Configuring Apache for Maximum Performance

1 Introduction

Apache is an open-source HTTP server implementation. It is the most popular web server on the Internet; the December 2005 Web Server Survey conducted by Netcraft shows that about 70% of the web sites on Internet are using Apache.

Apache server performance can be improved by adding additional hardware resources such as RAM, faster CPU, etc. But most of the time, the same result can be achieved by custom configuration of the server. This article looks into getting maximum performance out of Apache with the existing hardware resources, specifically on Linux systems. Of course, it is assumed that there is enough hardware resources - especially enough RAM that the server isn't swapping frequently. First two sections look into various Compile-Time and Run-Time configuration options. The Run-Time section assumes that Apache is compiled with prefork MPM. HTTP compression and caching is discussed next. Finally, using separate servers for serving static and dynamic contents is covered. Basic knowledge of compiling and configuring Apache and Linux are assumed. 2 Compile-Time Configuration Options 2.1 Load only the required modules:

The Apache HTTP Server is a modular program where the administrator can choose the functions to be included in the server by selecting a set of modules. The modules can be compiled either statically as part of the 'httpd' binary, or as Dynamic Shared Objects (DSOs). DSO modules can either be compiled when the server is built, or added later via the apxs utility, which allows compilation at a later date. The mod_so module must be statically compiled into the Apache core to enable DSO support.

Run Apache with only the required modules. This reduces the memory footprint, which improves the server performance. Statically compiling modules will save RAM that's used for supporting dynamically loaded modules, but you would have to recompile Apache to add or remove a module. This is where the DSO mechanism comes handy. Once the mod_so module is statically compiled, any other module can be added or dropped using the 'LoadModule' command in the 'httpd.conf' file. Of course, you will have to compile the modules using 'apxs' if they weren't compiled when the server was built. 2.2 Choose appropriate MPM:

The Apache server ships with a selection of Multi-Processing Modules (MPMs) which are responsible for binding to network ports on the machine, accepting requests, and dispatching children to handle the requests. Only one MPM can be loaded into the server at any time.

Choosing an MPM depends on various factors, such as whether the OS supports threads, how much memory is available, scalability versus stability, whether non-thread-safe third-party modules are used, etc.

Linux systems can choose to use a threaded MPM like worker or a non-threaded MPM like prefork:

The worker MPM uses multiple child processes. It's multi-threaded within each child, and each thread handles a single connection. Worker is fast and highly scalable and the memory footprint is comparatively low. It's well suited for multiple processors. On the other hand, worker is less tolerant of faulty modules, and a faulty thread can affect all the threads in a child process.

The prefork MPM uses multiple child processes, each child handles one connection at a time. Prefork is well suited for single or double CPU systems, speed is comparable to that of worker, and it's highly tolerant of faulty modules and crashing children - but the memory usage is high, and more traffic leads to greater memory usage. 3 Run-Time Configuration Options 3.1 DNS lookup:

The HostnameLookups directive enables DNS lookup so that hostnames can be logged instead of the IP address. This adds latency to every request since the DNS lookup has to be completed before the request is finished. HostnameLookups is Off by default in Apache 1.3 and above. Leave it Off and use post-processing program such as logresolve to resolve IP addresses in Apache's access logfiles. Logresolve ships with Apache.

When using 'Allow from' or 'Deny from' directives, use an IP address instead of a domain name or a hostname. Otherwise, a double DNS lookup is performed to make sure that the domain name or the hostname is not being spoofed. 3.2 AllowOverride:

If AllowOverride is not set to 'None', then Apache will attempt to open the .htaccess file (as specified by AccessFileName directive) in each directory that it visits. For example: DocumentRoot /var/www/html
<Directory />
AllowOverride all
</Directory>

If a request is made for URI /index.html, then Apache will attempt to open /.htaccess, /var/.htaccess, /var/www/.htaccess, and /var/www/html/.htaccess. These additional file system lookups add to the latency. If .htaccess is required for a particular directory, then enable it for that directory alone. 3.3 FollowSymLinks and SymLinksIfOwnerMatch: If FollowSymLinks option is set, then the server will follow symbolic links in this directory. If SymLinksIfOwnerMatch is set, then the server will follow symbolic links only if the target file or directory is owned by the same user as the link.

