

Performance

Your Web site can probably be made to run faster, if you are willing to make a few trade-offs, and spend a little time benchmarking your site to see what is really slowing it down.

There are a number of things that you can configure differently to get a performance boost. Although, there are other things to which you may have to make more substantial changes. It all depends on what you can afford to give up and what you are willing to trade off. For example, in many cases, you may need to trade performance for security, or vice versa.

In this chapter, we make some recommendations of things that you can change, and we warn against things that can cause substantial slow-downs. Be aware that Web sites are very individual, and what may speed up one Web site may not necessarily speed up another Web site.

Topics covered include hardware considerations, configuration file changes, and dynamic content generation, which can all be factors in getting every ounce of performance out of your Web site.



Very frequently, application developers develop programs in conditions that don't nearly enough reflect the conditions under which they will be run in production. Consequently, the application that seemed to run adequately fast with the test database of 100 records, runs painfully slowly with the production database of 200,000 records.

By ensuring that your test environment is at least as demanding as your production environment, you greatly reduce the chances that your application will perform unexpectedly slowly when you roll it out.

11.1 Determining How Much Memory You Need

Problem

You want to ensure that you have sufficient RAM in your server.

Solution

Find the instances of Apache in your process list, and determine an average memory footprint for an Apache process. Multiply this number by your peak load (maximum number of concurrent Web clients you'll be serving).

Discussion

Because there is very little else that you can do at the hardware level to make your server faster, short of purchasing faster hardware, it is important to make sure that you have as much RAM as you need.

Determining how much memory you need is an inexact science, to say the least. In order to take an educated guess, you need to observe your server under load, and see how much memory it is using.

The amount of memory used by one Apache process will vary greatly from one server to another, based on what modules you have installed and what the server is being called upon to do. Only by looking at your own server can you get an accurate estimate of what this quantity is for your particular situation.

Tools such as *top* and *ps* may be used to examine your process list and determine the size of processes. The `server-status` handler, provided by *mod_status*, may be used to determine the total number of Apache processes running at a given time.

If, for example, you determine that your Apache processes are using 4 MB of memory each, and under peak load, you find that you are running 125 Apache processes, then you will need, at a bare minimum, 500 MB of RAM in the server to handle this peak load. Remember that memory is also needed for the operating system, and any other applications and services that are running on the system, in addition to Apache, and so in reality you will need more than this amount.

If, by contrast, you are unable to add more memory to the server, for whatever reason, you can use the same technique to figure out the maximum number of child processes that you are capable of serving at any one time, and use the *MaxClients* directive to limit Apache to that many processes:

```
MaxClients 125
```

See Also

- <http://httpd.apache.org/docs/misc/perf-tuning.html>

11.2 Benchmarking Apache with ab

Problem

You want to benchmark changes that you are making to verify that they are in fact making a difference in performance.


Solution

Use *ab* (Apache bench), which you will find in the *bin* directory of your Apache installation:

```
ab -n 1000 -c 10 http://www.example.com/test.html
```

Discussion

ab is a command-line utility that comes with Apache and lets you do very basic performance testing of your server. It is especially useful for making small changes to your configuration and testing server performance before and after the change.

The arguments given in the previous example tell *ab* to request the resource *http://www.example.com/test.html* 1000 times (*-n 1000* indicates the number of requests) and to make these requests 10 at a time (*-c 10* indicates the concurrency level). 

Other arguments that may be specified can be seen by running *ab* with the *-h* flag. Of particular interest is the *-k* flag, which enables keepalive mode. See the following keep-alive recipe for additional details on this matter.

There are a few things to note about *ab* when using it to evaluate performance.

ab does not mimic Web site usage by real people. It requests the same resource repeatedly to test the performance of that one thing. For example, you may use *ab* to test the performance of a particular CGI program, before and after a performance-related change was made to it. Or you may use it to measure the impact of turning on *.htaccess* files, or content negotiation, for a particular directory. Real users, of course, do not repeatedly load the same page, and so performance measurements made using *ab* may not reflect actual real-world performance of your Web site.

You should probably not run the Web server and *ab* on the same machine, as this will introduce more uncertainty into the measurement. With both *ab* and the Web server itself consuming system resources, you will receive significantly slower performance than if you were to run *ab* on some other machine, accessing the server over the network. However, also be aware that running *ab* on another machine will introduce network latency, which is not present when running it on the same machine as the server.

