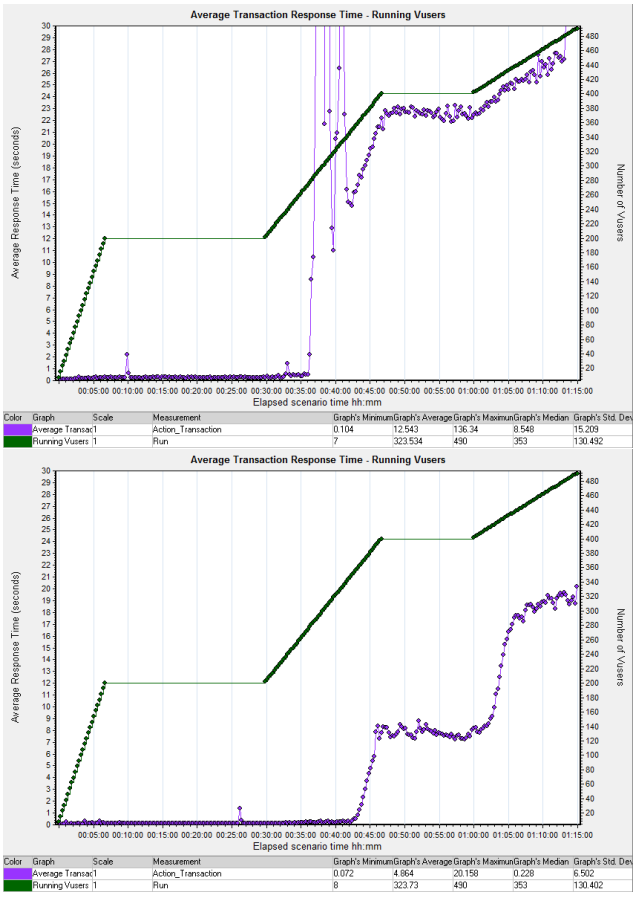

Figure 3. Average response times for enhanced Apache configuration. Top: normal script. Bottom: script reflecting optimized site.

All other configuration options were left at their default values, e.g. `LogLevel info`. In a real deployment it would be beneficial to adjust all these parameters automatically to optimize performance, for example as suggested in [5].

The average transaction response times when using the enhanced Apache configuration (and the number of running vusers) are shown in Figure 3. During the first 35–40 minutes both versions of the script achieved stable transaction times, with the regular version running at 270 msec per transaction, while the optimized one required only 138 msec — a reduction of 49% in average response time. At the same time the maximal num-

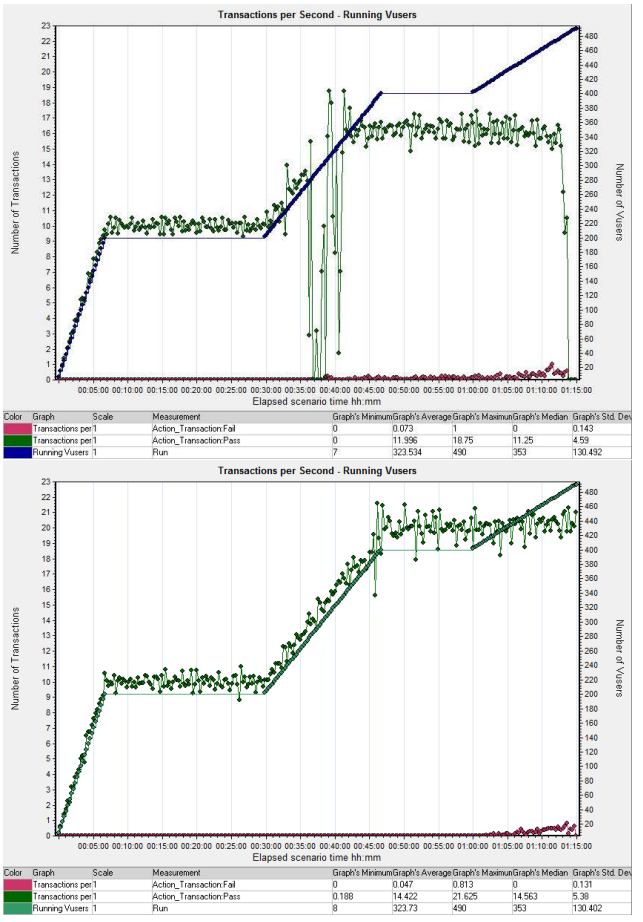

Figure 4. Completed transactions rate.

ber of supported vusers grows from about 260 to about 340.

