Linux Process Scheduler: An Insight into Optimized Web Services

Authors: Christopher Salvador Márquez Álvarez / César A. Espinosa Michel

October 14, 2016

1 Abstract

The accelerated rate of growth in the amount of web applications returns as a result an increase in the traffic that web servers must handle. This aggregated traffic, in addition to the demand of the clients to be served in a real time frame, leads to the requirement of a customized way to control web related resources. All processes of the web server are tied to the control that the OS scheduler has over them, and for the default settings, the scheduler is set to handle general purpose tasks instead of being optimized for web serving purposes. To address this issue, the use of custom settings into the scheduler will allow the daemons needed to run a web page (such as Apache, PHP and a SQL DB) to be handled by the OS as efficiently as possible. The results of the test will be the comparison in performance of a web server for different settings on the Linux Scheduler.

2 Introduction

Nowadays, most of business in the world have web pages and some of them provide web services. The world economy will be depending more and more from the connectivity of remote users trying to -for example- buy a large variety of items online or use, in general, online services. This means that in the upcoming future, the web servers must know how to handle multiple requests per second and thousands of requests each hour. The average rate is about 70 requests per second, which is over 250,000 page renders per hour and millions of users a day. Of course, the amount of users vary throughout the day, and normally there is a period of approximately 3 hours where this rate grows.

Apache HTTP Server Project is one of the most important HTTP servers in the market. Millions of web pages and services are running in these type of servers, and there are certain variables of the Linux Kernel that may optimize the access to memory of the users, the latency of the scheduler and the flexibility to swapping, in general, the thread management in the server. Other co-lateral services may be affected with this, such as PHP and SQL, which are the basis for the standard stack configuration LAMP (Linux, Apache, MySQL and PHP).

3 Theorethical Framework

In the last several years, the constant increase of the web services that are found across the internet demand the web servers to manage and optimize the requests per second these servers can handle. Day after day, more users are accessing the web pages of the companies, trying to acquire services, products or simply navigate through the interface, demanding the best quality of service. Apache server came out back in 1995, and it became an important basis for most HTTP servers all over the world. The latest version, Apache 2.x, is nowadays a consolidated general-purpose web server that provides flexibility, portability, and performance.

The basis of a good performance at hardware level for a web server, according to Apache's documentation, is the use of RAM. Each user that requests information to the web server naturally needs constant access to the main memory to perform all the possible operations that the web service itself provides to the user. This is the main reason of why a web server should never have to swap, because it will notably increase the latency of the user's request, decreasing the quality of services provided for the web service ("not fast enough"). Furthermore, swapping will also open the possibility of letting the server to load all the available memory. When swapping is not available, the scheduler keeps users in a ready queue until there is space available to access CPU and RAM, preventing the unnecessary creation of children and the overload of the server.

The rest of the performance of a web server, as Apache documentation stated, depends directly on the characteristics and hardware equipment of the server (CPU, network card, fast enough disks, etc.). So certainly, the possible software optimization that can be done at the operating system level must be focus entirely on the RAM and in the scheduler itself. Specifically, the handling of user requests and the way the web server distributes and manages the loads of these requests. Two variables will necessarily affect this performance: the time slice and the latency. Defining a proper time slice for a user requests may allow that more users access to the services provided by the server, preventing the amount of time that a user could take to perform certain process in the CPU. On the other hand, latency will provide a better service to the users, because it will let the user request to access and perform operations in the CPU in the least amount of time, reducing the waiting time of this request in the ready queue.

The modification of these three variables may optimize the performance of Apache HTTP server, as well as the performance of the co-lateral services such as PHP and MySQL. The result of this will allow the web server to handle more requests per second, providing service to a larger range of users and performing a better quality of service to them.

4 Objective

The objective of this research is to find the optimum configuration of a selection of the scheduler kernel variables, such that a basic web server stack composed of Apache, PHP and MySQL database can run as efficiently as possible.

5 Justification

Nowadays, the traffic in Internet is increasing exponentially and the load of requests that servers must process is huge. For websites to stay competitive, that is, to be able to serve as much users as possible concurrently, an optimized operating system is fundamental to accomplish such task. Any web stack that is used to power a web service is ultimately run like any other process in an operating system. Thus, the way that the operating system manages the processes will affect the performance of the web server.

