CS 695: Project
Sharing is Caring (KSM)

Parismita Das
Roll No: 22M0815

Manish Kumar
Roll No: 22M2110

3 May 2024

Contents

1 Introduction

Kernel Samepage Merging (KSM) is a memory management feature implemented in the Linux kernel that enhances the efficiency of memory usage by identifying and merging duplicate pages based on content. By allowing the system to consolidate identical data into fewer memory pages, KSM can significantly reduce the overall memory footprint of running applications, especially in virtualized environments where many instances might run similar or identical tasks. This service is particularly useful when combined with virtual machine platforms such as QEMU and KVM, where each VM instance can be treated as a separate process by the host system, thereby optimizing memory utilization across virtual machines.

2 Background

KSM operates by scanning memory pages and comparing their contents. Pages that are found to be identical and marked as mergeable via the `madvise` system call are then combined into a single page stored in physical memory. This single copy is then mapped into the address spaces of all processes that previously held copies of the merged page, effectively sharing it among multiple consumers while maintaining the illusion of private memory through copy-on-write semantics.

The implementation of KSM in the Linux kernel provides configurable parameters such as the frequency of memory scans and the aggressiveness of merging, which can be tuned to balance performance impacts against memory savings. We aim to checkout various combinations of these parameters to simulate various operational scenarios, exploring the behavior of KSM under different system loads and memory conditions.

3 Problem Description

The goal of this project is to study a comprehensive performance characterization of KSM to understand its behavior, workings and its impact thoroughly by using various workloads, setups and configurations such as scan rates to obtain performance and cost metrics. We aim to study the above to find out behavior and workings of this service.

We are going to look along the following points:

- **Workload Diversity**: For different kinds of load, with different duplicate pages, we want to check KSM's performance. Also on real life load like running some process on VM or multiple VMs
- **Performance Metrics**: To understand various performance metrics to checkout KSM's performance. for examle the CPU utiliation it takes due to scanning, or the profit observered from merging pages.
- **Configuration Parameters**: KSM's behavior is highly dependent on its configuration settings, such as scan rate and sleep intervals, hence we want to test which configuration settings are ideal in which scenarios.

We are using Qemu+KVM as workload and testing KSM's behavior and performance under various circumstances as mentioned above.

4 Design

The performance characterization study is designed to evaluate KSM under a variety of configurations and workload scenarios. The primary goal is to identify how different settings affect KSM's performance in terms of memory savings and CPU utilization. The study is structured around a series of controlled experiments where memory usage patterns are simulated using a custom C program, 'process.c' and orchestrated using shell scripts such as 'spawner.sh' for process management and 'monitor.sh' for performance monitoring.