Starting from certain point in time (about 36 minutes for the regular version and 42 for the optimized one) we see a sharp growth in the average response time, which stabilizes only when the load on the system becomes constant. This is most probably caused by contention for some system resource, causing requests to wait as was the case for the default Apache configuration. By looking at the number of completed transactions per second (Fig. 4), we find that the regular script stabilizes on a rate of about 16 transactions/second at the peak, while the optimized version reached around 20 transactions/second. However, the average response time of transactions is much lower for the optimized version.

These results suggest that with the optimized Apache version the CPU is less of a problem, and the network may become the bottleneck. Thus in the optimized version of the script the number of HTTP requests is reduced by 47%, and this indeed causes a nearly-factor-of-2 reduction in the response time, but it does *not* double the achieved throughput. On the other hand the total size of downloaded content is reduced by 22% only, and the throughput indeed grows by a similar factor. When any single resource saturates, the throughput depends on the saturated bottleneck and not on the potential service rate.

As explained above, these tests have the property of converging to a "closed system" once the load on the server was high enough. In the next section we overcome this limitation and create a scenario in which the number of new users trying to access the site does not depend on current performance of the server.

4.3 Flash Crowd Simulation

In the previous section we described performance tests aimed to compare the behavior of the server on regular and optimized content. However, the workload did not stress the server too much because of a feedback effect: longer response times cause longer delays before additional requests were made. In this section we describe a series of tests with the goal of simulating a slashdot effect. Thus we create an "open system", where new users continuously arrive at the web site at a certain rate, regardless of the current performance of the server. Apache was configured in the enhanced mode as above.

As is typically the case for flash crowds, all simulated users make a request for a single page, so in this case basically page=transaction=script. The load generators do not simulate any think time, as they are supposed to be simulating independent new users.

4.3.1 Optimization Levels

The simulation was repeated using three versions of the LoadRunner scripts, representing three possible levels of content adaptation at the server:

- 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 this link. The workload generated in this use case is expected to show high localization. In particular, we expect a large volume of users

to request a single page from the site (including all its embedded components). Note that this is not necessarily the home page, but rather whatever internal page was pointed to by the link.

To emulate this behavior, the LoadRunner script records a request to a single article on a local copy of the `www.top500.org` site. Overall the script opens 2 TCP connections, makes 59 HTTP requests and downloads a 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 requests 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 issued 29 HTTP requests, and the total size of the content was 188 KB. It still opened 2 TCP connections.

Extreme optimization Given the slashdot effect scenario, we decided to also test the system under a more extreme optimization scheme, which would be more aggressive in eliminating components of a page than the one used in the previous section. This was achieved 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 the CSS files to the body of the main HTML file, in order to preserve the basic formatting of the page. As result the size of the HTML page grew from 37.9 KB to 55.2 KB.
- The script was modified so that it makes all requests on a single TCP connection. It's possible to make the browser reuse the same connection for all requests by injecting a JavaScript code which will request all relevant images synchronously, after having the HTML page completely downloaded. This is advisable to reduce load as it reduces the number of opened connections.

See Figure 5 for a visual comparison of the regular and extreme optimized versions of the page.

Figure 5. Article from www.top500.org. Top: original version. Bottom: extreme optimized version. Note how the tabs at top-right are rendered sequentially due to simplified CSS usage.

