Performance−tuning Apache

Thomas Wouters <thomas@xs4all.net>
XS4ALL

Table of Contents

1.3 Tweaking Apache Configuration

In general it can be said that features cost speed. The fewer features that are actually enabled, the more quickly Apache can handle requests. In some situations, however, it is hard to figure out whether you need a specific feature or not, or how hard it would be to do without it. The following section covers some optional modules and configuration directives that can positively and negatively affect speed. Most of these, and some more, can also be found in the Performance Notes section of the Apache manual.

1.3.1 mod_status and ExtendedStatus

The standard Apache module *mod_status* enables a configuration directive `ExtendedStatus`, which causes Apache to maintain extra information on the requests Apache is handling and the performance of it. While this information can be useful for benchmarking or locating bottlenecks, it also takes extra effort to gather it, and thus reduces performance.

ExtendedStatus is off by default, and can be switched on or off using the server−global `ExtendedStatus` directive.

1.3.2 Hostname Lookups

Apache has the ability to do a DNS lookup for each request that comes in, in order to log the hostname of the remote host rather than the ipaddress, and to pass it to any CGI script that gets executed. DNS lookups can be quite time−consuming, however, especially if the name−servers for the ipaddress fail to respond. For the logfiles, hostname lookups are better done by a separate logfile−parser. For CGI scripts, as far as they really need the resolved hostname, most programming languages have a way to resolve it themselves.

Apache also allows turning hostname lookups on or off selectively, by including the `HostnameLookup` directive in File or Directory statements. Note that there is no real performance benefit in making CGI scripts do the resolving themselves, except by reducing the number of lookups done.

1.3.3 Symlinks

In order to improve security, Apache has specific checks to avoid being fooled by `UNIX` symbolink links. If these precautions are not taken, it is possible to fool the Apache server to serve files it is not supposed to serve, such as files containing password−data for restricted directories. Denying Apache access to those files is often impossible because they are needed by the mod_auth modules, or by CGI scripts which execute with the same permissions as Apache. In absence of such sensitive data or malicious users, or when using mod_disallow_uid to deny access to those files, turning off symlink security can be considered.

Apache's symlink handling is handled by two mutually exclusive options, `FollowSymlinks` and `SymlinksIfOwnerMatch`, which can be included in any `Options` directive, except one contained in a Location directive. FollowSymlinks disables the symlink checks, while SymlinksIfOwnerMatch enables symlinks only if the owners of the source directory and the target file or directory match. SymlinksIfOwnerMatch considered if speed is *not* an issue, but security and the ability to use symlinks inside documentroots are, as SymlinksIfOwnerMatch still has to examine each path element.

The symlink checking requires an extra filesystem lookup (via the `lstat()` systemcall) for each path element in the location being opened, resulting in several such lookups per request. None of these lookups are cached by Apache. Filesystem lookups can be relatively slow, and when using a remote file storage such as an NFS server, can be quite taxing for the NFS server as well as the webserver. A busy webserver with several deep document trees and symlink−checking enabled can easily bring down a heavier NFS server by the number of `lstat()` calls and their network traffic. It is possible to enable symlink checks only for specific directories (and their subdirectories), disabling it for safe paths, but with deep directory trees this can still be a burden.

1.3.4 AllowOverride

The Apache directive `AllowOverride` is one of the most powerful and versatile directives in the Apache arsenal. It allows for the runtime reconfiguration of Apache for a specific directory (and all its subdirectories.) If `AllowOverride` is enabled, Apache checks for a specific file (configurable, but the default of `.htaccess` is practically never changed) in each directory that it visits, and parses it for configuration directives. This feature is most commonly used to limit access to directories (ask for a password), but the `.htaccess` file can contain almost all Apache directives.

Unfortunately, this level of flexibility comes at a price. In order to see the `.htaccess` files, Apache has to scan for it in every directory, resulting in the same number of filesystem lookups as the symlinks case previously discussed. It should be noted that checking for `.htaccess` and symlinks has to be done in two separate and different `stat()` systemcalls, so disabling only one of the two still reduces the number of lookups significantly.