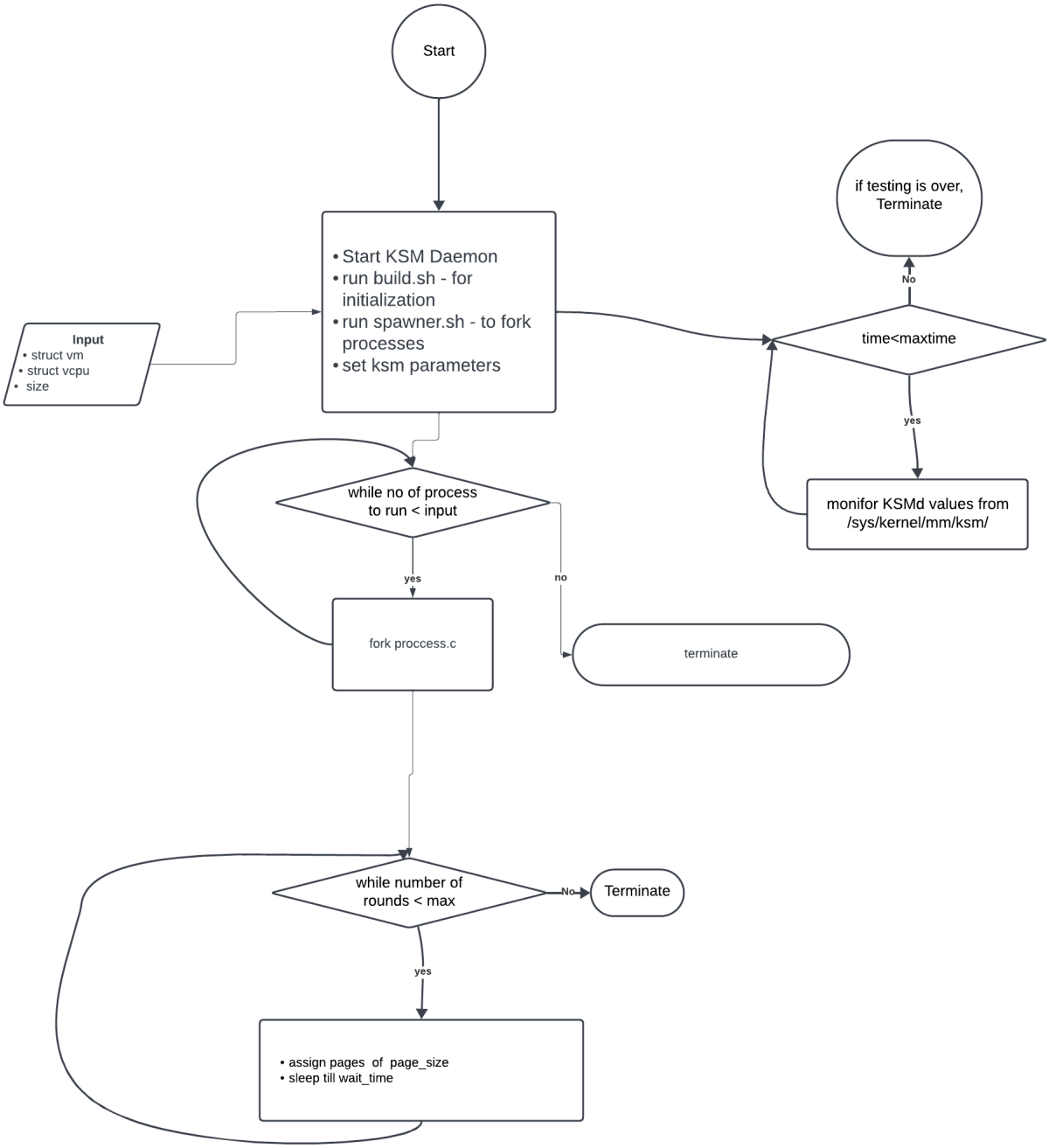

Figure 1: Design with Process as workload

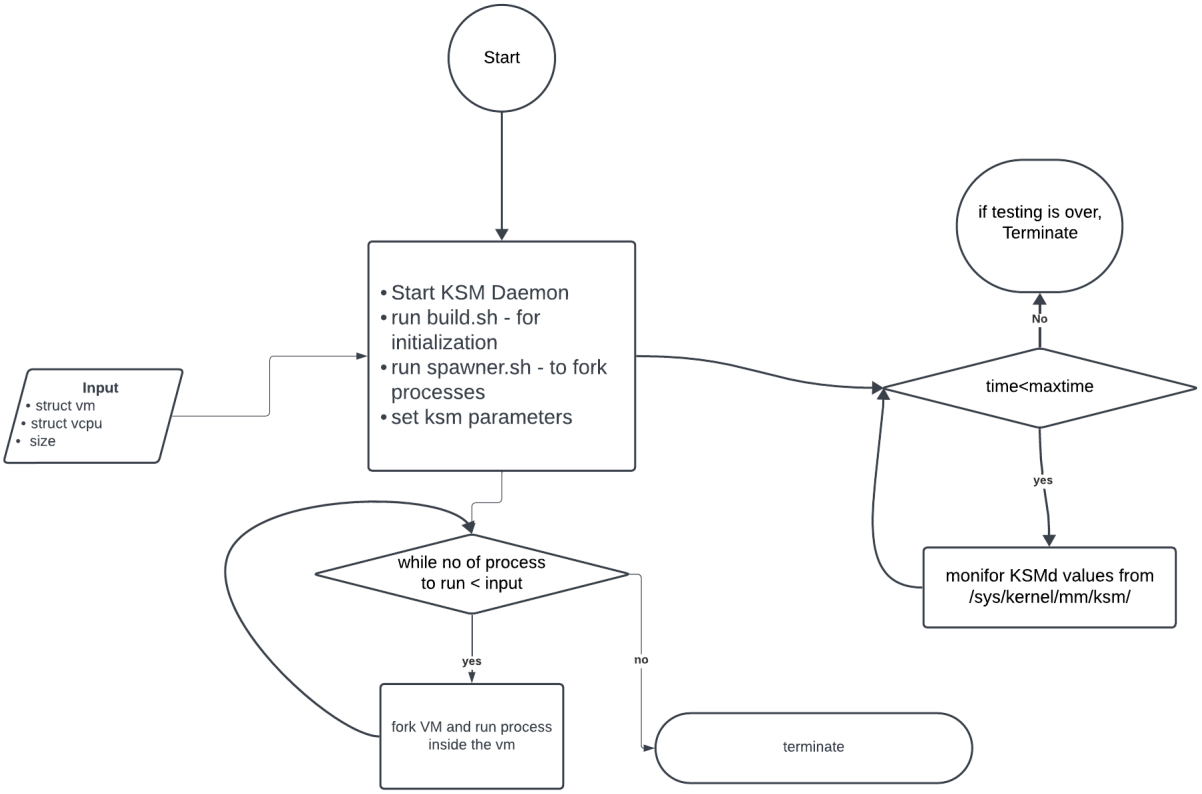

Figure 2: Design with QEMU+KVM VM as workload

5 Implementations

The implementation of the testing framework involves several components:

- **Data Generation:** `data_gen.py`
  Generates data by varying load and parameters for KSM performance testing using the data provided by the KSM daemon. Test configurations like the number of processes, pages per process, and test parameter ranges used. `data_gen.py` is used to call monitor.sh to generate CSV outputs of test results.
- **Workload Simulation:** `spawner.sh`
  Spawns multiple processes to simulate workloads for testing. spawner_vm.sh is used to spawn multiple VMs. The 'process.c' program simulates memory-intensive operations where memory pages can potentially be merged by KSM. It allocates a specified amount of memory and periodically modifies it to mimic real application behavior. spawner.sh forks proccesses using proccess.c
- **Monitoring:** `monitor.sh`
  The 'monitor.sh' script adjusts KSM parameters dynamically based on the configuration files and collects metrics such as the number of pages shared, the number of full scans completed, and CPU overhead. This data is crucial for evaluating the efficiency of KSM under different test conditions.
- **Build and Configuration:** `build.sh`
  to build the process.c

These components are compiled and managed using the 'build.sh' script (assuming availability), ensuring that all binary and script dependencies are properly set up for the experiments.

6 Experiments

The experiments are designed to measure KSM's performance across different configurations and workload:

6.1 Experiment 1

- **Goal**: To find out how much work the KSM need to do to scan the pages, We want to find this so that we can efficiently assign config to KSM so that it doesnt overwork or underwork
- **Setup**: Running proccess.c with random pages - no merging considered
- **Parameters**: CPU Utilization VS No of Pages to Scan for various per process total pages for constant time to scan setting as sleep_millisecond=1
- **Inference**: We can see that with increase in number of pages to scan CPU utilization increases untill it saturates. And with increase in total pages of the process, the CPU utilization increases.