4.3.2 LoadRunner Configuration

As noted above, load runner delays additional request iterations if previous ones have not ended yet, leading to a "closed system" scenario. In order to simulate an "open system" with the LoadRunner tool, the 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

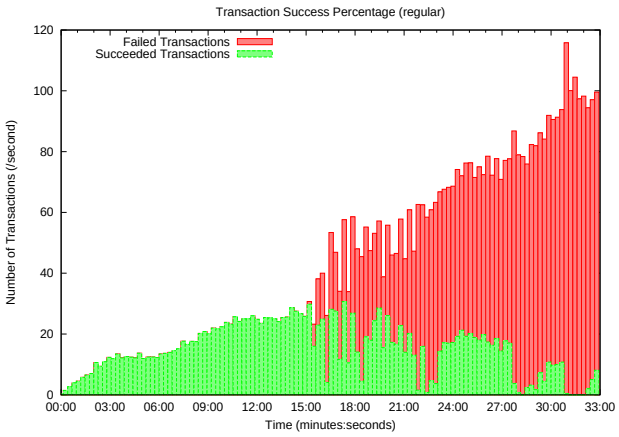

Figure 6. Transaction success rate for regular script.

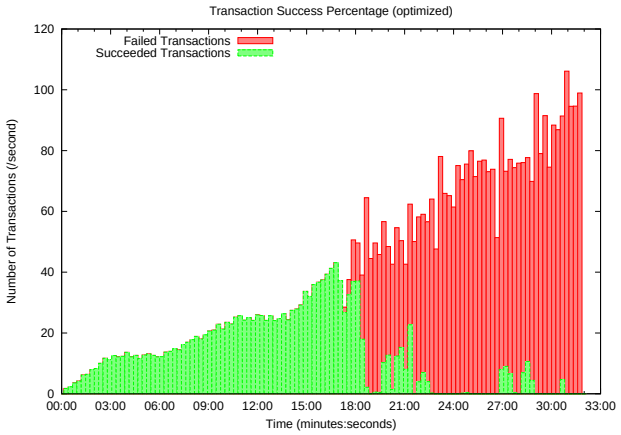

Figure 7. Transaction success rate for optimized script.

real-life case of a user waiting for a web page, and aborting the download it it takes too long;

- 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;
- The 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).

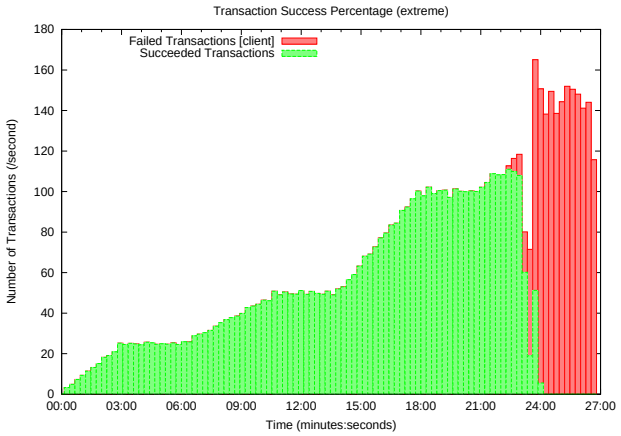

Figure 8. Transaction success rate for extremely optimized script.

4.3.3 Results

Figures 6, 7 and 8 show the success rate of attempted transactions for regular, optimized, and extremely optimized scripts respectively.

Figure 9 summarizes the data for success rates of transactions and provides a comparison between different versions of the script (and thus different levels of optimization). We can make the following observations:

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

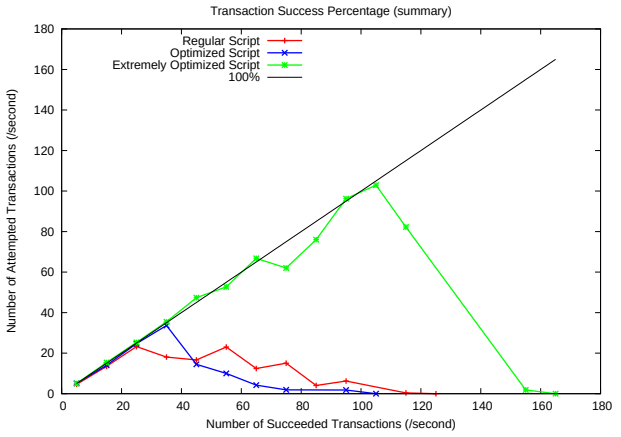

Figure 9. Summary of transaction success rates.