If SymLinksIfOwnerMatch is set, then Apache will have to issue additional system calls to verify whether the ownership of the link and the target file match. Additional system calls are also needed when FollowSymLinks is NOT set.
For example: DocumentRoot /var/www/html
<Directory />
Options SymLinksIfOwnerMatch
</Directory>
For a request made for URI /index.html, Apache will perform lstat() on /var, /var/www, /var/www/html, and /var/www/html/index.html. These additional system calls will add to the latency. The lstat results are not cached, so they will occur on every request.

For maximum performance, set FollowSymLinks everywhere and never set SymLinksIfOwnerMatch. Or else, if SymLinksIfOwnerMatch is required for a directory, then set it for that directory alone.
3.4 Content Negotiation:

Avoid content negotiation for fast response. If content negotiation is required for the site, use type-map files rather than Options MultiViews directive. With MultiViews, Apache has to scan the directory for files, which adds to the latency. 3.5 MaxClients:

The MaxClients sets the limit on maximum simultaneous requests that can be supported by the server; no more than this number of child processes are spawned. It shouldn't be set too low; otherwise, an ever-increasing number of connections are deferred to the queue and eventually time-out while the server resources are left unused. Setting this too high, on the other hand, will cause the server to start swapping which will cause the response time to degrade drastically.
The appropriate value for MaxClients can be calculated as:

MaxClients = Total RAM dedicated to the web server / Max child process size

The child process size for serving static file is about 2-3M. For dynamic content such as PHP, it may be around 15M. The RSS column
in "ps -ylC httpd --sort:rss" shows non-swapped physical memory usage by Apache processes in kiloBytes.

If there are more concurrent users than MaxClients, the requests will be queued up to a number based on ListenBacklog directive. Increase ServerLimit to set MaxClients above 256.
3.6 MinSpareServers, MaxSpareServers, and StartServers:

MaxSpareServers and MinSpareServers determine how many child processes to keep active while waiting for requests. If the MinSpareServers is too low and a bunch of requests come in, Apache will have to spawn additional child processes to serve the requests. Creating child processes is relatively expensive. If the server is busy creating child processes, it won't be able to serve the client requests immediately. MaxSpareServers shouldn't be set too high: too many child processes will consume resources unnecessarily.

Tune MinSpareServers and MaxSpareServers so that Apache need not spawn more than 4 child processes per second (Apache can spawn a maximum of 32 child processes per second). When more than 4 children are spawned per second, a message will be logged in the ErrorLog.

The StartServers directive sets the number of child server processes created on startup. Apache will continue creating child processes until the MinSpareServers setting is reached. This doesn't have much effect on performance if the server isn't restarted frequently. If there are lot of requests and Apache is restarted frequently, set this to a relatively high value.
3.7 MaxRequestsPerChild:

The MaxRequestsPerChild directive sets the limit on the number of requests that an individual child server process will handle. After MaxRequestsPerChild requests, the child process will die. It's set to 0 by default, the child process will never expire. It is appropriate to set this to a value of few thousands. This can help prevent memory leakage, since the process dies after serving a certain number of requests. Don't set this too low, since creating new processes does have overhead. 3.8 KeepAlive and KeepAliveTimeout:

The KeepAlive directive allows multiple requests to be sent over the same TCP connection. This is particularly useful while serving HTML pages with lot of images. If KeepAlive is set to Off, then for each images, a separate TCP connection has to be made. Overhead due to establishing TCP connection can be eliminated by turning On KeepAlive.