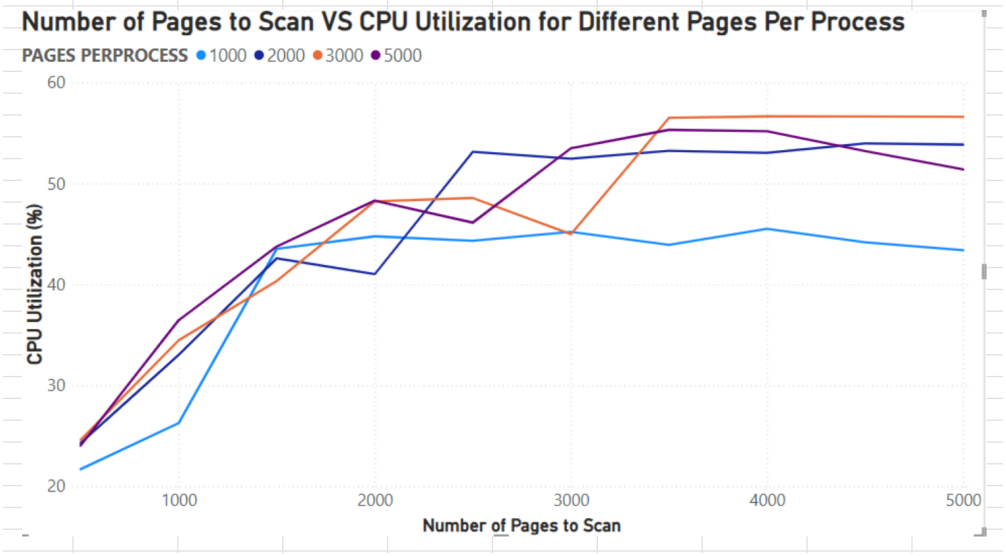

Figure 3: CPU Util VS Pages Scan

6.2 Experiment 2

- **Goal**: To find out whether or not the CPU utilization reduces upon increasing the sleep milliseconds, keeping the Pages per process constant.
- **Setup**: Running proccess.c with random pages - no merging considered.
- **Inference**: We can see that with increase in the sleep milliseconds, CPU utilization decreases. This is due to the fact that the pages are scanned and before the next round of page dirtying and the CPU sits idle after completing the job.

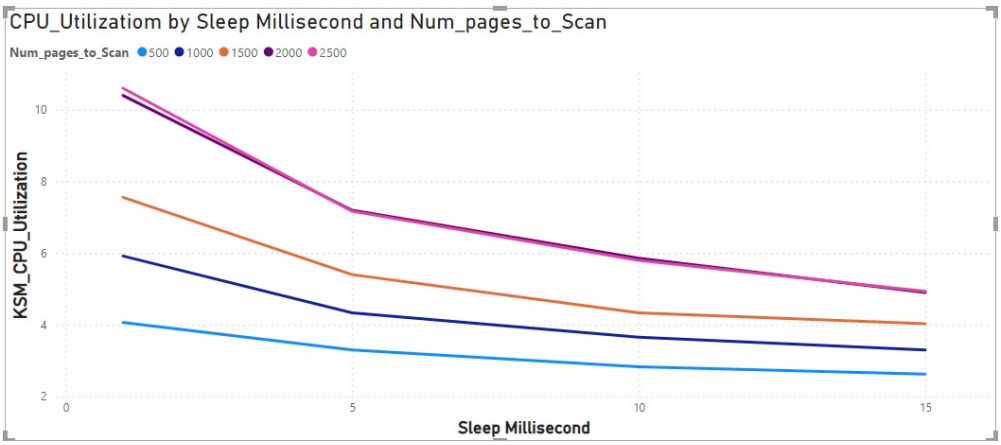

Figure 4: CPU Util VS Sleep Millisecond

6.3 Experiment 3

- **Goal**: To find out whether or not the CPU utilization reduces upon increasing the sleep milliseconds, keeping the Pages per process constant and varying the test type. First we do the experiment with test type "Random", followed by test type "Fixed-Random"
- **Setup**: Running proccess.c with random pages and Fixed Random pages - the case of merging and non-merging considered.
- **Inference**: We can see that with increase in the sleep milliseconds, CPU utilization decreases. This is due to the fact that the pages are scanned and before the next round of page dirtying and the CPU sits idle after completing the job. This follows in the case of both Fixed random workloads and Random workloads. It is also observed that the average CPU utilization is more in case of Random workloads because the pages are very less probable to be shared and the KSM module keeps scanning pages. In case of Fixed Random, there are shared pages, therefore the CPU needs to scan lesser number of pages in subsequent rounds.