Finally, there are many factors that can affect performance of the server, and you will not get the same numbers each time you run the test. Network conditions, other processes running on the client or server machine, and a variety of other things may influence your results slightly one way or another. The best way to reduce the impact

of environmental changes is to run a large number of tests and average your results. Also, make sure that you change as few things as possible—ideally, just one—between tests, so that you can be more sure what change has made any differences you can see.

Finally, you need to understand that, while *ab* gives you a good idea of whether certain changes have improved performance, it does not give a good simulation of actual users. Actual users don't simply fetch the same resource repeatedly, but they obtain a variety of different resources from various places on your site. Thus, actual site usage conditions may produce different performance issues than those revealed by *ab*.

See Also

- the manpage for the *ab* tool
- <http://httpd.apache.org/docs/2.2/programs/ab.html>

11.3 Tuning Keepalive Settings

Problem

You want to tune the keepalive-related directives to the best possible setting for your Web site.

Solution

Turn on the *KeepAlive* setting, and set the related directives to sensible values:

```
KeepAlive On
MaxKeepAliveRequests 0
KeepAliveTimeout 15
```

Discussion

The default behavior of HTTP is for each document to be requested over a new connection. This causes a lot of time to be spent opening and closing connections. *KeepAlive* allows multiple requests to be made over a single connection, thus reducing the time spent establishing socket connections. This, in turn, speeds up the load time for clients requesting content from your site.

In addition to turning keepalive on, using the *KeepAlive* directive, there are two directives that allow you to adjust the way that it is done.

The first of these, *MaxKeepAliveRequests*, indicates how many keepalive requests should be permitted over a single connection. There is no reason to have this number set low. The default value for this directive is 100, and this seems to work pretty well for most sites. Setting this value to 0 means that an unlimited number of requests will be permitted over a single connection. This might allow users to load all of their content

from your site over a single connection, depending on the value of *KeepAliveTimeout* and how quickly they went through the site.

KeepAliveTimeout indicates how long a particular connection will be held open when no further requests are received. The optimal setting for this directive depends entirely on the nature of your Web site. You should probably think of this value as the amount of time it takes users to absorb the content of one page of your site before they move on to the next page. If the users move on to the next page before the *KeepAliveTimeout* has expired, when they click on the link for the next page of content, they will get that next document over the same connection. If, however, that time has already expired, they will need to establish a new connection to the server for that next page.

You also should be aware that if users load a resource from your site and then go away, Apache will still maintain that open connection for them for *KeepAliveTimeout* seconds, which makes that child process unable to serve any other requests during that time. Therefore, setting *KeepAliveTimeout* too high is just as undesirable as setting it too low.

In the event that *KeepAliveTimeout* is set too high, you will see (i.e., with the *server-status* handler—see Recipe 11.4) that a significant number of processes are in *keepalive* mode, but are inactive. Over time, this number will continue to grow, as more child processes are spawned to take the place of child processes that are in this state.

Conversely, setting *KeepAliveTimeout* too low will result in conditions similar to having *KeepAlive* turned off entirely, when a single client will require many connections over the course of a brief visit. This is less easy to detect than the opposite condition. In general, it is probably better to err on the side of setting it too high, rather than too low.

Because the length of time that any given user looks at any given document on your site is going to be as individual as the users themselves, and varies from page to page around your Web site, it is very difficult to determine the best possible value of this directive for a particular site. However, it is unlikely that this is going to make any large impact on your overall site performance, when compared to other things that you can do. Leaving it at the default value of 15 tends to work pretty well for most sites.

See Also

- <http://httpd.apache.org/docs/2.2/mod/core.html#keepalive>
- <http://httpd.apache.org/docs/2.2/mod/core.html#maxkeepaliverequests>
- <http://httpd.apache.org/docs/2.2/mod/core.html#keepalivetimeout>

The default value is 5, not 15, in the 2.2 and trunk config files.

11.4 Getting a Snapshot of Your Site's Activity

Problem

You want to find out exactly what your server is doing.

Solution

Enable the `server-status` handler to get a snapshot of what child processes are running and what each one is doing. Enable *ExtendedStatus* to get even more detail:

```
<Location /server-status>
    SetHandler server-status
    Order deny,allow
    Deny from all
    Allow from 192.168.1
</Location>
```

```
ExtendedStatus On
```

Then, view the results at the URL *http://servername/server-status*

Discussion

Provided by *mod_status*, which is enabled by default, the `server-status` handler provides a snapshot of your server's activity. This snapshot includes some basic details, such as when the server was last restarted, how long it has been up, and how much data it has served in that time. Following that, there will be a list of the child processes and what each one is doing. At the bottom of the page is a detailed explanation of the terms used and what each column of the table represents.



