Date: 05 March 2002
Author: Julian Midgley
Version: 1.0

Zeus Technology

Benchmarking Web Servers on Linux

Zeus Technology
Zeus House
Cowley Road
Cambridge
England
CB4 0ZT

Phone: +44 1223 525000
Fax: +44 1223 525100
Email: info@zeus.com
Web: http://www.zeus.com/

zeus

Table of contents

1 Overview

1.1 Executive summary

This document discusses how to go about benchmarking web servers running on Linux. It seeks to identify some of the common pitfalls that can lead to erroneous results, and provides advice on tuning web server systems for optimum performance.

1.2 Copyright notice

2 Principles of HTTP benchmarking

In benchmarking a web server, the aim is typically to determine the number of requests per second the server is able to sustain, whilst satisfying certain constraints. Typical constraints include:

- the requirement that a client should have to wait no longer than 'n' seconds for a response. ('n' is typically low, around 5-8 seconds);
- no valid request should receive an error message as a response;
- the requests should be selected at random from a large document root, to ensure that the web server's caching mechanism is fully exercised.

In order to conduct web server benchmarking, a system with at least the following components is required:

- a server running the web server software under test;
- clients running load generating software in sufficient number so as to avoid their saturation;
- a network connecting the clients to the server which is free of other traffic, which will not be saturated by the planned tests.

Under no circumstances should one attempt to benchmark a server by running the load generating software on the server itself. The web server and benchmark software will compete for CPU, and any results obtained will be extremely inaccurate.

2.1 A typical benchmarking installation

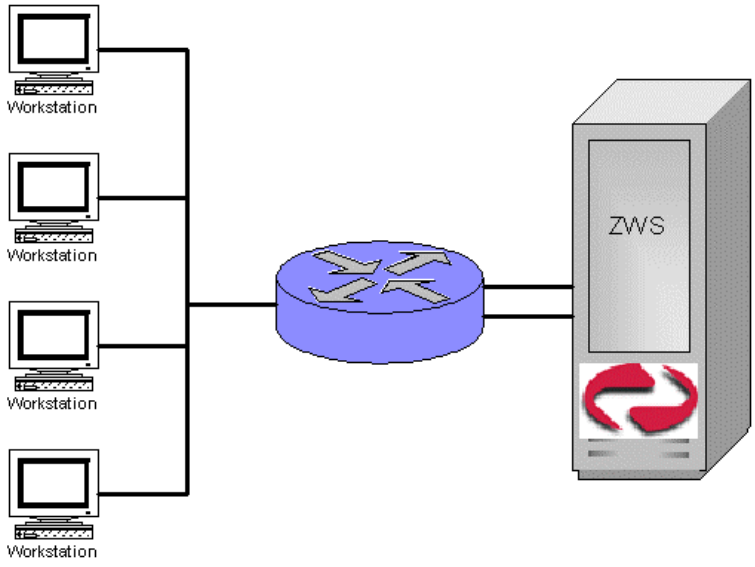

Figure 1 - A typical benchmarking network

Figure 1 illustrates a typical network layout for web server benchmarking. A number of clients are connected via a switch to the server under test. The server in this example has two network cards, to increase the bandwidth available to it, and decrease the likelihood of the network becoming the bottleneck during the tests. In practice, the actual bandwidth to the server that will be required will be a function of the server hardware, and the type of tests being performed. It is possible to saturate a server from a single client without saturating the network when dynamic content generators are being

tested or when requesting large numbers of very small files, but a web server serving large static files may be able to supply data at several hundred Mbps.

The network in use is private, and no machines other than those required for the tests are connected to it, in order to eliminate the effects of externally-generated network traffic spikes on the tests.

3 Likely bottlenecks

In order for the benchmark results to be useful, it is critical that the server under test is the only bottleneck in the system. It is all too easy to allow the network or the client machines to become the bottleneck, giving entirely inaccurate results. Preventing this can be difficult, and usually requires significant expenditure on hardware. For example, conducting an accurate SPECweb99 test against a modest 750 MHz Pentium III server will require more than 100 Mbps of bandwidth to the server (i.e. several network cards) and at least four client machines.

