Comparative Performance and Scheduling Analysis Across Linux Kernel Versions 4.19 and 6.6

Isaac Stephens
Junior, Computer Science, CS Department
Missouri University of Science and Technology
Rolla, MO 65409
issq3r@mst.edu

Abstract – This project presents a controlled performance comparison between Linux kernel versions 4.19 and 6.6 using identical virtualized hardware environments. The study evaluates CPU throughput, context switch behavior, scheduling latency, and I/O-bound workload performance in order to analyze the long-term evolution of the Linux scheduling subsystem, particularly the Completely Fair Scheduler (CFS). All experiments where conducted within QEMU virtual machines running Debian 12 to ensure consistency across test conditions. Although Linux 2.6 was initially included in the proposed scope, it was excluded due to fundamental incompatibilities with modern toolchains and virtualization platforms. Benchmark results show that raw CPU throughput remains nearly identical between Linux 4.19 and Linux 6.6, indicating strong long-term performance stability. Linux 6.6 demonstrates slightly improved context switch behavior, suggesting scheduler refinements and improved system efficiency. Overall, the findings confirm that modern Linux kernels achieve incremental scheduler improvements while maintaining consistent baseline performance across generations.

I. Introduction

The Linux kernel has undergone substantial evolution since its inception, introducing new scheduling algorithms and performance improvements that directly impact system efficiency. This project aims to compare and evaluate the performance characteristics of Linux kernel versions 4.19, and 6.6 through controlled benchmarking experiments. The evaluation focuses on CPU throughput, context switch behavior, scheduling latency, and I/O workload performance. These metrics are critical for understanding how Linux scheduling and process management have evolved across kernel generations.

The original scope of the project included benchmarking Linux 2.6 in addition to 4.19 and 6.6. However, after extensive attempts to compile and boot Linux 2.6.39 on modern systems, it was determined that Linux 2.6 is incompatible with current toolchains and virtualization environments. As a result, Linux 2.6 is included only as a historical reference point rather than an experimental datapoint.

The Completely Fair Scheduler (CFS), introduced after the *O*(1) scheduler used prior to the 2.6.23 Linux kernel, represents one of the most significant scheduling advances in Linux history. The project evaluates how CFS and its subsequent refinements impact real world performance by comparing Linux 4.19 and Linux 6.6 under identical virtualized hardware conditions.

II. Historical context of Linux 2.6

The Linux 2.6 kernel series, released between 2003 and 2011, marked a major transition in Linux scheduling and multiprocessing support. Prior to 2.6, Linux schedulers scaled poorly as CPU counts increased, using scheduling algorithms that ran in $O(n)$ time. The 2.6 series introduced the $O(1)$ scheduler with kernel 2.6.0, which guaranteed constant time scheduling decisions regardless of the number of active processes. This scheduler relied on per-CPU runqueues and priority arrays, significantly improving responsiveness and scalability at the time [1].

Despite these advances, the $O(1)$ scheduler required complex heuristics to classify tasks as interactive or CPU-bound. As workloads diversified and multicore processors became standard, these heuristics became difficult to tune. These limitations motivated the eventual replacement of O(1) with the CFS in kernel 2.6.23.

III. Why Linux 2.6 was excluded

Although Linux 2.6 remains historically important, it is not practical for experimental benchmarking in a modern environment. Its build system is incompatible with modification to compiler flags and kernel headers. Linux 2.6.0 and 2.6.39 failed to reliably produce a valid bootable bzImage on Debian 12.

Modern GCC enforces PIE, stack protection, and strict section handling that Linux 2.6 cannot support without legacy patches. Running such a kernel would require a fully outdated user-space or a dedicated historical build environment. Since the remainder of this study relies on standardized QEMU virtual machines for fair

comparison, Linux 2.6 could not be included as an empirical datapoint.

IV. Experimental Setup

A. Host System

All virtual machines were executed using QEMU on a host system running Debian GNU/Linux 13 "Trixie" with Linux kernel 6.12.43. The physical hardware used for testing consisted of a 13th Generation Intel Core i5-1340P processor with 16 hardware threads, Intel Iris Xe Graphics, and ~16 GiB of DDR4 memory.

B. Base Virtual Machine

A base virtual machine (VM) was created using Debian 12 "Bookworm" due to its long-term stability and compatibility with kernel compilation. The Debian 12.9 netinst ISO was used as the installation medium. The VM was configured with:

- 2 virtual CPUs
- 4096 MiB of virtual memory
- 53.7 GiB virtual disk

Only essential system utilities were installed to minimize background CPU activity and benchmark noise. No graphical desktop environment was used.