The operative system's process scheduler is the piece of software in charge of managing what process should run in the computer, to schedule their order of execution and to properly utilize the hardware (in this case CPU and Main Memory) in such a way that all processes that are created can be executed and fulfill their purpose. However, depending on the settings of the scheduler is the preference that is given to some concepts, such as fairness and equal access to resources. These settings when properly modified can benefit or harm some other process depending on the nature of that process. This is the premise in which this investigation is sustained. That, with proper customization, a web server can gain substantial performance just by modifying the scheduler behavior in a way that benefits the web serving daemons.

6 Development

The methodology to accomplish the stated objective went as follows. The variables chosen to be modified in the different test were selected in basis to the services required to be run in the server. Apache works by spawning a new thread per user connection, so for each connection that is established between a client and the server, a new lightweight process is generated. The operations performed by PHP and SQL are called from each of these processes; thus variables that affect process scheduling (specially those that involve a context switch) were selected. These three settings are:

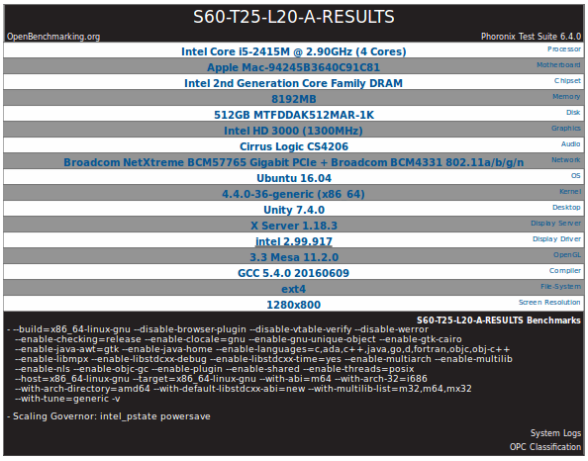
- **Timeslice**: Defined as the maximum time that a single process is allowed to take control of the CPU at once. The name of the Linux kernel variable is kernel.sched_rr_timeslice_ms. Its default value is set to 25ms.
- **Latency**: Latency denotes the time between a process is selected by the scheduler to run in a CPU and when that process starts to run in its assigned CPU. The variable related to this setting is kernel.sched_latecy_ns. The default value is set to 20ms.
- **Swappiness**: The swappiness variable controls how likely it is for a process to be swapped from main memory into the swapping space. The closer the value is to 0 is the less likely that a process will be swapped from main memory, the opposite happens at the value of 100 when it is the most likely that the process will be swapped. The name of the variable is vm.swappiness and its set to 60 by default.

With these variables, the parameters selected for testing were the default value of the variable as well as a high value close to the upper bound allowed by the setting and a low value close to the lower bound. Then, all the permutations

of the three values are to be tested by a bench-marking software, in this case, Phoronix Test Suite. This software includes, among many other tests, an Apache Test, a PHP Test and a test for the MySQL database.

Finally, as all the data from the bench-marking suite is produced, statistics will be generated from this data that allow to shed some light over those parameters that are tightly related to the daemons performance and better optimize a computer to be a web server.

To perform the testing, two laptop PC with different hardware were utilized, to ensure that the changes in performance are replicated in distinct hardware devices and that indeed the changes were produced by tuning the OS scheduler. The first computer for testing is an Asus Laptop, the second is a MacBook Pro. Both computers were tested with the same bench-marking software and both were running Ubuntu 16.04 as their base OS. The full specs can be read in the following tables.

