CS 695 Assignment 4
What is my working set?

Isha Arora
210050070

Karan Godara
210050082

Indian Institute of Technology, Bombay

April 14, 2024

Contents

1 Aim of Experiment 2

2 Building the setup 2

3 Experiments 3
3.1 Experiments Designed: 3
3.2 Experiment Details 4
3.3 Results Obtained, Observations and Conclusions 6
3.3.1 seq_access 6
3.3.2 random_access 7
3.3.3 rare_access 9
3.3.4 alternate_page_access 10
3.3.5 every_fifth_access 12
3.3.6 temporal_local_access 13
3.3.7 chunk_array_access(spatial locality) 15
3.3.8 mix_seq_random_seq 16
3.3.9 mix_rare_chunk_rare 18
3.3.10 mix_seq_rare_random_seq_rare_seq_random 19

4 Further Experiments 20
4.1 Experiments Designed 20
4.2 Results Obtained, Observations and Conclusions 21

5 Final Conclusions 22

1 Aim of Experiment

The aim of the experiment is to evaluate the effectiveness and agility of the memory working set estimation method proposed in the paper "Memory Resource Management in VMware ESX Server" (memmgmt). This method utilizes a statistical sampling approach to estimate the working set size of a virtual machine (VM) without guest involvement. At regular intervals, a small number of the virtual machine's physical pages are randomly selected. These selected pages are tracked, and subsequent accesses trigger page re-mappings and increment a touched page count. At the end of each sampling period, a statistical estimate of the fraction of actively accessed memory is calculated.

The experiment aims to address a gap identified in the paper which lacks comprehensive reasoning and experimental validation for its proposed memory working set estimation method. Specifically, the paper introduces a method for estimating the working set size of virtual machines (VMs). However, it does not go deep into the underlying rationale behind the method, nor does it provide empirical evidence to support its effectiveness. It brings to our mind several questions such as how can such a small number of sampled pages over a large time period offer a 'good' estimate of the working set of a VM? Good enough that a very important decision of how much memory should be allocated to a VM can be taken solely on this basis. Why did the paper choose to take a slow moving average, a fast moving average and a current estimate and finally choose the maximum of all these to be its final estimate? Didn't the maximum highly overestimate the size of the VM?

Thus, the experiment seeks to fill this gap by conducting a systematic evaluation of the proposed method. It aims to investigate the efficacy and agility of the method across various VM memory usage patterns under varying parameters like sampling periods or the sample size of randomly selected pages.

2 Building the setup

For conducting the experiments, we build upon the foundational setup established in Assignment 2, which involved a mini-VM equipped with rudimentary functionalities. **To enhance this setup for the experimental investigations, we implemented several modifications and improvements**:

1. We **reduced the granularity of page size** from 2MB to 4KB to have ample page availability for analysis.
2. To accommodate the decrease in granularity and properly configure the memory setup for the virtual machine, we **increased the number of page levels** from 3 to 4.
3. To expand the capacity of VM memory and accommodate a larger number of pages, we **augmented the VM size** from 2MB in the base setup to 32MB. This increase resulted in expanding the number of pages available for experimentation from 512 to 8192.
4. Within the "setup_long_mode" function, we **carefully populated the page table entries to effectively manage the memory configuration**. This adjustment became necessary as we transitioned from utilizing 2MB pages to 4KB pages and simultaneously augmented the total size of the VM.
5. We **relocated the page table pages** to the upper region of the VM memory address space and **configured the stack pointer** to point just below these pages. This strategic adjustment was made to mitigate the risk of inadvertently modifying critical addresses, such as those belonging to the page table, during experimental operations involving the allocation and random access of variables. By segregating these sensitive addresses from the dynamic memory region, we ensured the stability of the memory management infrastructure.

6. During the initialization process in the "vm_init" function, when configuring the kvm_userspace_memory_region, we **activated the logging of dirty pages**. This feature allows us to monitor and track the pages accessed within a specific time frame. To enable this functionality, we set the "KVM_MEM_LOG_DIRTY_PAGES" flag while creating the memory slot for the VM.

7. Implemented within a **dedicated thread, the estimator functionality periodically reads the dirtied pages and leverages this data to estimate the working set size, aligning with the methodology outlined in the paper**. This thread systematically collects information on page access patterns and utilizes statistical analysis to derive estimations of the actively accessed memory within the virtual machine.

8. To facilitate the operation of the estimator thread, we **implemented a custom binary search tree (BST) based set data structure**. This implementation was necessitated by the absence of native support for a set data structure in standard C libraries. The BST set enables efficient sampling of randomly required unique pages for the experiment, enhancing the functionality of the estimator thread.

9. We **introduced a hypercall** named "HC_Sleep" designed to enable the VM to enter a sleep state for a duration of 1 second. This hypercall was instrumental in emulating periods of non-memory accessing behavior, replicating the behavior observed in conventional VMs during certain intervals. By incorporating this functionality, we ensured a more accurate simulation of real-world VM behavior within the experimental environment.

3 Experiments

3.1 Experiments Designed:

In order to evaluate the effectiveness of the method outlined in the paper, we created and integrated diverse memory access patterns into the VM. Those being:

1. **seq_access** : In this implementation, the VM systematically accesses all available memory addresses in a sequential manner. This access pattern is executed in an infinite loop, simulating the behavior observed in **VMs that continuously access nearly all accessible memory**.

2. **random_access** : In this implementation, the VM randomly selects memory addresses accessible to it and accesses them continuously within an infinite while loop. This approach **mirrors the behavior of VMs that utilize all available memory, albeit in a non-sequential manner**.

3. **rare_access** : In this implementation, the VM intermittently accesses a small subset of available memory locations randomly. After each round of access, the VM invokes the 'HC_sleep' hypercall to sleep for 1 second before resuming the pattern. This approach replicates the behavior of **VMs that sporadically access limited memory resources in a random fashion**.

4. **alternate_page_access** : In this implementation, instead of accessing every page sequentially as in 'seq_access,' we alternately access memory pages. This simulates the behavior of **VMs that access memory in a somewhat sequential order** but do not access the entire memory space.

5. **every_fifth_access** : In this implementation, we follow a pattern similar to 'alternate_page_access,' but instead of accessing every other page, we access every fifth page. This approach mimics the behavior of **VMs that access memory in a sequential or fixed pattern**, but only a subset of the allocated memory.

6. **temporal_local_access** : In this implementation, we continuously access a selected set of memory addresses within an infinite loop. This emulates the behavior of **VMs that repeatedly access a few specific memory addresses from their allocated memory space in a temporal pattern**.

7. **chunk_array_access** : In this implementation, we repeatedly select a random location in memory and access the next 2000 addresses starting from it within an infinite loop. This emulates the behavior of memory-intensive **VMs that access memory randomly but exhibit spatially localized access patterns.** For instance, VMs continuously access different arrays and access a range of contiguous elements from each array.

In addition to the memory access patterns described above, we further enhanced our simulation to emulate typical real-life memory access scenarios by combining multiple patterns described above together. These implementations are :

1. **mix_seq_random_seq** (Medium-High-Medium access pattern) : **This pattern exhibits a Medium-High-Medium access pattern**, characterized by a sequence of memory access behaviors within the VM environment. Initially, the VM accesses all allocated memory in a sequential manner. After a period of time, it transitions to randomly accessing memory addresses. Finally, it reverts to sequential memory access again. The sequential part of the pattern represents a Medium memory access pattern, as only a subset of pages is accessed between two sampling periods. In contrast, the random access phase is considered High, as it involves accessing at least one address from each page, resulting in access to all available pages between sampling times. This pattern mirrors the behavior of VMs that first create and initialize a large array, resulting in sequential memory access. Subsequently, the VM performs operations on this array, leading to random accesses of its elements. Finally, the VM iterates through each element of the array, performs additional operations, and deletes the array.