C. Benchmarking Tools

The following standardized benchmarking tolls were installed on all virtual machines:

- sysbench for CPU and file I/O testing
- vmstat for monitoring context switch activity
- perf for kernel-level context switch and migration counts
- cyclictest for real time scheduling latency measurements

V. Kernel Compilation

A. Linux 4.19.309

Linux 4.19.309 was compiled using the default configuration shown in Figure 1 below:

```
$ make defconfig
$ make -j"$(nproc)"
$ sudo make modules_install
$ sudo make install
```

Fig. 1. Compilation Configuration for Linux 4.19.309

The kernel was then selected at boot through GRUB.

B. Linux 6.6

Linux 6.6 was compiled using the currently running Debian kernel configuration as a baseline as shown in Figure 2:

```
$ cp /boot/config-$(uname -r) .config
$ make olddefconfig
$ make -j"$(nproc)"
$ sudo make modules_install
$ sudo make install
```

Fig. 2. Compilation Configuration for Linux 6.6.0

Initial boot attempts resulted in a blank display due to missing framebuffer and DRM support. This was resolved by enabling SimpleDRM, VirtIO-GPU, and framebuffer console support. After these adjustments, Linux 6.6 successfully booted under QEMU with full console access.

VI. Benchmarking Methodology

To ensure statistically meaningful and repeatable results, each benchmark was executed ten independent times on each kernel version (Linux 4.19.309 and 6.6.0). All trials were conducted using identical virtual machine configurations and identical software environments. Only one kernel version was installed and tested on a given VM at a time to eliminate cross-kernel contamination.

Before each trial, the virtual machine was fully rebooted and allowed to idle for sixty seconds after login to ensure background services has stabilized and disk caches were in a consistent state. No additional user processes were run during testing. All benchmarks were executed from a clean shell using an automated script to guarantee identical execution order and timing across trials.

The benchmarking workflow for each kernel consisted of the following steps:

1. Boot the VM into the target kernel version.
2. Allow the system to idle for sixty seconds.
3. Execute the automated benchmark script.
4. Save the output to a uniquely numbered results file.
5. Reboot and repeat until ten trials were completed.

Each trial produced a separate output file in the format: `` `results_<kernel-version>_run<N>.txt` `` where *N* ranged from 1 to 10 (as determined by the script in Figure 3). After all trial were collected, the numerical results were aggregated to compute the mean and standard deviation for each measured metric. All comparative analysis in this report is based on these average values rather than on single-run measurements.

```
#!/bin/bash
# run_tests.sh

KERNEL=$(uname -r)
RUN=$(ls results_${KERNEL}_run*.txt 2>/dev/null | wc -l)
RUN=$((RUN + 1))
OUT="results_${KERNEL}_run${RUN}.txt"

echo "Kernel: $(uname -r)" > "$OUT"

echo "--- SYSBENCH CPU ---" >> "$OUT"
sysbench cpu --threads=1 run >> "$OUT"
sysbench cpu --threads=2 run >> "$OUT"

echo "--- VMSTAT CONTEXT SWITCHES ---" >> "$OUT"
vmstat 1 10 >> "$OUT"

echo "--- PERF CONTEXT SWITCHES ---" >> "$OUT"
sudo perf stat -a -e context-switches,cpu-migrations,task-clock sleep 5 >> "$OUT"
2>&1

echo "--- SYSBENCH FILEIO ---" >> "$OUT"
sysbench fileio --file-test-mode=seqwr prepare
sysbench fileio --file-test-mode=seqwr run >> "$OUT"
sysbench fileio cleanup

echo "--- CYCLCTEST LATENCY ---" >> "$OUT"
sudo cyclictest -t2 -p 80 -i 1000 -l 5000 >> "$OUT"

# EoF
```

Fig. 3. Standardized Benchmark Script (run_tests.sh)

A. Benchmark Script

The same benchmarking script, shown in Figure 3, was used unchanged for every trial on both kernel versions.

B. CPU Throughput Measurement

CPU performance was measured using sysbench with one and two worker threads. This workload generates a deterministic, CPU-bound prime number computation task. The reported metric is events per second, which reflects raw integer computation throughput. The ten trial results for each thread count were averaged to obtain the final reported values.

C. Context Switch Measurement

Context switching behavior was measured using both vmstat and perf. The vmstat utility sampled the system state once per second for ten seconds, recording the number of context switches per second. The perf tool was used to collect kernel-level statistics for context switches, CPU migrations, and task clock activity during a five second system-wide measurement window. These measurements were repeated across all ten trials and later averaged.