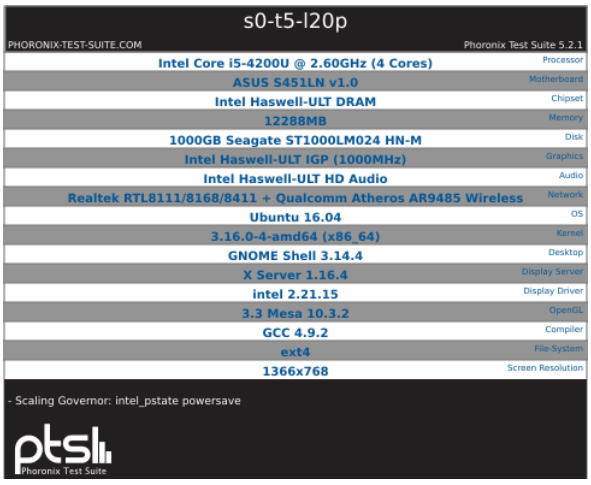


S60-T25-L20-A-RESULTS	
OpenBenchmarking.org	Phoronix Test Suite 6.4.0
Intel Core i5-2415M @ 2.90GHz (4 Cores)	Processor
Apple Mac-94245B3640C91C81	Motherboard
Intel 2nd Generation Core Family DRAM	Chipset
8192MB	Memory
512GB MTFDDAK512MAR-1K	Disk
Intel HD 3000 (1300MHz)	Graphics
Cirrus Logic CS4206	Audio
Broadcom NetXtreme BCM57765 Gigabit PCIe + Broadcom BCM4331 802.11a/b/g/n	Network
Ubuntu 16.04	OS
4.4.0-36-generic (x86_64)	Kernel
Unity 7.4.0	Desktop
X Server 1.18.3	Display Server
intel 2.99.917	Display Driver
3.3 Mesa 11.2.0	OpenGL
GCC 5.4.0 20160609	Compiler
ext4	File-System
1280x800	Screen Resolution

S60-T25-L20-A-RESULTS Benchmarks

- --build=x86_64-linux-gnu --disable-browser-plugin --disable-vtable-verify --disable-werror --enable-checking=release --enable-clocale=gnu --enable-gnu-unique-object --enable-gtk-cairo --enable-java-awt=gtk --enable-java-home --enable-languages=c,ada,c++,java,go,d,fortran,objc,obj-c++ --enable-libmpx --enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-multiarch --enable-multilib --enable-nls --enable-objc-gc --enable-plugin --enable-shared --enable-threads=posix --host=x86_64-linux-gnu --target=x86_64-linux-gnu --with-abi=m64 --with-arch-32=i686 --with-arch-directory=amd64 --with-default-libstdcxx-abi=new --with-multilib-list=m32,m64,mx32 --with-tune=generic -v
- Scaling Governor: intel_pstate powersave

System Logs
OPC Classification



s0-t5-l20p	
PHORONIX-TEST-SUITE.COM	Phoronix Test Suite 5.2.1
Intel Core i5-4200U @ 2.60GHz (4 Cores)	Processor
ASUS S451LN v1.0	Motherboard
Intel Haswell-ULT DRAM	Chipset
12288MB	Memory
1000GB Seagate ST1000LM024 HN-M	Disk
Intel Haswell-ULT IGP (1000MHz)	Graphics
Intel Haswell-ULT HD Audio	Audio
Realtek RTL8111/8168/8411 + Qualcomm Atheros AR9485 Wireless	Network
Ubuntu 16.04	OS
3.16.0-4-amd64 (x86_64)	Kernel
GNOME Shell 3.14.4	Desktop
X Server 1.16.4	Display Server
intel 2.21.15	Display Driver
3.3 Mesa 10.3.2	OpenGL
GCC 4.9.2	Compiler
ext4	File-System
1366x768	Screen Resolution

- Scaling Governor: intel_pstate powersave

ptsl
Phoronix Test Suite

7 Results

The results of the following graph were obtained by testing the Apache Benchmark program of Phoronix Test Suite. As described in the official website, the test measures "how many requests per second a given system can sustain when carrying out 1,000,000 requests with 100 requests being carried out concurrently" (OpenBenchmarking.org, n.d.). In total, 27 tests were done, and they are the result of combining the three variables (Swappiness, Time Slice and Latency) with each one of their possible values, resulting 3^3 total tests. The Apache Test does three tests and obtains the average score (which are the requests per second); if the tests have a high standard deviation, it repeats the test until it gets coherent results, to avoid outliers. Each of the test lasted about 3 minutes and the three of them would be finished by the eleven minutes mark. The statistics of the obtained results can be seen in the following table and graph.