2. **mix_rare_chunk_rare** (Low-High-Low access pattern) : In this implementation, we emulate a **Low-High-Low access pattern** within the VM. Initially, the VM exhibits minimal memory consumption or sporadic memory usage. However, after a certain period, it abruptly transitions to utilizing all available memory in a random order, representing a High memory access phase. Subsequently, the VM returns to a state of low memory consumption. This pattern replicates the behavior of VMs that may initially be idle or awaiting input/output (IO) operations. Upon completion of these tasks, they undergo a sudden surge in memory usage, accessing memory in a random fashion. After completing the workload, the VM reverts to a state of minimal memory usage or intermittent access, potentially waiting for the next IO operation.

3. **mix_seq_rare_random_seq_rare_seq_random** : In this implementation, we aim to simulate the diverse memory access patterns exhibited by VMs over their lifespan. This composite pattern combines various access behaviors, ranging from minimal memory usage to near-full memory utilization, with intermittent transitions between different access modes. The pattern incorporates sequential memory access, sporadic or rare memory access, random memory access, and sequential-random-sequential transitions. **By integrating these diverse access patterns, this implementation serves as a comprehensive representation of general memory access behaviors observed in real-world VMs.**

3.2 Experiment Details

Each estimation of working set size is called a **'run'**. We have the following configuration parameters:

1. **Sampling Period**: This is the time between the two consecutive runs (estimations of working set size). Initially, we set the sampling period as 30 seconds(as it is set in the 'memmgmt' paper).

2. **Sample Size**: This is the number of pages that are sampled in each run. Initially, we set the sample size as 100(as it is set in the 'memmgmt' paper).

For each run of the experiment setups listed above, we report the following:

1. Number of pages accessed in the run by the VM
2. Number of pages accessed from the sampled pages in the period

From the above, we compute the following:

1. **Slow-moving average of** the number of pages accessed from the sampled pages: The slow-moving average was computed using a weighting of 0.9 for previous values and 0.1 for the current estimated value. The formula for the slow-moving average (SMA) is:

$$SMA_t = 0.9 \times SMA_{t-1} + 0.1 \times x_t$$

   where SMA_t represents the slow moving average at time t, and x_t represents the current estimated value at time t.

2. **Fast-moving average** of the number of pages accessed from the sampled pages: The fast-moving average was computed using a weighting of 0.1 for previous values and 0.9 for the current estimated value. The formula for the fast-moving average (FMA) is:

$$FMA_t = 0.1 \times FMA_{t-1} + 0.9 \times x_t$$

   where FMA_t represents the fast moving average at time t, and x_t represents the current estimated value at time t.

3. **Max estimate** of the number of pages accessed from the sampled pages: The maximum estimate was computed as the maximum value among the current sample, the slow-moving average, and the fast-moving average.

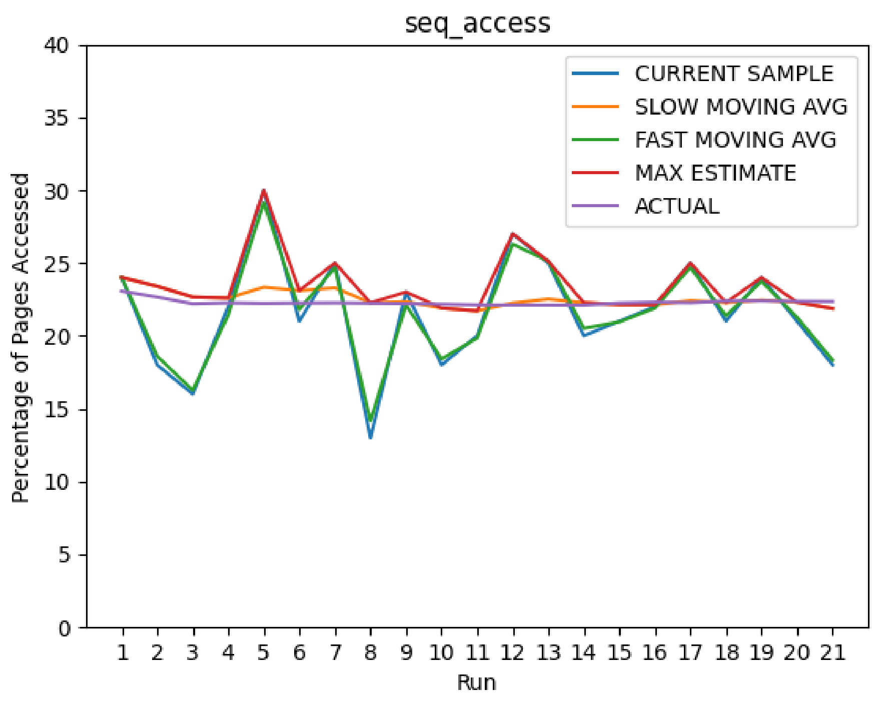
From the above values, as the number of sampled pages and the actual size of memory of the VM are known, the following values were computed:

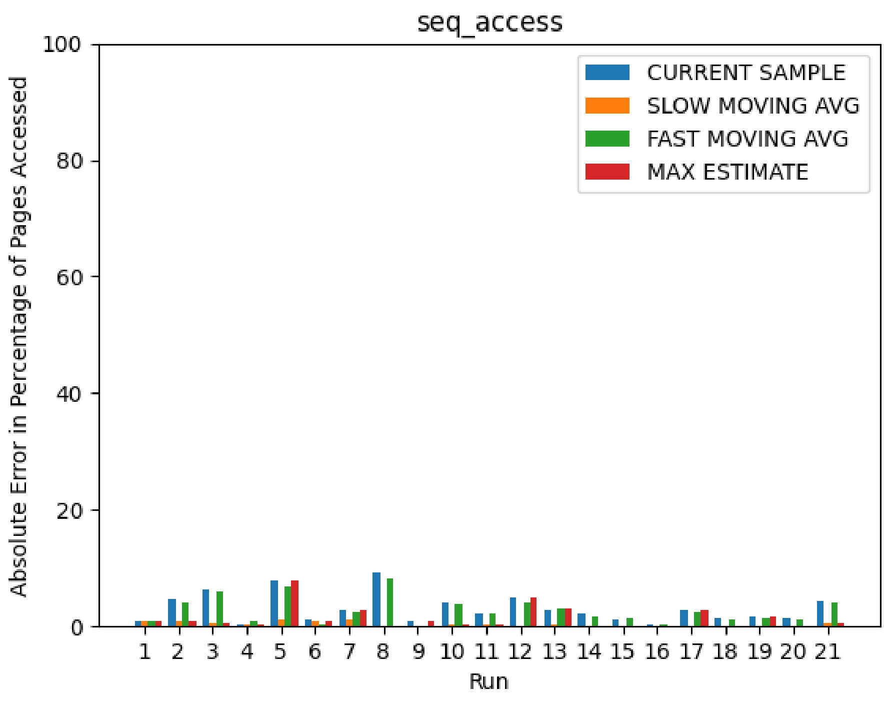
1. Percentage of pages accessed in the current sample
2. Actual percentage of pages accessed by the VM in the current run
3. Slow-moving average of the percentage of pages accessed in the samples over many runs
4. Fast-moving average of the percentage of pages accessed in the samples over many runs
5. Working size estimate of the current run
6. Actual working set size of the VM in the current run
7. Slow-moving average of the working set size estimates
8. Fast-moving average of the working set size estimates
9. Working set size estimate computed using the max estimate

Using the above values, the **errors** in the working set size estimate and the percentage of pages accessed of the current sample, slow moving average, fast moving average, and the max estimate were also computed.