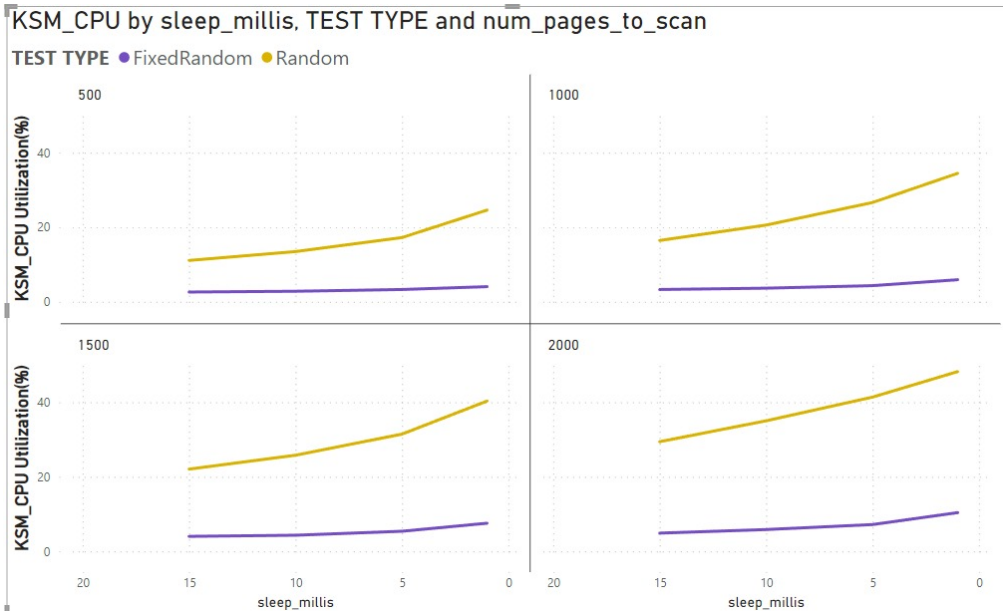

Figure 5: CPU Util VS Sleep Millisecond (Fixed Random and Random Workloads)

6.4 Experiment 4

- **Goal**: To find out the general profit as we increase the number of processes keeping the number of pages to scan constant.
- **Setup**: Running proccess.c with Fixed Random pages (meaning there are blocks of same content pages but not every page is same) - the case of merging considered.
- **Inference**: It is observed that as we increase the number of processes, the general profit increases, as the total number of pages shared across all processes keeps on increasing as the number of shared pages remain constant and the total pages shared increases.

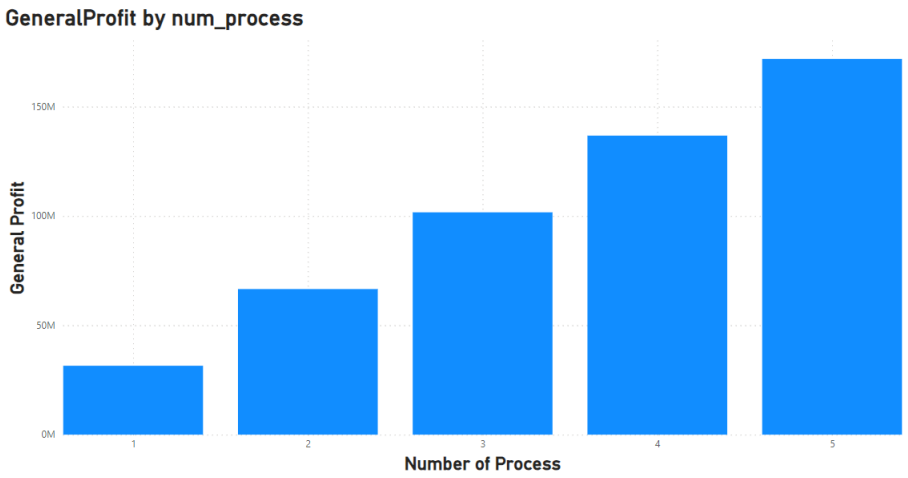

Figure 6: General Profit Vs Number of Processes

6.5 Experiment 5

- **Goal**: To find out the general profit as we increase the number of processes keeping the number of pages to scan constant.
- **Setup**: Running proccess.c with Fixed Random pages (meaning there are blocks of same content pages but not every page is same) - the case of merging considered.
- **Inference**: It is observed that as we increase the number of processes, the general profit increases, as the total number of pages shared across all processes keeps on increasing as the number of shared pages remain constant and the total pages shared increases.

