ECE 1759 - Advances in Operating Systemsl

"Project Report"

Comprehensive Performance Evaluation and Benchmarking of Linux Kernels

Mohammadreza Sabramooz(1011567680) & Mohammad hooman Keshvari(1011293869)

Supervisor : Dr. Ding Yuan

UNIVERSITY OF TORONTO

Contents

1 Introduction

The Linux kernel is at the core of many modern computing systems, powering everything from personal devices to large-scale enterprise servers. Understanding the performance characteristics of different Linux kernel versions is crucial for optimizing system performance and ensuring reliability across diverse workloads. This project aims to conduct a detailed performance evaluation of various Linux kernel versions, shedding light on their impact on system-level operations and real-world applications.

In this study, we benchmark several system calls, which are fundamental interfaces between user-space applications and the kernel. These benchmarks provide insight into the efficiency of critical kernel operations. In addition, we evaluate reading and writing to the MySQL database and train a machine learning model using PyTorch on various Linux kernels to compare their performance. We also analyzed the performance of the munmap syscall and identified the reasons behind its decreased efficiency in newer kernel versions.

2 Benchmarking tools

2.1 Existing tools

- **LEBench**[1] is a microbenchmark suite that measures the performance of core Linux kernel operations, such as context switching, process creation, memory allocation, and I/O. It benchmarks these functions across Linux kernel versions to identify performance changes, offering insights for optimizing system performance.

- **Sysbench** is a versatile and widely used benchmarking tool designed to evaluate the performance of various system components, including CPUs, memory, disk I/O, and databases. It is particularly effective for testing database systems such as MySQL and PostgreSQL by simulating transactional workloads like OLTP (Online Transaction Processing) scenarios. Sysbench allows users to customize parameters such as table size, number of threads, and workload duration, making it adaptable for diverse benchmarking needs. The tool provides detailed performance metrics, including query throughput and latency.

2.2 Self-developed tools

- **PyTorch Benchmarker**: For benchmarking the PyTorch framework, we implemented a process to train the VGG11 model, one of the most renowned architectures for image classification and computer vision tasks. Using the CIFAR-10 dataset, we evaluated the model's training performance by measuring the overall training time and calculating

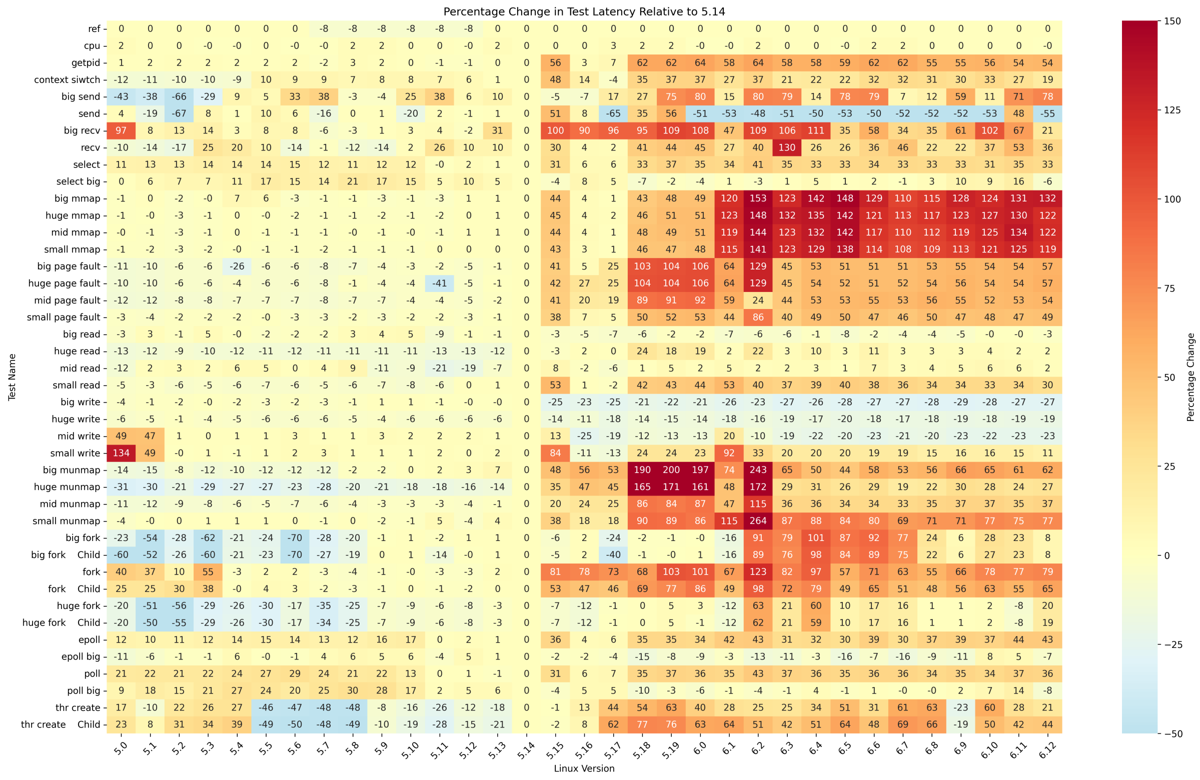
[1] X. J. Ren, K. Rodrigues, L. Chen, C. Vega, M. Stumm, and D. Yuan, "An Analysis of Performance Evolution of Linux's Core Operations," in Proceedings of the 27th ACM Symposium on Operating Systems Principle

