Effective tuning for popular Web sites

Fast as an arrow

There is nothing more annoying than an Apache server that fails to reply, or responds too slowly. Webmasters find themselves increasingly under pressure to get things running smoothly. The possible causes are many and easily as variegated as the optimization strategies designed to resolve these bottlenecks.

BY CHRISTIAN KRUSE

When users start to complain about slow download times, it is time for the Webmaster of the site in question to start doing something about it. There are many possible causes: network bottlenecks, server overloads, or simply too high a volume of traffic in user requests. This article explains the different ways of minimizing your Apache Web server's response and transfer times, and also shows you how to test the efficacy of your tuning strategy using simple benchmarks.

Network and Router Issues

Rather than attempting to cure the symptoms, admins should instead look for a systematic approach. The most probable source of the problem is a network bottleneck, possibly caused by a router or a line having gone down. *tracepath* or *traceroute* are both useful diagnostic tools that check where a connection goes, and how long it takes to do so, by reducing the Time to Live and waiting for the routers to respond with a *TIME_EXCEEDED* error.

The Time to Live field does not contain a temporal value, but instead specifies the number of hops that a packet is allowed to travel across, thus allowing admins to understand and trace network connectivity problems. Listing 1 shows an example: the router at hop 4, *ar-essen2.g-win.dfn.de*, shows a fairly slow response, but as 0.2 seconds are not really all that critical, it must be assumed that the bottleneck is occurring elsewhere.

Listing 1: Excerpt from *tracepath*

```
01 ckruse@shine:~ $ tracepath www.defunced.de
02 1?: [LOCALHOST]      pmtu 1500
03 1:  fogg.defunced.de (192.168.1.1)                          2.625ms
04 2:  10.3.11.1 (10.3.11.1)                                   5.109ms
05 3:  gwin-gw-gig00-112.HRZ.Uni-Dortmund.DE (129.217.129.190)
   13.403ms
06 4:  ar-essen2.g-win.dfn.de (188.1.44.33)
   222.119ms
07 5:  cr-essen1-ge4-0.g-win.dfn.de (188.1.86.1)               12.839ms
08 6:  cr-frankfurt1-po8-1.g-win.dfn.de (188.1.18.89)          24.964ms
09 7:  188.1.80.42 (188.1.80.42)                               24.872ms
10 8:  gi-0-3-ffm2.noris.net (80.81.192.88)                    26.466ms
11 9:  ge0-2-151-nbg5.noris.net (62.128.0.209)                 40.772ms
12 10:  no.gi-5-1.RS8K1.RZ2.hetzner.de (213.133.96.25)         31.012ms
13 11:  et-1-16.RS3K1.RZ2.hetzner.de (213.133.96.230)          32.433ms
14 12:  srv001.occuris.de (213.133.103.124)                    38.802ms
   reached
15 Resume: pmtu 1500 hops 12 back 12
```

Table 1: Essential Apache Modules

Module Name	Module Name
env_module	alias_module
config_log_	rewrite_module
module	access_module
mime_module	auth_module
includes_module	setenvif_module
autoindex_module	headers_module
dir_module	expires_module
cgi_module	php4_module
action_module	gzip_module

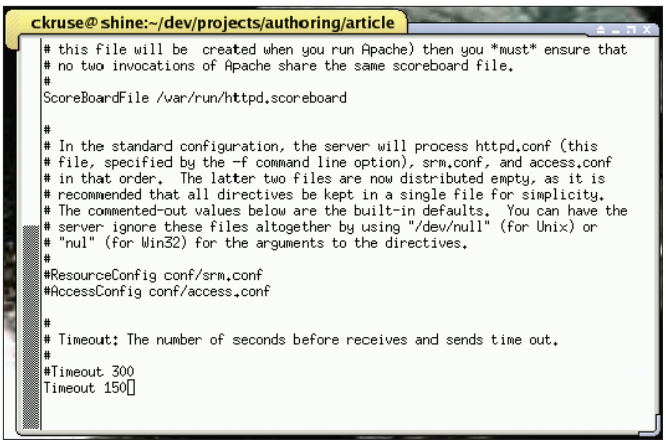

Figure 1: Changing the connection *Timeout* in the Apache configuration file can boost performance

The next step is to analyze the server load. The best way to do this is to check the server during normal production operations, using SSH, for example. The *top* and *ps* tools provide a few initial clues, *uptime* tells you something about the current server load. And the Apachetop tool reviewed in last month's Linux Magazine [3] also adds an extremely useful contribution.

