CFS Scheduler Performance Evaluation by Varying its Parameters CSE506

Jagadeesh Reddy Vanga, Muskan Gupta
{jagadeeshreddy.vanga, muskan.gupta.1}@stonybrook.edu

Abstract

The CFS (Completely Fair Scheduler) is a process scheduler used in the Linux kernel since version 2.6.23. It is designed to provide fair CPU resource allocation among processes in a multiprogramming environment. The CFS scheduler aims to distribute CPU time fairly among all running processes regardless of their priority. Here, we evaluate the performance of CFS Scheduler by varying its different parameters on Linux Virtual Machine.

1 Introduction

Key Features of the CFS Scheduler:

Fairness: The CFS scheduler treats all processes fairly, ensuring that each process receives an equal share of the CPU time over a given period. It avoids situations where a single process monopolizes the CPU resources, leading to poor performance of other processes.

Time-based Scheduling: The CFS scheduler uses a virtual runtime concept for each process, which represents the amount of CPU time a process has received. It employs a red-black tree data structure to keep track of processes based on their virtual runtimes. The process with the smallest virtual runtime gets the CPU next.

Dynamic Priorities: The CFS scheduler does not rely on static priorities like traditional schedulers do. Instead, it uses a concept called "niceness" to assign dynamic priorities to processes. The niceness value ranges from -20 to 19, with lower values indicating higher priority. The scheduler automatically adjusts the niceness of processes based on their resource usage and behavior.

Sleep Control: The CFS scheduler efficiently handles situations where a process is put to sleep, such as when it is waiting for I/O or other resources. When a process is asleep, its virtual runtime does not increase, allowing the scheduler to accurately account for the time spent waiting. Once the process wakes up, it resumes its execution with the same relative position in the scheduling queue.

Constant Time Complexity: The CFS scheduler ensures constant time complexity for core operations, such as process enqueueing, dequeueing, and context switching. This property allows the scheduler to scale well even in environments with a large number of processes.

Control Groups (cgroups) Support: CFS integrates with control groups, a Linux kernel feature that provides resource management and accounting capabilities. With cgroups, administrators can allocate CPU shares to different groups of processes, allowing fine-grained control over resource allocation.

The CFS scheduler has been widely adopted due to its fairness and responsiveness. It provides a smooth and efficient scheduling mechanism, enabling Linux systems to effectively handle diverse workloads and optimize overall system performance.

In this project, we are testing the performance of CFS in Linux by changing different parameters such as delay frequency, runtime, time slicing etc.

2 Motivation

Performance Optimization: Adjusting kernel scheduler variables can potentially improve system performance by fine-tuning the scheduling behavior to better match the specific workload requirements. By customizing these variables, you can optimize CPU allocation, reduce latency, and improve overall system responsiveness.

Real-Time Workloads: If you are working with real-time or time-sensitive applications, modifying kernel scheduler variables can help ensure that critical tasks receive the necessary resources and are executed within their speci-

fied deadlines. By fine-tuning parameters like sched_rt_runtime_us and sched_latency_ns, you can prioritize real-time tasks and provide them with the required CPU time and scheduling latency.

Resource Allocation: Changing kernel scheduler variables allows you to allocate system resources more efficiently. By adjusting parameters like sched_min_granularity_ns and sched_wakeup_granularity_ns, you can control the granularity of task scheduling, providing better resource utilization and fair distribution among competing tasks.

Workload-specific Optimization: Different workloads have different characteristics and requirements. Customizing kernel scheduler variables enables you to tailor the scheduling behavior to match your specific workload. Whether it's a compute-intensive workload, interactive applications, or multi-threaded scenarios, adjusting variables such as sched_child_runs_first or sched_rr_timeslice_ms can optimize resource allocation and improve overall performance.

System Responsiveness: A well-tuned kernel scheduler ensures that tasks are scheduled promptly, reducing latency and improving system responsiveness. By modifying variables like sched_latency_ns and sched_min_granularity_ns, you can minimize the time between a task becoming eligible for execution and its actual execution, resulting in a more responsive system for both interactive and background tasks.

Customization and Fine-tuning: Modifying kernel scheduler variables allows you to customize the behavior of the scheduler based on your specific requirements and system characteristics. It gives you the flexibility to fine-tune the scheduling parameters to achieve the desired trade-off between performance, fairness, and real-time responsiveness.