the average training time per epoch. This analysis provides valuable insights into the efficiency of PyTorch when paired with different Linux kernel versions.

- **Futex system call Benchmarker:** The benchmarking of the futex syscall measures its latency across different Linux kernel versions to evaluate performance. The process involves using a shared futex variable to perform a high number of iterations (e.g., 1,000,000) of $futex - wait$ (waiting for a value change) and $futex - wake$ (waking a waiting thread), with precise timing captured using high-resolution timers like $clock - gettime$. The total execution time is recorded, and the average latency per syscall is calculated by dividing the total time by the number of iterations.

3 Performance Analysis

In this project, we conduct a comprehensive evaluation of Linux kernel versions ranging from **5.0** to **6.12**, focusing on their performance and behavior through detailed analysis of system calls. We compiled and installed Linux kernels on **Ubuntu 18.04** and conducted a series of experiments. Our testbed is built on a high-performance computer equipped with an **Intel i9-9820 processor**, featuring a 3.3 GHz base clock and 20 cores.



3.1 Evaluation of System Calls Using LEBench

In our initial experiment, we utilized the LEBench tool to benchmark a range of system calls and generated the above heatmap to visualize the results.

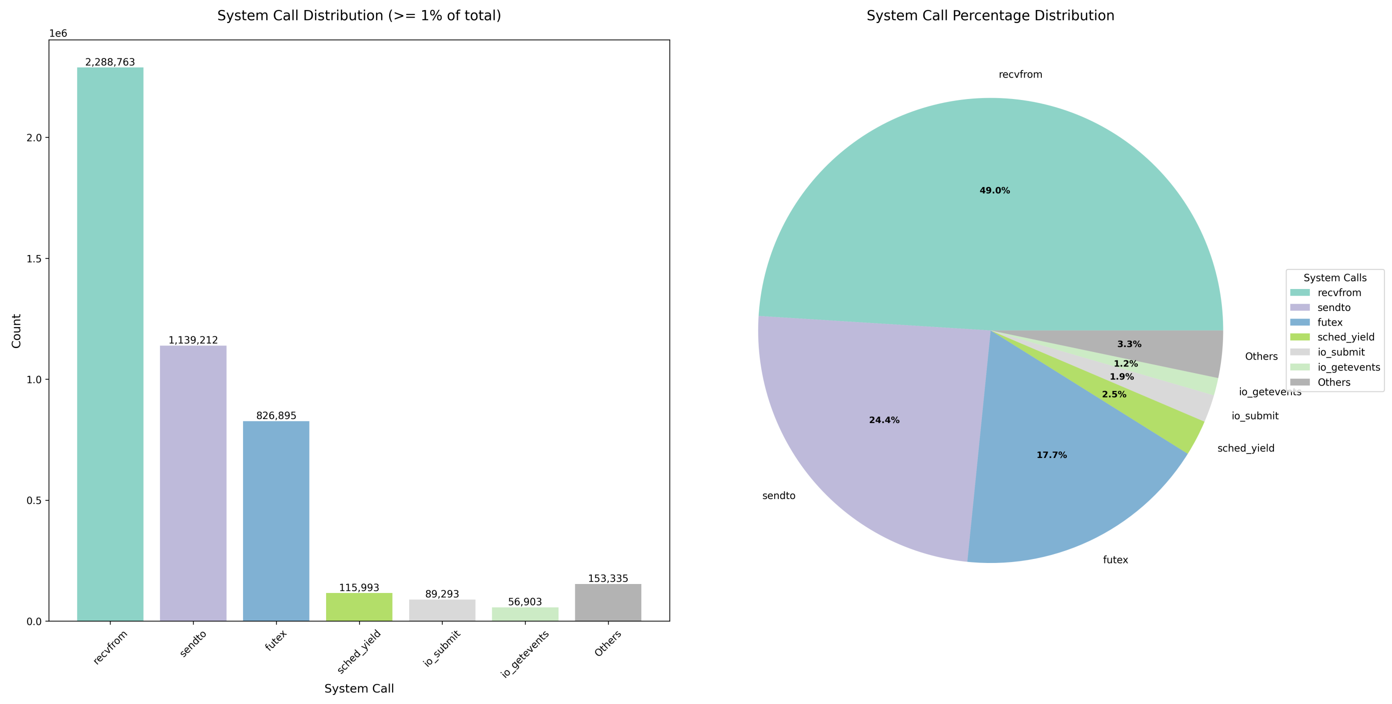
3.2 Performance Evaluation of the MySQL Across Different Linux Kernels

The experiment aimed to benchmark MySQL performance using the sysbench tool under an OLTP (Online Transaction Processing) workload with a read-write mix. The test environment included a MySQL database named '$sysbench - test$' and utilized sysbench's OLTP read-write test to simulate realistic database operations. The database was preloaded with a table containing 1,000,000 rows to create a moderately large dataset. The benchmark was executed with 4 concurrent threads over a duration of 120 seconds, capturing key performance metrics such as transactions per second (TPS), queries per second (QPS), and query latency (minimum, average, maximum, and 95th percentile). The experiment involved three main stages: (1) data preparation, where the database was populated with test data; (2) benchmark execution, where read and write queries were run concurrently to measure system performance; and (3) cleanup, where test data was removed, restoring the database to its original state. This setup ensured a balanced and reproducible evaluation of MySQL's performance under realistic workload conditions. Our experiment was conducted on the Long-Term Support (LTS) versions of Linux, specifically versions 5.10, 5.15, 6.1, 6.6, and 6.12. Table 1 shows the benchmark results across linux kernel versions. To analyze the