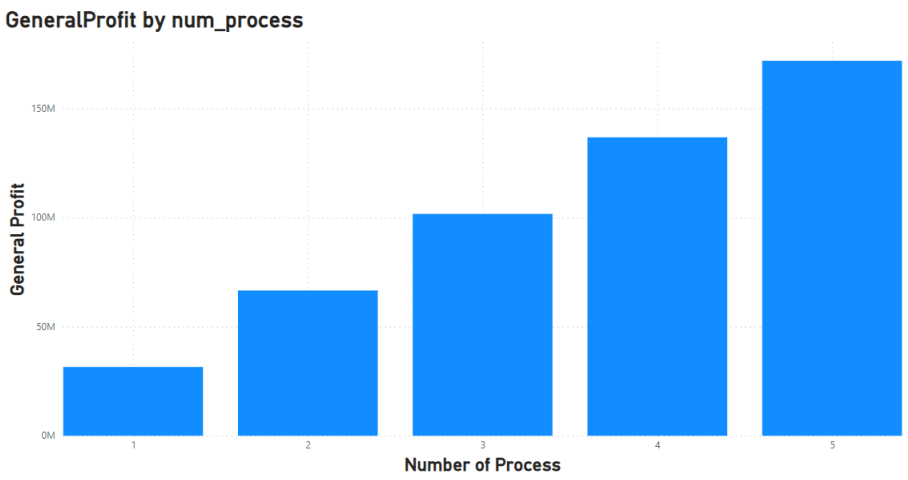

Figure 7: General Profit Vs Number of Processes

The total number of pages are also shown in the graph. As the number of processes increases, the total pages shared is shown to be increasing.

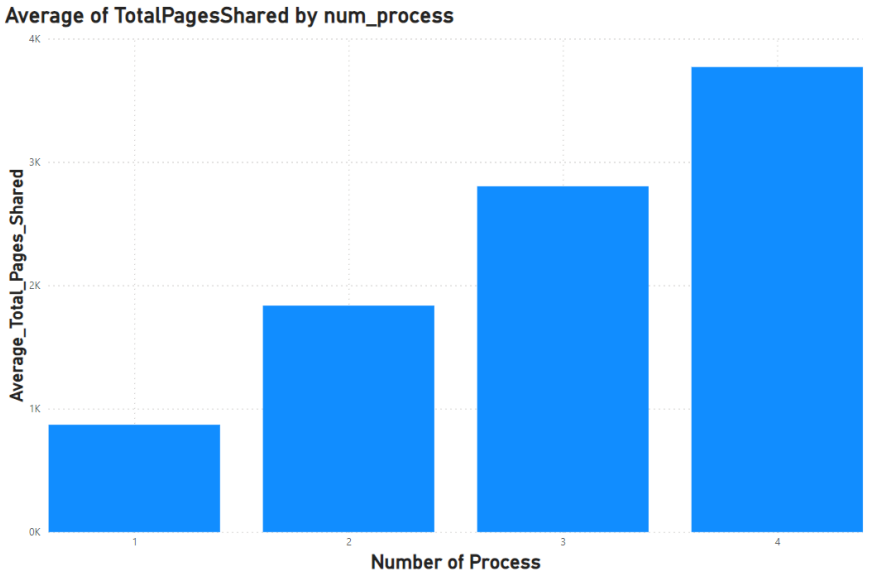

Figure 8: Total Pages Shared Vs Number of Processes

According to the experiments that we did using the processes, we found that the optimal pages to be scanned was 2000. The same value was used in the further experiment where we use different VMs instead of processes.

6.6 Experiment 6

- **Goal** To see the trend of the scan rate.
- **Setup** Running proccess.c with Fixed Random pages (meaning there are blocks of same content pages but not every page is same) - the case of merging considered.

- **Inference** We observe that the scan rate decreases as the sleep time reduces.