3.1 Network

When testing even the most modest of servers, at least 100 Mbps will be required between the clients and server. A 400 MHz Celeron is easily able to saturate a 10 Mbps network when serving static content, and will saturate a 100 Mbps network when serving large files. For SPECweb99 benchmarking, several network cards or gigabit ethernet to the server will be required.

When conducting the tests, monitor network I/O[1] to see if it comes close to the network limits; tools such as `httperf` report this for you. (Remember that on an Ethernet network, maximum throughput in practice is typically no more than 80-90% of the rated maximum (i.e. a 100Mbps network will typically support a maximum throughput of no more than 90Mbps).)

3.2 Client machines

It is critical that the client machines themselves should not become the bottleneck. The likelihood of this occurring varies with the benchmarking software in use. To eliminate this as a possibility, several clients are used. To determine whether or not the clients are the bottleneck, run the same benchmark twice with different numbers of clients (such that the requested load on the server is the same in each case). If the results differ, then the test was client-bottlenecked, and the number of clients should be increased. Never use system load or percentage CPU usage on the clients to determine whether or not they are bottlenecked; several benchmarking tools (such as `httperf`) report 100% CPU usage even when they have spare capacity[2].

[1] This could be done, for instance, by using `netstat -s` or possibly `ifconfig` depending on the brand of UNIX you are using.
[2] `httperf` executes in a tight loop wherever possible to avoid being affected by operating system scheduling characteristics.

4 Factors to measure when benchmarking a web server

A number of different quantities can be measured when examining a web server's performance. This section provides a brief description of each.

- **Requests per second**

  The number of requests per second that the web server is able to sustain. Useful as a raw measure of performance. Remember that in measuring this, one can choose either to send a large number of requests down a single connection (using HTTP/1.1 persistent connections) or initiate a new connection for each request. Real world servers typically see between 5 and 20 requests per connection, so if you are using persistent connections, it is sensible to limit the number of requests sent down each connection to around this value.

- **Simultaneous connections**

  The number of simultaneous connections the web server can handle without errors is an important metric. Some real world servers have to deal with tens of thousands of simultaneous connections.

- **Data throughput**

  The number of bytes transferred per unit time. Usually useful only in conjunction with other metrics (it's easy to achieve high throughput figures when transferring a single large file – less easy to do so when trying to transfer numerous small files down a large number of concurrent connections).

- **The number of conforming connections**

  This metric is used in the SPECweb99 benchmark. The benchmark measures the number of simultaneous connections that the server can support; in order for a connection to be considered 'conformant' it must sustain a throughput of 320,000 bits per second.

5 Conducting a benchmark

This section explains how to conduct a variety of benchmarks against a server.

5.1 Simple static benchmarking using `httperf`

In a simple static benchmark, the client or clients make a large number of requests for the same file from the server, and the achieved throughput (typically the number of requests per second served) is measured.

This gives some measurement of the server's raw performance, but is very different from real world conditions; since only a single file is requested, the web server's caching algorithm isn't exercised at all.

The following command can be used to set up a static benchmark using '`httperf`':

```
httperf --server testhost.mydomain.com --uri /index.html \
        --num-conn 5000 --num-call 10 --rate 200 --timeout 5
```

This command instructs `httperf` to benchmark the URL http://testhost.mydomain.com/index.html. The option `--num-conn 5000` instructs `httperf` to attempt 5000 connections, `--num-call 10` causes it to issues 10 requests per connection, and `--rate 200` specifies that `httperf` should attempt to create 200 new connections every second (combined with `--num-call 10`, this results in a demanded request rate of 2000 requests per second). `--timeout 5` sets a five second timeout; any requests that aren't answered within this time will be reported as errors.