Table 1: Benchmark Results Across Linux Kernel Versions

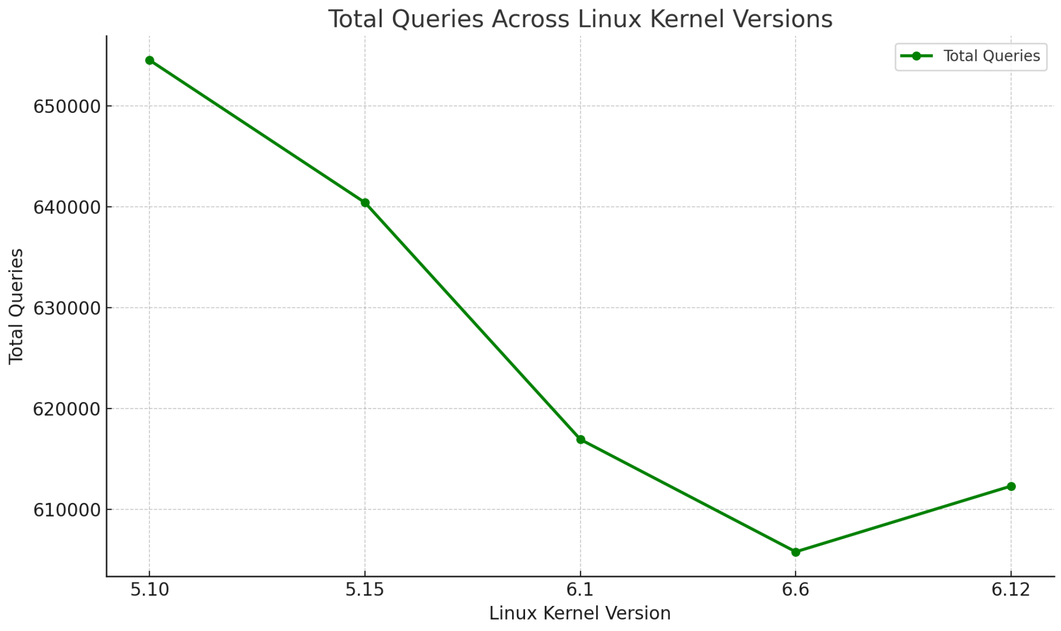
Kernel Version	**5.10**	**5.15**	**6.1**	**6.6**	**6.12**
Read Queries	458,192	448,308	431,872	424,074	428,638
Write Queries	130,912	128,088	123,392	121,164	122,468
Other Queries	65,456	64,044	61,696	60,582	61,234
Total Queries	654,560	640,440	616,960	605,820	612,340
Transactions/sec	272.68	266.78	257.03	252.37	255.09
Queries/sec	5,453.55	5,335.69	5,140.53	5,047.31	5,101.75
Total Experiment Time (s)	120.02	120.03	120.02	120.02	120.02
Avg. Latency (ms)	14.67	14.99	15.56	15.84	15.68
Max Latency (ms)	249.87	228.29	222.63	232.51	229.48
95th Percentile (ms)	26.20	27.17	27.66	28.16	27.66
Events (avg)	8,182.0	8,005.5	7,712.0	7,572.8	7,654.3
Execution Time (s)	119.99	119.99	119.98	119.99	119.99

results, it is essential to identify the system calls invoked during the MySQL benchmarking execution. To achieve this, we utilized the **strace** tool to monitor and record the system calls made by MySQL while running the benchmarking workload. This approach allowed us to capture detailed information about the system calls executed during the benchmarking process, providing valuable insights into the interaction between MySQL and the underlying operating system. The plot below illustrates the distribution of system calls observed during the MySQL benchmarking process, providing insights into the frequency and type of system calls executed.



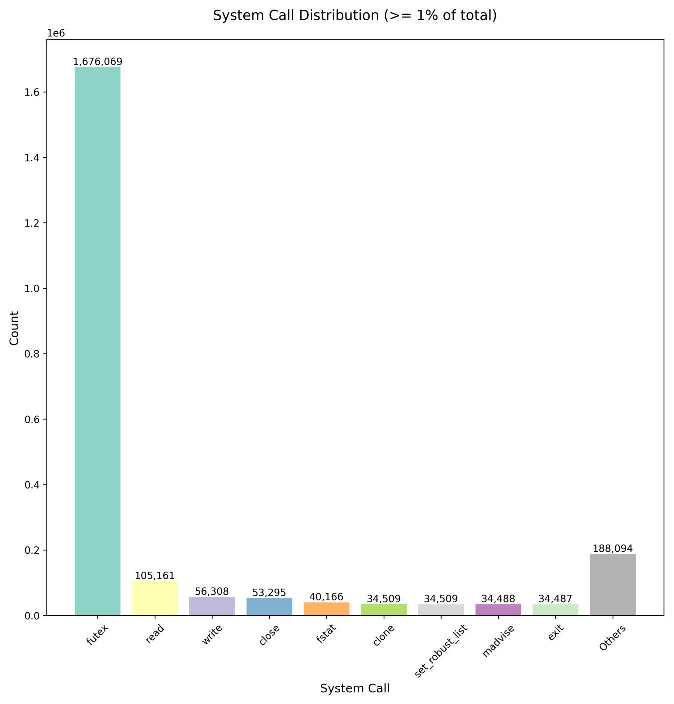
Based on the above plot, the **recv** system call is the most frequently invoked syscall during the MySQL benchmarking process. This indicates that the performance of the **recv** syscall has a significant impact on the overall database performance.