The server status display shows activity across the entire server—including virtual hosts. If you are providing hosting services for others, you may not want them to be able to see this level of detail about each other.

It is recommended that, as in the default configuration file that comes with Apache, you restrict access to this handler. Part of the information contained on this page is a list of client addresses and the document that they are requesting. Some users feel that it is a violation of their privacy for you to make this information readily available on your Web site. Additionally, it may provide information such as `QUERY_STRING` variables, `PATH_INFO` variables, or simply URLs, which you wished to not be made public. It is therefore recommended that you add to the above recipe some lines such as:

```
Order deny,allow
Deny from all
Allow from 192.168.1
```

This configuration allows access only from the 192.168.1 network, or whatever network you put in there, and denies access from unauthorized Internet users.

See Also

- http://httpd.apache.org/docs/2.2/mod/mod_status.html

- <http://httpd.apache.org/server-status/>

11.5 Avoiding DNS Lookups

Problem

You want to avoid situations where you have to do DNS lookups of client addresses, as this is a very slow process.

Solution

Always set the *HostNameLookups* directive to **Off**:

```
HostNameLookups Off
```

And make sure that, whenever possible, *Allow from* and/or *Deny from* directives use the IP address, rather than the hostname of the hosts in question.

Discussion

DNS lookups can take a very long time - anywhere from 0 to 60 seconds - and should be avoided at all costs. In the event that a client address cannot be looked up at all, it can take up to a minute for the lookup to time out, during which time the child process that is doing the lookup cannot do anything else.

There are a number of cases in which Apache will need to do DNS lookups, and so the goal here is to completely avoid those situations.

HostNameLookups

Before Apache 1.3, *HostNameLookups*, which determines whether Apache logs client IP addresses or hostnames, defaulted to **on**, meaning that each Apache log entry required a DNS lookup to convert the client IP address to a hostname to put in the logfile. Fortunately, that directive now defaults to **off**, and so this is primarily an admonition to leave it alone.

If you need to have these addresses converted to hostnames, then this should be done by another program, preferably running on a machine other than your production Web server. That is, you really should copy the file to some other machine for the purpose of processing, so that the effort required to do this processing does not negatively effect your Web server's performance.

Apache comes with a utility called *logresolve*, which will process your logfile, replacing IP addresses with hostnames. Additionally, most logfile analysis tools will also do this name resolution as part of the log analysis process.

Allow and Deny from hostnames

When you do host-based access control, using the *Allow from* and *Deny from* directives, Apache takes additional precautions to make sure that the client is not spoofing its hostname. In particular, it does a DNS lookup on the IP address of the client to obtain the name to compare against the access restriction. It then looks up the name that was obtained, just to make sure that the DNS record is not being faked.*

For the sake of better performance, therefore, it is much better to use an IP address, rather than a name, in *Allow* and *Deny* directives.

See Also

- Chapter 3

11.6 Optimizing Symbolic Links

Problem

You wish to balance the security needs associated with symbolic links with the performance impact of a solution, such as using *Options SymLinksIfOwnerMatch*, which causes a server slowdown.

Solution

For tightest security, use *Options SymLinksIfOwnerMatch*, or *Options -FollowSymLinks* if you seldom or never use symlinks.

For best performance, use *Options FollowSymLinks*.

Discussion

Symbolic links are an area in which you need to weigh performance against security and make the decision that makes the most sense in your particular situation.

In the normal everyday operation of a Unixish operating system, symbolic links are considered to be the same as the file to which they link.[†] When you *cd* into a directory, you don't need to be aware of whether that was a symlink or not. It just works.

Apache, by contrast, has to consider whether each file and directory is a symlink or not, if the server is configured not to follow symlinks. And, additionally, if *Option SymLinksIfOwnerMatch* is turned on, Apache not only has to check if the particular file is a symlink, but also has to check the ownership of the link itself and of the target, in

* For example, the owner of the IP address could very easily put a PTR record in their reverse-DNS zone, pointing their IP address at a name belonging to someone else.

[†] Of course, this is not true at the filesystem level, but we're just talking about the practical user level.

the event that it is a symlink. Although this enforces a certain security policy, it takes a substantial amount of time and so slows down the operation of your server.

In the trade-off between security and performance, in the matter of symbolic links, here are the guidelines.

If you are primarily concerned about security, never permit the following of symbolic links. It may permit someone to create a link from a document directory to content that you would not want to be on a public server. Or, if there are cases in which you really need symlinks, use *Options SymLinksIfOwnerMatch*, which requires that someone may only link to files that they own and will presumably protect you from having a user link to a portion of the filesystem that is not already under their control.