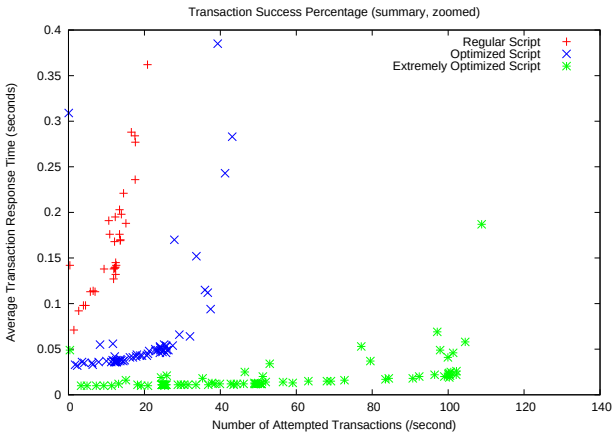

Figure 10. Average transaction response time.

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

Figure 10 shows the average transaction response times (of successful transactions), comparing different versions of the script. When the load on the server is relatively low, the average response times are:

- Regular script: around 100 msec.;
- Optimized script: around 40 msec. (reduced by 60%);
- Extremely optimized: 13 msec. (reduced by 87%).

Response times for the regular script start to increase when the load is as low as 15 transactions per second, and reach a value of 1 second at a load of 25 transactions per second. The optimized one reaches this value at 37 transactions per second, and the extremely optimized one at a load of 110 transactions per second.

4.4 Switching Between Modes

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

4.4.1 Performance Indicators

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

1. CPU Utilization;
2. Number of incoming TCP connections per second;
3. Status of running processes of the Apache web server.

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

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

The Apache web server includes a *mod_status* module, which, once enabled, allows collecting various statistics regarding the current state of the server by accessing the `http://server/server-status` page (`http://server/server-status?auto` 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.

4.4.2 Server Modes

The optimized version of the site is activated by redirecting all requests to an alternative file tree, in which the modified content is stored. To implement this, the following lines were added to the `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 actually activate the path-rewriting rule; in this case we want to test for existence of the `/var/www/html/opt.do` file. The last line redirects all requests for the `top500` site to the `/var/www/html/opt/`

top500 directory, in which the optimized versions of the original files are located.

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

We should also pay a special attention to configuration of the caching algorithm used by the server. The 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 our context only if the cache can be cleared as part of the transition between normal and optimized modes. In our tests we used the path-based caching scheme, in which the files are cached in the memory according to their physical location in the file system.

4.4.3 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. Collect performance measurements as explained above.
2. While in the *Normal* mode record the throughput of incoming TCP connections per second, and keep the highest value as an indicator for a "safe" load on the server.
3. When in the *Normal* mode, switch to *Optimized* mode if the CPU utilization exceeds a threshold of 85%, by writing the /var/www/html/opt.do file.
4. When in the *Optimized* mode, 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;
   - The rate of incoming TCP connections is lower that the maximal "safe" rate which was stored while running in the *Normal* mode.

   The switch back to *Normal* mode is done by deleting the previously created /var/www/html/opt.do file. At the same time also reduce the "safe" value of number of connections per second by 30%.

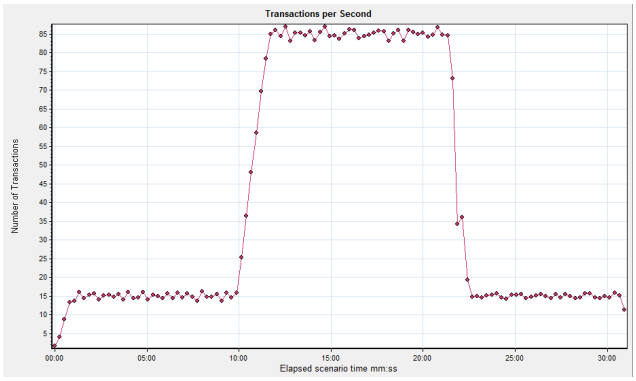

Figure 11. Average Number of Transactions per Second

The script uses the number of concurrent connections as a measure for the 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, so the average number of requests per transaction is also different.

Note that the number of TCP connections per user changes too. With the regular site each client usually opens two simultaneous connections to the server, but 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.