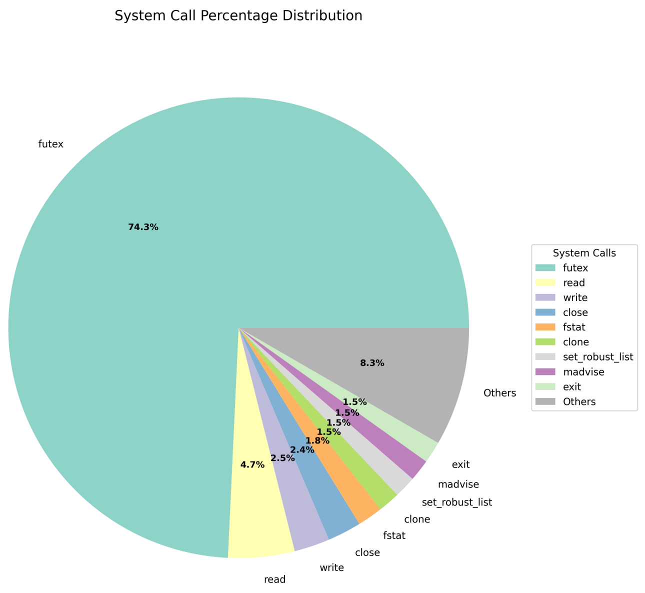
By analyzing the results in our benchmark results, it is evident that the performance of MySQL tends to decrease as the kernel version increases. This trend suggests that changes or updates in newer kernel versions may introduce factors impacting MySQL's query processing efficiency. The plot below illustrates the total query trends across different Linux kernel versions, highlighting the decline in performance as the kernel version progresses. As the performance of the ‘recv‘ syscall decreases with newer Linux kernel versions, and given that ‘recv‘ is the most frequently invoked system call during MySQL benchmarking, it is reasonable to attribute the decline in MySQL performance to kernel upgrades.



3.3 Performance Evaluation of the PyTorch Framework Across Different Linux Kernels

To evaluate the performance of the PyTorch framework across different Linux kernel versions, we implemented a training process using the VGG11 model with the CIFAR10 dataset. The model was trained for 2 epochs with a batch size of 64, and both the overall training time and the average training time per epoch were measured. Additionally, the training process was analyzed using strace to identify the most frequently invoked system calls during the training process. This comprehensive evaluation helps us understand the impact of different kernel versions on PyTorch's performance and the underlying system behavior. The plot below illustrates the distribution of system calls invoked during the training process, providing insights into the frequency and types of system calls utilized. This analysis highlights the underlying system behavior and the role of specific syscalls in the overall training performance.





From the plot above, it is evident that the most frequently invoked system call during the training process is futex, indicating its critical role in synchronization operations.