Unfortunately, there is no guarantee that a bottleneck will be permanent; in fact, sporadic interruptions are quite common. The answer to this dilemma is to use a program such as MRTG [4] to log server load over a longer period of time. You can take a look at a sample MRTG installation at [5].

Analyzing the MRTG graphs will typically reveal that the server is either overloaded, or the traffic volumes are too high. The load average and processor load graphs are a good indicator of the former case. The graph with the number of active processes provides a good clue as to the identity of the miscreant. If the number of processes remains constant, you should start looking for a program that has gone wild.

If the number of processes varies, you should check for CGI scripts hogging resources. The Apache Access log and the process list (*top*) for heavy load periods are useful in this case. They tell you which URLs have been requested most often. This kind of information allows admins to locate the script or program responsible for the heavy load. A high traffic volume will leave a different footprint. The server's load graphs will reveal consistently low values, but the network traffic values will be noticeably too high.

Is Your Server Overloaded?

The fact that your Web server is overloaded, is not necessarily anything to worry about. Large and popular documents, such as SELFHTML, always tend to provoke a heavy load. And in some cases, there is very little an admin can do about it. The first thing to consider, is recompiling the software with appropriate compiler flags set, as many of the popular distributions compile the Apache packet to run on a 386 CPU for compatibility reasons.

Attentive sysadmins will tend to compile mission-critical software like Apache themselves, ensuring that the appropriate compiler flags are set when they do so. If you have an Athlon server (see the "Test Environment" box below), you might like to use the same flags as yours truly:

```
CFLAGS="-march=athlon ➥
-fexpensive-optimizations -O3"
```

These flags tell the compiler to create machine code that will run only on Athlon CPUs. This allows the compiler to use optimization methods that cost a lot of time. The optimization level is set to three. But it should be obvious that these compiler flags are no replacement for manually optimizing the algorithm in question.

Test Environment

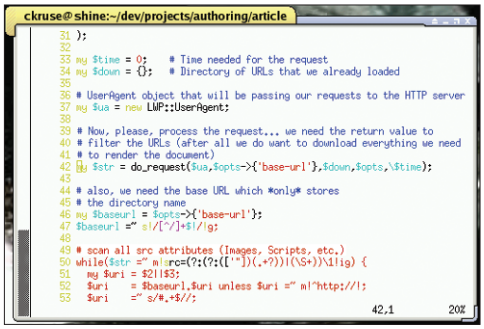

Figure 2: Excerpt from the author's Perl benchmark script, which you can download from the Linux Magazine Web site

Benchmark tests are used to determine the efficacy of optimization approaches. It makes sense to measure the time the server takes between receiving a request and serving up the response. To test this, I set up two PCs and attached them to an internal 100 Mbit Ethernet wire. My server PC (an AMD Athlon 600 with 64 MBytes of SD-RAM) runs FreeBSD 4.6 and Apache 1.3.27.

The Web server characteristics described in this article also apply to Apache 2, unless a specific reference to the contrary is made. To simulate ISDN speeds, and thus make differences in response times more visible, I also installed Mod_bandwidth.

The client machine (an AMD Duron 800 with 312 MBytes SD-RAM) runs Gentoo Linux 1.4 with a Kernel 2.4.22, and Perl 5.8.0. My major benchmarking tool is a Perl script I wrote myself; you can download the script at [1]. Figure 2 shows an excerpt. The script downloads the URL passed to it as an argument, along with any references needed to render the results. The script measures the time required to do this (see Figure 3).