3 Methodology

We are choosing some parameters that affects the performance of scheduler, the most and then we vary those parameters to see the impact of it on the performance of the scheduler and operating system as a whole.

For this project, we have used Ubuntu 18.06 because, After kernel version 5.10, some of the parameters have been moved to /kernel/sched.h from include/linux/sched/sysctl.h [1]. Which made changing these variables from sysctl haeder. In Ubuntu 18.06, the default kernel is 5.4 which makes this a favorable OS with relatively latest kernel features. We have deployed this on a Virtual Machine with the following system configuration:

Processor: Intel Core i5-1145G7 (4 Cores)
Motherboard: Intel 440BX (6.00 BIOS)
Chipset: Intel 440BX/ZX/DX
Memory: 8GB
Disk: 107GB VMware Virtual S
Graphics: VMware SVGA II
Audio: Ensoniq ES1371/ES1373
Network:Intel 82545EM
OS: Ubuntu 18.0
Kernel: 5.4.0-148-generic(x86 64)
Desktop: GNOME Shell 3.28.4
Display Server: X Server 1.20.8
Compiler: GCC 7.5.0
File-System: ext4
Screen Resolution: 1718x878
System Layer: VMware

In this experiment, we have selected 4 variables which can affect the system performance by changing how CFS works. The variables are:

```
kernel.sched_rr_timeslice_ms
kernel.sched_latency_ns
kernel.sched_wakeup_granularity_ns
kernel.sched_rt_runtime_us
```

To test these changes, we are using **Apache HTTP Server** and benchmark the throughput after changing the values of the above kernel variables. Each run have 6 different concurrences like 4, 20, 100, 200, 500, 1000 requests. The resulting throughput is compared with other profiles. *** Note: here _us, _ms means micro seconds and milli seconds**

3.1 kernel.sched_rr_timeslice_ms

The 'kernel.sched_rr_timeslice_ms' parameter in the kernel is used to set the time slice or time quantum for tasks scheduled under the Round-Robin (RR) scheduling policy. The RR scheduling policy is a preemptive scheduling algorithm that assigns a fixed time slice to each task in a cyclic manner, allowing them to run in a round-robin fashion.

The parameter specifies the length of the time slice in milliseconds for tasks scheduled under the RR policy. Each task is allowed to run for the duration of one time slice before being preempted and replaced by the next task in the scheduling queue.

By default, the value of kernel.sched_rr_timeslice_ms is usually set to a relatively small value, such as 100 milliseconds. However, this default value can vary depending on the Linux distribution and kernel version.

The default value for the time slice is 100ms which we have changed to 150 ms, 10ms and 50ms which produced the output given as follows:

	round_robin_150 ms	round_robin_10m s	round_robin_50m s	Ran with out-of-box config
Apache HTTP Server - 4 (Reqs/sec)	13451	13260	13352	4365
Normalized	100%	98.58%	99.26%	32.45%
Standard Deviation	0.1%	2.1%	0.8%	0.6%
Apache HTTP Server - 20 (Reqs/sec)	16491	16180	16317	5354
Normalized	100%	98.11%	98.94%	32.47%
Standard Deviation	0.4%	0.3%	1.7%	0.5%
Apache HTTP Server - 100 (Reqs/sec)	18768	18920	18794	13568
Normalized	99.2%	100%	99.33%	71.71%
Standard Deviation	1.2%	1.2%	0.1%	25.9%
Apache HTTP Server - 200 (Reqs/sec)	19184	19425	19291	15664
Normalized	98.76%	100%	99.31%	80.64%
Standard Deviation	0.4%	0.7%	0.7%	8.5%
Apache HTTP Server - 500 (Reqs/sec)	17973	17838	15069	6844
Normalized	100%	99.25%	83.84%	38.08%
Standard Deviation	0.1%	0.4%	21.5%	46.3%
Apache HTTP Server - 1000	17498	17845	17655	5685
Normalized	98.05%	100%	98.94%	31.86%
Standard Deviation	0.3%	0.4%	0.5%	1.1%

Figure 1: Results according to the time slice

The throughput with highest score in given concurrency spectrum is highlighted in green and lowest score is given red.

3.2 kernel.sched_latency_ns

The kernel.sched_latency_ns parameter in the Linux kernel is used to set the target scheduling latency for the scheduler. Scheduling latency refers to the time it takes for a task to start running after it becomes eligible for execution.