KeepAliveTimeout determines how long to wait for the next request. Set this to a low value, perhaps between two to five seconds. If it is set too high, child processed are tied up waiting for the client when they could be used for serving new clients. 4 HTTP Compression & Caching

HTTP compression is completely specified in HTTP/1.1. The server uses either the gzip or the deflate encoding method to the response payload before it is sent to the client. Client then decompresses the payload. There is no need to install any additional software on the client side since all major browsers support these methods. Using compression will save bandwidth and improve response time; studies have found a mean gain of %75.2 when using compression.

HTTP Compression can be enabled in Apache using the mod_deflate module. Payload is compressed only if the browser requests compression, otherwise uncompressed content is served. A compression-aware browser inform the server that it prefer compressed content through the HTTP request header - "Accept-Encoding: gzip,deflate". Then the server responds with compressed payload and the response header set to "Content-Encoding: gzip".

The following example uses telnet to view request and response headers: bash-3.00$ telnet www.webperformance.org 80
Trying 24.60.234.27...
Connected to www.webperformance.org (24.60.234.27).
Escape character is '^]'.
HEAD / HTTP/1.1
Host: www.webperformance.org
Accept-Encoding: gzip,deflate

HTTP/1.1 200 OK
Date: Sat, 31 Dec 2005 02:29:22 GMT
Server: Apache/2.0
X-Powered-By: PHP/5.1.1
Cache-Control: max-age=0
Expires: Sat, 31 Dec 2005 02:29:22 GMT
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 20
Content-Type: text/html; charset=ISO-8859-1

In caching, a copy of the data is stored at the client or in a proxy server so that it need not be retrieved frequently from the server. This will save bandwidth, decrease load on the server, and reduce latency. Cache control is done through HTTP headers. In Apache, this can be accomplished through mod_expires and mod_headers modules. There is also server side caching, in which the most frequently-accessed content is stored in memory so that it can be served fast. The module mod_cache can be used for server side caching; it is production stable in Apache version 2.2. 5 Separate server for static and dynamic content

Apache processes serving dynamic content take from 3MB to 20MB of RAM. The size grows to accommodate the content being served and never decreases until the process dies. As an example, let's say an Apache process grows to 20MB while serving some dynamic content. After completing the request, it is free to serve any other request. If a request for an image comes in, then this 20MB process is serving static content - which could be served just as well by a 1MB process. As a result, memory is used inefficiently.

Use a tiny Apache (with minimum modules statically compiled) as the front-end server to serve static contents. Requests for dynamic content should be forwarded to the heavy-duty Apache (compiled with all required modules). Using a light front-end server has the advantage that the static contents are served fast without much memory usage and only the dynamic contents are passed over to the big server.

Request forwarding can be achieved by using mod_proxy and mod_rewrite modules. Suppose there is a lightweight Apache server listening to port 80 and a heavyweight Apache listening on port 8088. Then the following configuration in the lightweight Apache can be used to forward all requests (except requests for images) to the heavyweight server:

```
ProxyPassReverse / http://%{HTTP_HOST}:8088/
RewriteEngine on
RewriteCond %{REQUEST_URI} !.*\.(gif|png|jpg)$
RewriteRule ^/(.*) http://%{HTTP_HOST}:8088/$1 [P]
```

All requests, except for images, will be proxied to the backend server. The response is received by the frontend server and supplied to the client. As far as client is concerned, all the responses seem to come from a single server. 6 Conclusion

Configuring Apache for maximum performance is tricky; there are no hard and fast rules. Much depends on

understanding the web server requirements and experimenting with various available options. Use tools like ab and httperf to measure the web server performance. Lightweight servers such as tux or thttpd can also be used as the front-end server. If a database server is used, make sure it is optimized so that it won't create any bottlenecks. In case of MySQL, mtop can be used to monitor slow queries. Performance of PHP scripts can be improved by using a PHP caching product such as Turck MMCache. It eliminates overhead due to compiling by caching the PHP scripts in a compiled state.

Article derived from linuxgazette.net