The second benchmark tool used here is supplied with the Apache server: *ab* [2]. It is mainly designed for load analysis. To allow this, the tool simulates a large number of simultaneous requests, and measures the number of requests per second handled by the server and how long Apache takes to handle them.

Figure 3: The author's benchmark script in action. It downloads the URL passed to it and measures the time required to do so

Many Apache modules create temporary files while handling requests; PHP creates disposable files to facilitate session management. This is why the next optimization step moves the */tmp* directory to a memory disk, typically with a size of around 128 MBytes. Although Linux does use efficient caching strategies, the RAM disk will often ensure better server response times.

Tuning the Apache HTTPD

Apache itself offers the greatest potential for optimization. Although the basic configuration is fairly useful, some distributions tend to load too many modules. For normal operations. Only the modules detailed in Table 1 are really necessary. Removing any superfluous modules will reduce the size of the HTTPD process and the time needed to handle a request.

Standard CGIs: Performance Killers

CGI scripts running on your Web server need a lot of careful attention. These indisputably useful programs, often cause noticeable performance issues, as launching them requires the server to load bulky interpreter binaries.

I have removed any references to existing sites to protect the innocent, but the following example is a real-life case study: A Web server running somewhere in Germany has a very large and complex Perl CGI script. As the server was hit by load issues, the Web team responsible for it, developed a new routine for the script that checked the Load Average, and killed the CGI process in case of overloading. This proved to be a highly inefficient approach: the peak load was still far too high, and launching the interpreter and interpreting the script consumed masses of resources.

The next step the programmers took, was to write a 804 byte binary in C, that made a decision based on performance. If the load was too high, the binary issued an error message, if not, the binary launched the original CGI script. This proved to be effective, as the system handled peak loads far more gracefully after this point.

Now if the programmers at this site had taken this a step further and written a *mod_loadavg* Apache module, they could have reduced the load even further. Modules of this type, remove the need to call the interpreter. There are Apache modules for almost every language: *mod_perl*, *mod_php*, *mod_ruby*, *mod_fastcgi* and so on.

Background: For each request it handles, Apache needs to check every single active module, to find out if it is required for the request. But be careful if you have Apache 2: the *config_log_module* is called *log_config_module* in this case.

Experience shows that reducing the value of the connection timeout in the *Timeout* directive makes a big difference – after all, do you really need to wait 300 seconds for a new packet and/or request? Sluggish requests simply tie up the child process preventing it from accepting new connections that Apache could have handled in the meantime. A timeout of between 120 and 150 seconds should be fine, even for slower Internet connections (see Figure 1).

Admins should also ensure that *KeepAlive* is set to *On*. This directive enables or disables Keep Alive requests. Keep Alive requests allow clients to use the connection for multiple requests. Otherwise the client needs to set up a new connection for each new request – and this entails going through a complete handshake, and wasting three packets. Assuming 50 requests per second, that would be 9000 packets per minute just to set up connections. Thus, assuming that three clients generate these 50 requests, setting *KeepAlive* to *On* would reduce the number of connections per second to three, and thus the number of packets for creating these connections to 540.

Reducing Timeouts

The *KeepAliveTimeout* should also be a fairly low value: 15 seconds is typically fine. The *MaxSpareServers* directive restricts the number of idle *httpd* processes. That is, the number of *httpd* processes drops during periods with light loads.

This also means that Apache will need to fork a few new *httpd* processes before it can handle a heavier load. In the case of services plagued by load issues, admins are well advised to comment out the directive, and monitor the results, to see if the server's responsiveness improves. Make sure that you leave *MaxSpareServers* set, unless your server is continually under fire, as the number of *httpd* processes will otherwise increase continually.

