Indian Institute of Technology Bombay

Virtualization and Cloud Computing, 2024

Report On

Working Set Size: Estimation and Analysis

Author: Nitish Bhat (23M0750)

Contents

1 Introduction

Working set size (WSS) is the amount of memory a system is actively using for its computation. Estimation of WSS enables cloud providers to efficiently allocate memory based on their current use. This allows them to over-provision resources, make server consolidation and improve memory utilization.

In this project, we implement two different methods of WSS estimation on a simple KVM hypervisor running a guest with varying workloads. We contrast the benefits and drawbacks of each of these methods and compare their performance overhead. We also comprehensively analyse the effect of sampling parameters such as sample size and sample rate on performance and accuracy.

2 Methodology

We have considered two different methods of estimation WSS, one using access bit of the page table entry and another by invalidation page table entries and counting the page faults as described in below sections.

2.1 Access bit based estimation

Every page table entry in x86 architecture contains an access bit at 5th position from the least significant bit. The bit is set whenever that page is accessed (read or write). This can be used to estimate the active memory usage size. We periodically sample a uniform set of pages p and count the number of entries with an access bit set x and reset it back to zero. The working set is estimated as

$$WSS = \frac{x}{p}\left(\frac{\text{total\_memory}}{\text{page\_size}}\right)$$

Although this method is efficient, one has to keep in mind that Direct Memory Access bypasses setting access bit and can lead to inaccurate estimates in extreme scenarios where majority of memory usage is as a result of device IO.

2.2 PTE Invalidation-based estimation

In this method, we manually reset the PTE valid bit of a fixed sample of pages, periodically. Whenever these pages are accessed by the guest it causes a minor page fault. At this point, we resolve the page fault by obtaining the address from cr2 register and setting the valid bit while keeping track of the number of minor page faults. Then we estimate the working set size as

$$WSS = \frac{minorpg_faults}{sample_size} * \frac{total\_memory}{page\_size}$$

3 Design and Implementation Challenges

This section describes various design and implementation challenges and choices made in the project.

3.1 Realiable way of measuring performance

In order to check the overhead of different methods of WSS estimation, different sample rates, and sizes, we needed a reliable way to measure performance of guest. For this purpose, we introduced a counter in guest which was incremented regularly. The overflows of this counter per second was considered to be a performance metric for guest.

$$\text{overflows per second} = \frac{counter\_overflows}{total\_time}$$

3.2 Random Memory access in guest

One of the challenges faced was how to make the guest access memory randomly as the simple guest running without OS has no access to system calls and could not use rand() library function. This was solved by generating random numbers using a Linear Congruential Generator (LCG) in the guest and accessing memory locations based on it. LCG was calculated as

$$\text{random number } = (a \cdot \text{seed} + c) \mod m$$

$$seed = \text{random number}$$

and configured with *a =1103515245, c= 12345 and m =2147483648* so that it randomly generates 32 bit integers

3.3 Passing parameters to guest

While initially, we considered using hypercalls from the guest to get the parameters, we discarded it for mainly two reasons. Hypercalls would add additional overhead and would make guest more dependent on hypervisor than we wanted. So the parameters were passed by reserving a fixed location in guest and injecting parameters whenever we needed to change the guest behaviour. The guest would periodically read these parameters and change its memory access pattern.

3.4 Avoiding compiler optimizations in guest

Since the guest only read memory locations and its results were not used anywhere, the compiler would optimize away these instructions. In order to prevent this the guest computed the sum of all contents read and wrote it to a fixed dummy location.

Another problem of the compiler using register values to read parameters instead of memory locations was solved by marking these parameters as volatile.

4 Experimental Setup

4.1 KVM and sampling

The setup consists of a simple KVM hypervisor running a VM with guest program without an operating system. The VM is allocated a memory of 512 MB. A timer is configured to generate a signal at sampling intervals and interrupt the guest when the user space KVM computes the WSS based on sample size. The memory access parameters to the guest are passed by writing it to specific locations in the mapped memory. We considered different sampling intervals of 1, 2, and 5 seconds and sampling size of 100, 1000, 10000 pages per interval and plotted the results.