`AllowOverride` can be set to multiple values. Only setting it to `None` will turn off the checking for `.htaccess` files altogether. Like the `FollowSymlinks` and `SymlinksIfOwnerMatch` directives, `AllowOverride` can be specified on a per−directory basis. Even if the VirtualHosts want or might want to use the `.htaccess` functionality, it is a good idea to turn off `AllowOverride` for directories that will never contain `.htaccess` files. The default Apache configuration does this for the system root directory, only enabling AllowOverride for the server's `DocumentRoot`.

1.3.5 mod_include

mod_include is the Apache module that parses *Server−Side Includes* files (generally .shtml files.) On some sites, mostly those based on old NCSA Webserver configuration files, the mod_include module gets invoked on every .htm(l) file, causing an extra pass over each such request. In general, this is not necessary, though a check to make sure no .htm(l) files contain SSI statements might be required to make sure.

1.3.6 MaxClients and Max/MinSpareServers

Apache has several statements to configure the number of clients being handled and the number of child−processes it forks off to deal with those requests. If the machine running the Apache installation is not intended for anything but running Apache, it is generally a good idea to make these numbers as high as possible, but not so high that the machine starts paging memory out to its swap space.

1.3.6.1 MaxClients

`MaxClients` limits the number of children Apache creates to handle requests, and because each child only accepts one connection at a time, the total number of concurrent connections. Setting this too low has a disastrous effect on performance. While all clients are busy handling connections, new connections will be put in the TCP queue and eventually timeout, while the machine itself will seem perfectly responsive and fast. It is generally a good idea to set MaxClients as high as the machine can handle in terms of memory. Setting it too high will cause the machine to swap needlessly, which effectively kills performance. If the machine can handle the number of processes in memory, but can't handle the sheer number of requests, it is often not useful to lower MaxClients. Instead, tuning Apache elsewhere, throttling one or more virtualhosts or simply sharing the load with more or bigger servers is the way to go.

1.3.6.2 MaxSpareServers and MinSpareServers

`MaxSpareServers` and `MinSpareServers` control how many spare (unused) child−processes Apache will keep alive while waiting for more requests to put them to use. Each child−process consumes resources, so having MaxSpareServers set too high can cause resource problems. On the other hand, if the number of unused servers drops below MinSpareServers, Apache will fork new child−processes (at a variable rate with a maximum of 32 forks per second) until MinSpareServers is satisfied. Forking a new child can be an relatively expensive operation, depending on the underlying operating system, and if a sudden burst of requests use up all the spare servers, the clients will be waiting for this. And new requests will pile up in the mean time.

The best values for these are highly system specific, and can change frequently even for specific sites. Fortunately it is usually not critical to get them exactly right. MinSpareServers should be a number high enough to satisfy sudden bursts of requests, to give Apache enough time to create new children. MaxSpareServers should be a number high enough to cover the normal fluctuation in numbers of requests, but not the daily fluctuations.

1.3.6.3 StartServers

`StartServers` controls the number of child−processes that Apache forks before starting to accept connections. If this number is lower than MinSpareServers, Apache will start forking off children while accepting connections, until the number of idle servers is at least MinSpareServers (but starts killing them off if the number exceeds MaxSpareServers.) It is a good idea to set StartServers to the minimum amount of servers you need active to start requests. When frequently restarting Apache during peak hours it might be prudent to raise this number to an average peak number, as killing off children while the server is not busy is less of a performance hit than creating children while busy.

1.3.6.4 Apache 2.0

Apache 2.0 has a more modular setup where the actual handling of connections is handled by Multi−Processing Modules (MPMs). The traditional pre−forking model of Apache 1.3 is available as an MPM, as is a multi−threaded variant. Which MPM is optimal for any particular system depends on a number of things, such as the efficiency of threads versus the efficiency of processes. For some platforms the choice is made simple by the available platform−specific MPMs, such as for Windows.

1.3.7 More performance tweaking

The *Performance Notes* of the *Apache manual* contains some more hints on performance−tuning, but most of those, especially the *Serialized Accept* section, are very low level and beyond the scope of this paper. See the *Apache manual* for more information.