The *MinSpareServers* directive is a different thing altogether. It specifies the number of idle processes the server should have. These processes are a kind of buffer for peak loads. Note that this behavior depends on your choice of MPM module [6, 7], for Apache 2. If you use the traditional Prefork model, there is no change. But if you use the Worker MPM, the directive names in *MaxSpareThreads* and *MinSpareThreads* are different.

In this case the value refers to the number of threads per Apache process. Generally speaking, Apache 2 should have less trouble creating new threads

Listing 2: Results prior to tuning

```
01 Concurrency Level:      10
02 Time taken for tests:   2.842457 seconds
03 Complete requests:      100
04 Failed requests:        0
05 Write errors:           0
06 Total transferred:      11023620 bytes
07 HTML transferred:       10994986 bytes
08 Requests per second:    35.18 [#/sec] (mean)
09 Time per request:       284.246 [ms] (mean)
10 Time per request:       28.425 [ms] (mean, across all concurrent
   requests)
11 Transfer rate:          3787.22 [Kbytes/sec] received
12
13 Connection Times (ms)
14               min  mean[+/-sd] median   max
15 Connect:        0    0   1.0      0       5
16 Processing:    39  278 342.6    105    1012
17 Waiting:      -83  -22  19.1    -21       1
18 Total:         39  278 342.8    106    1012
```

Listing 3: Results after tuning

```
01 Concurrency Level:      10
02 Time taken for tests:   1.58400 seconds
03 Complete requests:      100
04 Failed requests:        0
05 Write errors:           0
06 Total transferred:      11290258 bytes
07 HTML transferred:       11261068 bytes
08 Requests per second:    94.48 [#/sec] (mean)
09 Time per request:       105.840 [ms] (mean)
10 Time per request:       10.584 [ms] (mean, across all concurrent
   requests)
11 Transfer rate:          10416.67 [Kbytes/sec] received
12
13 Connection Times (ms)
14               min  mean[+/-sd] median   max
15 Connect:        0    0   0.3      0       3
16 Processing:    34   98  21.1     95     155
17 Waiting:     -102  -46  17.9    -42       0
18 Total:         34   98  21.0     95     155
```

than new processes, as a thread does not have its own memory area, which Apache would need to copy. Also, a thread does not have a process ID of its own, and instead shares a common ID with the other threads.

Finding a Compromise

The next directive that can affect performance is *MaxRequestsPerChild*, which terminates a child process after a certain number of requests and forks a new child process. Unfortunately, some libraries and modules are susceptible to memory leaks, and this is why you need the directive. But you should test your own system, to find out whether it is affected, before enabling the directive.

If you notice that the number of Apache processes has grown rapidly during a period of heavy load (you can use a tool like *top* to do this), you have very little alternative but to enable the directive. In this case you should set a high value. The value itself will depend on your system configuration and is a question of trial and error. The value is a compromise between performance and memory resources.

The *HostnameLookups* directive is critical. If you are looking for speed, it has to be set to *Off*. Enabling hostname lookups would force a reverse DNS lookup for each request. This in turn causes at least one DNS request and would probably lead to more DNS requests to nameservers at remote locations on the network. This not only results in a lot of traffic, but also wastes a lot of time.

How Effective is Apache Tuning?

It is hard to quantify the effect of optimization measures. The script at [1] is not useful for testing this, as it does not place any load on the server. However, the Apache benchmark *ab* (Version 2, [2]) should supply the results you need (see the "Test Environment" box). Run the benchmark before you start tuning:

```
ckruse@shine:~ $ ab2 -n 100 ➥
-c 10 http://rain/article/
```

The program initiates 100 request phases, launching ten requests simultaneously in each phase. Listing 2 contains an excerpt from the results, showing the relevant values. The *Requests per second* and *Time per request* figures are interesting: approximately 35 requests per second and 28 milliseconds per request on average. Following this, fine tune your server and repeat the benchmark with the same pattern, as shown in Listing 3. It looks like the tuning steps have been successful in this case: 100 percent. A cautious estimate would be that the site should be able to handle twice as many requests as previously.

