Tecnológico de Monterrey

Operating Systems Lecture

(TC2008)

Modifying Linux Kernel

Authors:

Adair Ibarra Bautista

Oscar Arturo Zapata Buenrostro

Professor:
Victor Rodríguez Bahena

Abstract

In this document we will focus on modifying the Linux Kernel through memory and scheduler parameters. The main objective is to study the performance of a computer during the execution of AIO-Stress Benchmark. It was necessary to run the test several times since three of the parameter mentioned in this project were modified 5 times. After completing the test, the results were displayed on graphs, showing that all the variables have a noticeable influence on the performance of the computer.

Contents

1 Introduction

The AIO-Stress Test is a simple workload generator for systems. It imposes a configurable amount of CPU, memory, I/O, and disk stress on the computer. This test allowed us to determine which parameter has a greater influence on the performance of the computer and helped us to define how the computer behaves under specific circumstances.

The purpose of this paper is to present a case study to explore the benefits or disadvantages of modifying kernel and memory parameters of the scheduler (commonly known as "scheduler tuning"). In order to accomplish this, we will use an AIO Stress tool for our benchmarks and experiments

2 Theoretical Framework

Scheduling policies are divided into two major categories:

- Real time policies
  1. SCHED-FIFO
  2. SCHED-RR
- Normal policies

2.1 Real time policies

Real time threads are scheduled first, and normal threads are scheduled after all Real-time threads have been scheduled. The Real time policies are used for time-critical tasks that must complete without interruptions. [1]

2.1.1 SCHED-FIFO policy

This policy is also referred to as static priority scheduling, because it defines a fixed priority (between 1 and 99) for each thread. The scheduler scans a list of SCHED-FIFO threads in priority order and schedules the highest priority thread that is ready to run. This thread runs until it blocks, exits, or is preempted by a higher priority thread that is ready to run.

In the Linux kernel, the SCHED-FIFO policy includes a bandwidth cap mechanism. This protects Real time application programmers from Real-time tasks that might monopolize the CPU. [1]

2.1.2 SCHED-RR policy

SCHED-RR is a round-robin variant of the SCHED-FIFO policy. SCHED-RR threads are also given a fixed priority between 1 and 99. However, threads with the same priority are scheduled round-robin style within a certain quantum, or time slice.

2.2 Scheduler tuning

The scheduler offers a number of parameters which allow to tune its behavior to actual needs:

- *sched rt runtime us*: The maximum CPU time that can be used by all the real-time tasks (1 second by default). When this amount of time is used up these tasks must wait for *sched rt period us* before the are allowed to be executed again. [2]
- *sched rt period us*: The scheduler waits this amount of time (0.95 s by default) before scheduling any of the real-time tasks again. [2]
- *sched min granularity ns*: This parameter decides the minimum time a task will be be allowed to run on CPU before being pre-empted out. By default, it is set to 4ms. So by default, any task will run at least 4ms before getting pre-empted out. [3]

- *sched latency ns*: This parameter, together with *sched min granularity ns*, decides the scheduler period, the period in which all run queue tasks are scheduled at least once. [3]

- *sched migration cost ns*: Determines how long a migrated process has to be running before the kernel will consider migrating it again to another core.

- *sched nr migrate*: This option can be set to specify the number of tasks that will move at a time.

- *sched rr timeslice ms*: Threads that have the same priority are scheduled round-robin style within a certain time slice. You can set the value of this time slice in milliseconds with the *sched rr timeslice ms*.

- *Swappiness*: Is a parameter that controls the relative weight given to swapping out runtime memory, as opposed to dropping pages from the system page cache. Swappiness can be set to values between 0 and 100 inclusive. A low value causes the kernel to avoid swapping, a higher value causes the kernel to try to use swap space.

3 Objective

The objective of this project is to ”get our hands dirty” modifying real variables of a real operating system and see how this changes actually affect the performance of a given process or service of the operating system.

Along all this, it is important to know how benchmarks are done and how to read the results, including taking decisions about the best values that can improve the operating system.