4.2 Workloads

Guest ran 4 different workloads *high, low, normal, and rapid*. The high workload consisted of guest VM accessing about 80 percent of its memory while the low workload accessed only 20 percent. The normal workload was designed to mimic real applications with consuming average memory but requiring high memory for certain short periods of time. The rapid workload was purely designed for testing the accuracy and changed access pattern every 3 seconds.

We also tested by varying percentages of random memory access of a workload namely 0,20,50 and 100 percent. At 0 percent, the guest accessed memory contiguously and at 100 percent every memory access was randomized.

4.3 System configuration

The experiment was carried out on a consumer system running 13th Gen Intel(R) Core(TM) i9-13900HX with 24 cores, maximum clock frequency of 5.4 GHz, equipped with 32 GB of RAM, and running Linux 22.02 LTS operating system.

5 Results and Discussion

This section describes the results obtained from the experiment with different configurations mentioned in the previous section and discusses the reasons behind it.

5.1 Estimation methods

We conducted the experiment measuring WSS with the access bit method and page invalidation methods for high and low workloads. The result is plotted in Figure **??**. The performance is negligibly affected by our estimation of WSS. Moreover at low memory usage, since there are not many page faults both, the access bit method and invalidation method perform similarly. However, the overhead of page faults can be seen with high memory usage.

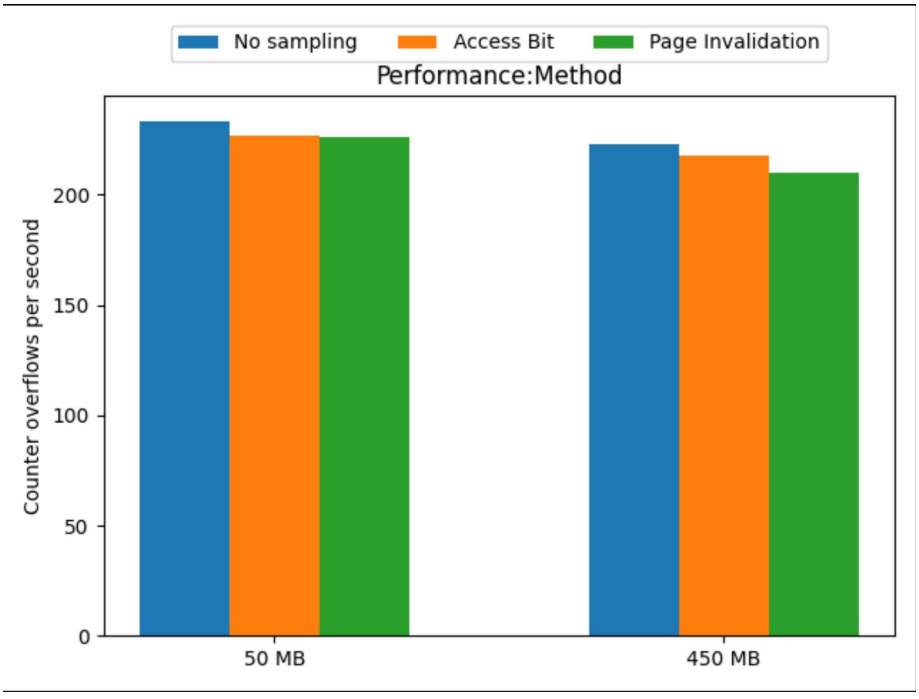

Figure 1: Performance of different estimation methods

5.2 Sample rate and Sample Size

The results for varying sample rate and sample sizes are plotted in Figure **??** and Figure **??** respectively. As expected the performance decreases with more frequent sampling as well as increased sample size.