Optimization at HTTP Level

Optimization at HTTP level is mainly concerned with reducing the amount of traffic and the number of requests. This can be achieved by using caching and conditional headers. Caching headers tell the User Agent (UA) that it does not need to re-request certain content for a certain period of time. This is useful for static HTML or CSS files. The *Expires* header indicates the point in time when a document should be regarded as obsolete and thus re-requested. See Listing 4.

If you want to tell the UA – whether this be a browser, a proxy, or another client – that the document is obsolete immediately after Apache has served it up, and thus needs to be re-requested every time, set the value of this field to the value in the *Date* header. Values such as *now* or *0* are invalid, but they are accepted to improve error tolerance and equated with an obsolete value. It is then up to the client to honor this information, or not.

The *Cache-Control* header is more strict. It specifies caching instructions – that any RFC conform HTTP client must honor. The syntax for *Cache-Control* is slightly different from *Expires*. The valid-

Listing 4: Setting the *Expires* and *Cache-Control* headers

```
01 ExpiresActive On
02 ExpiresByType text/html "access plus 1 month"
03 ExpiresByType text/css "access plus 6 month"
04 ExpiresByType text/javascript "access plus 6 month"
05 ExpiresByType image/gif "access plus 6 month"
06 ExpiresByType image/jpeg "access plus 6 month"
07 ExpiresByType image/png "access plus 6 month"
08
09 <Files ~ "\.(js|css|gif|jpe?g|png)$">
10  Header append Cache-Control "public"
11 </Files>
```

ity of a document is specified by *max-age=value*. The value is a validity period in seconds from the point in time shown in the *Date* field. Specifying a value of *0*, indicates that the document should not be cached at all, however, this setting is not recommended.

Supporting Ignorance

The *public* attribute in the Cache-Control header can provide a speed boost under certain circumstances, as it provides proxy caches with more freedom choice. If a proxy with this instruction receives a request that should not be cached – for example an authenticated request that uses HTTP-Auth, it can ignore this and cache anyway.

Both headers require the *mod_expires* and *mod_headers* modules. Admins can use *mod_expires* to specify when a document will expire. However, the module is not aware of the *public* attribute, and this means you also need the *mod _headers* module, to allow image, Javascript, and CSS files to leverage the *public* attribute. Listing 4 shows an example of a HTTP request to a server configured in this way. The request shown in Listing 5 should be cached for one month by the UA, without the UA having to contact the server in the meantime.

However, experience shows that caching UAs tend to cache results for a while, typically about two days, and then use a conditional header to ask whether the cached version is up to date, or if they should request an update. The *If* header group is used to do this. Operations such as *If-Match*, *If-Modified-Since*, *If-None-Match*, *If-Range*, and *If-Unmodified-Since* are permissible.

However, in the wild, you will normally not see any of these apart from *If-None-Match* (Opera Version 7, or later, and Mozilla), and *If-Modified-Since* (Mozilla, Opera Version 7, and Internet Explorer 5.5 or later). The other headers are explained at [8].

Conditional Headers

Both conditional headers refer to fields that the server has served up due to a prior request. *If-Modified-Since* refers to the *Last-Modified* field, which contains the last change date for the document. The *If-None-Match* header refers to *ETag*, a checksum for the document.

The User Agent stores these values, and formulates a request accordingly, as you can see in Listing 6. As you will note, the server simply responds with *304 Not Modified* and a few other headers, but it does not serve up any content.

There are few caveats with respect to CGI scripts here: the CGI script should send a *Last-Modified* or *ETag* header to allow the browser to use its conditional get algorithms. Also, the script will need to parse the header. And it is required to format the date appropriately (typically as the number of seconds elapsed since 1970), and then check whether the version specified by the *If-Modified-Since* header is obsolete.