4 Methodology

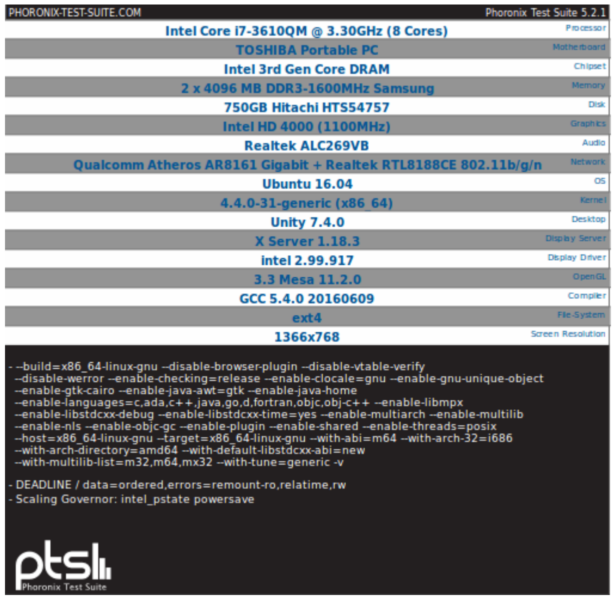
In this project we modified three different scheduler variables to observe the effect this variables have performance-wise. The variables we changed are: vm.swappiness, kernel.sched-latency-ns and finally, kernel.sched-rt-runtime-us.

The benchmark we ran to test this values is phoronix AIO-Stress.

The process followed for this tests was giving five different values to each variable, leaving all the scheduler variables with their default values (even the ones being evaluated in this project) and running the test. Once finished with the five tests, we analyzed the graphs produced by phoronix.

The detailed information about the values given to each variable will be seen in the following sections.

All this tests were performed in a laptop computer. The technical specifications are shown in Figure 1.



PHORONIX-TEST-SUITE.COM	Phoronix Test Suite 5.2.1
Intel Core i7-3610QM @ 3.30GHz (8 Cores)	Processor
TOSHIBA Portable PC	Motherboard
Intel 3rd Gen Core DRAM	Chipset
2 x 4096 MB DDR3-1600MHz Samsung	Memory
750GB Hitachi HTS54757	Disk
Intel HD 4000 (1100MHz)	Graphics
Realtek ALC269VB	Audio
Qualcomm Atheros AR8161 Gigabit + Realtek RTL8188CE 802.11b/g/n	Network
Ubuntu 16.04	OS
4.4.0-31-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
1366x768	Screen Resolution

```
- --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

- DEADLINE / data=ordered,errors=remount-ro,relatime,rw
- Scaling Governor: intel_pstate powersave
```

Figure 1: Technical specifications of the computer

4.1 Swappiness

This was the first variable we modified. We changed the value of the variable and then, ran the test. We repeated this process five times.

The default value of this variable is 60. The possible values for the scheduler swappiness ranges from 0 to 100.

Maintaining the default values for the scheduler variables, we gave to the swappiness the values seen in Table 1.

Value
20%
40%
60%*
80%
100%

Table 1: Values given to the swappiness variable

4.2 Latency

Returning the swappiness variable to its default value, we then changed the values of the latency time.

The default value for this variable is 24ms.

The values given to this variable for the AIO-Stress benchmark can be seen in Table 2.

Value
6ms
12ms
24ms*
48ms
72ms

Table 2: Values given to the latency variable

4.3 Runtime

The last variable we changed is the runtime variable. The values this variable can take range from 0 to 1'000,000 [μs]. The default value is 950,000μs.

The values that were used for the test are shown in 2

Value
0.01s
0.25s
0.50s
0.75s
0.95s*

Table 3: Values given to the runtime variable

5 Results

Once done with all the benchmarks, phoronix helped graphing the results and giving some interesting overviews about the tests.

Having these graphs, and analyzing them we couldn't find any patterns about the results. We found out that there is no sure result about this test.

In the following sections the results will be explained along with their graphs and a result overview provided by phoronix. Since we are using the AIO-Stress benchmark it is important to know that all the results are given in MB/s.

For all the results in this test, the greater the number is, the better.

5.1 Swappiness

In Figure 2 we have a bar chart representing the performance of the computer in this benchmark for all the different runs. This chart is quite useful given that you can see more easily the difference between each run.

With the default value of swappiness (60), we get a data transmission rate of 381.88MB/s.