The results of such a test, run from a single client machine, are given below:

```
Maximum connect burst length: 1

Total: connections 4986 requests 39620 replies 39620 test-duration 29.294 s

Connection rate: 170.2 conn/s (5.9 ms/conn, <=1022 concurrent connections)
Connection time [ms]: min 922.1 avg 4346.7 max 8045.6 median 4414.5 stddev 1618.6
Connection time [ms]: connect 643.6
Connection length [replies/conn]: 10.000

Request rate: 1352.5 req/s (0.7 ms/req)
Request size [B]: 58.0

Reply rate [replies/s]: min 1195.0 avg 1344.7 max 1393.1 stddev 84.1 (5 samples)
Reply time [ms]: response 370.3 transfer 0.0
Reply size [B]: header 167.0 content 2048.0 footer 0.0 (total 2215.0)
Reply status: 1xx=0 2xx=39620 3xx=0 4xx=0 5xx=0

CPU time [s]: user 1.35 system 27.95 (user 4.6% system 95.4% total 100.0%)
Net I/O: 3002.2 KB/s (24.6*10^6 bps)

Errors: total 1038 client-timo 1024 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 14 addrunavail 0 ftab-full 0 other 0
```

The first thing to notice is that the request rate (1352.5 requests/sec) is less than the requested request rate (2000 requests/sec). There are two possible explanations for this - one is that the server is saturated, and is unable to sustain 2000 requests/sec; the other possibility is that the client is saturated. It is possible to distinguish between these possibilities by running the test a second time, this time using two client machines, with `--rate` set to 100 on each of them, and summing the results obtained. If the combined request rate is around 2000, then the client was saturated in the first test - if it's closer to

1300, then the server was saturated. (In the latter case, one should conduct at least one further test to confirm that the clients can happily sustain at least 1000 requests per second.)

In any event, for the server tested in the example above, it was in fact the server which was saturated. So we can deduce that the server under test is unable to sustain 2000 requests per second for a file of 2048 bytes. This is useful information, but it would be more useful to know at what number of requests per second the server became saturated, so that we know its limits. This is not always obvious. When the server is overloaded, queuing effects come into play, causing performance to drop off sharply. To establish the point at which the server becomes saturated, we would need to iteratively zero in on the limit.

It's usual, therefore, to conduct a number of tests, increasing the demanded number of requests per second each time, and recording the number of replies per second actually received. A graph of the results clearly defines the web server's behaviour, and similar graphs for different web servers may be used to compare performance.

The error information returned by `httperf` should always be examined. In this case we see that there were 1038 errors, with 1024 of these being client timeouts (caused by responses that took longer than five seconds to be returned to `httperf`). In addition, there were 14 'fd-unavail' errors; these are caused by `httperf` trying to open a new file descriptor, and receiving an error because the per process limit on the number of open files had been exceeded. Notice that the reported number of concurrent connections is close to 1024, which is the default open files limit per process on Linux; `httperf` use a file descriptor for each connection it has open concurrently. Refer to the section on Performance Tuning below for information on how to increase the file descriptor limits on Linux.

It is also a good idea to keep an eye on the network I/O statistics; you should watch these to ensure that the network itself isn't saturated. The 'Net I/O' line above shows a throughput of 24.6 Mbps, which is well within bounds for the 100 Mbps network the test was run on. If this were to get close to 80-90 Mbps, it would be sensible to increase the bandwidth between the client and server (by installing more network cards, or moving to gigabit Ethernet).

5.2 Static benchmarking with apachebench

Apachebench is a simple benchmarking utility, which allows the number of requests and the number of simultaneous connections to be varied. It is useful for performing quick tests of a server, but the statistics it generates are not as reliable as those produced by `httperf`. Apachebench tends to generate requests in bursts rather than smoothly over time, often triggering hysteresis[3].

The following is an example of a typical use of apachebench:

```
ab -n 1000 -c 200 http://www.test.com/foo/bar.html
```

The flags have the following meanings:

- `-n` Specifies the number of requests to make

[3] The lagging of an effect behind its cause, as when the change in magnetism of a body lags behind changes in the magnetic field.