3.3 Results Obtained, Observations and Conclusions

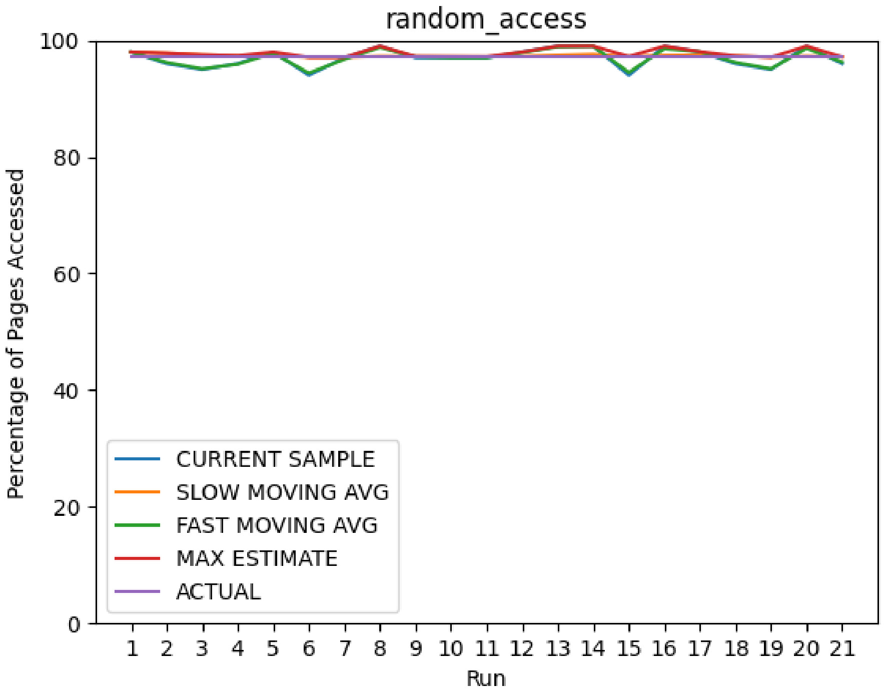
3.3.1 seq_access

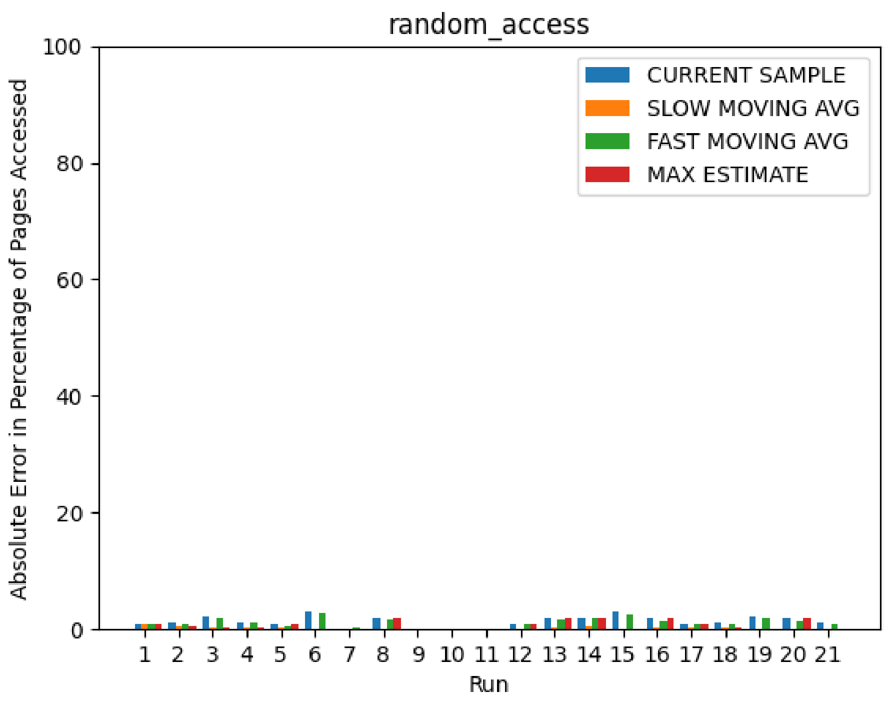




The graphs above illustrate that the memory access pattern remains relatively constant due to sequential memory location accesses, resulting in the actual percentage of pages remaining almost stable over time. Among the different averaging methods used to model memory access, the **slow-moving average demonstrates superior performance in capturing the actual memory access behavior**. Unlike the fast-moving average and current estimate, which respond quickly to individual sample fluctuations and hence exhibit more significant variability, the slow-moving average provides a more stable representation of the overall memory access pattern. Its **inherent stability** enables it to accurately model the stable actual access behavior, offering a reliable depiction of the system's memory utilization over time. Unlike, the current and fast-moving average estimates which undershoot the actual memory access, the **'max estimate'** never undershoots the actual memory access, and hence is stable and **performs reasonably well**.

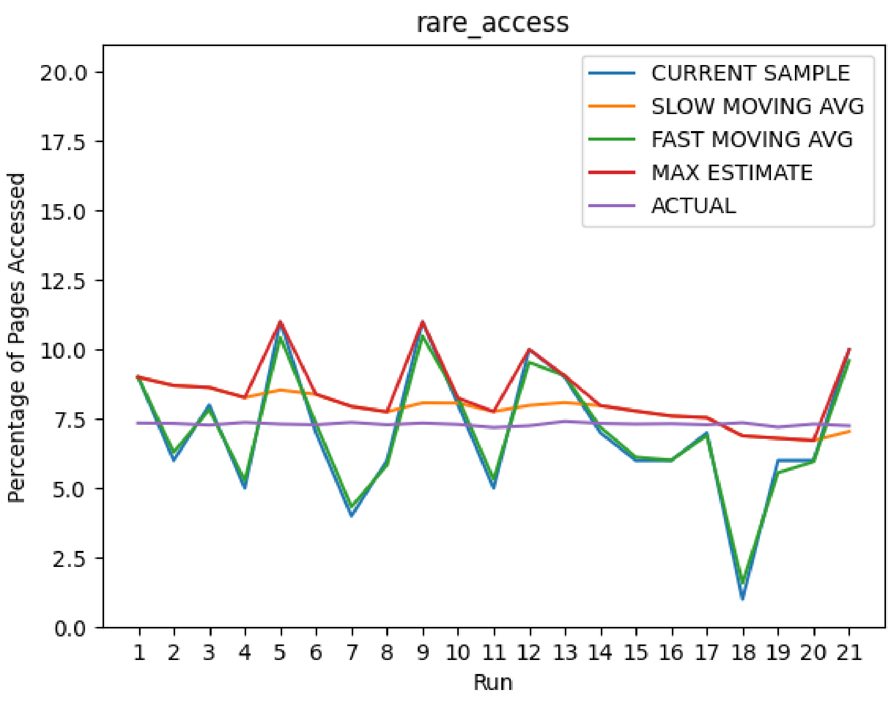
3.3.2 random_access

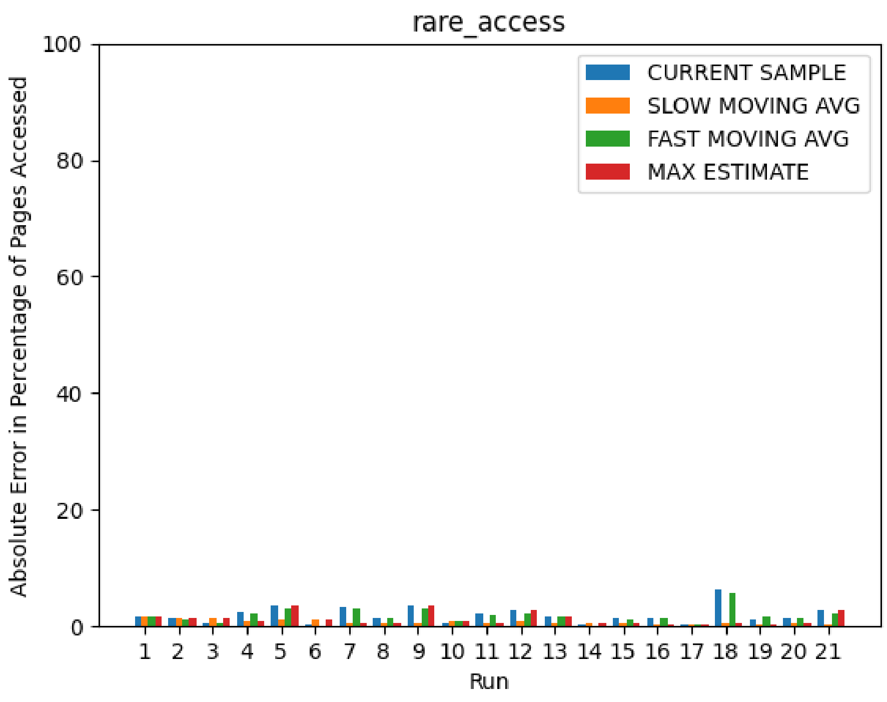




In the case of random access, nearly the entire memory is accessed in each run. Given that random access typically involves accessing almost all of the pages in each run, **all of the estimation methods perform well in modeling this behavior**. This is because, in random access scenarios, as in each run almost all of the pages are accessed, and almost all of the pages are sampled hence the percentage of sampled pages accessed is the same as the percentage of actual memory pages accessed, and therefore all of the estimates, fast-moving average, slow-moving average, max estimate, and current estimate, effectively capture the extensive memory access observed in each run.