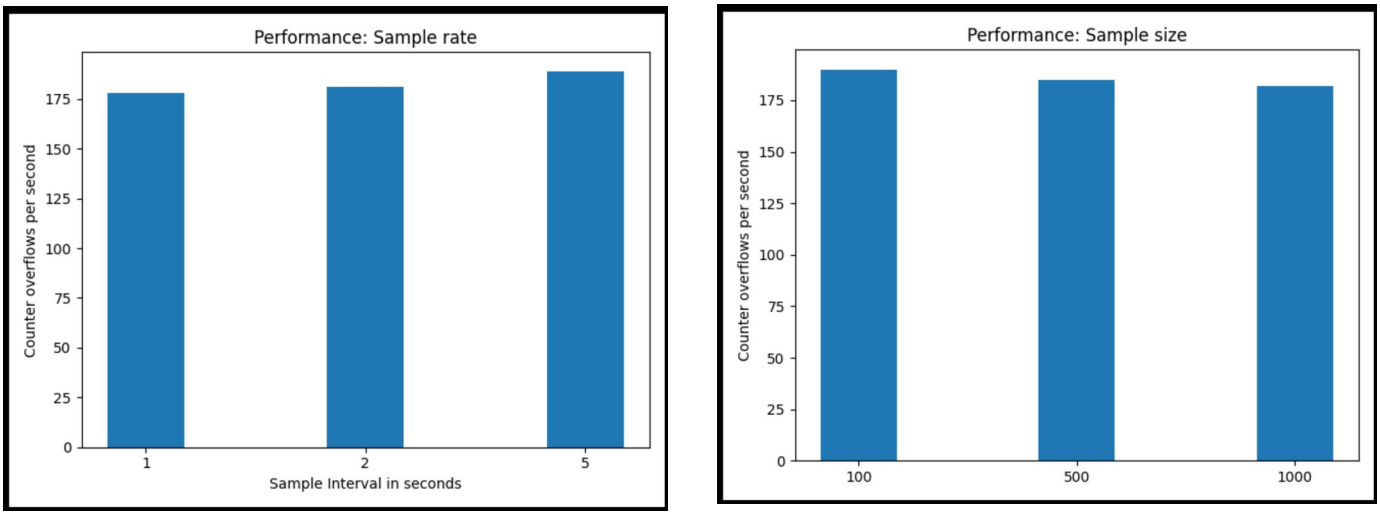

Figure 2: Performance vs Sample Size and Sample rate

5.3 Memory access pattern

We plotted graphs of estimated WSS varying sample sizes when the memory access is contiguous and when memory access is random in Figure?? and Figure ?? respectively. When memory is contiguously accessed small sample size is sufficient to determine WSS. But with random access increasing sampling size increases accuracy. We found a sample size of 1000 to be fairly accurate for our experiment.