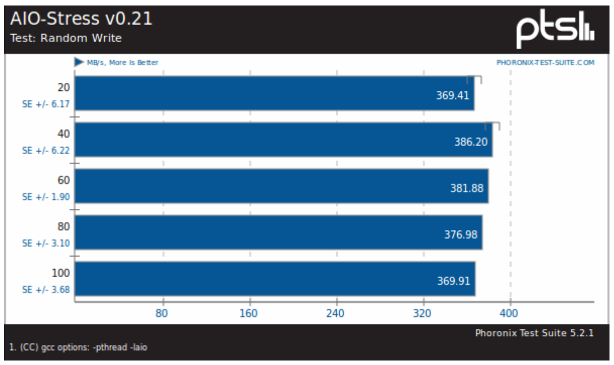
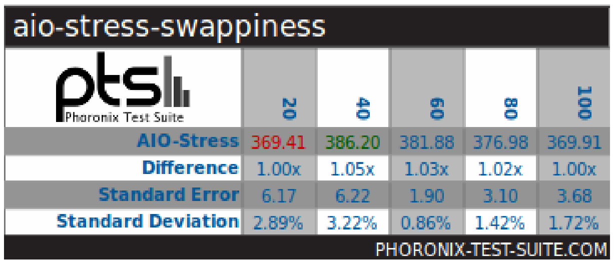


Figure 2: Bar chart of performance for swappiness variable.

In Figure 3 we can see that the best variable value for swappiness is 40, giving a transmission rate of 386.20MB/s. While the worst value is a swappiness of 20 with a transmission rate of 369.41, 12.47MB/s less than the default value.

Recalling that the swappiness of the scheduler is "how much" swapping between the ROM and RAM is performed between processes, we can get from this test that a good swapping percentage is 40%. This means, trying to maintain as much information possible in RAM, in order to avoid access to ROM but not too much information.



aio-stress-swappiness

Phoronix Test Suite	20	40	60	80	100
AIO-Stress	369.41	386.20	381.88	376.98	369.91
Difference	1.00x	1.05x	1.03x	1.02x	1.00x
Standard Error	6.17	6.22	1.90	3.10	3.68
Standard Deviation	2.89%	3.22%	0.86%	1.42%	1.72%

PHORONIX-TEST-SUITE.COM

Figure 3: Result overview of different runs.

5.2 Latency

Regarding to latency, at first, we were thinking that the results of this test were going to be quite obvious. Of course with less latency everything would work better, right?

Wrong.

With a default latency value of 24ms, we started the benchmark. Later on we started increasing this value just to prove that our theory was right. The surprise came when we started to decrease this value. In Figure 4 we can see the bar chart for this test, Even if we decreased the value of latency it didn't improve.

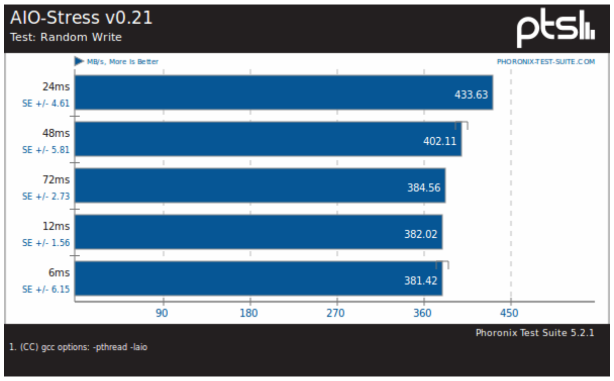
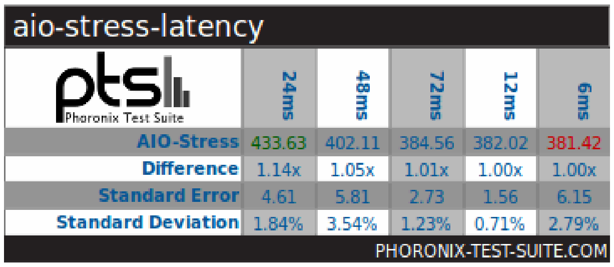


Figure 4: Bar chart of performance for latency variable.

In Figure 5 we can see that the best result (and for a big difference) is the default value of 24ms. The second best is a value of 48ms with a difference of 31.52MB/s.



aio-stress-latency

Phoronix Test Suite	24ms	48ms	72ms	12ms	6ms
AIO-Stress	433.63	402.11	384.56	382.02	381.42
Difference	1.14x	1.05x	1.01x	1.00x	1.00x
Standard Error	4.61	5.81	2.73	1.56	6.15
Standard Deviation	1.84%	3.54%	1.23%	0.71%	2.79%

PHORONIX-TEST-SUITE.COM

Figure 5: Result overview of different runs.

