IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 30, 2019, accepted July 5, 2019, date of publication July 11, 2019, date of current version July 30, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2928221

More Accurate Estimation of Working Set Size in Virtual Machines

AHMED A. HARBY, SHERIF F. FAHMY, (Member, IEEE), AND AHMED F. AMIN
Computer Engineering Department, Arab Academy for Science, Technology and Maritime Transport, Cairo, Egypt
Corresponding author: Sherif F. Fahmy (fahmy@aast.edu)

ABSTRACT Accurate working set size estimation is important to increase the consolidation ratio of data centers and to improve the efficiency of live migration. Thus, it is important to come up with a technique that provides an accurate estimation of the working set size of virtual machines that can respond to changes in memory usage in real-time. In this paper, we study the problem of working set size estimation in virtual machines and come up with a method that allows us to better estimate the working set size of virtual machines in Linux. Toward that end, we design a finite state machine that can be used to accurately estimate the working set size and that is responsive to changes in workload. We then implement the algorithm on Linux using QEMU-KVM as our hypervisor. The system is tested using the sysbench benchmark for memory, CPU, and database workloads. The results indicate that our algorithm provides better results in terms of average working set size estimations and is competitive with existing techniques in terms of page faults.

INDEX TERMS WSS, virtual machines, memory management.

I. INTRODUCTION

Virtualization of computing resources is an important topic that is expected to become more relevant as we move towards an increase in the virtualization of everything – from the virtualization of standard machines in data centers [1] to the virtualization of networking functions in the form of NFVs (network function virtualization) [2] that can be used to compose networking components as required.

This paper covers an important topic related to virtualization. Specifically, it studies how we can more accurately estimate the size of the working set of virtualized machines. This is important for two reasons, namely

- To reduce memory pressure on host machine(s)
- To improve migration performance

We shall now elaborate on these items. First, by estimating the working set size accurately, a host operating system can reclaim memory from the guest OS and either use it for its own purposes or allocate it to other virtual machines, therefore increasing the consolidation ratio of virtual machines in data centers. This is important as it allows the data center to host more virtual machines and hence increases its efficiency and its profit margin.

In addition, there is currently a trend in the live migration research domain to only migrate hot pages and maintain cold pages on a distributed memory layer across the entire data center. The idea behind this is to bring down the time required for live migration by only migrating a subset of the memory footprint of the virtual machine [3]. In order to be able to do this, the host OS needs to have an accurate estimation of the size and identify of the hot pages – estimating the working set size is an important first step in order to achieve this.

Therefore, it is extremely important to design an algorithm that can accurately determine the size of the working set of a machine. If the algorithm over-estimates the size of the working set, it will not be possible to get the maximum benefit in terms of reclaiming memory and faster migration.

On the other hand, if the algorithm under-estimates the working set size, this will result in an increase in page faults and a consequent drop in performance as we pay the price of handling this increase. Thus, the algorithm needs to carefully track the size of the working set without erring too much in one direction or the other.

Also, it is important to note that the working set size changes as the mix of programs executing on an operating system changes. Even if the same set of applications is executing, the working set size will change based on the stage of execution of the programs. Thus, the developed algorithm needs to adapt quickly to changes in memory requirements in order to accommodate the changing needs of the system.

In this paper, we propose a new working set size estimation algorithm that allows us to accurately track the size of the

The associate editor coordinating the review of this manuscript and approving it for publication was Tae Hyoung Kim.

working set of each virtual machine running on a host by adaptively estimating the number of pages needed by the virtual machines and then reclaiming additional memory from the virtual machine using ballooning [4].

Our algorithm improves on the state of the art by attempting to more accurately estimate the size of the working set size and being less pessimistic when interpreting changes in the memory consumption pattern of virtual machines. This paper describes the algorithm and its implementation of a Linux system with a QEMU-KVM hypervisor.

The results of our research indicate that the proposed algorithm performs better in terms of estimating tighter working set sizes, at little to no effect on the number of incurred page faults. The rest of the paper is organized as follows. In section II, we review the literature. Section III presents the proposed solution, section IV contains some theoretical analysis, section V presents the experimental result and section VI discusses the overhead of the algorithm. We conclude the paper in section VII.