- `-c` Specifies the number of concurrent connections to open

The following is an example of apachebench output:

```
This is ApacheBench, Version 1.3c <Revision: 1.41 > apache-1.3
Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright (c) 1998-1999 The Apache Group, http://www.apache.org/

Server Software:        Zeus/3.4
Server Hostname:        test.xenoclast.org
Server Port:            80

Document Path:          /
Document Length:        2049 bytes

Concurrency Level:      200
Time taken for tests:   24.537 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      2189000 bytes
HTML transferred:       2049000 bytes
Requests per second:    40.75
Transfer rate:          89.21 kb/s received

Connection Times (ms)
              min   avg   max
Connect:        0    28   201
Processing:   237  3997  9634
Total:        237  4025  9835
```

5.3 Automating static benchmarking with autobench

Autobench is an Open Source Perl script designed to assist in the automation of benchmarking with `httperf`. It runs `httperf` a number of times against the target server, increasing the requested request rate each time, and produces output in the form of a CSV or TSV file which can be imported directly into a spreadsheet for further analysis or graphing.

Autobench also enables the easy comparison of two different web servers - it can test two servers and amalgamate the results in the same table, enabling comparative graphs to be drawn quickly and easily.

5.3.1 Obtaining autobench

Autobench can be obtained from http://www.xenoclast.org/autobench/. Download the autobench tarball to your client machine, untar it, and run make; make install to install the autobench script.

5.3.2 Procedure for benchmarking with autobench

First, you should establish the range over which you wish to conduct the tests, and verify that neither the client machine nor the network become bottlenecks at the upper limit of this range.

Decide what values of `--num-call` and `--time-out` you are going to use during the tests. `--num-call` specifies the number of requests that will be sent down an individual connection - in practice this should be similar to the number of elements a browser might request from a typical HTML page – somewhere between 5 and 20 on average. `--time-out` should probably be somewhere between 5 and 10 seconds - research has indicated that most people will give up on a site if it takes more than 8 seconds to download a page.

Now estimate the highest number of requests per second that your test server or servers can sustain. Depending on the hardware and the web server

software you are using, this will probably be between 500 and 2000 requests per second. The number of requests per second demanded by `httperf` is the product of `--num-call` and `--rate`, so choose the value of `--rate` that will generate the number of requests per second you are after.

For example, if you were to assume that the web server would be saturated by 1500 requests per second, and you were sending 10 requests down each connection, then you would set `--rate` to 150. Now conduct an `httperf` test of the server using these values:

```
httperf --server testhost --uri/test/file.html --num-call 10 \
        --num-conn 5000 --timeout 5 --rate 150
```

Record the number of requests per second that the server achieves, and confirm that this is less than the demanded number of requests per second (if not, then the server isn't yet saturated, and you should increase the rate accordingly).

Check that your network isn't saturated by looking at the Net I/O that `httperf` reports.

Finally, check that the client machine isn't the bottleneck. The easiest way to do this is to run the same test from two client machines simultaneously with each requesting half the rate of the previous test. Sum the results, and check that they correspond to those you obtained the first time; if they are significantly better, then it is likely that the client machine was saturated during the first test.

Once you have established an upper limit for the tests, and have confirmed that neither the client machine or network become bottlenecked at this value, you are ready to run autobench to benchmark your server.

The first time you run autobench it will create a configuration file, autobench.conf, in your home directory. You should edit this file to set the values for `--num_call`, `--timeout` and the hostname and URI of your server.

You can then run autobench, using a command similar to:

```
autobench --single_host --file bench_results.tsv --low_rate 40 \
          --high_rate 200 --rate_step 20 [--quiet]
```