This also applies to the *If-None-Match* header: the script evaluates the checksum and thus ascertains whether the version described in the header is obsolete. Hashing algorithms such as MD5 are useful as checksum algorithms. The CGI environment variables, *HTTP_IF_MODIFIED_SINCE* and *HTTP_IF_NONE_MATCH*, provide access to the values in both headers.

Listing 5: Sample HTTP Request

```
01 ckruse@shine:~ $ telnet rain
   80
02 Trying 192.168.1.3...
03 Connected to rain.defunced.de.
04 Escape character is '^]'
05 GET /article/ HTTP/1.1
06 Host: rain
07 Connection: close
08
09 HTTP/1.1 200 OK
10 Date: Sun, 09 Nov 2003
   19:52:04 GMT
11 Server: Apache/1.3.27 (Unix)
   PHP/4.2.3 AuthMySQL/2.20
12 Cache-Control: max-age=2592000
13 Expires: Tue, 09 Dec 2003
   19:52:04 GMT
14 Last-Modified: Tue, 04 Nov
   2003 23:50:42 GMT
15 ETag: "a3a068-1a9d9-3fa83b52"
16 Accept-Ranges: bytes
17 Content-Length: 109017
18 Connection: close
19 Content-Type: text/html
20
21 <I>HTML contents<I>
```

20 Times the Performance

The script ([1], see the "Test Environment" box) can be used to test the efficiency of the tuning measures described thus far:

```
ckruse@shine:~ $ ./measure.pl ➥
--base-url http://rain/article/
Getting http://rain/article/...
[...]
Time elapsed: 4.642203 seconds
```

Enabling the conditional header means that the whole request can be processed in less than a second.

```
ckruse@shine:~ $ ./measure.pl ➥
--send-if-modified-since ➥
--send-if-none-match --base-url➥
 http://rain/article/
Getting http://rain/article/...
[...]
Time elapsed: 0.181876 seconds
```

Compression Techniques

Content encoding is the other method of tuning HTTP. This requires the UA to send the *Accept-Encoding* header when requesting a document. The header specifies the type of processing acceptable for the content, and can contain values such as *gzip* [9] or *compress*. Both values specify algorithms that compress the document content before the server serves it up, and then expand it again clientside.

The method saves a lot of traffic – up to 90 percent. But there is a major downside: first of all, not all UAs can handle it – Netscape 4 sends *Accept-Encoding: gzip*, although the browser does not handle this method correctly. Secondly, compression will impact your server's performance, as the Apache server needs to compress each document before serving it up. And thirdly, the technique can suffer from a lack of acceptance at various places en route: many content filters simply remove the Accept-Encoding header.

And most proxies do not react kindly to the *Vary* header that compression requires. *Vary* lists the headers that depend on this variant of the request. Older versions of the Squid HTTP proxy disable any kind of caching and re-request the document every time – this is not really in the Web site operator's best

interests. Thus, you should carefully consider the pros and cons of using *mod_gzip* with respect to your target group.

Tuning at HTML Level

HTML files provide the greatest optimization potential. A great deal of HTML code – especially in the commercial sector – is provided by HTML designers using WYSIWYG editors like Dreamweaver. And that is exactly what the code looks like. You can often shorten the code by up to 50 percent, without affecting the rendered version, of course. Generally speaking, it makes sense to remove as many formatting sequences as possible from the HTML files, and place them in CSS files instead; this is something that WYSIWYG editors actually can do – much in contrast to their users.

In contrast to embedded formatting sequences, CSS files only need to be loaded once, and this means that the HTTP server only needs to transmit the formatting sequences once. Also, try to avoid superfluous nesting of tables, such as generated by Dreamweaver. Web designers should not accept any code generated by programs of this kind, as is, but instead use HTML-Tidy [10] for example, to tidy up the code.