II. LITERATURE REVIEW

There are multiple papers that attempt to address the issue of determining the working set size of virtual machines. In this section of the paper, we discuss some of the methods present in the literature. The first method for determining the working set size of virtual machines is referred to as self-ballooning [5]. In this method, the host operating system assumes that the working set size of an application is equal to the virtual memory requests it makes.

Therefore, self-ballooning assumes that the value of *committed_AS*, a variable that Linux increments every time an application calls malloc and decrements every time memory is freed, is equal to the working set size of the virtual machine. The host then inflates or defaults a balloon driver [6], [7] to enforce this estimate of the working set size in the guest OS.

This technique uses values that Linux already provides, and so does not require modification to the codebase. However, it has two disadvantages that may seem contradictory at first. The first is that *committed_AS* only takes into account anonymous memory, and thus may underestimate the memory requirements of applications that have a large page cache – as would occur in disk I/O intensive applications.

The second is that the anonymous memory requested by an application, as reported by Linux in *committed_AS*, is typically much larger than the working set size of the application in terms of anonymous memory. Therefore, this method would overestimate the amount of anonymous memory needed.

In order to solve this problem, a solution, zballoond, was proposed in [8], [9]. This solution assumes that the working set size of a system is equal to the value of *committed_AS* initially, and then incrementally increases and decreases this estimate based on the number of observed page faults and refaults – thus, even if the algorithm starts with a value that is too small or too large, it incrementally approaches the true size of the working set of the system.

The algorithm resets the estimate of the working set size calculated so far back to *comitted_AS* any time the value of *commited_AS* changes. This occurs because zballoond interprets such a change as a complete change in the memory usage pattern of the system, and thus invalidates any calculations made so far. The algorithm that is presented in this paper is based on the core ideas of zballoond.

Another method used for estimating the working set size of virtual machines is employed by VMware [7]. The algorithm works by periodically invalidating a subset of the memory pages of a virtual machine. When any of these pages are swapped back in, this event is trapped in the hypervisor. The hypervisor then calculates the percentage of the invalidated pages that were swapped back in and uses this percentage as an indication of the system-wide percentage of allocated pages that are actually in the working set.

For example, if a guest OS has 1000 pages allocated, the algorithm may choose 100 pages as the sample to test. It would then invalidate these 100 pages, and count the number of pages from among those 100 that are swapped back into memory. If this value is, for example, 80 pages. It would conclude that 80% of the allocated pages are part of the working set – this amounts to about 800 pages in the example used here.

While the implementation of this technique is non-intrusive to the codebase of the guest OS, its implementation has an impact on the performance of the virtual machine depending on which pages are selected for invalidation. Page faults are expensive, and deliberately inducing them to calculate the percentage of pages that constitute the working set may be an expensive operation.

The second issue with this technique is, since it selects a sample of already allocated pages to invalidate, it cannot estimate a working set size larger than the currently allocated pages. In the best case, all the invalidated pages will be swapped back in and the hypervisor would calculate the percentage of allocated pages that constitute the working set size as 100%, thus setting the working set size to be equal to the total number of pages currently allocated – when in fact it is larger.

Another technique that can be used to calculate the working set size is Geiger, presented in [10]. This algorithm monitors the pages that are swapped out of memory, and then, if they are swapped back, calculates how much space would be needed to prevent their swapping out in the first place.

Unfortunately, the main problem with this technique is it only works if the initial memory allocated to the virtual machine is less than its working set size. If it were larger, there will be no swapping activity and the working set size will be maintained equal to the current memory allocation.

In [11], another method for computing the working set size is presented. In this technique, each virtual machine is only allocated a small amount of memory and the rest of the memory requirements of the guests are managed in the hypervisor as an *Exclusive Cache*. When the small amount of memory that is assigned to the guests is used up, the virtual

machine sends memory to the exclusive cache rather than to swap space. Thus, the hyervisor is fully aware of the memory requirements of all guests. Like Geiger [10], this algorithm only works if the current allocation is less than actual needs of the virtual machine.