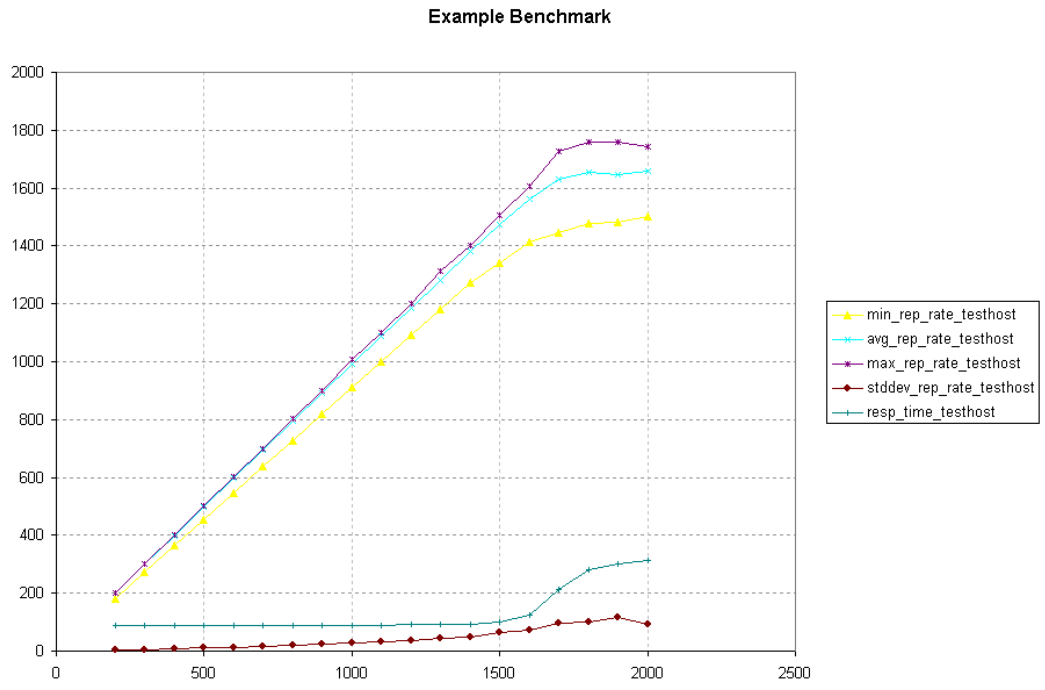

Figure 2 - Example graph drawn from autobench output

The optional `--quiet` option will prevent autobench from displaying the `httperf` output after each test. `--low_rate` and `--high_rate` set the lower and upper rate bounds for the test, and `--rate_step` sets the amount to increase the rate by on each iteration.

The autobench output can then be read into a spreadsheet or graphing package for graphing. A example graph drawn from autobench output is shown in figure 2.

5.4 Benchmarking using replayed sessions

Tools such as `httperf` also allow more sophisticated testing to be conducted using a series of requests for different pages, designed to simulate a real user's progress through a site. This type of testing is useful for estimating the actual performance that a web server will achieve in practice, and is particularly useful for testing dynamically generated sites where a significant proportion of the requests will be for pages generated through database access.

The principle is straightforward - a file is created containing the sequence of requests to be performed (often with specified delays (simulating read time) between individual requests). It is possible to generate such a file automatically from web logs.

This file is then passed to the benchmarking program, and the tester is able to specify how many concurrent sessions should be attempted. The results presented depend on the benchmarking software in use - `httperf` returns data similar to that obtained from benchmarking a single URL, with the addition of some session specific information.

An example of the `httperf` command to generate such a benchmark follows:

```
httperf --server www.test.com --wsesslog 1000,2,session.log \
        --max-piped-calls 5 --rate 20
```

The file `session.log` contains the list of URLs - see below for a discussion of its contents. The first two arguments to the `--wsesslog` option specify the total number of sessions to attempt, and the second is the default delay (in seconds) between each request in a session (if none is specified in `session.log` itself).

`--max-piped-calls` specifies the maximum number of requests which will be sent down an individual TCP connection (using HTTP 1.1 persistent connections), and `--rate` is the rate at which new sessions will be created (in sessions per second).

The man page for `httperf` specifies the format of the `session.log` file in detail; a brief overview is given here. Each non-blank line in the file is the relative path (URI) to be requested from the server; optionally, it may also specify the HTTP method to be used, and a delay (`httperf` will wait this long before requesting the next URL from the server). If a line begins with whitespace then it will be requested as part of a burst of requests sent out immediately after the last request for a line which did not begin with whitespace. A blank line separates one session from the next, and lines beginning with '#' are comments.