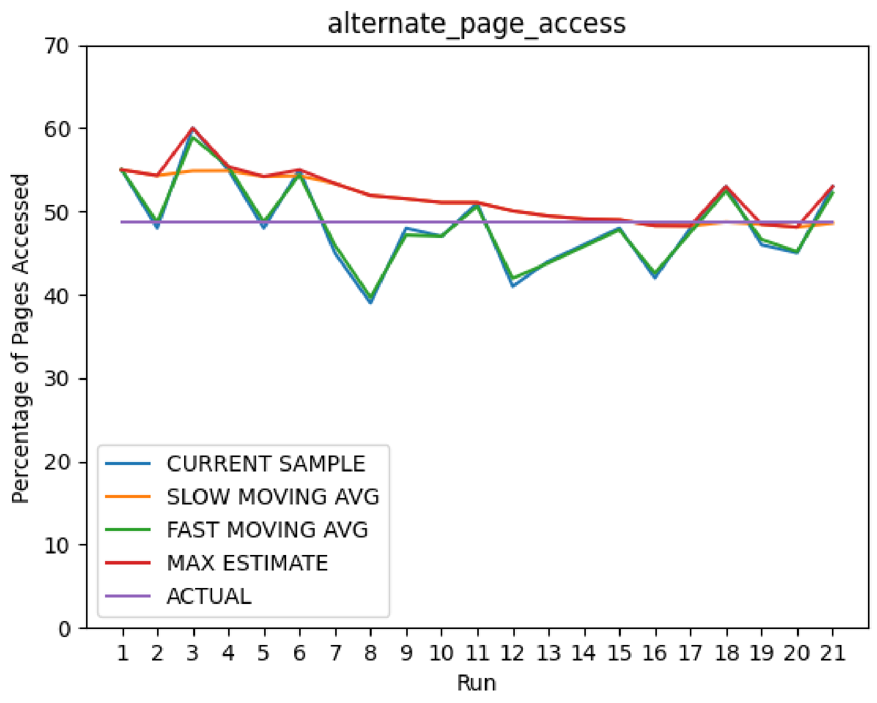
3.3.3 rare_access

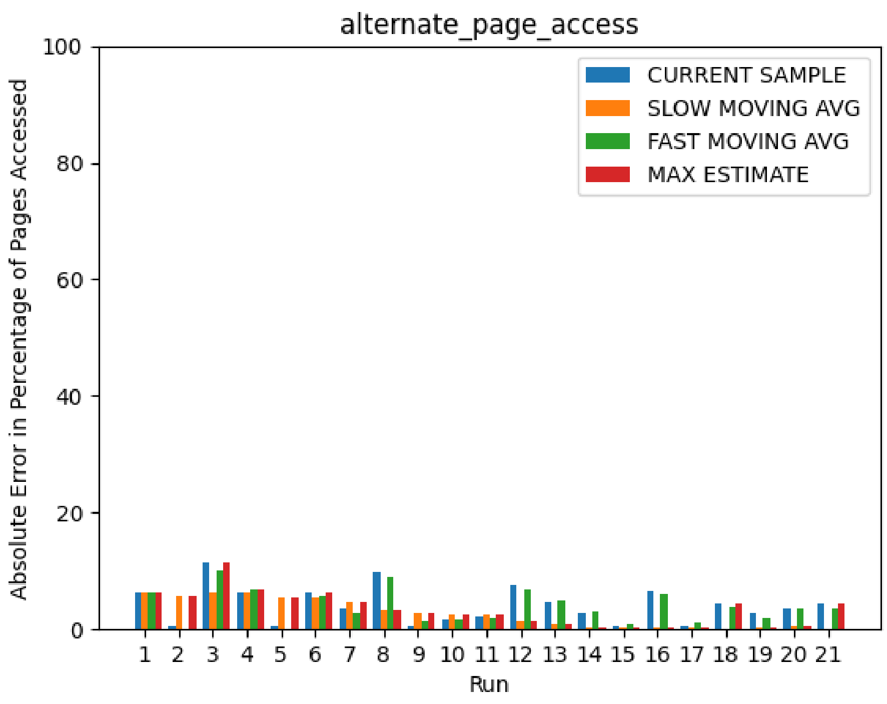




This is similar to the case of sequential access. But here it is to be noted that the **slow-moving average does not perform as well as it did earlier**. This is because in the initial runs, the initial samples were not exactly accurate. But as we can see in the figures, the slow-moving average in a few runs starts estimating the actual stable memory access really well. The fast-moving average fluctuates quite highly as the memory access is rare and hence a lot of times sampled pages do not correctly represent the actual memory access. We can see that in one run, fast moving average underestimates the memory access by 7 percent. The **max estimate also performs well** as it closely follows the higher and closer slow-moving average.