Finally, in [12], a new approach that is a combination of both Geiger and the VMware technique, BADIS, is presented. The rationale is that each of these component algorithms only address one of the following issues

- Handle the case in which VM is wasting memory
- Handle the case in which VM needs memory

The VMware technique handles the first point very well, while the Geiger algorithm handles the second. Thus the authors combine them into one and called it BADIS.

It should be noted, that self-ballooning and zballoond handle both issues at the same time out of the box. While self-ballooning does not track the actual working set accurately since it relies on *committed_AS* only, zballoond uses *committed_AS* as a starting point and incrementally changes its estimate of the working set size based on the realtime behavior of the system. Thus, we asked ourselves if zballoond could be modified to produce even better results. This led to the following question:

Is it absolutely necessary for zballoond to reset the estimated value of the working set size each time committed_AS changes?

And the following follow-up question

If we do not reset the working set size every time the value of committed_AS changes, but rather incrementally respond to the changes, will there be a price to pay in terms of page faults?

In this paper, we attempt to answer the above two questions.

III. PROPOSED SOLUTION

The proposed solution is based on zballoond [8]. The idea is to smooth the spikes introduced into the estimation of the working set size by the algorithm proposed in [8], thus we propose a number of changes to the finite state machine presented in that work.

In order to present the proposed solution, we will first quickly review the algorithm proposed in [8]. The algorithm presented in [8] relies on the fact that if the physical memory assigned to a virtual machine is larger than its working set, the number of page faults and refaults should be near zero.

Therefore, the algorithm proposes incrementally decreasing the amount of memory available to a virtual machine by using ballooning. The rate at which the memory is decreased varies, with a fast state reducing the memory available at a fast rate and a slow state reducing the memory available at a lower rate. The proposed algorithm also has a mechanism for increasing the memory available – by deflating the balloon from the guest OS – when the number of page faults and refaults rises, this state is referred to as the CoolDown state in the paper.

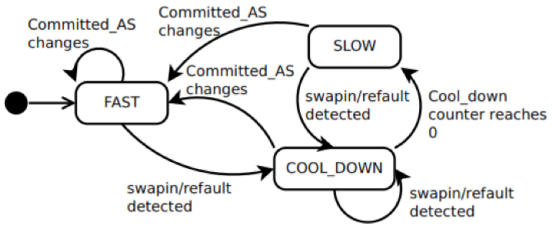

FIGURE 1. Zballooning FMS [8].

The algorithm described above should converge around the actual size of the working set of the virtual machine by increasing and decreasing the size of the balloon driver inside the guest OS based on the number of page faults that occur in the system. The initial size of the working set size is assumed to be *committed_AS*, which is exposed by the Linux *proc* filesystem. At the start of the algorithm, the size of the working set is decreased at a fast rate of 5% every one second if no page faults or refaults occur during that interval.

Whenever a page fault or refault occurs, it is taken as an indication that the working set size estimate is too small for the actual needs of the guest, therefore it is increased by the size of the page faults and refaults in order to go back to a value that accommodates the actual needs of the guest – this is the CoolDown state. The algorithm remains in this state until there are no more page faults or refaults. The number of page faults and refaults are checked every eight seconds while in this state.

The algorithm exists the above state when eight seconds pass without a page fault or refault occurring. However, since the algorithm has just left a state in which the working set size was too tight for the virtual machine, the algorithm does not go back to aggressively decreasing the size of the working set by 5%, instead, it decreases the size of the working set size by 1% for every one second during which no page faults or refaults occur.

If at any time the value of *committed_AS* changes, the algorithm assumes that the entire memory usage pattern of the system has changed and therefore resets the working set size back to *committed_AS*. The finite state diagram below depicts the algorithm.

It is this last part of the algorithm that we attempt to optimize in our proposed algorithm. Instead of resetting the working set size when *committed_AS* changes, we modify the working set size to reflect the change in *committed_AS*. After making this change, we go to either the fast state or the slow state based on the size of the increase in *committed_AS*. The idea is not to lose all the work done so far by resetting the working set size each time the value of *committed_AS* changes, but rather to respond incrementally to such changes. The diagram below depicts the finite state machine of our algorithm.

The idea of our proposed algorithm is to add another state to the finite state machine proposed in [8]. We refer to this state as the *Recovery* state, this state is entered when the *committed_AS* changes – specifically, when it increases.