An example will make things clearer (the example below has been taken from the `httperf` man page):

```
# session 1 definition (this is a comment)
/foo.html think=2.0
     /pict1.gif
     /pict2.gif
/foo2.html method=POST contents='Post data'
     /pict3.gif
     /pict4.gif

# session 2 definition
/foo3.html method=POST contents="Multiline\ndata"
/foo4.html method=HEAD
```

The above file specifies two sessions. The first session begins with a request for `foo.html`, immediately followed by requests for `/pict1.gif` and `/pict2.gif` (in a burst - presumably these are images forming part of the same page); `httperf` will then wait for the 'think time' of 2 seconds, before issuing a POST request for `foo2.html`, followed immediately by its associated images. The second session then begins, and comprises a POST request and a HEAD request separated by the default think time (the second argument to the `--wsesslog` option).

The program `sesslog` supplied in the autobench package can be used to generate session log files for use with `httperf` from an NCSA Common Log Format, or Combined Log Format log file. See the `sesslog` man page for more details.

6 Performance tuning

6.1 General tuning tips

The essentials of tuning Apache for optimum performance are discussed in depth at http://httpd.apache.org/docs/misc/perf-tuning.html. Some of the advice given here is definitely Apache-specific, but parts (or at least the principles behind them) are more generally applicable.

Some simple rules follow:

- Only enable those modules you absolutely require. Each additional module implies additional processing overhead, even though you may not be making active use of the module on your site.

- Where possible, use static content instead of dynamic content. If you are generating weather reports that are updated once an hour, then it is better to write a program that generates a static file once an hour than to have users run a CGI to generate the report on the fly.

- When choosing an API for your dynamic applications, choose the fastest and most appropriate one available. CGI may be easy to program for, but it forks a process for each request - usually an expensive and unnecessary procedure. FastCGI is a better alternative, as is Apache's mod_perl - both provide persistency and can increase performance significantly. With Zeus Web Server, in-process ISAPI provides tight integration with the server and is therefore extremely efficient and high performance (even when run out-of-process), though can be more difficult to write for. More information on the APIs supported by Zeus Web Server and examples of their use can be found at http://support.zeus.com/doc/api/.

- If your server performance is a critical issue for you (or you just want to get the maximum number of bangs for your bucks), then choose a high performance web server instead of Apache. Apache's process model does not stand up to increased load; it uses an individual process for each connection, leading to massive context switching when you have a large number of simultaneous connections. Furthermore, Apache is limited to 256 simultaneous connections unless recompiled with a different value for `MaxClients`[4]. Single process per CPU servers such as Zeus and Boa are significantly faster; Zeus has held the SPECweb96 and 99 records for the majority of the last five years[5]. Furthermore, recent studies[6] have shown that Apache 2, whether using the same process model as Apache 1, or using the new multithreaded architecture, offers little performance advantage when scaled to large numbers of simultaneous connections. For enterprise web sites, therefore, or those looking to service high numbers of concurrent clients, Zeus Web Server remains the highest performance web server available.

[4] Arguably this limit was imposed for a reason, so increasing the value is likely not to yield the benefits of linear scalability that web servers based on a single-process model can achieve.
[5] http://www.spec.org/osg/web99/results/web99.html
[6] Please refer to the 'Comparing High Performance Web Servers' which can be found at http://support.zeus.com/doc/tech/zws_comparison.pdf

6.2 Tuning file descriptor limits on Linux

Linux limits the number of file descriptors that any one process may open[7]; the default limits are 1024 per process. These limits can prevent optimum performance of both benchmarking clients (such as `httperf` and Apachebench) and of the web servers themselves (Apache is not affected, since it uses a process per connection, but single process web servers such as Zeus use a file descriptor per connection, and so can easily fall foul of the default limit).

The open file limit is one of the limits that can be tuned with the `ulimit` command. The command `ulimit -aS` displays the current limit, and `ulimit -aH` displays the hard limit. The hard limit can be lowered below and raised above the limit displayed by the superuser, but not beyond the limit imposed by the kernel (which cannot be increased without tuning kernel parameters in `/proc`).