The table below presents the benchmarking results of the PyTorch framework on Long-Term Support (LTS) versions of Linux kernels, highlighting performance variations across different kernel releases. Based on our observations, the performance of the PyTorch framework remains relatively consistent across different Linux kernel versions, with only minor variations

Table 2: Benchmarking Results of PyTorch on Different Linux Kernel Versions

Kernel Version	Epoch 1 Time (s)	Epoch 2 Time (s)	Average Time per Epoch (s)	Total Time (s)
5.10	129.25	127.82	128.54	257.07
5.15	136.92	136.75	136.84	273.68
6.1	138.85	136.21	137.53	275.05
6.6	120.28	118.90	119.59	239.17
6.12	124.20	121.62	122.91	245.82

in execution time. To thoroughly analyze the results above, it is essential to evaluate the performance of the futex system call (Most frequent system call in the training process). Since LEBench does not natively support benchmarking the futex system call, we developed a dedicated benchmarking tool for this purpose. The table below presents the benchmarking results for the futex system call on Long-Term Support (LTS) versions of the Linux kernel. Based on our experiment, the performance of the 'futex' syscall remains consistent across

Linux Kernel Version	Iterations	Time (seconds)	Average Syscall Time (seconds)
5.10	1,000,000	162.384040	0.000162
5.15	1,000,000	162.179226	0.000162
6.1	1,000,000	162.475411	0.000162
6.6	1,000,000	163.274819	0.000163
6.12	1,000,000	164.472484	0.000164

Table 3: Futex Benchmark Results for Different Linux Kernel Versions

different Linux kernel versions. This stability in 'futex' performance justifies why the training process performance also shows minimal variation between Linux kernels, as synchronization overhead, a critical factor in training, remains unchanged.

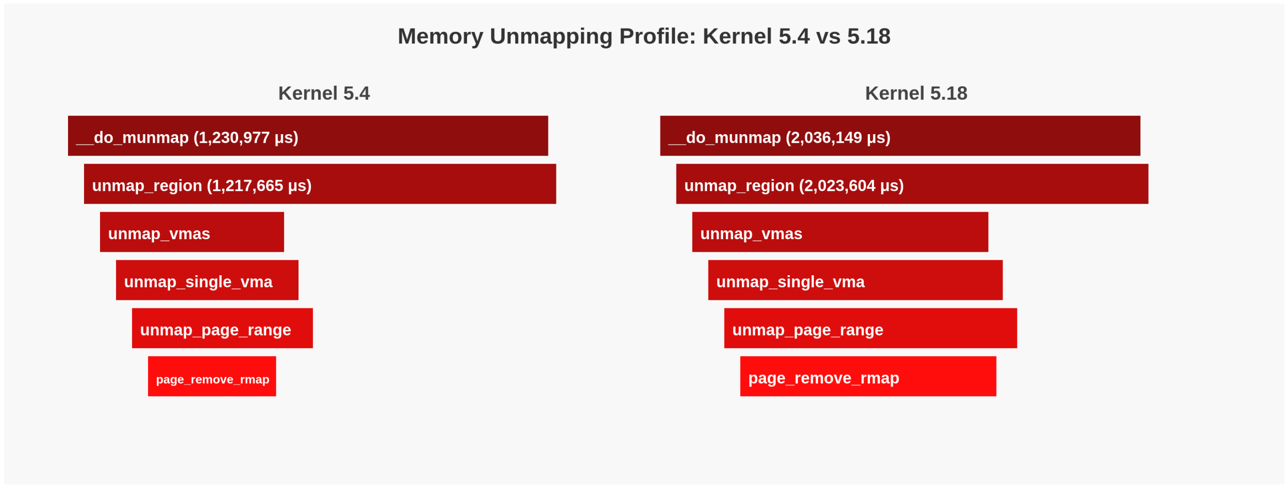
4 Analysis of munmap Syscall Performance Degradation

This analysis explores the concerning performance drop we discovered in munmap operations between Linux kernels 5.4 and 5.18. Our heatmap findings motivated us to dig deeper into what changed in the newer kernel to cause such significant slowdown. We approached this investigation methodically: first by function tracing a large-scale [2] munmap operation (starting at __do_munmap) to understand the execution flow, then analyzing the timing of each function call. This initial analysis guided us to specific bottleneck functions that warranted deeper investigation. In the following sections, we will walk through our detailed findings and examination of these performance-critical code paths.