When a task becomes eligible for execution, the scheduler performs various operations, such as selecting the appropriate CPU, checking the task's priority, and preparing the CPU for the task's execution. These operations introduce some overhead and can cause a delay before the task actually starts running.

The kernel.sched_latency_ns parameter sets a target value for the maximum delay caused by these operations. It specifies the desired maximum scheduling latency in nanoseconds. The scheduler aims to keep the actual scheduling latency below this value, although it may vary depending on system load, CPU speed, and other factors.

Lower values for sched_latency_ns can reduce scheduling latency and improve the responsiveness of the system, especially in time-critical workloads. However, setting it too low can increase the overhead of the scheduler and potentially impact overall system performance. By default the latency was set to 18ms in Ubuntu 18.06 which we have altered to measure its performance under different circumstances. The results are shown in Figure 2.

	latency_ns_20 ms	latency_ns_5m s	latency_ns_1m s	latency_ns_10 ms	Ran with out-of-box
Apache HTTP Server - 4 (Reqs/sec)	13443	10193	9818	8600	4365
Normalized	100%	75.83%	73.03%	63.97%	32.47%
Standard Deviation	0.7%	0.3%	1%	9.2%	0.6%
Apache HTTP Server - 20 (Reqs/sec)	15299	17253	15112	10665	5354
Normalized	88.68%	100%	87.59%	61.82%	31.03%
Standard Deviation	6.6%	0.3%	2.2%	0.3%	0.5%
Apache HTTP Server - 100 (Reqs/sec)	17833	17945	16653	13050	13568
Normalized	99.37%	100%	92.8%	72.72%	75.61%
Standard Deviation	6.6%	0.7%	1.5%	1.9%	25.9%
Apache HTTP Server - 200 (Reqs/sec)	19266	18351	17696	12893	15664
Normalized	100%	95.25%	91.85%	66.92%	81.3%
Standard Deviation	0.2%	0.6%	2.7%	1.7%	8.5%
Apache HTTP Server - 500 (Reqs/sec)	18045	13820	15713	12317	6844
Normalized	100%	76.59%	87.08%	68.26%	37.93%
Standard Deviation	0.8%	13.7%	0.2%	2.2%	46.3%
Apache HTTP Server - 1000 (Reqs/sec)	17940	14932	15336	12733	5685
Normalized	100%	83.23%	85.48%	70.97%	31.69%
Standard Deviation	0.8%	0.5%	0.9%	11.7%	1.1%

Figure 2: Results for Kernel Schedule Latency

Here we tested the system by increasing and decreasing the default value of the parameter. We have increased it from 18ms to 20ms and then decreased it to 5, 10 and 1ms. The above table indicates the throughput with highest score in given concurrency spectrum is highlighted in green and lowest score is given red.

We can observe that the throughput have increased significantly when the value is set to 20ms. The default setting mostly performed worse than any other congifuration.

3.3 kernel.sched_wakeup_granularity_ns

The sched_wakeup_granularity_ns parameter specifies the minimum time interval between wakeups for a given task. When a task is waiting to be executed on a CPU, it is said to be in a "wait queue". The scheduler in the Linux kernel maintains a list of tasks waiting in each wait queue and decides which task to execute next based on a number of factors, including the priority of the task and the amount of time it has already spent waiting.

The sched_wakeup_granularity_ns parameter determines the minimum time that must elapse between successive wakeups of a task that is waiting in a particular wait queue. This parameter helps prevent tasks from being excessively preempted and rescheduled, which can cause unnecessary overhead and reduce overall system performance.

By default, the sched_wakeup_granularity_ns parameter is set to 3 millisecond (3000000 nanoseconds) on Ubuntu 18.06. We have changed

it between 1ms to 4ms and tested the throughput. The results are displayed in Figure 3