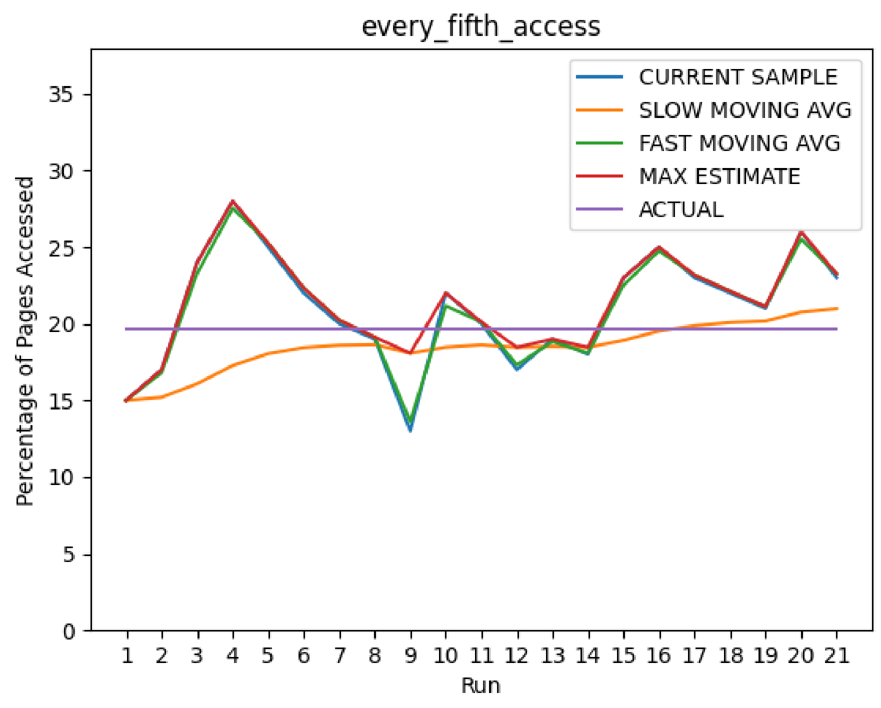
3.3.4 alternate_page_access

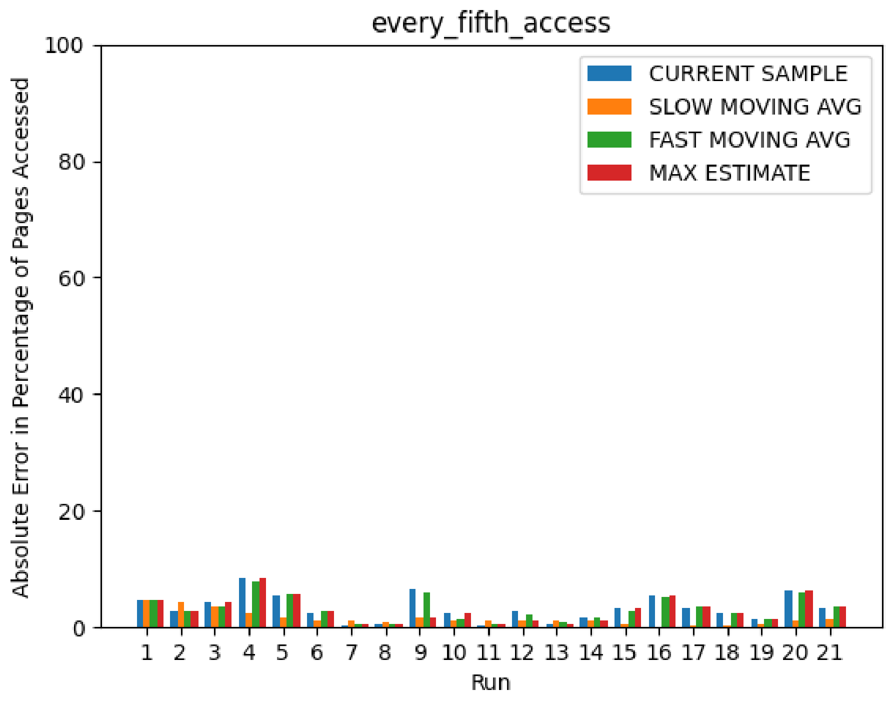




The results obtained in the case of alternate access are very similar to the case of rare access and the same explanation follows.

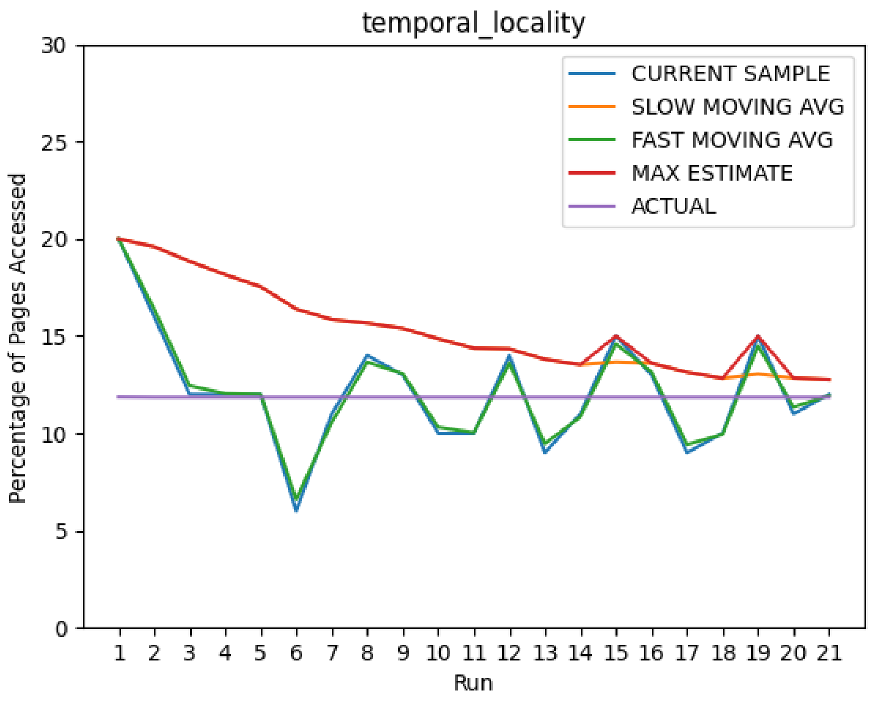
3.3.5 every_fifth_access

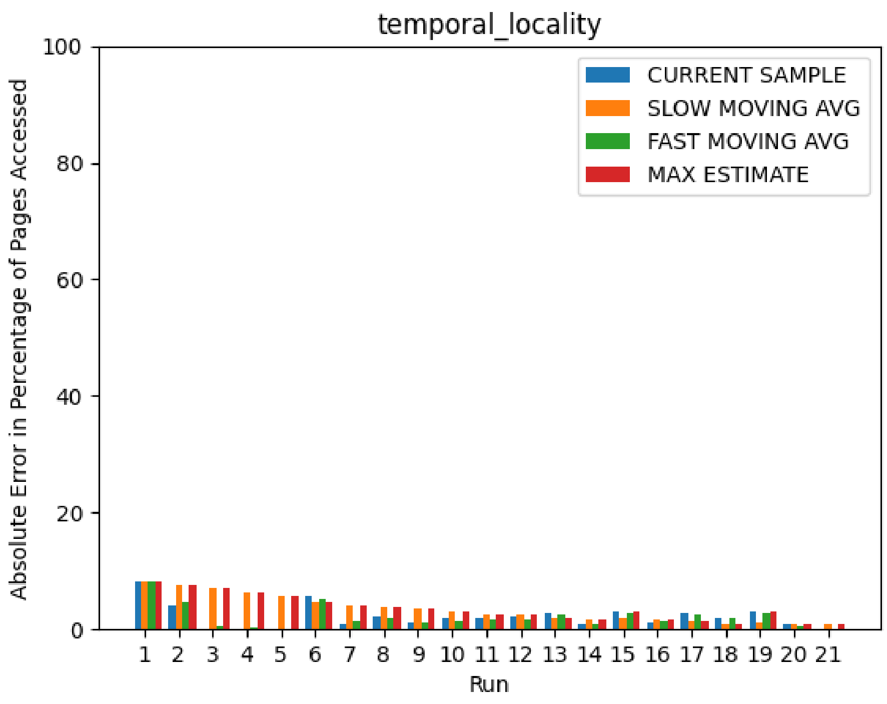




This case is interesting as in this, in the initial runs the samples underestimate the memory access. As **slow moving average** progresses slowly, it **underestimates** the VM working set for many runs. **Fast moving average** as seen before, keeps on **fluctuating**. But here the **'max estimate' still performs well** as it does not underestimate the working set size in the initial runs like the slow-moving average and neither does it fluctuate as much as the fast-moving average.