If you are concerned about performance, then always use *Options FollowSymLinks*, and never use *Options SymLinksIfOwnerMatch*. *Options FollowSymLinks* permits Apache to follow symbolic links in the manner of most Unixish applications—that is, Apache does not even need to check to see if the file in question is a symlink or not.

See Also

- <http://httpd.apache.org/docs/2.2/mod/core.html#options>

11.7 Minimizing the Performance Impact of .htaccess Files

Problem

You want *per*-directory configuration but want to avoid the performance hit of *.htaccess* files.

Solution

Turn on *AllowOverride* only in directories where it is required, and tell Apache not to waste time looking for *.htaccess* files elsewhere:

```
AllowOverride None
```

Then use *<Directory>* sections to selectively enable *.htaccess* files only where needed.

Discussion

.htaccess files cause a substantial reduction in Apache's performance, because it must check for a *.htaccess* in every directory along the path to the requested file to be assured of getting all of the relevant configuration overrides. This is necessary because Apache configuration directives apply not only to the directory in which they are set, but also to all subdirectories. Thus, we must check for *.htaccess* files in parent directories, as well as in the current directory, to find any directives that would trickle down the current directory.

For example, if, for some reason, you had *AllowOverride All* enabled for all directories and your *DocumentRoot* was */usr/local/apache/htdocs*, then a request for the URL *http://example.com/events/parties/christmas.html* would result in the following files being looked for and, if found, opened and searched for configuration directives:

```
/.htaccess
/usr/.htaccess
/usr/local/.htaccess
/usr/local/apache/.htaccess
/usr/local/apache/htdocs/.htaccess
/usr/local/apache/htdocs/events/.htaccess
/usr/local/apache/htdocs/events/parties/.htaccess
```

Now, hopefully, you would never have *AllowOverride All* enabled for your entire filesystem, so this is a worst-case scenario. However, occasionally, when people do not adequately understand what this configuration directive does, they will enable this option for their entire filesystem and suffer poor performance as a result.

The recommended solution is by far the best way to solve this problem. The *<Directory>* directive is specifically for this situation, and *.htaccess* files should really only be used in the situation where configuration changes are needed and access to the main server configuration file is not readily available.

For example, if you have a *.htaccess* file in */usr/local/apache/htdocs/events* containing the directive:

```
AddEncoding x-gzip tgz
```

You should instead simply replace this with the following in your main configuration file:

```
<Directory /usr/local/apache/htdocs/event>
    AddEncoding x-gzip tgz
</Directory>
```

Which is to say, anything that appears in a *.htaccess* can, instead, appear in a *<Directory>* section, referring to that same directory.

If you are compelled to permit *.htaccess* files somewhere on your Web site, you should only permit them in the specific directory where they are needed. For example, if you particularly need to permit *.htaccess* files in the directory */www/htdocs/users/leopold*, then you should explicitly allow them for only this directory:

```
<Directory /www/htdocs/users/leopold>
    AllowOverride All
</Directory>
```

One final note about the *AllowOverride* directive: this directive lets you be very specific about what types of directives you permit in *.htaccess* files, and you should make an effort only to permit those directives that are actually needed. That is, rather than using the *All* argument, you should allow specific types of directives as needed. In particular,

the `Options` argument to `AllowOverride` should be avoided, if possible, as it may enable users to turn on features that you have turned off for security reasons.

See Also

- <http://httpd.apache.org/docs/2.2/howto/htaccess.html>
- <http://httpd.apache.org/docs/2.2/howto/htaccess.html>

11.8 Disabling Content Negotiation

Problem

Content negotiation causes a big reduction in performance.

Solution

Disable content negotiation where it is not needed. If you do require content negotiation, use the `type-map` handler, rather than the `MultiViews` option:

```
Options -MultiViews
AddHandler type-map var
```

Discussion

If at all possible, disable content negotiation. However, if you must do content negotiation—if, for example, you have a multilingual Web site—you should use the `type-map` handler, rather than the `MultiViews` method.

When `MultiViews` is used, Apache needs to get a directory listing each time a request is made. The resource requested is compared to the directory listing to see what variants of that resource might exist. For example, if `index.html` is requested, the variants `index.html.en` and `index.html.fr` might exist to satisfy that request. Each matching variant is compared with the user's preferences, expressed in the various `Accept` headers passed by the client. This information allows Apache to determine which resource is best suited to the user's needs.