4.4.4 Results

The script always starts a transaction by requesting the same HTML page: one of the articles available on the top500 site. It then searches for a 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 decides what set of HTTP requests to send to simulate loading additional components of the page. This way the set of requests generated by LoadRunner is identical to that of a real web browser, and depends on the mode in which the server is currently running.

Figure 11 shows the load imposed on the server during this test. In the first 10 minutes the users were arriving in a constant rate of 15 transactions per second on average. At that time the load started increasing

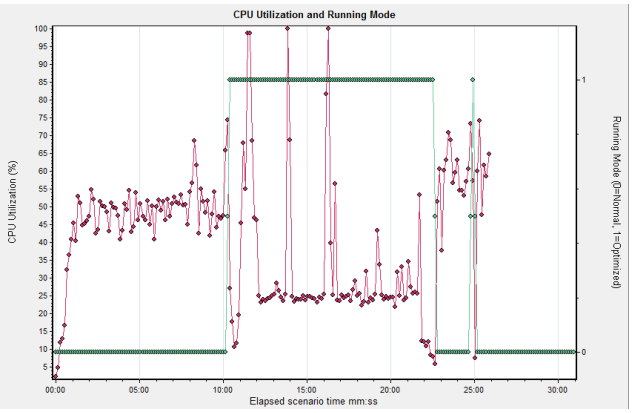

Figure 12. CPU Utilization and Running Mode

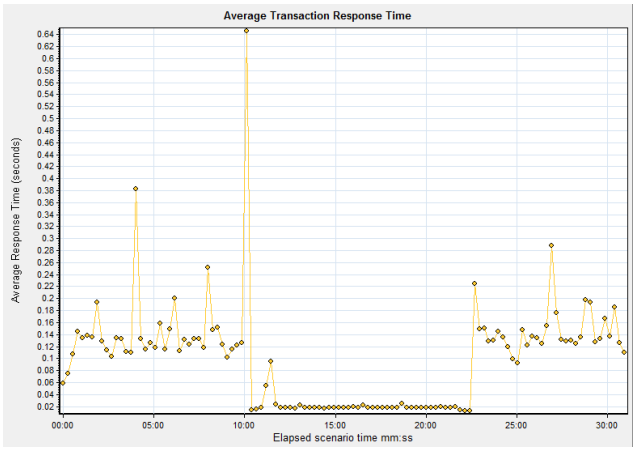

Figure 13. Average Transaction Response Time

sharply, and reached the rate of 85 transactions per second within 1:45 minutes. The high load remained for 10 minutes, and then dropped back to the initial rate, in which it remained till the end of the test.

The average CPU utilization is shown in Figure 12, along with an indicator of the server's running mode (0 for normal mode, 1 for optimized). As the graph shows, at time of 10:16 minutes the server already started running in the optimized mode, a mere 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 a CPU utilization perspective.

  In particular, if we define efficiency as the ratio of transactions per CPU percent, we find the following. The original efficiency was $\frac{15}{50} = 0.3$. The efficiency of the enhanced version was $\frac{85}{25} = 3.4$. Thus the efficiency of using the CPU improved by a factor of 11.3. Barring other bottlenecks, this indicates the maximal possible improvement in throughput.
- 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 21:30 minutes, and the server switched back to the normal version of the site at 22:45 minutes. We can also see that at time of 25:00 minutes the server 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 13. As in the previous tests, the average response times for the optimized version are lower than response times of an unmodified page.

5. Conclusions

Content adaptation serves users with a degraded version of the requested web page when the server is overloaded and cannot afford to serve the full data. Using guidelines for content adaptation based on data collected from a random selection of web sites [7], we have conducted an empirical study of a complete content adaptation implementation. We have shown how the different elements of content adaptation can be supported easily by an Apache web server, using the mod_rewrite rule-based rewriting engine to select normal or optimized versions of pages, and a set of three readily-available performance metrics to trigger the switch between modes. This facilitates automatic content adaptation with extremely short delay between the onset of overload and the switch to optimized mode.