1.4 Reducing network load

The following three modules can be used to reduce the network load generated by Apache. Of these, only mod_gzip will have any effect if the bandwidth bottleneck is outside of your control, like the user's modem connection.

1.4.1 mod_gzip

The mod_gzip module attempts to reduce bandwidth use by compressing data that is being sent out. If the browser claims to accept 'gzip' encoding, files can get compressed using the Lempel−Ziv coding (LZ77), the same algorithm used by the UNIX 'gzip' command. This compression comes at a cost of processing time on both the server and the client, however. The modules allows specifying which files are eligible for compression, so that files that are already (partially) compressed, and which would have little to gain by compression (like .gz files, .jpeg files, etc.) can be skipped.

The module is especially effective when using it to compress text−files (and thus HTML files) which are easily compressible. The mod_gzip module and the accompanying documentation can be found at http://www.remotecommunications.com/apache/mod_gzip/.

In Apache 2.0, mod_gzip's functionality is replaced by a new standard module, mod_deflate, which is documented in the standard documentation.

1.4.2 mod_bandwidth

Mod_bandwidth is a bandwidth throttling module, useful for keeping the traffic of a whole Apache installation or of specific VirtualHosts or directories in check. It allows for two *rates*, one for general data and one for files larger than a specified value, making it convenient for quenching file−downloads so the rest of the site is still responsive.

Throttling can also be done at the OS level, rather than the application level. For throttling entire Apache installations or IP−based virtualhosts this is often more efficient. However, throttling a name−based virtualhost or a directory within a virtualhost is not possible that way.

The throttling comes at a price of extra calculations on every packet send, and a local scratchboard to keep track of bandwidth usage. It is a good example of non−speed oriented optimizations. Mod_bandwidth can be found at http://www.cohprog.com/mod_bandwidth.html. Documentation is included in the C source file.

1.4.3 mod_proxy

Another method to reduce traffic for a specific server and increase the speed with which pages are served is by using a *front proxy*. The standard Apache module *mod_proxy* can serve as a front proxy. A front proxy keeps a cache of recently requested pages and returns the page from that cache if at all possible. Front proxies are only useful if the real storage of the data is slower than the cache of the proxy. For example, frequently requested data that resides on a remote NFS server or on a slow disk or CD can be significantly sped up by a front proxy.

Another common use of mod_proxy (with the help of mod_rewrite) is to split up requests to several servers, based on the URL, so that static content is taken from one server while auto−generated and server−intensive pages are taken from another.

Mod_proxy is useful for more than just playing front proxy, but it is not suited for every task. Since it was designed to proxy for other servers, not the server it is loaded into, it can be tricky to incorporate into existing setups. And since mod_proxy is a cache, the logfiles of the actual server no longer contains information of hits that were satisfied in the proxy. Furthermore, since all requests have to pass through the proxy server, it is still a bandwidth and speed bottleneck. Both mod_proxy and the documentation for it (which includes examples of configuring it as a front proxy) are part of the standard Apache distribution, though the module is not compiled in by default.

1.5 Customizing Apache for better performance

1.5.1 Site specific modules

While Apache is a great piece of software that is both very flexible and very efficient, it can't always be both at the same time. *hard−coding* site−specific information, thereby avoiding the sometimes complicated task of figuring it out at run−time, can in some situations provide a big performance boost. An example of this is the mod_userdir module. This module translates URLs in the form of `http://host/~<user>/` into a user−specific path on the filesystem, allowing users to have a homepage with its documentroot inside their own UNIX homedirectory. Its typical use is simply like this:

```
                UserDir WWW
```

Designating the directory 'WWW' in a users homedirectory to be the documentroot for the above URL. But in order to find the homedirectory, mod_userdir has to query the system's password file. This can be a very slow operation, especially when that information is stored on a NIS or LDAP server. mod_userdir allows for this, by giving a *template* for the full path, with the username as wildcard:

```
                UserDir /usr/home/*/WWW
```

This avoids any lookups and simply does regular filesystem checks. But not all homedirectory structures fit in such a restricted template, and mod_userdir does not sport a full regular−expression rewriting engine such as mod_rewrite does. For instance, XS4ALL divides homedirectories into one−letter subdirectories, the first letter of the username. This is not uncommon when dealing with a large number of homedirectories, as filesystem limits and directory−access times quickly become important.