Mean	10219.10
Standard Dev.	967.98
Max Value	11877.68
Min Value	7717.33
Range	4160.35
Max Absolute Inc.	8.22%
Max Relative Inc.	27.68%

Table 1: Apache Statistics

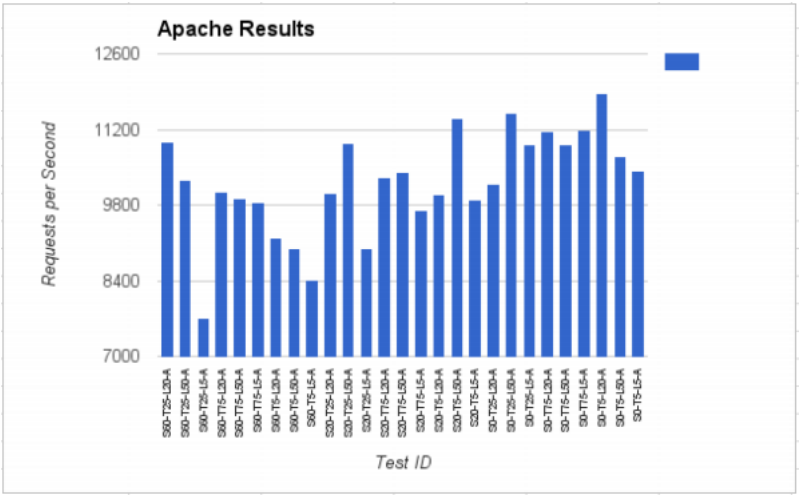


Figure 1: Apache Results

The testing for PHP performance was done very similarly to the Apache Benchmark. There is an estimated test count of three and at the end the

benchmark returns the three results and their average. In this case the score is measured just as points and not as a particular unit like Apache. Each of the test lasted about 90 seconds and the three of them would be finished by the five minutes mark. As the description of the benchmark states, the test consists of a large number of simple tests in order to bench various aspects of the PHP interpreter. The number of iterations used per unit test is 1,000,000. From the combinations made with the scheduler variables, the best combination was setting swappiness to 20, latency to 20 ms and timeslice to 5ms. The statistics of the obtained results can be seen in the following table and graph.

Mean	99873.63
Standard Dev.	1339.30
Max Value	101581.00
Min Value	96620.00
Default	96620.00
Range	4961.00
Max Performance Inc.	5.13%

Table 2: PHP Statistics

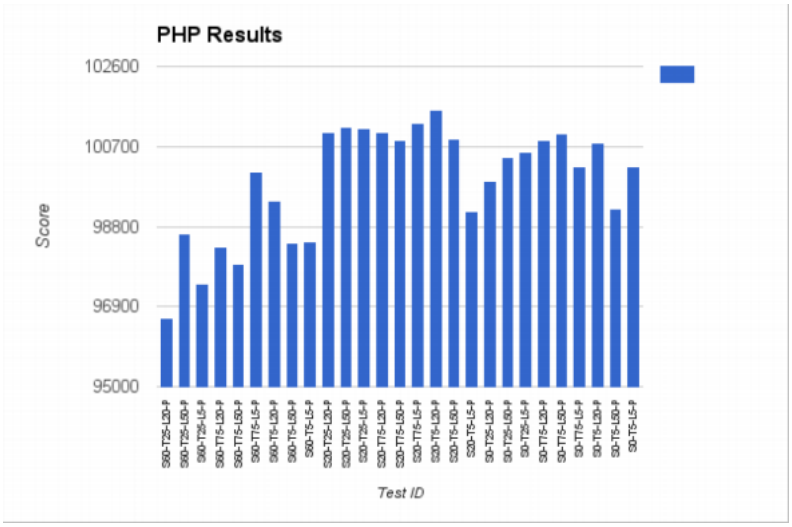


Figure 2: PHP Results

The final results for the overall web server analysis were based on both Apache and PHP results. In order to mix these results, an special value was created, which consists on the sum of the PHP test score and 10 times the requests per second of the Apache tests. This was just to acknowledge a final punctuation of the tests and to have a real comparison between the tests.The statistics of the obtained results can be seen in the following table and graph.