D. File I/O Performance Measurement

Sequential write performance was measured using sysbench file I/O in synchronous mode. Easch test generated a 2 GiB workload consisting of 128 files at 16 MiB per file. Periodic and final fsync operations were enabled to ensure that actual disk writes occurred rather than remaining in cache. Throughput in MiB/s and operation rates were recorded for each of the ten trials and averaged.

E. Scheduling Latency Measurement

Real-time scheduling latency was measured using cyclictest with two threads pinned at priority 80. Each run executed 50,000 iterations with a 10 millisecond interval. The test measures the delay between expected and actual wake-up times of high-priority threads. All scheduling latency metrics were collected across the ten trials to evaluate both consistency and worst-case jitter behavior.

F. Statistical Treatment

For each benchmark category, the following values were computed: the arithmetic mean, and the standard deviation across all ten trials. All kernel performance comparisons in the Results and Analysis sections are based on these average values. Differences that fall within one standard deviation are treated as statistically insignificant unless otherwise stated.

VII. Results

Table I, shown below, provides a summary of the mean and standard deviation across all benchmarks.

A. CPU Throughput

Figures 4 and 5 show the distribution of single-threaded and dual-threaded CPU throughput measured using Sysbench across ten trials for both kernel versions.

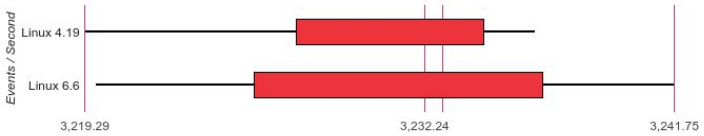

Fig. 4. Distribution of single-threaded Sysbench CPU throughput for Linux 4.19 and Linux 6.6

Fig. 5. Distribution of dual-threaded Sysbench CPU throughput for Linux 4.19 and Linux 6.6

B. Context Switching Behavior

Figures 6, 7, and 8 summarize the distributions of context switches, CPU migrations, and task clock activity measured using perf during system-wide sampling.

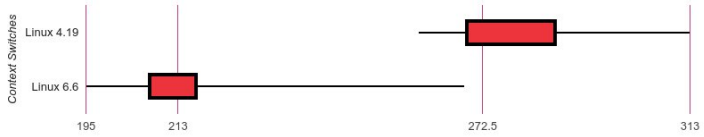

Fig. 6. Distribution of context switch counts measured by perf for Linux 4.19 and Linux 6.6

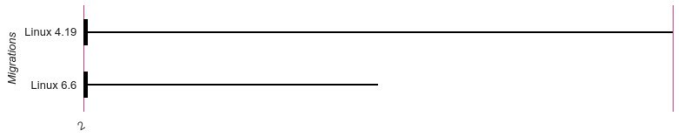

Fig. 7. Distribution of CPU migration counts measured by perf for Linux 4.19 and Linux 6.6

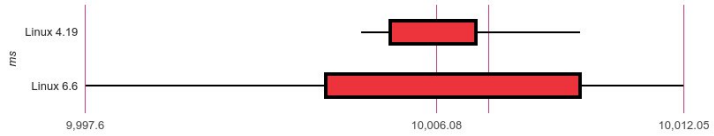

Fig. 8. Distribution of task clock duration measured by perf for Linux 4.19 and Linux 6.6

C. File I/O Performance

Figure 9 presents the distribution of sequential file I/O execution time measured using Sysbench across ten trials.

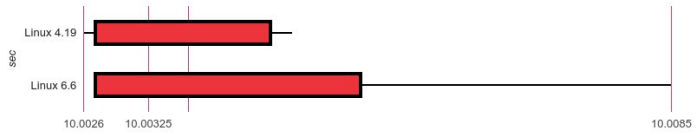

Fig. 9. Distribution of Sysbench sequential file I/O execution time for Linux 4.19 and Linux 6.6

D. Scheduling Latency

Fig. 10 illustrates the distribution of worst-case real-time scheduling latency observed using cyclictest across all trials.