To still avoid slow NIS lookups, which XS4ALL also uses, mod_userdir was locally modified to simply hardcode a users homedirectory to */home/u/username*. A solution using mod_rewrite, or extending mod_userdirs syntax would also have been possible, but would have cost more effort for no apparent gain. With the number of hits on user homepages XS4ALL gets, it should be noted that avoiding NIS lookups is important for the performance of the NIS servers as much as the performance of the website.

1.5.2 Mass Virtualhosting

One possible problem with Apache performance comes from large configuration files, especially many `<Directory>`, `<Location>` or `<VirtualHost>` blocks. Even though Apache uses hash−tables to optimize most of the lookups, these can still become large data−structures to wade through, and a lot of information to read in and parse at every restart. The actual impact is very system dependent, however.

The modules and directives in this section allow for more generic configuration. The main problem with these modules is that they require your server configuration to be very homogeneous.

1.5.2.1 Using regular expressions in Directives

The `<Directory>`, `<Location>` and `<Files>` directives can all take an extended regular expression instead of a regular directory, location or filename. While this takes some extra processing on each request, collapsing large lists of such statements into a single regexed statement can be well worth it. Reduced startup time and reduced process size can easily negate the added regexp match, especially when keeping the regular expression simple.

1.5.2.2 mod_vhost_alias

The mod_vhost_alias module is specifically intended to Apache to handle many VirtualHosts with just a few lines in the config file. Instead of specifying a <VirtualHost> block for each VirtualHost, each VirtualHosts' documentroot is calculated from the hostname passed in with the request, using a mod_rewrite−like rewrite−rule. Adding a new VirtualHost is then as simple as creating the directory for it. A common way to use this module is by mapping each request to `/usr/local/WWW/`*`www.domain.com/page`* where *www.domain.com* is taken from the 'Host:' header and *page* from the actual page requested.

The price paid is again that of loss of flexibility. VirtualHosts can no longer be configured with specific options (other than through `.htaccess`) and creating aliases for VirtualHosts has to be done using symlinks on the filesystem level (if symlinks are enabled in the Apache config.) The directory structure also has to be catered to the needs of this module, making it somewhat tricky to incorporate into existing installations.

1.5.2.3 mod_rewrite

A more generic approach to mass virtual hosting is mod_rewrite. Though this module has many, many uses, and is probably the most difficult to configure and understand modules in the standard Apache distribution, it can be an efficient replacement for situations where mod_vhost_alias is not versatile enough. The `RewriteMap` feature of mod_rewrite can use a mapping to calculate which directory to extract the files out of. A mapping can be a text−file of "`<hostname> <WITESPACE> <Rewrite−to>`" values, or a DBM hash containing `<hostname>` values as the keys and `<Rewrite−to>` values as the resulting values, or even an external program. (Though the latter is not likely to improve performance.)

As an incomplete example, here are some of the rewrite rules for doing mass virtualhosting using mod_rewrite:

```
RewriteMap   vhost        txt:/usr/local/apache/conf/vhost.map
RewriteCond  ${vhost:%1}  ^(/.*)$
RewriteRule  ^/(.*)$      %1/$1
```

Note: this example will not work properly, and is only intended to give an example of mod_rewrite's syntax. A complete example can be found in the URL Rewriting Guide which is part of the *Apache manual*.

Aside from the more extensive rewriting possibilities of mod_rewrite, it suffers the same problem as mod_vhost_alias: it is rigid in its configuration, disallowing per−VirtualHost customization except as part of rewrite rules. In addition, to activate a VirtualHost, the extra step of adding the server to the rewrite map has to be taken. And to define Virtualhost *aliases*, extra RewriteRules and a RewriteMap have to be added. Lastly, the syntax is clearly a downside, requiring very good inline documentation to be maintainable. Fortunately, the *URL Rewriting Guide* sets a good example in that, too.