Listing 6: A conditional GET

```
01 ckruse@shine:~ $ telnet rain
   80
02 Trying 192.168.1.3...
03 Connected to rain.defunced.de.
04 Escape character is '^]'.
05 GET /article/ HTTP/1.1
06 Host: rain
07 Connection: close
08 If-Modified-Since: Tue, 04 Nov
   2003 23:50:42 GMT
09 If-None-Match: "a3a068-1a9d9-
   3fa83b52"
10
11 HTTP/1.1 304 Not Modified
12 Date: Sun, 09 Nov 2003
   20:28:43 GMT
13 Server: Apache/1.3.27 (Unix)
   PHP/4.2.3 AuthMySQL/2.20
14 Connection: close
15 ETag: "a3a068-1a9d9-3fa83b52"
16 Expires: Tue, 09 Dec 2003
   20:28:43 GMT
17 Cache-Control: max-age=2592000
18
19 Connection closed by foreign
   host.
```

Images and Graphs

Web designers can often save a lot of traffic by paying attention to images. The major formats, which nearly all browsers support, are gif, jpeg and PNG. Simple line drawings and cartoons are the domain of gif, whereas jpeg is used for photos and high-color images; however the jpeg algorithm does impact image quality.

The gif format is particularly unsuitable as its color palette is restricted to 256 colors. Any other colors are created by dithering, and this tends to inflate the file size unnecessarily. If quality is your major concern, the compressed PNG format is your best option, although the file size may be slightly larger than that of a comparable jpeg file.

You often see files published at a very high resolution, and scaled down on the HTML page using *width* and *height*. There are two disadvantages to this approach: for one thing, it means transferring unnecessarily large image files across the wire. And for another, it impacts the viewing quality in the Web browser, as typical rendering engines will produce far inferior results to graphics programs when reducing image files.

Today's Web designers do not need to go to the opposite extreme, as used to be the case in the age of 14,400 baud modems and expensive Web space. People used to scale images down, and allow the browser to scale them back up again. But again, typical clients perform this task inadequately. It makes sense to choose a scale that allows image files to be viewed at their original size – this is a perfect compromise between file size and quality.

Spaces and Newlines

As HTML is a free format language to a larger extent, you can remove any syntactically superfluous space and newline characters. The UA does not really care if your HTML code is nicely formatted or haywire. This provides more scope for reducing traffic. This is similar to converting Windows newlines to Unix newlines, which occupy one byte rather than two. Any modern editor should be able to handle this task.

Apache 2 users can enable the *mod_blanks* module to handle this task. Users of other versions can achieve a similar effect by using two separate versions: one for development, and another for publishing. After each change in the developer file, simply run a short script (see [11] for an example) against the file. This removes any space and newline characters from the final version.

If you apply all of these tuning methods to your site, you can look forward to better performance for medium to high load servers. But you will need to perform before and after benchmarks, using the techniques discussed in this article, to find out exactly how big the benefit is for your system. ■

INFO

[1] The author's benchmarking script: *http://www.linux-magazin.de/Service/Listings/2004/01/Apache-Tuning*

[2] Apache benchmarking tool: *http://httpd.apache.org/docs/programs/ab.html*

[3] Charly Kühnast, "The Sysadmin's Daily Grind: Apache Tracking": Linux Magazine, Issue 38, p 54

[4] Multi Router Traffic Grapher: *http://people.ee.ethz.ch/~oetiker/webtools/mrtg/*

[5] Sample Installation of MRTG: *http://www.defunced.de/mrtg/*

[6] Multiprocessing Module: *http://httpd.apache.org/docs-2.0/mpm.html*

[7] T Grahammer, "Apache 2.0 – Rules of Succession": Linux Magazine, Issue 24, October 2002, p29

[8] HTTP RFC: *ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt*

[9] U. Keil, "Dynamic Webpage Compression with Mod_gzip and Apache": Linux Magazine, Issue 24, October 2002, p26

[10] HTML-Tidy: *http://sourceforge.net/projects/tidy/*

[11] Script for removing spaces: *http://www.defunced.de/blanks.pl.gz*

THE AUTHOR

Christian Kruse studies computer science in Dortmund. As a freelance administrator, he is responsible for a number of Linux and FreeBSD servers.