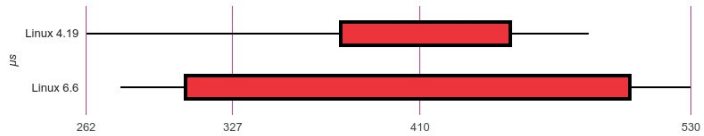

Fig. 10. Distribution of worst-case scheduling latency measured by cyclictest for Linux 4.19 and Linux 6.6

Metric	Linux 4.19 Mean	Linux 4.19 Std Dev	Linux 6.6 Mean	Linux 6.6 Std Dev
Sysbench 1T (events/sec)	3230.663	5.496687993	3231.876	7.456161062
Sysbench 2T (events/sec)	6479.23	4.432170274	6477.699	10.52119496
Context Switches	279.6	17.98888546	216.7	19.90002792
CPU Migrations	2.3	0.674948558	2.1	0.316227766
Task Clock (ms)	10007.267	3.164943566	10006.848	21.76406049
File I/O Time (s)	10.00356	0.000857904	10.00424	0.001886914
Cyclictest Max (µs)	415.8	101.9191401	444.7	207.6246881

Table I: Summary of Mean and Standard Deviation Across All Benchmarks

VIII. Discussion

The results of this study demonstrate that the Linux kernel has achieved strong long-term performance stability while continuing to refine internal scheduling efficiency across major kernel generations. Across all benchmarks, Linux 4.19 and Linux 6.6 exhibited nearly identical raw computational throughput and I/O performance, while Linux 6.6 showed a substantial reduction in context switching overhead. At the same time, Linux 6.6 exhibited slightly degraded worst-case real-time scheduling determinism as indicated by increased cyclictest latency variance. These findings highlight the trade-offs between throughput optimization, scheduler efficiency, and real-time predictability in modern Linux kernel development.

CPU throughput measurements using Sysbench revealed no statistically significant performance difference between Linux 4.19 and Linux 6.6 in either single-threaded or dual-threaded workloads. As reported in Table I, both kernels averaged about 3231 events per second in the single threaded test and approximately 6478 events per second in the dual-thread test. All observed differences fell well within one standard deviation.

In contrast to throughput, context switching behavior revealed a clear improvement in Linux 6.6. The average number of context switches per perf sampling window decreased from 279.6 on Linux 4.19 to 216.7 on Linux 6.6, representing a reduction of approximately 22.5%. This is a substantial improvement and strongly suggests that scheduler refinements in Linux 6.6 are successful reducing unnecessary task preemption and scheduler bookkeeping overhead. The corresponding CPU migration counts remained low and statistically similar between both kernels, indicating that while task switching became more efficient, overall task affinity and load distribution across cores remained consistent. This aligns with expected behavior with modern CFS refinements and the introduction of newer scheduling models such as the Earliest Eligible Virtual Deadline First (EEVDF) Scheduler, which prioritize fairness while minimizing scheduling overhead [2][3].

Task clock measurements further reinforce the fairness of the experimental methodology. Both kernels exhibited nearly identical task clock durations of approximately 10 seconds per perf sampling window, confirming that timing bias did not influence the context-switch measurements. The higher standard deviation on Linux 6.6 in task clock measurements likely reflects increased scheduler adaptability and power-management behavior rather than systematic measurement error.

Sequential file I/O performance was also statistically indistinguishable between the two kernels. Linux 4.19 and Linux 6.6 produced average execution times of 10.00356 sec and 10.00424 sec respectively, with extremely small standard deviations (< 0.01). These results indicate that block-layer write performance under synchronous workloads remains highly stable across kernel versions. Because the workload was executed within a virtualized environment with identical backing storage, these results suggest that modern Linux kernel changes have not introduced any significant performance regressions or enhancements in basic sequential disk write behavior at this scale.

The most notable behavioral difference between the two kernels appeared in the real-time scheduling latency measurements. Linux 6.6 exhibited a higher average maximum latency (444.7 μs) than Linux 4.19 (415.8 μs), along with a much larger standard deviation and several high-latency outliers approaching one millisecond. Although the mean increase of approximately 6.9% is modest, the increased variance observed in Figure 10 indicates reduced worst-case determinism on Linux 6.6 under the tested conditions. This behavior is likely attributable to more aggressive power management, deeper idle states, and changes to internal scheduler heuristics in newer kernels. While such behavior is acceptable for general-purpose computing workloads, it may be undesirable for strict real-time applications that rely on highly deterministic wake-up behavior.

Importantly, the real-time latency results must be interpreted alongside the improved context switching performance observed in Linux 6.6. The data suggests that Linux 6.6 favors reduced scheduling overhead and improved average efficiency at the expense of slightly

increased worst-case latency variability. This reflects a fundamental design trade-off in modern kernel development: optimizing for throughput and general system efficiency can introduce addition scheduling jitter under specific timing-sensitive workloads. Linux 4.19, by contrast, demonstrates slightly stronger worst-case determinism, but at a cost of higher overall context switching overhead.

The virtualized nature of the test environment introduces additional consideration when interpreting these results. Although QEMU provides a consistent and repeatable testing platform, virtualization may amplify scheduling jitter and mask certain hardware-level timing behaviors. However, because both kernels were evaluated under identical virtualized conditions, the retaliative comparisons between kernels remain valid. Furthermore, by controlling background services, rebooting between trials, and standardizing execution order through automation, the study minimized sources of experimental noise and ensures fair kernel-to-kernel comparisons.