1.6 Speeding up CGI scripts

CGI scripts are any program that gets executed on−demand by the webserver, and that uses the Common Gateway Interface to transfer information from and to the webserver and the browser that did the original request. Even though compiled C programs that use CGI are not technically scripts, they are often referred to as CGI scripts as well. A large portion of today's web programs are actually CGI scripts, though with the advent of PHP and other HTML−embedded server−side languages, those are getting more popular.

This section gives some hints on how to speed up the execution of CGI scripts. *mod_fastcgi* is a general FastCGI module, which uses FastCGI rather than normal CGI to connect to the CGI scripts. FastCGI uses some tricks to reduce the fork/exec overhead of a CGI script, but is not entirely backward compatible with normal CGI. *mod_php* and other such language−specific modules use language−specific information to speed up the execution process.

1.6.1 FastCGI

FastCGI is a slightly more complex alternative to normal CGI. With normal CGI, the webserver communicates with the CGI script through environment variables, and the client browser with the CGI script through its *standard input*. With FastCGI, each script acts as a *daemon*, being started once and handling multiple requests. Instead of environment variables, the server passes all information about the request through standard input, allowing FastCGI scripts to even be executed on different servers, over extra TCP connections.

The FastCGI interface allows for far more efficient use of resources, especially for oft−requested scripts, but might require a rewrite of the script in question to work properly. There are FastCGI API libraries for most popular languages, most of which allow a script to be used both though CGI and FastCGI without the need for modifications, but scripts that do not (yet) use these APIs do need to be modified, if not rewritten. FastCGI is best explained on its website, www.fastcgi.com, which also contains the Apache module and the API libraries for most languages.

1.6.2 mod_*<language>*

For several scripting languages (including Perl, Python, PHP, Tcl and Ruby) there are separate interpreter modules that give the language more control over Apache, as well as a performance boost. In general this is done by using specific knowledge about the language, and by keeping the language's *Interpreter* or *Virtual Machine* hanging around, passing it scripts as they get invoked. This avoids the execution overhead, and in some languages the compiling phase.

Each of these modules defines a *Handler*, to which specific file extensions and mime−types can be mapped, so that files of that type automatically get parsed by the right module. Because the interface to the scripts is far more like CGI than FastCGI, CGI scripts often need no or little modification to work properly.

The modules generally also allow embedding the language directly in HTML, using special tags to indicate the start and end of such embedded code. PHP started this trend by being specifically designed for the task. Though the resulting file looks like a HTML page, it should not be thought of as such: it is a script. The performance of the resulting script is often somewhat worse than a normal script that outputs HTML, because the HTML file has to undergo extra parsing to extract the HTML snippets.

The Perl module, mod_perl, can be found at: perl.apache.org.
For Python there are two competing modules, mod_snake and mod_python. mod_snake was originally written for Apache 2.0, has been ported to Apache 1.3 but is currently not being maintained. mod_python provides less

functionality, but is also more lightweight because of it.
For Tcl there are actually several modules, each with their own special focus. They can all be found under tcl.apache.org.
PHP is a language that got popular mainly because it was easily embeddable in HTML. It is currently in its fourth incarnation and is still undergoing improvements and expansions. PHP can be found at www.php.net
Ruby is another OO scripting language that is growing in popularity, with its own language module. mod_ruby can be found at www.modruby.net.

1.6.3 Java

Though the Java support could have been listed in the mod_<*language*> section, it is currently a special case in Apache. A separate Apache project, Jakarta, covers all Apache/Java integrations, to provide an open−source, portable platform for JavaServlets. The author of this document is insufficiently educated in Java and Jakarta to say anything about it at all, let alone about its performance. For more information, see jakarta.apache.org

1.7 About this document

This document is intended as companion to the Performance−tuning Apache ApacheCon presentation, and is written and maintained by Thomas Wouters. The latest version can be downloaded from http://www.xs4all.nl/~thomas/apachecon/. The author can be reached at thomas@xs4all.net.