So with the results from this test, we

realize that the system actually needs latency to be able to work properly or with a good performance. And that not necessarily has to be the smallest number possible, but just the right number that allows the system to work as it should. Maybe, reducing the delay actually makes some processes slower and that impacts to the data transmission rate.

5.3 Runtime

Finally, we put to test the kernel.sched-rt-runtime-us variable. This test is the one that generated a smaller difference between the values given to the variable;but still, we found a value that improves the performance.

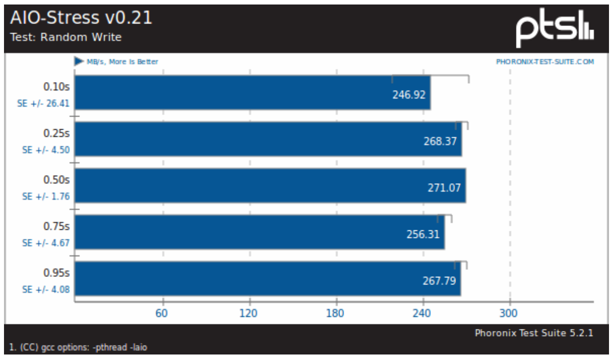
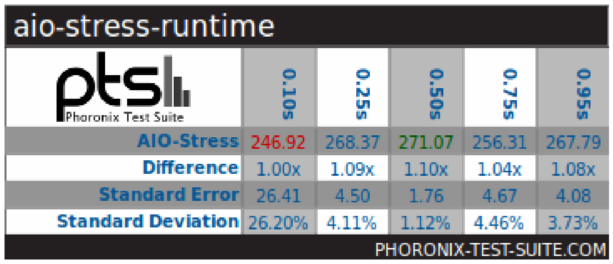


Figure 6: Comparison graph of performance obtained with each test run.

The improvement in with this variable, was only of 3.28MB/s. To get this improvement, we changed from the default runtime value (0.95s) to 0.50s. We also found another value (0.25s) that performs better than the default, this value gives an improvement of 0.58MB/s.

The worst value in this test was 0.10s.



aio-stress-runtime

	0.10s	0.25s	0.50s	0.75s	0.95s
AIO-Stress	246.92	268.37	271.07	256.31	267.79
Difference	1.00x	1.09x	1.10x	1.04x	1.08x
Standard Error	26.41	4.50	1.76	4.67	4.08
Standard Deviation	26.20%	4.11%	1.12%	4.46%	3.73%

PHORONIX-TEST-SUITE.COM

Figure 7: Result overview of different runs.

From this last test we get that we can reduce almost half a second the time that we are letting real-time processes to be executed in the CPU. For the AIO-Stress test, we give this processes more time than they need, so reducing this variable improves the data transmission.

6 Conclusion

It is very important to keep in mind that these results are only for the AIO-Stress benchmark. These results doesn't mean that changing the values of the variables for the ones we got, will improve the operative system. These results are only dependent of I/O operations.

This analysis is very useful for very specific situations where you want a specific process or service of the operative system to be faster either in a server, a micro-controller or even a personal computer. For this situations, you will have to look for the benchmark related to the service you want to improve and perform several test runs with a lot of different values to get the optimum combination of values in the variables.

Although some improvements can be not very significant, for a purpose-specific application, a difference of 3.28MB/s can be a transcendental difference.

For future work, it would be interesting to keep looking for the best value for the different variables, considering that in this tests we left a big gap between each test run. However We have a good reference to start in order to keep improving data transmission for this benchmark.

Another interesting thing that could be done is analyze if the individual values that we got, actually improve when they are all put together. Or if the combination of the values that we got is worse than the default values.

References

[1] (2016). CPU Scheduling. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-cpu-scheduler.html

[2] Kobus, J. Szklarski, R. (2016). Completely Fair Scheduler and its tuning (1st ed.). https://www.fizyka.umk.pl/~jkob/prace-mag/cfs-tuning.pdf

[3] Linux Scheduler. (2012). Oakbytes. https://oakbytes.wordpress.com/2012/06/06/linux-scheduler-cfs-and-latency/