IX. Conclusion

This study presented a controlled performance comparison between Linux kernel versions 4.19.309 and 6.6.0 using identical QEMU-based virtualized environments. By evaluating CPU throughput, context switching behavior, scheduling latency, and sequential file I/O performance across repeated trials, this project analyzed how the Linux scheduling subsystem has evolved over several major kernel generations. Although Linux 2.6 was originally included in the scope as a historical reference point, modern tool-chain and virtualization incompatibilities prevented its inclusion as an experimental datapoint.

The experimental results demonstrate that baseline CPU throughput and I/O-bound performance have remained highly stable across the tested kernel versions. Linux 4.19 and Linux 6.6 produced nearly identical Sysbench throughput in both single-threaded and dual-threaded workloads, and their sequential file I/O execution times were statistically indistinguishable. These results confirm that modern Linux kernel development has successfully preserved core computational and storage performance while evolving internal scheduling mechanisms.

In contrast to the stable throughput results, scheduling behavior revealed and important trade-off. Linux 6.6 achieved a significant reduction in context switching, with approximately a 22.5% decrease compared to Linux 4.19, indicating improved scheduler efficiency and reduced overhead. However, real-time latency measurements showed greater worst-case variability on Linux 6.6, including several high-latency outliers. This suggests that modern kernels favor average-case efficiency and power-management responsiveness at the expense of stricter worst-case determinism. Linux 4.19, by comparison, exhibited slightly stronger worst-case predictability but higher overall scheduling overhead, highlighting the fundamental trade-off between throughput optimization and real-time determinism in modern operating system design.

X. Appendix

This appendix provides the full raw benchmark measurements used to compute the summary statistics reported in Table I. Tables II and III show the benchmark measurements for Linux 4.19.309 and Linux 6.6.0 respectively.

Run	1T CPU	2T CPU	Ctx Switches	CPU Migrations	Task Clock (ms)	File I/O (s)	Cyclictest Max (μs)
1	3224.78	6478.83	269	2	10004.92	10.0047	470
2	3234.53	6483.98	270	2	10014.83	10.0031	451
3	3235.89	6481.66	265	2	10004.26	10.0027	374
4	3230.75	6468.7	275	2	10006.49	10.0041	284
5	3236.47	6481.97	260	4	10009.56	10.0031	262
6	3233.73	6475.62	313	2	10006.08	10.0027	417
7	3229.95	6481.51	287	2	10005.85	10.0034	387
8	3219.29	6480.75	308	2	10007.08	10.0026	618
9	3227.3	6481.75	280	3	10004.72	10.0045	485
10	3233.94	6477.53	269	2	10008.88	10.0047	410

Table II. Raw Benchmark Measurements for Linux 4.19.309

Run	1T CPU	2T CPU	Ctx Switches	CPU Migrations	Task Clock (ms)	File I/O (s)	Cyclictest Max (μs)
1	3231.53	6486.05	213	2	10006.94	10.0027	968
2	3236.77	6484.32	207	2	10009.36	10.0028	277
3	3240.16	6474.5	216	3	10054.3	10.0027	392
4	3232.11	6491.97	269	2	10009.82	10.0054	530
5	3225.72	6480.59	213	2	10012.05	10.0056	515
6	3241.75	6482.48	217	2	10007.75	10.0031	325
7	3233.78	6482.09	205	2	10006.43	10.0047	305
8	3219.7	6456.93	223	2	10000.27	10.0042	304
9	3235.66	6464.75	195	2	9963.96	10.0027	327
10	3221.58	6473.31	209	2	9997.6	10.0085	504

Table III. Raw Benchmark Measurements for Linux 6.6.0

References

[1] J. Aas, “Understanding the Linux 2.6.8.1 CPU Scheduler,” Silicon Graphics, Inc. (SGI), Feb. 17, 2005.

[2] I. Stoica, H.A. Wahab, “Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation,” Dept. of Computer Science, Old Dominion Univ., Norfolk, VA, USA, Tech. Rep. TR-95-22, 1995.

[3] S. Hegde, “Re: [PATCH 00/17] sched: EEVDF using latency-nice,” Linux Kernel Mailing List, Apr. 3, 2023. [Online]. Available: https://lore.kernel.org/lkml/a79014e6-ea83-b316-1e12-2ae056bda6fa@linux.vnet.ibm.com/. [Accessed: Nov. 30, 2025]