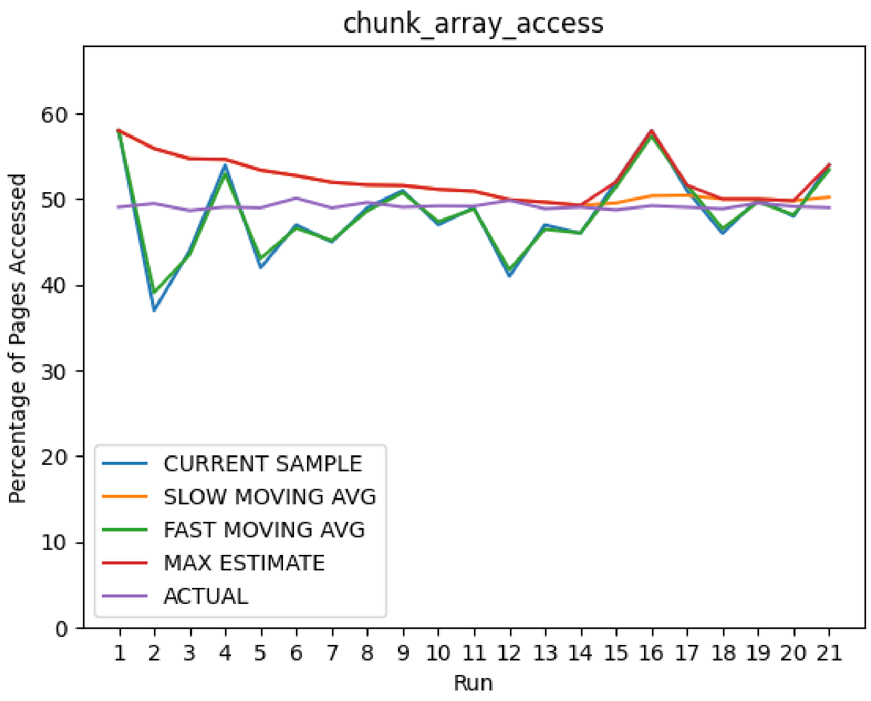
3.3.6 temporal_local_access

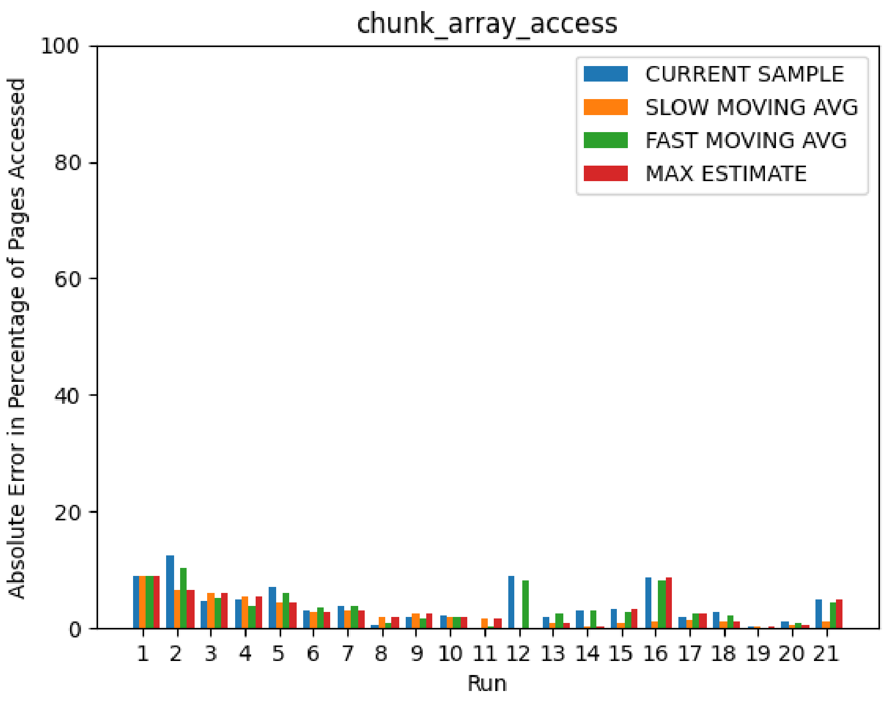




The sampling method does not take into account temporal locality of access of pages yet as we can see that our sampling method still performs well in this case as the errors are small in magnitude. As we have seen before the slow and 'max estimate' perform well in cases like this where the memory access pattern remains constant.

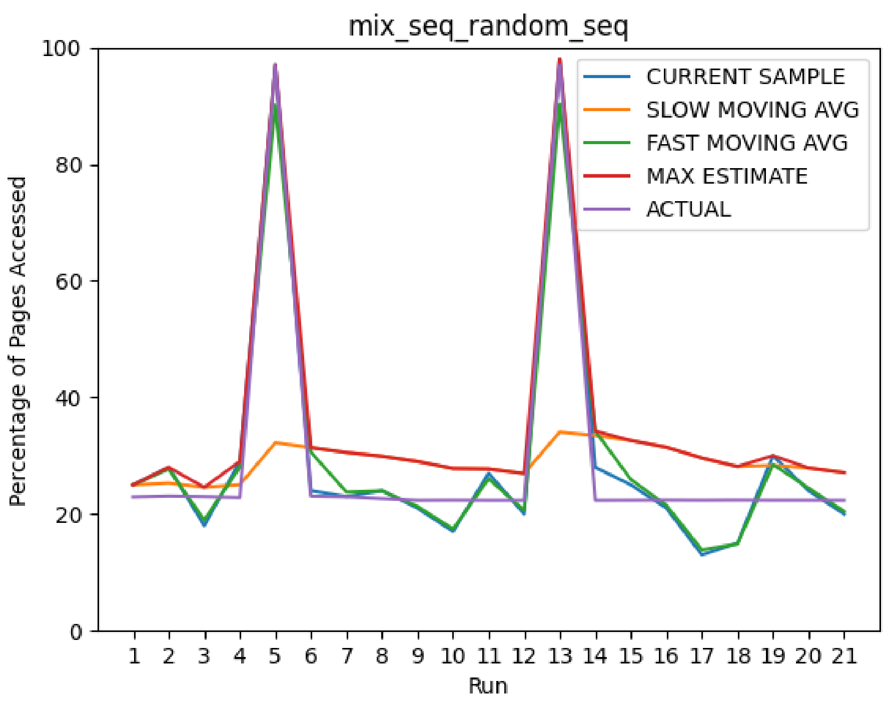
3.3.7 chunk_array_access(spatial locality)

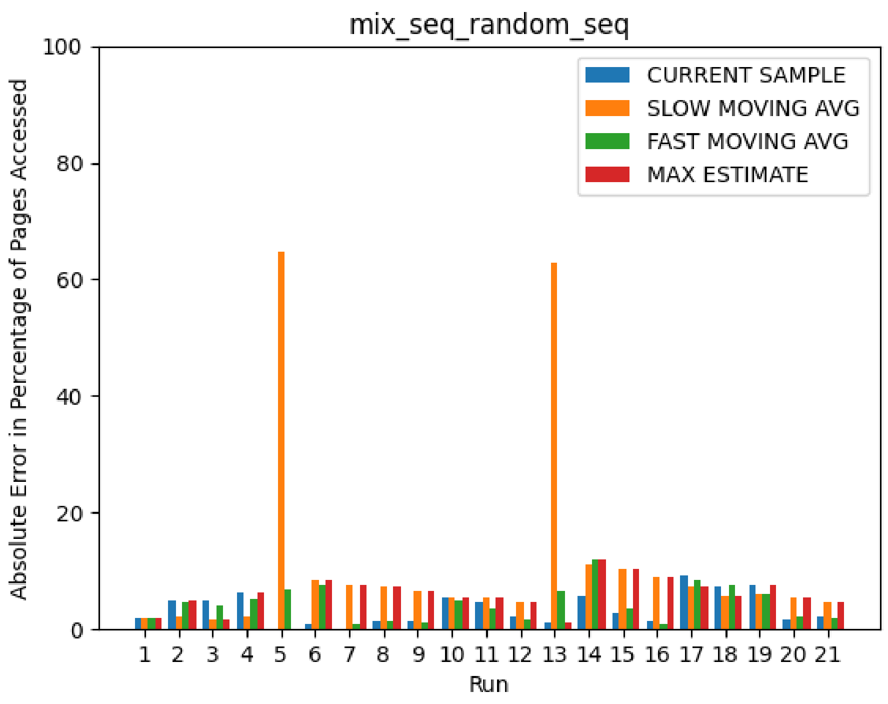




The sampling method does not take into account spatial locality of access of pages yet as we can see that our sampling method still performs well in this case as the errors are small in magnitude. As we have seen before the slow and 'max estimate' performs well in cases like this where the memory access pattern remains constant.