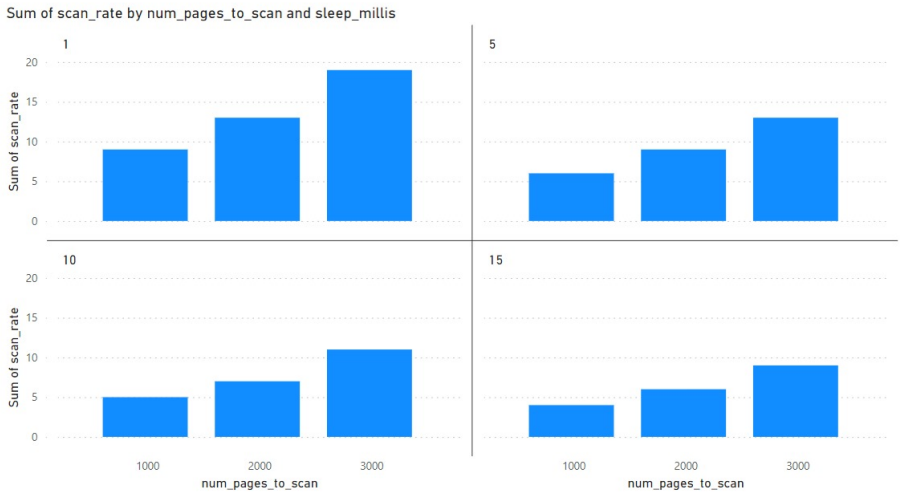

Figure 9: Scan Rate Vs sleep Milliseconds

6.7 Experiment 7

- **Goal** To do performance testing with real life load, hence we are using VMs and sysBench for load generation.
- **Setup** VMs are run using VirtManager. Every VM has SysBench installed and multiple database was created on each VM and sysBench was used to run read/write workllload on these databases
- **Inference** Similar to the processes, when multiple VMs are used, we observe that the total pages shared increases and the trend remains the same. This verifies the previous experiments even in the case of VMs.

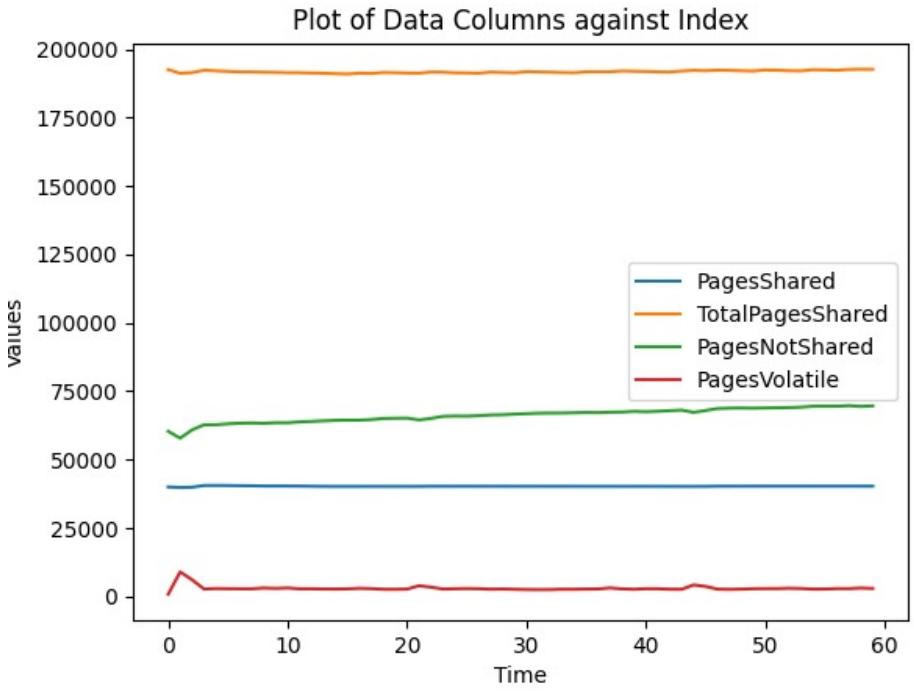

Figure 10: Total Pages Shared Vs Time

6.8 Experiment 8

- **Goal** To check how many pages were shared and its relation with scan rates
- **Setup** VMs are run using VirtManager. Every VM has SysBench installed and multiple database was created on each VM and sysBench was used to run read/write workload on these databases. similar setup to expt 6.
- **Inference** pages that are volatile decreases with increase in pages to scan and sleep millisecs but there wasn't noticable change for pages shared with increase in scan rate.