	4millisecond_wk_ gran	2millisecond_wk_ gran	1millisecond_wk_ gran	Ran with out-of-box config
Apache HTTP Server - 4 (Reqs/sec)	8731	4608	11064	4365
Normalized	78.91%	41.64%	100%	39.45%
Standard Deviation	5.7%	1.9%	1.7%	0.6%
Apache HTTP Server - 20 (Reqs/sec)	10365	5437	12626	5354
Normalized	82.09%	43.06%	100%	42.4%
Standard Deviation	2.2%	2.3%	2.2%	0.5%
Apache HTTP Server - 100 (Reqs/sec)	12250	5927	13322	13568
Normalized	90.29%	43.69%	98.19%	100%
Standard Deviation	15.5%	0.6%	1.6%	25.9%
Apache HTTP Server - 200 (Reqs/sec)	13650	5990	13874	15664
Normalized	87.14%	38.24%	88.57%	100%
Standard Deviation	2.1%	0.4%	1%	8.5%
Apache HTTP Server - 500 (Reqs/sec)	12797	5786	13083	6844
Normalized	97.81%	44.23%	100%	52.31%
Standard Deviation	0.5%	0.8%	20.4%	46.3%
Apache HTTP Server - 1000	12585	5748	16700	5685
Normalized	75.36%	34.42%	100%	34.04%
Standard Deviation	7.3%	1.2%	10.7%	1.1%

Figure 3: Results for Schedule Wakeup Granularity

The above results show the that best performing value is 1ms in concurrencies of 4, 20, 500 and 1000 and rest are led by out-of-the-box 3ms.

Optimizing the schedule wakeup granularity requires a balance between responsiveness and efficiency. In general, it is recommended to set the granularity to a value that is appropriate for the specific workload and system configuration. This may involve adjusting the kernel parameters and/or selecting a different scheduling algorithm or configuration.

3.4 kernel.sched_rt_runtime_us

The sched_rt_runtime_us parameter in the Linux kernel is used to set a run-time limit for real-time tasks scheduled under the real-time scheduling policy. The real-time scheduling policy is used for tasks that require deterministic and low-latency execution, such as audio and video processing, control systems, and other time-critical applications. Few Linux Operating Systems defaults this to 0, which means that there is no limit on the run-time of real-time tasks. In Ubuntu 18.06 the default value is 950000us/950ms. Setting sched_rt_runtime_us can help prevent real-time tasks from monopolizing CPU resources and ensure that other system tasks have sufficient CPU time to run. However, it can also introduce additional overhead and potentially impact the responsiveness and predictability of the system.

We have changed the value from 100ms to 1s and the results of the benchmark are shown in Figure 4.

As the results show, the Server throughput

	RUNTIME 1S INSTEAD OF 950MS	RUNTIME 100MS	RUNTIME 700 MS	RUNTIME 500 MS	RAN WITH OUT-OF-BOX CONFIG
Apache HTTP Server 4 (Reqs/sec ↑)	3843	8823	9787	13355	4365
Apache HTTP Server 20 (Reqs/sec ↑)	5430	10898	10818	16391	5354
Apache HTTP Server 100 (Reqs/sec ↑)	5501	13908	13228	18786	13568
Apache HTTP Server 200 (Reqs/sec ↑)	5570	14050	14680	19248	15664
Apache HTTP Server 500 (Reqs/sec ↑)	4806	13761	14862	17919	6844
Apache HTTP Server 1000 (Reqs/sec ↑)	6751	15780	4703	17832	5685

Figure 4: Results for Schedule Wakeup Granularity

peaked when the value is set to 500ms and performed worst when set to 1second.

4 Inferring Results and Future Work

The Out-of-the-box configurations of Ubuntu 18.06 were not optimized for the Apache Server Application. An OS should not optimize by just keeping one application as reference but to Generalize these variables such that different types of Applications at different system load can run smoothly.

The testing/experiment can be taken forward in terms of adding more test suites and applications to check their performance by varying not just the 4 variables we chosen but every available kernel.sched_ variable.

All of our tests were performed on Phoronix Test Suite [2] and results were uploaded to Openbenchmark and Github[3]

5 Conclusion

Different parameters have different aspect on performance and in this project we tried to analyse those impacts on throughput by changing the value of different parameters. The most common thing which can be noted from the results of all the parameters is that, by reducing the value of parameter if it is related to latency then the performance improves upto a certain threshold and then it starts degrading. One of the most possible reason for this is that if we put a lot of pressure for CPU cycles or performs context switching more frequently, than there are more chances that the performance will decrease.

Overall, the experiment had provided really good results and we tested and evaluated the

Ubuntu system under different conditions.

References

1. Change in Linux Kernel: https://github.com/torvalds/linux/commit/18765447c3b7867b3f8cccde52dc9d822852e71b

2. Phoronix Test Suite: https://www.phoronix-test-suite.com/

3. CSE506 CFS Experiment Results: https://github.com/jagadeesh-r1/CSE506-CFS-Experiment-Results