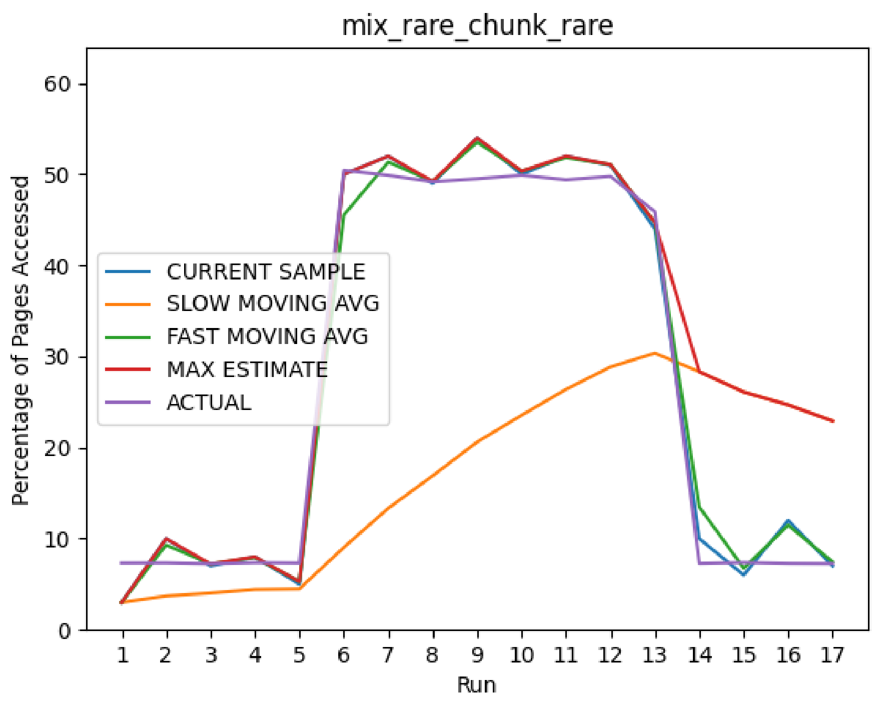
3.3.8 mix_seq_random_seq

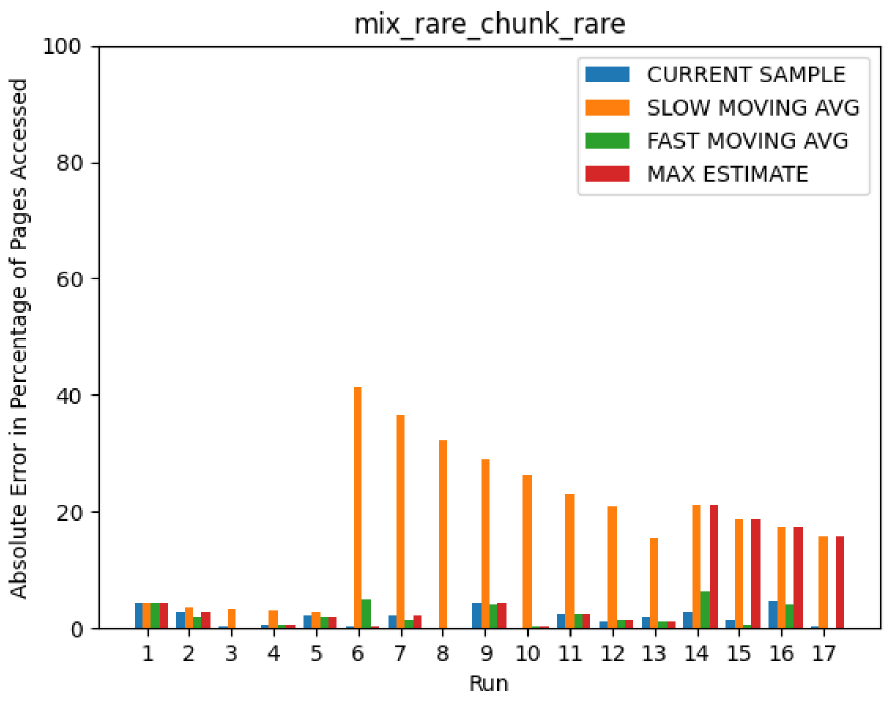




This case depicts the advantage of having fast moving average. In the case when memory access changes from sequential to random, fast moving average and current estimate **quickly adapt** to the change and increase their working size estimate. On the other hand, the slow-moving average as the name says, doesn't quickly adapt to the change and heavily underestimates the working set size(error of about 70 percent). The **'max estimate' like the fast-moving average also quickly adapts** to the change in working set size and performs well.

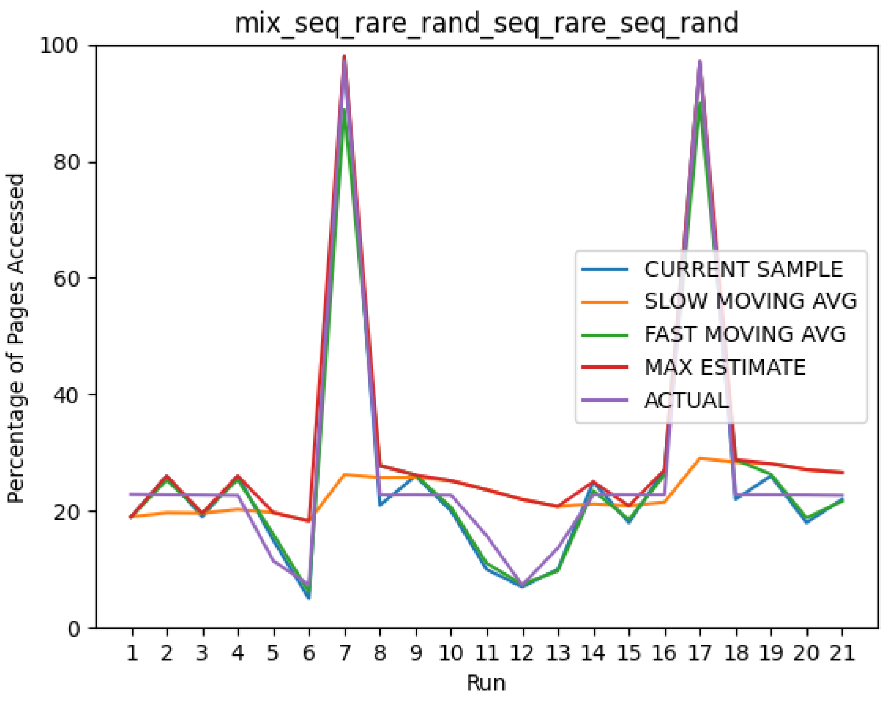
3.3.9 mix_rare_chunk_rare

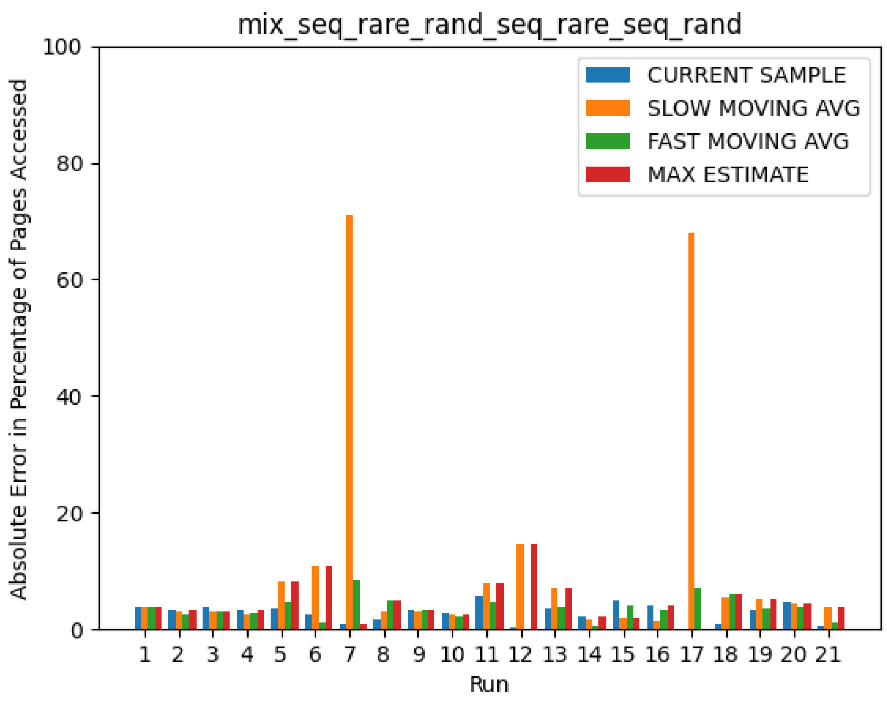




In the previous case, there was a sudden surge in the memory access but in this case, we see there is a larger high peek, and thus the memory access remains high for a long time. In the previous case, one may say that the slow-moving average didn't perform well just for 1-2 runs, but here we can see that **even if there is a prolonged high memory access, the performance of the slow-moving average remains poor**. The other quickly adapting estimates perform well as reasoned before.

3.3.10 mix_seq_rare_random_seq_rare_seq_random





This is the case in which memory access has varying patterns as seen in many real-life VMs. As observed before, the **slow-moving average performs very poorly** in the case of sudden memory access change. The **fast-moving average** adapts very quickly to sudden memory changes but has **a lot of fluctuations** even in the case of a constant memory access pattern. However, the **'max estimate' is stable in case of fluctuations** due to the slow-moving average and **can adapt quickly** due to the fast-moving average, and thus models the memory access pretty well.