However, this process can be very time-consuming, particularly for large directories or resources with large numbers of variants. By putting the information in a `.var` file and allowing the `type-map` handler to be used instead, you eliminate the requirement to get a directory listing, and greatly reduce the amount of work that Apache must do to determine the correct variant to send to the user.

The `.var` file just needs to contain a listing of the variants of a particular resource and describe their important attributes.

If you have, for example, English, French, and Hebrew variants of the resource `index.html`, you may express this in a `.var` file called `index.html.var` containing information about each of the various variants. This file might look like the following:

```
URI: index.html.en
Content-language: en
Content-type: text/html
```

```
URI: index.html.fr
Content-language: fr
Content-type: text/html
```

```
URI: index.html.he.iso8859-8
Content-language: he
Content-type: text/html; charset=ISO-8859-8
```

This file should be placed in the same directory as the variants of this resource, which are called *index.html.en*, *index.html.fr*, and *index.html.he.iso8859-8*.

Note that the Hebrew variant of the document indicates an alternate character set, both in the name of the file itself, and in the **Content-type** header field.

Enable the *.var* file by adding a *AddHandler* directive to your configuration file, as follows:

```
AddHandler type-map .var
```



Each of the file extensions used in these filenames should have an associated directive in your configuration file. This is not something that you should have to add—these should appear in your default configuration file. Each of the language indicators will have an associated *AddLanguage* directive, while the character set indicator will have an *AddCharset* directive.

In contrast to **MultiViews**, this technique gets all of its information from this *.var* file instead of from a directory listing, which is much less efficient.

You can further reduce the performance impact of content negotiation by indicating that negotiated documents can be cached. This is accomplished by the directive:

```
CacheNegotiatedDocs On
```

Caching negotiated documents can cause unpleasant results, such as people getting files in a language that they cannot read or in document formats that they don't know how to render.

If possible, you should completely avoid content negotiation in any form, as it will greatly slow down your server no matter which technique you use.

See Also

- http://httpd.apache.org/docs/2.2/mod/mod_negotiation.html
- http://httpd.apache.org/docs/2.2/mod/mod_mime.html#addhandler

- http://httpd.apache.org/docs/2.2/mod/mod_mime.html#addcharset
- http://httpd.apache.org/docs/2.2/mod/mod_mime.html#addlanguage
- <http://httpd.apache.org/docs/2.2/mod/core.html#optionsr>

11.9 Optimizing Process Creation

Problem

You're using Apache 1.3, or Apache 2.0 with the *prefork* MPM, and you want to tune *MinSpareServers* and *MaxSpareServers* to the best settings for your Web site.

Solution

Will vary from one site to another. You'll need to watch traffic on your site and decide accordingly.

Discussion

The *MinSpareServers* and *MaxSpareServers* directives control the size of the server pool, so that incoming requests will always have a child process waiting to serve them. In particular, if there are fewer than *MinSpareServers* idle processes, Apache will create more processes until that minimum is reached. Similarly, if there are ever more than *MaxSpareServers* processes, Apache will kill off processes until there are fewer than that maximum. These things will happen as the site traffic fluctuates on a normal day.

The best values for these directives for your particular site depends on the amount and the rate at which traffic fluctuates. If your site is prone to large spikes in traffic, *MinSpareServers* needs to be large enough to absorb those spikes. The idea is to never have a situation where requests come in to your site, and there are no idle server processes waiting to handle the request. If traffic patterns on your site are fairly smooth curves with no abrupt spikes, the default values may be sufficient.

The best way to watch exactly how much load there is on your server is by looking at the *server-status* handler output. (See Recipe 11.4.)

You also should set *MaxClients* to a value such that you don't run out of server resources during heavy server loads. For example, if your average Apache process consumes 2 MB of memory and you have a total of 256 MB of RAM available, allowing a little bit of memory for other processes, you probably don't want to set *MaxClients* any higher than about 120. If you run out of RAM and start using swap space, your server performance will abruptly go downhill and will not recover until you are no longer using swap. You can watch memory usage by running a program such as *top*, which shows running processes and how much memory each is using.

See Also

- the section called “Setting the number of threads on single-child MPMs” in Recipe 11.10
- the section called “Number of threads when using the worker MPM” in Recipe 11.10

11.10 Tuning Thread Creation

Problem

You’re using Apache 2.0 with one of the threaded MPMs, and you want to optimize the settings for the number of threads.

Solution

Will vary from server to server.