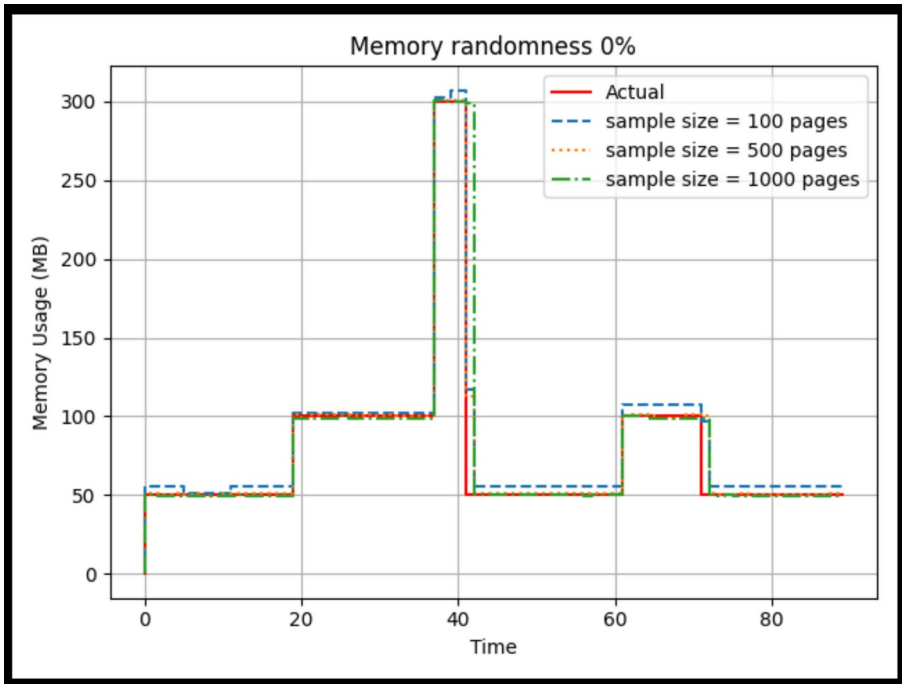

Figure 3: Different sample sizes for contiguous memory access

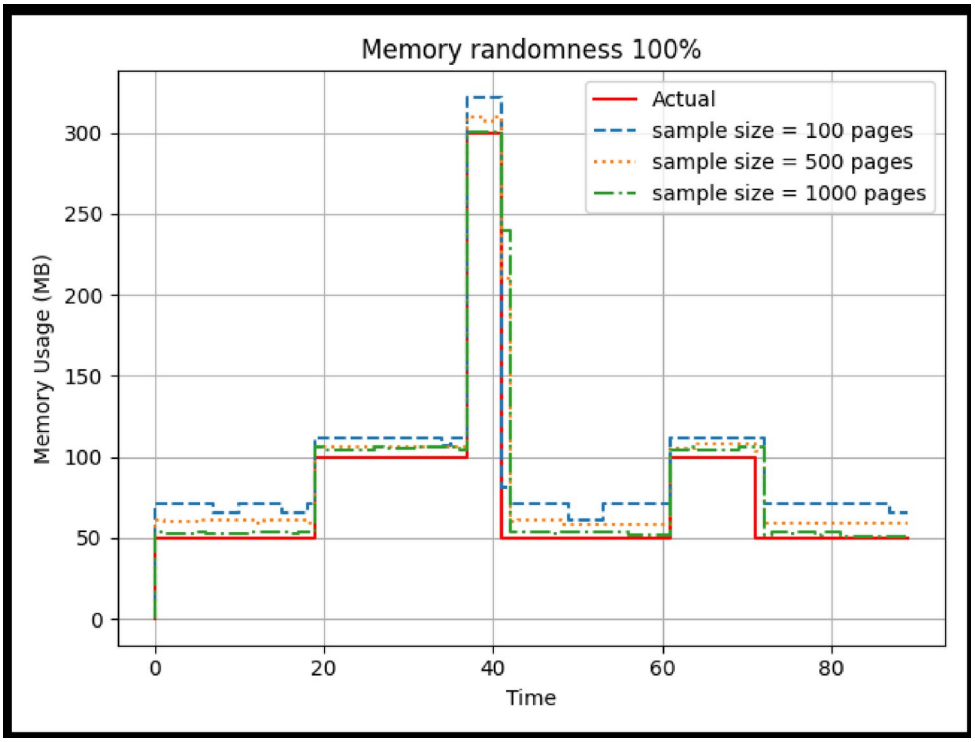

Figure 4: Different sample sizes for random memory access

We also plotted graphs of WSS varying sampling rate under normal memory access conditions and against a workload which changed its access pattern rapidly in Figure ?? and Figure ?? respectively. As expected slower sampling rate resulted in reduced accuracy with rapid varying workload.