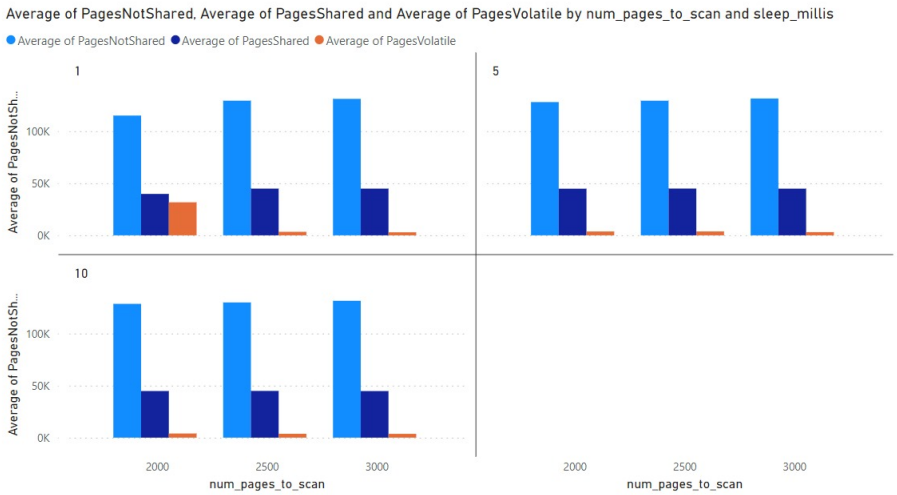

Figure 11: Pages Shared/Not Shared Vs Pages to scan

7 Discussion

The experiments conducted have provided valuable insights into the performance of KSM (Kernel Samepage Merging) under various configurations and workloads. These insights can guide system administrators and developers on optimizing KSM settings to balance performance and resource utilization effectively.

Firstly, the correlation between CPU utilization and the number of pages scanned is evident from Experiment 1. As the number of pages increases, so does CPU usage, up to a point of saturation. This suggests that while KSM can significantly aid in memory deduplication, it requires careful tuning of parameters like the number of pages to scan to avoid excessive CPU consumption.

In Experiment 2 and Experiment 3, adjusting the sleep milliseconds between scans showed a clear decrease in CPU utilization. This indicates that by controlling the scan frequency, we can reduce CPU load, potentially at the cost of slightly slower reaction times to memory changes that could benefit from deduplication.

Moreover, the distinction between different types of workloads—Random and Fixed-Random—highlights how the nature of the workload affects KSM's efficiency. The Fixed-Random setup, which simulates a more predictable and repetitive memory usage pattern, allows KSM to perform more effectively by merging pages more frequently than in the completely Random setup, where less predictability leads to fewer opportunities for deduplication.

From Experiment 4, we observe that the memory savings and general profit from KSM increase as more processes participate, reflecting KSM's ability to effectively deduplicate memory across a growing number of processes. This scalability also extends to virtualized environments, as demonstrated in Experiment 5, where similar benefits are observed with increasing VM counts. The consistency of these results across both physical and virtual environments underscores KSM's robust deduplication mechanism and its adaptability to diverse operational conditions.

The findings from Experiments 1 through 5 provide a comprehensive view of how KSM's performance scales with increasing numbers of processes and VMs, as well as its efficacy under different workload types. These insights confirm that KSM is not only effective in optimizing memory usage but also demonstrates significant scalability and resource efficiency across varied setups. This makes it a valuable tool for both smaller systems and large-scale deployments.

8 Conclusion

KSM is a powerful tool for memory management that, when configured correctly, can provide substantial benefits in terms of memory savings and potentially improved system performance due to reduced memory pressure. The experiments demonstrate that:

- Performance vs. Cost Trade-off: Increasing the scan rate can improve deduplication effectiveness but also raises CPU usage, presenting a trade-off between memory efficiency and CPU resource consumption.
- Workload Sensitivity: The effectiveness of KSM is highly dependent on the memory access patterns and homogeneity of the processes. Systems running applications with similar or repetitive memory patterns can benefit more from KSM.
- Optimization of Parameters: Configuring the sleep intervals and scan settings according to the system's workload can optimize the benefits of KSM. Systems administrators should consider these parameters carefully based on the typical use case of their systems.
- KSM demonstrates excellent scalability properties as the number of processes or VMs increases, indicating its suitability for both small and large-scale systems.

Future work could explore adaptive algorithms that adjust KSM settings in real-time based on current system load and memory usage patterns, potentially increasing the efficiency of this tool in dynamic environments. Additionally, integrating more detailed system metrics and machine learning models could predict optimal KSM configurations for given workloads, further enhancing system performance and resource utilization.