Call Chain Analysis

When we looked at the function call traces and their timing, we created flamegraphs for both kernels to compare them side by side.

[2] 100 pages mmap then unmap



The flamegraphs showed something interesting - we identified that the majority of execution time in both kernels was spent in the unmap_region function and its callees. Further analysis revealed that the performance difference ultimately concentrated because a huge chunk of time was being spent in the **page_remove_rmap** function, which just handles cleaning up the links between virtual and physical memory pages. This seemed pretty odd to us since this basic memory management task shouldn't be taking up so much time, so we decided to look into it further. Seemingly this function is our bottleneck.

```
do_munmap
 unmap_region
     unmap_vmas
         unmap_page_range
             page_remove_rmap  [Primary Bottleneck]
```

4.1 page_remove_rmap Analysis

To understand why page_remove_rmap was running about twice as slow in kernel 5.18, we needed to dig deeper. We ran another function trace to see exactly what was happening inside this function in both kernel versions. Here is our traces:

4.1.1 Kernel 5.4 Execution Profile

```
page_remove_rmap() [3.946 µs]
 lock_page_memcg() [0.444 µs]
 PageHuge() [0.443 µs]
 unlock_page_memcg() [1.307 µs]
     unlock_page_memcg() [0.443 µs]
```

4.1.2 Kernel 5.18 Execution Profile

```
page_remove_rmap() [6.775 µs]
 lock_page_memcg() [2.281 µs]
    folio_memcg_lock() [1.386 µs]
 PageHuge() [0.462 µs]
 unlock_page_memcg() [2.245 µs]
     folio_memcg_unlock() [2.245 µs]
```

4.2 Key Findings

The analysis reveals that while the core operation (PageHuge) maintains consistent performance (0.443 µs vs 0.462 µs), the significant slowdown stems from changes in the memory locking mechanism:

- Lock acquisition time increased by 413% (0.444 µs to 2.281 µs)
- Lock release time increased by 71.8% (1.307 µs to 2.245 µs)

Looking at the traces, we spotted what might be causing the slowdown: the switch from the simple page lock to the new folio data structure and its locking mechanisms. The folio structure was introduced as a way to handle larger memory pages (like huge pages) more efficiently by grouping related pages together. However, the folio's locking system is more complex than the old page lock: it needs to handle multiple pages at once and maintain consistency across the entire folio structure. This extra complexity in the locking mechanism, while designed for better memory management in certain cases, seems to be adding overhead to our basic munmap operations.

5 Conclusion

In this project, we conducted a comprehensive evaluation of Linux kernel 5 and 6 performance, benchmarking system calls, database operations, and machine learning workloads across multiple kernel versions. The results reveal that while certain syscalls, such as futex, exhibit stable performance across kernels, others, like munmap, show notable regressions due to changes in kernel mechanisms, such as the introduction of the folio structure. Similarly, MySQL workloads exhibit slight but noticeable performance differences influenced by kernel updates. These results emphasize the critical need for detailed performance analysis with each kernel update to understand its impact on diverse workloads, ensuring that optimizations do not unintentionally degrade performance in specific use cases.

6 Future Works

- **Microarchitecture Influence:** Investigate how different hardware architectures (e.g., AMD vs. Intel) impact the performance of system calls, database operations, and machine learning training across various kernels.

- **Dynamic Workloads:** Evaluate performance under dynamic and real-time workloads to assess kernel adaptability in diverse and evolving scenarios, such as cloud-native environments.

- **Energy Efficiency:** Incorporate energy consumption measurements during benchmarking to identify the most power-efficient kernel versions for specific workloads.

- **Kernel Security Overhead:** Examine the performance trade-offs introduced by security enhancements in newer kernels, such as Spectre/Meltdown mitigations, and their impact on workloads.