Discussion

The various threaded MPMs on Apache 2.0 handle thread creation somewhat differently. In Apache 1.3, the Windows and Netware versions are threaded, whereas the Unixish version is not. Tuning the thread creation values will vary from one of these versions to another.

Setting the number of threads on single-child MPMs

On MPMs that run Apache with a single threaded child process, such as the Windows MPM (*mpm_winnt*), and the Windows and Netware versions of Apache 1.3, there are a fixed number of threads in the child process. This number is controlled by the *ThreadsPerChild* directive and must be large enough to handle the peak traffic of the site on any given day. There really is no performance tuning that can be done here, as this number is fixed throughout the lifetime of the Apache process.

Number of threads when using the worker MPM

The *worker* MPM has a fixed number of threads per child process but has a variable number of child processes, so that increased server load can be absorbed. A typical configuration might look like the following:

```
StartServers 2
MaxClients 150
MinSpareThreads 25
MaxSpareThreads 75
ThreadsPerChild 25
ServerLimit 16
```

The *MinSpareThreads* and *MaxSpareThreads* directives control the size of the idle pool of threads, so that incoming clients will always have an idle thread waiting to serve their request. The *ThreadsPerChild* directive indicates how many threads are in each child process, so when the number of available idle threads drops below *MinSpareThreads*, Apache will launch a new child process, populated with *ThreadsPerChild* threads. Similarly, when server load is reduced and the number of idle threads is greater than *MaxSpareThreads*, Apache will kill off one or more child processes to reduce the idle pool to that number or less.

The goal, when setting these values, is to ensure that there are always idle threads ready to serve any incoming client's request, without having to create a new one. The previous example will work for most sites, as it will ensure that there is at least one completely unused child process, populated with 25 threads, waiting for incoming requests. As soon as threads within this process start to be used, a new child process will be launched for future requests.

The values of *MaxClients* and *ServerLimit* should be set so that you will never run out of RAM when a new child process is launched. Look at your process list, using *top*, or a similar utility, and ensure that *ServerLimit*, multiplied by the size of an individual server process, does not exceed your available RAM. *MaxClients* should be less than, or equal to, *ServerLimit* multiplied by *ThreadsPerChild*.

Setting the number of threads when using Netware or the perchild MPM

Whereas with most of the other MPMs the *MinSpareThreads* and *MaxSpareThreads* directives are server-wide, in the *perchild* and *netware* MPMs, these directives are assessed per child process. Of course, with the *netware* MPM, there is only one child process, so it amounts to the same thing.

With the *netware* MPM, threads are created and reaped as needed, to keep the number of spare threads between the limits imposed by *MinSpareThreads* and *MaxSpareThreads*. The total number of threads must be kept at all times below the limit imposed by the *MaxThreads* directive.

See Also

- <http://httpd.apache.org/docs/2.2/mpm.html>

11.11 Caching Frequently Viewed Files

Problem

You want to cache files that are viewed frequently, such as your site's front page, so that they don't have to be loaded from the filesystem every time.

Solution

Use *mod_mmap_static* or *mod_file_cache* (for Apache 1.3 and 2.0, respectively) to cache these files in memory:

```
MMapFile /www/htdocs/index.html
MMapFile /www/htdocs/other_page.html
```

For Apache 2.0, you can use either module or the *CacheFile* directive. *MMapFile* caches the file contents in memory, while *CacheFile* caches the file handle instead, which gives slightly poorer performance but uses less memory:

```
CacheFile /www/htdocs/index.html
CacheFile /www/htdocs/other_page.html
```

Discussion

For files that are frequently accessed, it is desirable to cache that file in some fashion to save disk access time. The *MMapFile* directive loads a file into RAM, and subsequent requests for that file are served directly out of RAM, rather than from the filesystem. The *CacheFile* directive, by contrast, opens the file and caches the file handle, saving time on subsequent file opens.

In Apache 1.3, this functionality is available with the *mod_mmap_static* module, which is labelled as experimental and is not built into Apache by default. To enable this module, you need to specify the `--enable-module=mmap_static` flag to *configure* when building Apache. *mod_mmap_static* provides only the *MMapFile* directive.

In Apache 2.0, this functionality is provided by the *mod_file_cache* module, which is labelled as experimental, and is not built into Apache by default. To enable this module, you need to specify the `--enable-file-cache` flag to *configure* when building Apache. *mod_file_cache* provides both the *MMapFile* and *CacheFile* directives.