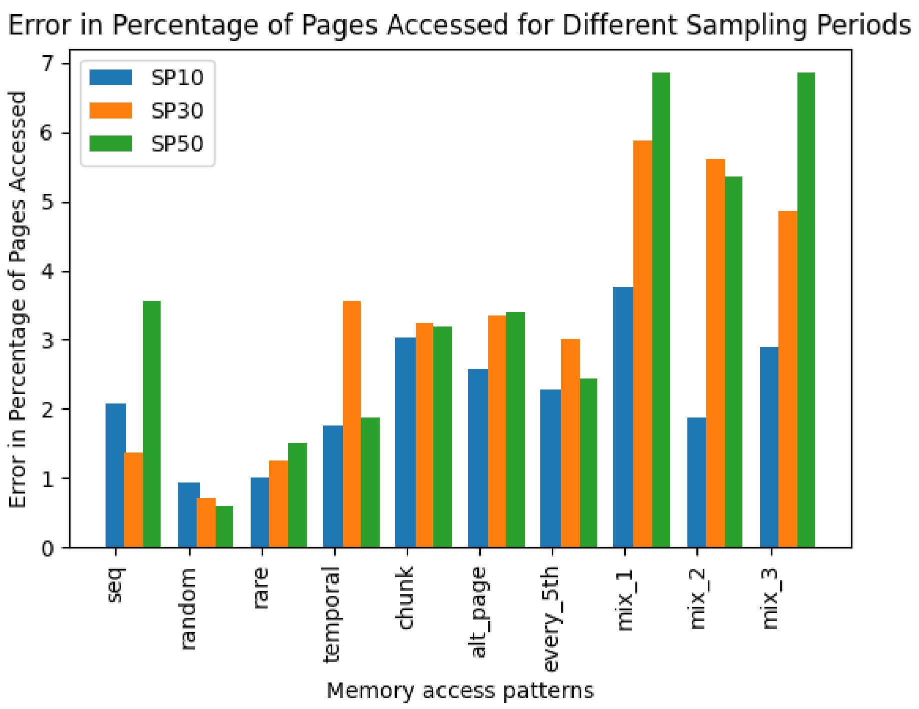
4 Further Experiments

4.1 Experiments Designed

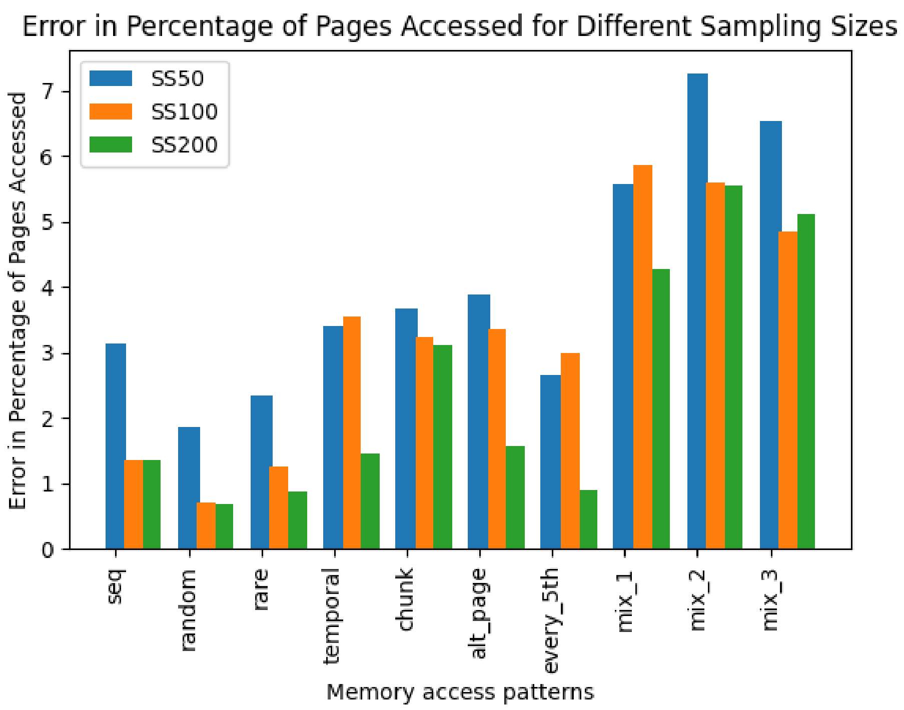
Now, we experimented by changing the values of the configuration parameters. We did 4 experiments:

1. **SS50** - Set the sampling size to be 50
2. **SS200** - Set the sampling size to be 200
3. **SP10** - Set the sampling period to be 10
4. **SP50** - Set the sampling period to be 50

4.2 Results Obtained, Observations and Conclusions



As shown in the above graph, when the memory access pattern keeps varying as in the case of **mix accesses, the error increases as the sampling period increases**. This happens because a smaller sampling period allows us to monitor the change in memory accesses more frequently and thus have a better estimate of the current working set of the VM. But in the case when the **memory access pattern is constant is the VM, this order is not strictly followed** as seen in the case of random and temporal accesses. As the memory access pattern is not changing, the benefit of a smaller sampling period in being able to detect a change in memory access pattern easily is lost and hence we can't say for sure that it would have a smaller error. In such cases, a larger sampling period may help to model the memory access pattern better as it gives a longer opportunity to observe the memory accesses.



As can be seen in the above graph, **as the sampling size increases, the error decreases**. This is because having a larger sampling size allows us to make better estimates of the actual memory access and hence gives better results. However it is to be noted that we can't arbitrarily increase the sampling size as it comes at the cost of efficiency. Having a larger sample size requires more computation and hence the efficiency of the sampling process decreases.

5 Final Conclusions

We started out with the aim to investigate the efficacy and agility of the working set estimation method as suggested in the paper 'Memory Resource Management in VMware ESX Server" (memmgmt)' as the paper lacked comprehensive reasoning and experimental validation for its proposed method. It seemed counterintuitive that randomly sampled 100 pages over a 30 second period for a memory which actually spans over thousands of pages would provide a correct estimate of a working set size of a VM, correct to the extent that we can snatch away or provide more memory to it on 'just' on this basis. Add to it the plethora of memory accesses that a VM can have and how fast it might switch between these memory accesses. But it turns out as a result of our comprehensive testing that in fact, 'max estimate' computed for 100 pages over a 30 second period is in fact a really good estimate of the working set size of a VM. A summary of the result of our experiments is as follows:

1. **Moving Averages vs. Current Estimate**: Moving averages, particularly the slow-moving average, provide a more stable estimate of the working set size compared to the current estimate. While the current estimate may fluctuate significantly due to individual sample variations, moving averages offer a smoother representation of the overall memory access pattern.

2. **Slow-Moving Average Performance**: The slow-moving average performs well in capturing stable memory access patterns but struggles to adapt to sudden changes in memory access behavior. It

may underestimate or lag behind the actual working set size, especially during transitions between different access modes.

3. **Fast-Moving Average Performance**: The fast-moving average quickly adapts to changes in memory access patterns but tends to exhibit fluctuations, even in cases of stable memory access. While it provides a rapid response to variations, its reliability may be compromised due to high variability.
4. **Max Estimate Advantage**: The max estimate combines the stability of the slow-moving average with the adaptability of the fast-moving average. It remains stable during fluctuations yet adjusts quickly to changes in memory access patterns, making it a robust estimator of the working set size.
5. **Effect of Sampling Period**: In general, as the sampling period increases, the error in estimating the working set size tends to increase. However, this trend may not hold in cases of stable memory access patterns, where a larger sampling period can provide a better estimation.
6. **Effect of Sampling Size**: Increasing the sampling size generally decreases the error in estimating the working set size. However, larger sampling sizes come at the cost of efficiency due to increased computational overhead.

Overall, the experiments demonstrate that the max estimate method based on statistical sampling provides the best of both worlds, stability and fluctuations and can effectively estimate the working set size of a virtual machine, providing valuable insights into memory resource management. The choice of moving averages offers a balance between stability and adaptability, depending on the specific characteristics of the memory access patterns observed. Additionally, optimizing sampling parameters such as period and size can further enhance the accuracy and efficiency of the estimation process.