To test our implementation we used the HP LoadRunner tool. Several load generators were used to create increasing load levels until the system saturated. This demonstrated that the optimized version of the site could sustain loads that were about 2–4 times higher

than those sustained with the normal version — using the same hardware and infrastructure, and without operator intervention. If a larger improvement in throughput is needed, larger compromises of quality are required (but still possible without affecting the actual content of the page). Thus content adaptation may be limited to an intermediate range of load fluctuations. Really large load surges would likely overwhelm even the most optimized version of the contents.

A byproduct of our experiments is to emphasize the need for careful evaluations. There are many pitfalls where one needs to configure the Apache server and load generating scripts correctly in order to obtain reliable results. It should also be noted that achieving the best possible absolute throughput numbers depends on correctly setting available Apache configuration settings.

The current work used optimized versions tailored by hand based on predefined guidelines. But in large and dynamic web sites, there is a need for automatic generation of optimized versions of new or updated pages. In future work we intend to implement this mainly based on removing decoration images and embedding scripts in the main HTML page.

Another avenue for additional work is to improve the mode switching algorithm, and especially the consideration of different performance indicators. The current implementation focuses on the CPU as the main potential bottleneck. However, it is possible that the network or even the system bus may become a bottleneck too. Identifying such situations and using them as triggers for mode switching will again improve the system throughput and reduce the danger of clients who do not receive service.

Acknowledgments

This research was supported by a grant from the Israel Internet Association.

References

[1] T. F. Abdelzaher and N. Bhatti, "*Web content adaptation to improve server overload behavior*". *Comput. Networks* **31(11-16)**, pp. 1563–1577, May 1999.

[2] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, "*Performance guarantees for web server end-systems: A control-theoretical approach*". *IEEE Trans. Parallel & Distributed Syst.* **13(1)**, pp. 80–96, Jan 2002.

[3] Apache Software Foundation, "*Apache module mod_rewrite*". URL httpd.apache.org/docs/2.2/mod/mod_rewrite.html, 2009. (visited 6 May 2010).

[4] E. A. Brewer, "*Lessons from giant-scale services*". *IEEE Internet Comput.* **5(4)**, pp. 46–55, Jul/Aug 2001.

[5] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus, "*Managing web server performance with AutoTune agents*". *IBM Syst. J.* **42(1)**, pp. 136–149, 2003.

[6] U. Erra, G. Iaccarino, D. Malandrino, and V. Scarano, "*Personalizable edge services for web accessibility*". *Universal Access Inf. Soc.* **6(3)**, pp. 285–306, Nov 2007.

[7] M. Gopshtein and D. G. Feitelson, "*Empirical quantification of opportunities for content adaptation in web servers*". In 3rd *Ann. Haifa Experimental Syst. Conf.*, May 2010.

[8] "*HP LoadRunner software*". URL https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17^8_4000_100__, 2010. (visited 6 May 2010).

[9] W. LeFebvre, "*CNN.com: Facing a world crisis*". *;Login:* **27(1)**, p. 83, Feb 2002. (summary of invited talk at LISA 2001).

[10] D. Malandrino, F. Mazzoni, D. Riboni, C. Bettini, M. Colajanni, and V. Scarano, "*MIMOSA: Context-aware adaptation for ubiquitous web access*". *Personal & Ubiquitous Copmut.* **14(4)**, pp. 301–320, May 2010.

[11] B. Noble, "*System support for mobile, adaptive applications*". *IEEE Personal Comm.* **7(1)**, pp. 44–49, Feb 2000.

[12] R. Pradhan and M. Claypool, "*Adaptive content delivery for scalable web servers*". In *Intl. Network Conf.*, Jul 2002.

[13] Royal Pingdom Blog, "*Exploring the software behind Facebook, the world's largest site*". URL http://royal.pingdom.com/2010/06/18/the-software-behind-facebook/, 18 Jun 2010. (Visited 27 Sep 2010).

[14] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso, "*Analysis of caching and replication strategies for web applications*". *IEEE Internet Comput.* **11(1)**, pp. 60–66, Jan-Feb 2007.

[15] S. Sounders, "*High-performance web sites*". *Comm. ACM* **51(12)**, pp. 36–41, Dec 2008.

[16] Wikipedia, "*Slashdot effect*". URL http://en.wikipedia.org/wiki/Slashdot_effect. (visited 31 Jan 2010).