These directives take a single file as an argument, and there is not a provision for specifying a directory or set of directories. If you wish to have the entire contents of a directory mapped into memory, the documentation provides the following suggestion. For the directory in question, you would run the following command:

```
% find /www/htdocs -type f -print \
> | sed -e 's/./mmapfile &/' > /www/conf/mmap.conf
```

In your main server configuration file, you would then load the file created by that command, using the *Include* directive:

```
Include /www/conf/mmap.conf
```

This would cause every file contained in that directory to have the *MMapFile* directive invoked on it.

Note that when files are cached using one of these two directives, any changes to the file will require a server restart before they become visible.

See Also

- http://httpd.apache.org/docs/2.2/mod/mod_mmap_static.html
- http://httpd.apache.org/docs/2.2/mod/mod_file_cache.html

11.12 Distributing Load Evenly Between Several Servers

Problem

You want to serve the same content from several servers and have hits distributed evenly among the servers.

Solution

Use DNS round-robin to have requests distributed evenly, or at least fairly evenly, among the servers:

```
www.example.com. 86400 IN A 192.168.10.2
www.example.com. 86400 IN A 192.168.10.3
www.example.com. 86400 IN A 192.168.10.4
www.example.com. 86400 IN A 192.168.10.5
www.example.com. 86400 IN A 192.168.10.6
www.example.com. 86400 IN A 192.168.10.7
```

Add the following to your configuration file:

```
FileETag MTime Size
```

Discussion

This example is an excerpt from a BIND zone file. The actual syntax may vary, depending on the particular name server software you are running.

By giving multiple addresses to the same hostname, you cause hits to be evenly distributed among the various servers listed. The name server, when asked for this particular name, will give out the addresses listed in a round-robin fashion, causing requests to be sent to one server after the other. The individual servers need be configured only to answer requests from the specified name.

Running the *host* command on the name in question will result in a list of possible answers, but each time you run the command, you'll get a different answer first:

```
% host www.example.com
www.example.com has address 192.168.10.2
www.example.com has address 192.168.10.3
www.example.com has address 192.168.10.4
www.example.com has address 192.168.10.5
www.example.com has address 192.168.10.6
www.example.com has address 192.168.10.7
% host www.example.com
www.example.com has address 192.168.10.7
```

```
www.example.com has address 192.168.10.2
www.example.com has address 192.168.10.3
www.example.com has address 192.168.10.4
www.example.com has address 192.168.10.5
www.example.com has address 192.168.10.6
```



Make sure that when you update your DNS zone file, you also update the serial number, and restart or reload your DNS server.

One of the document aspects used to determine cache freshness is the ETag value the server associates with it. This usually includes a calculation based on the document's actual disk location, which may be different on the different back-end hosts. The *FileETag* settings give cause that information to be omitted, so if the documents are truly identical they should all be given the same ETag value, and be indistinguishable when it comes to caching them.

See Also

- *DNS and Bind* by Paul Albitz and Cricket Liu (O'Reilly)
- Recipe 10.3

11.13 Caching Directory Listings

Problem

You want to provide a directory listing but want to reduce the performance hit of doing so.

Solution

Use the `TrackModified` argument to *IndexOptions* to allow browsers to cache the results of an auto-generated directory index:

```
IndexOptions +TrackModified
```

Discussion

When sending a directory listing to a client, Apache has to open that directory, obtain a directory listing, and determine various attributes of the files contained therein. This is very time consuming, and it would be nice to avoid this when possible.

By default, the Last Modified time sent with a directory listing is the time that the content is being served. Thus, when a client, or proxy server, makes a *HEAD* or conditional *GET* request to determine if it can use the copy that it has in cache, it will always decide to get a fresh copy of the content. The `TrackModified` option to *Index-*

Options cause *mod_autoindex* to send a Last Modified time corresponding to the file in the directory that was most recently modified. This enables browsers and proxy servers to cache this content, rather than retrieving it from the server each time, and also ensures that the listing that they have cached is in fact the latest version.

Note that clients that don't implement any kind of caching will not benefit from this directive. In particular, testing with *ab* will show no improvement from turning on this setting, as *ab* does not do any kind of content caching.

See Also

- the manpage for the *ab* tool
- <http://httpd.apache.org/docs/2.2/programs/ab.html>

11.14 Speeding Up Perl CGI Programs with *mod_perl*

Problem

You have existing functional Perl CGI programs and want them to run faster.

Solution

If you have the *mod_perl* module installed, you can configure it to run your Perl CGI programs, instead of running *mod_cgi*. This gives you a big performance boost, without having to modify your CGI code.

There are two slightly different ways to do this.