The following is an example of the output of `ulimit -aH`. You can see that the current shell (and its children) is restricted to 1024 open file descriptors.

```
core file size (blocks) unlimited
data seg size (kbytes) unlimited
file size (blocks) unlimited
max locked memory (kbytes) unlimited
max memory size (kbytes) unlimited
open files 1024
pipe size (512 bytes) 8
stack size (kbytes) unlimited
cpu time (seconds) unlimited
max user processes 4094
virtual memory (kbytes) unlimited
```

6.2.1 Increasing the file descriptor limit

The file descriptor limit can be increased using the following procedure:

1. Edit `/etc/security/limits.conf` and add the lines:

```
*       soft    nofile 1024
*       hard    nofile 65535
```

2. Edit `/etc/pam.d/login`, adding the line:

```
session required /lib/security/pam_limits.so
```

3. The system file descriptor limit is set in `/proc/sys/fs/file-max`. The following command will increase the limit to 65535:

```
echo 65535 > /proc/sys/fs/file-max
```

4. You should then be able to increase the file descriptor limits using:

```
ulimit -n unlimited
```

The above command will set the limits to the hard limit specified in `/etc/security/limits.conf`.

Note that you may need to log out and back in again before the changes take effect.

[7] As do other operating systems. Refer to the 'Running SPECweb99 with Zeus' white paper available at http://support.zeus.com/doc/tech/SPECweb99.pdf

For further information on increasing the system file descriptor limit for non-Linux versions of UNIX, please refer to our FAQ entry on increasing this limit: http://support.zeus.com/faq/zws/v4/entries/filedescriptors_shortage.html.

7 Useful benchmarking tools

The following are some of the commonly used benchmarking tools:

- **httperf**

  A useful tool, capable of doing everything from simple requests for a single file to simulating client sessions. Available from: http://www.hpl.hp.com/personal/David_Mosberger/httperf.html

- **Apachebench**

  A simple benchmarking tool, distributed with the Apache Web Server. The source is in the file `src/support/ab.c` in the Apache source tree.

- **Autobench**

  A Perl script for automating a series of `httperf` tests against the same server (or a pair of servers). It parses the output of `httperf`, and summarises the results of the tests in a CSV or TSV file that can be imported directly into a spreadsheet for further analysis/graphing. Available from: http://www.xenoclast.org/autobench/

8 References and resources

1. 'The Linux HTTP Benchmarking HOWTO', Julian Midgley's original paper on Linux Benchmarking: http://www.xenoclast.org/doc/benchmark/

2. 'httperf - A Tool for Measuring Web Server Performance', David Mosberger's paper on `httperf`: http://www.hpl.hp.com/personal/David_Mosberger/httperf/

3. The SPECweb99 benchmark: http://www.spec.org/osg/web99/

4. 'Running SPECweb99 with Zeus', a white paper describing the SPECweb99 benchmark and how to optimise Zeus Web Server to perform well in the tests: http://support.zeus.com/doc/tech/SPECweb99.pdf

5. 'Comparing High Performance Web Servers', a study by Zeus Technology in to the relative performances of high performance web servers such as Zeus Web Server, Apache 2.0, Tux and Chromium X15: http://support.zeus.com/doc/tech/zws_comparison.pdf

6. 'Web Server Performance and Scalability', a white paper from Zeus Technology, useful in particular for its discussion of Class-Based Queuing, a method of benchmarking real-world conditions, such as simulating multiple, simultaneous, slow modem connections using the `ratelimit` tool: http://support.zeus.com/doc/tech/perf.pdf

7. 'Improving Web Server Performance with Zeus Load Balancer', a study by Zeus in to how the performance of a cluster of Apache web servers are susceptible to overload due to slow modem connections, and how a load balancing solution can improve response times and overall performance: http://support.zeus.com/doc/tech/lbperf.pdf

8. Apache Performance Notes: http://httpd.apache.org/docs/misc/perf-tuning.html

9. Linux Performance Tuning website: http://linuxperf.nl.linux.org/

9 Contact Information

9.1 Office Contact Details

Zeus Technology Ltd.
Zeus House
Cowley Road
Cambridge
CB4 0ZT
UK

Tel.: +44 (0)1223 525000
Fax: +44 (0)1223 525100

9.2 Zeus products and services

For more information about our products, please visit:

http://www.zeus.com/products/

To find out more about our consultancy, technical support and training services, please visit:

http://www.zeus.com/services/