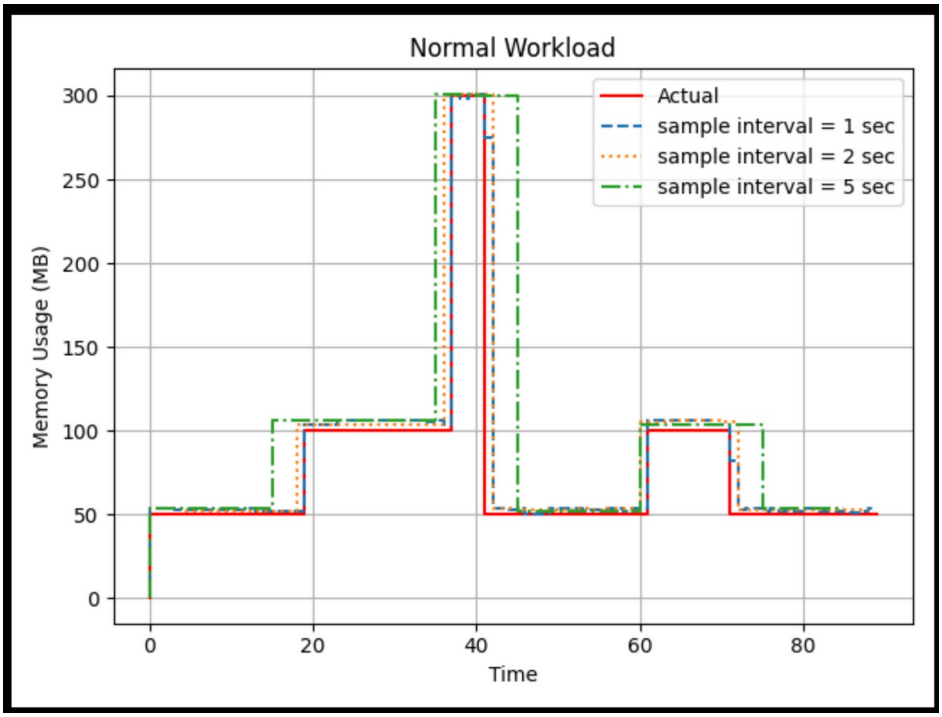

Figure 5: Different sample rate for normal memory access

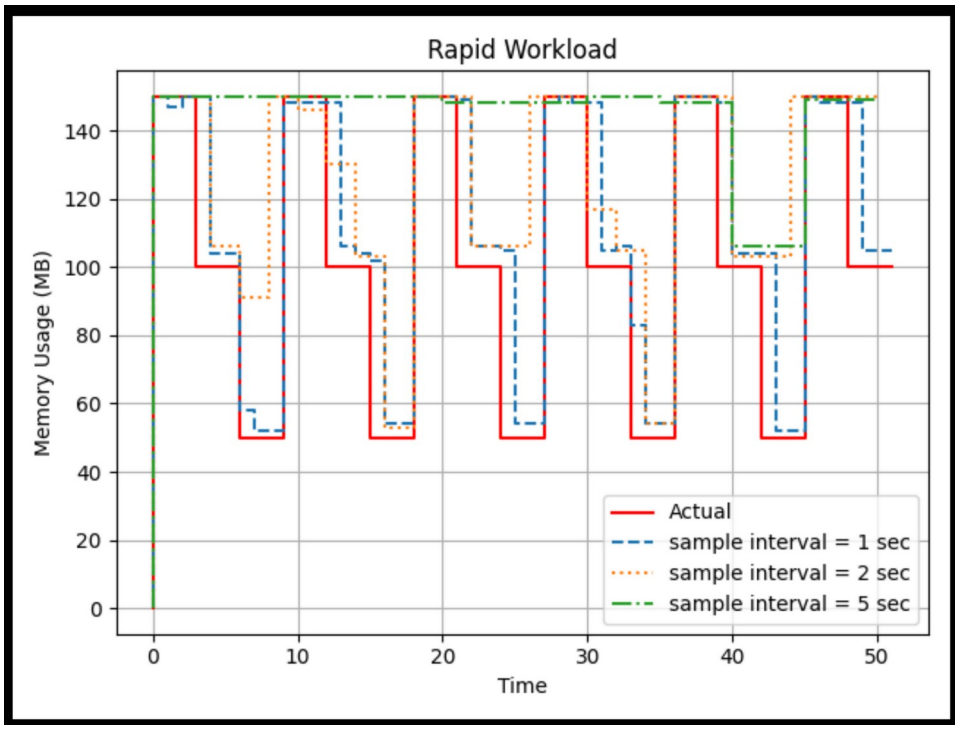

Figure 6: Different sample rate for rapid changing memory access

6 Conclusion

In this project, we have implemented a method for estimating working set size in a simple KVM setup and compared its performance, and accuracy with varying workloads and presented their analysis.