Mean	202064.66
Standard Dev.	10293.78
Max Value	219575.8
Min Value	174609.3
Range	44966.5
Max Absolute Inc.	6.40%
Max Relative Inc.	41.55%

Table 3: Web Server Statistics

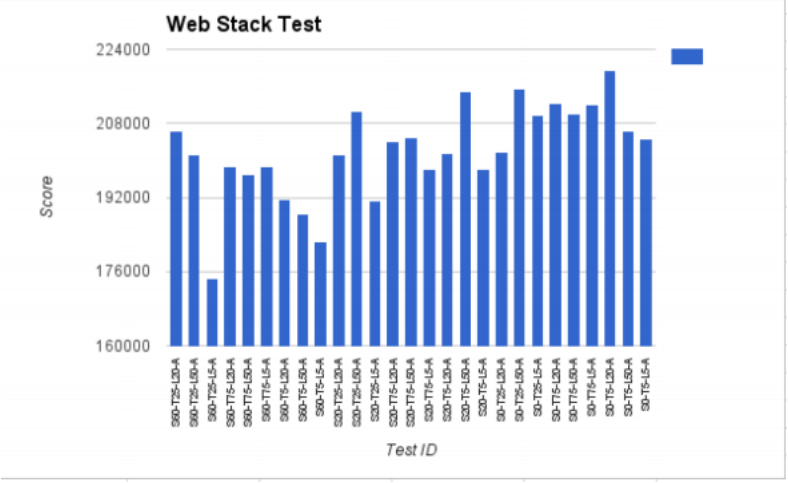


Figure 3: Web Server Results

8 Conclusions

The results obtained through the experiment led to the following conclusions. It can be observed that in general the performance of the server tends to increase as the value of the swappiness is reduced, regardless of the values of the other test variables. This tells something interesting about the behavior of the system and swapping. The less the server has to swap the processes, the more performance it will get. Swapping is an expensive operation in terms of memory and CPU, as it has to deallocate the resources from the RAM and the CPU registers (like a context switch) and transfer them to secondary storage, usually a disk much slower than RAM. Eventually, the server process will be swapped back into RAM and continue to be executed by the CPU. Thus, each time swapping happens, the process has to be transfer into and out of the disk, which lowers performance considerably. As seen in the state of the art, Apache recommends to swap as little as possible, theory that gets validated with the experiment results.

Another important insight that the results showed, was that the optimal configuration in these tests is when latency is 20 ms, and the time slice is 5 ms. The top marks for both PHP and Apache tests (S20-T5-L20-P and S0-T5-L20-A, respectively) had this configuration, as well as other remarkable tests like S20-T5-L20-A from Apache and S0-T5-L20-P from PHP (both ranked in

the top 5 tests). Recalling the theoretical framework and development sections, the time slice variable defined the time in which a process could stay in the CPU, and the latency was the time that the scheduler let the processes in the ready queue. In a web server, it is important to serve to the most amount of users with the best possible service; this configuration allows many processes to work concurrently (thanks to the short time slice) but it also coordinates them correctly so they do not wait much time, but enough to allow these processes to access the CPU.

References

[1] CPU Scheduling. (n.d.). Retrieved October 20, 2016, from https://access.redhat.com/documentation/enUS/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-cpu-scheduler.html

[2] Tuning the Task Scheduler. (n.d.). Retrieved October 20, 2016, from https://doc.opensuse.org/documentation/html/openSUSE_121/opensuse-tuning/cha.tuning.taskscheduler.html

[3] Tuning Virtual Memory. (n.d.). Retrieved October 20, 2016, from https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-memory-tunables.html

[4] The Apache HTTP Server Project. (n.d.). Retrieved October 20, 2016, from https://httpd.apache.org/

[5] Apache Performance Tuning. (n.d.). Retrieved October 20, 2016, from http://httpd.apache.org/docs/2.4/misc/perf-tuning.html

[6] Apache Benchmark Test Profile. (n.d.). Retrieved October 20, 2016, from https://openbenchmarking.org/test/pts/apache-1.6.1