For Apache 1.3 and *mod_perl* Version 1:

```
Alias /cgi-perl/ /usr/local/apache/cgi-bin/
<Location /cgi-perl>
    Options ExecCGI
    SetHandler perl-script
    PerlHandler Apache::PerlRun
    PerlSendHeader On
</Location>

Alias /perl/ /usr/local/apache/cgi-bin/
<Location /perl>
    Options ExecCGI
    SetHandler perl-script
    PerlHandler Apache::Registry
    PerlSendHeader On
</Location>
```

For Apache 2.0 and *mod_perl* Version 2, the syntax changes slightly:

```

PerlModule ModPerl::PerlRun
Alias /cgi-perl/ /usr/local/apache2/cgi-bin/
<Location /cgi-perl>
    SetHandler perl-script
    PerlResponseHandler ModPerl::PerlRun
    Options +ExecCGI
</Location>

PerlModule ModPerl::Registry
Alias /perl/ /usr/local/apache2/cgi-bin/
<Location /perl>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    Options +ExecCGI
</Location>

```

Discussion

By using *mod_perl*'s CGI modes, you can improve the performance of existing CGI programs without modifying the CGI code itself in any way. Given the previous configuration sections, a CGI program that was previously accessed *via* the URL *http://www.example.com/cgi-bin/program.cgi* will now be accessed *via* the URL *http://www.example.com/cgi-perl/program.cgi* to run it in *PerlRun* mode or *via* the URL *http://www.example.com/perl/program.cgi* to run it in *Registry* mode.

The primary difference between *PerlRun* and *Registry* is that, in *Registry*, the program code itself is cached after compilation, whereas in *PerlRun* mode, it is not. While this means that code run under *Registry* is faster than that executed under *PerlRun*, it also means that a greater degree of code quality is required. In particular, global variables and other careless coding practices may cause memory leaks, which, if run in cached mode, could eventually cause the server to run out of available memory.

When writing Perl CGI code to run under *mod_perl*, and, in general, when writing any Perl code, it is recommended that you place the following two lines at the top of each program file, following the *#!* line:

```

use strict;
use warnings;

```

Code that runs without error messages, with these two lines in them, runs without problems under *Registry*.



strict is not available before Perl 5, and *warnings* is not available before Perl 5.6. In versions of Perl earlier than 5.6, you can get behavior similar to *warnings* by using the *-w* flag to Perl. This is accomplished by adding it to the *#!* line of your Perl programs:

```

#!/usr/bin/perl -w

```

See Also

- *Programming Perl*, Third Edition, by Larry Wall, Tom Christiansen, and Jon Orwant (O'Reilly)

11.15 Caching Dynamic Content

Problem

You want to cache dynamically generated documents that don't actually change very often.

Solution

In Apache 2.2, use the following recipe:

```
CacheEnable disk /  
CacheRoot /var/www/cache  
CacheIgnoreCacheControl On  
CacheDefaultExpire 600
```

In 2.3 and later, you can use something like this:

```
CacheEnable disk /  
CacheRoot /var/www/cache  
CacheDefaultExpire 600  
CacheMinExpire 600
```

Discussion

Caching is usually explicitly disabled for dynamic content. Dynamic content, by definition, is content that is generated on demand—that is, created fresh each time it is requested. Thus, caching it is contrary to its very nature

However, it is often—even usually—the case that dynamically generated content doesn't actually change very much from one minute to the next. This means that you end up wasting an awful lot of time generating content that hasn't actually changed since the last time it was requested. If you're doing this several times per second, you're probably causing your server a great deal more work than is really necessary.

The recipes above solve this problem in two slightly different ways. The solution for 2.3 is better, but, as of this writing, 2.3 hasn't been released yet, so it's not a terribly practical solution yet.

The recipe for Apache 2.2 takes the approach of disabling cache control—that is, it tells Apache to ignore the request made by the dynamic content that it not be cached, and caches it anyway. Then, for good measure, a default cache expiration time of 5

minutes (600 seconds) is set, so that any content that is cached will be retained at least that long.

The solution for 2.3 is slightly more elegant. It sets a minimum cache expiration time of five minutes, as well as setting the default expiration time. This ensures that all content is cached at least for 5 minutes, but the content itself may specify a longer time, if desired. The main difference here is that in the 2.2 solution if the content itself sends a request that it be cached longer this will be ignored, whereas in the 2.3 solution it will be honored.

You'll first see the 2.3 solution working when the 2.4 version of the server is released. 2.3 is a development branch, and it will be called 2.4 when it is ready for general use.

Make sure that the directory specified as the CacheRoot exists, and is writeable by the Apache user.