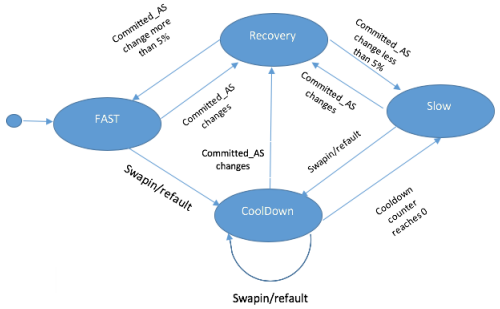

FIGURE 2. Our FMS.

Instead of resetting the estimate of the working set size and losing all the work done so far, the *Recovery* state simply adjusts the current working set size by the amount of change that occurred in the value of *committed_AS*.

Once this is done, the algorithm transitions to either the *fast* or *slow* state depending on the value of the change that occurred to *committed_AS*. If the value increases by more than 5%, the algorithm exits the *Recovery* state and goes to the *fast* state, otherwise it goes to the *slow* state.

If the value of *committed_AS* decreases, our algorithm does not decrease the estimated size of the working set since it is already as tight as possible, rather, it responds incrementally to the change in the state of the system, triggered by changes in the page fault behavior, in order to bring back the algorithm to a steady state. Note that the algorithm developed in [8] would reset the estimated working set size to the new *committed_AS* even if it decreases – further note that the value of *committed_AS* is expected to be larger than the working set size in most cases, so this would still usually result in an increase in the estimated size of the working set until the system stabilizes.

Using this modified finite state machine, we expect to minimize the number of spikes in the size of the estimated working set that would occur every time the finite state machine is reset to the start based on a change in *committed_AS*.

IV. THEORETICAL DISCUSSION

In this section of the paper, we attempt to make some claims about the properties of the proposed algorithm. The claims are a way for us to clarify our thought process regarding the algorithm. They are not a rigorous mathematical framework, but just a way for us to semi-formalize our thinking about the proposed algorithm. We will now present our claims regarding the algorithm.

Claim 1: The proposed algorithm is expected to produce a smaller average working set size, averaged over time, than zballoond.

Proof: Zballoond interprets each change in the value of *committed_AS* as a complete change in the memory consumption pattern of the system and therefore resets the estimated size of the working set. This means that the working set size will continuously spike when there is a change in *committed_AS*. This happens even if the value of *committed_AS* decreases, as it is expected that the estimated size of the working set will be smaller than this value.

On the other hand, the algorithm proposed in this paper does not interpret a change in *committed_AS* as a complete change in the memory consumption pattern of the entire system, but rather as an incremental change in the system – therefore, it attempts to incrementally change the working set size estimate calculated so far in order to accommodate this incremental change. Therefore, it will not always reset to the value of *committed_AS* thus retaining all the work done so far in estimating the working set size resulting in a smaller average working set size averaged over time. □

Corollary 1: A corollary of the above claim is that the standard deviation, calculated over time, of the size of the working set size of the proposed algorithm will also be smaller than of zballoond.

Claim 2: We claim that because the average estimated working set size of the proposed algorithm is smaller than other algorithms in the literature, the number of reported page faults should be slightly higher. We make no claim about how higher it is expected to be, or even that it will always be higher, but in general we expect the number of page faults to be higher due to the tighter range of values that the algorithm keeps for the working set size.

Proof: The algorithm presented in this paper incrementally responds to changes in memory consumption patterns of the system. Specifically, it does not reset the value of the estimated working set size each time the value of *committed_AS* changes. Therefore, it is expected to respond more slowly – and thus provide a tighter range for the working set size – to changing memory consumption patterns. This incremental response and tighter range of values for the working set size may mean that there is an increase in the number of page faults as the algorithm responds to changes in the memory consumption environment of the system. □

Corollary 2: A corollary of the above claim is that the standard deviation, calculated over time, of the number of page faults of the proposed algorithm should be larger than that reported by zballoond.

V. EXPERIMENTAL RESULTS

In order to determine if our algorithm functions as expected, we tested it on the following setup: The guest OS is a two core Intel(R) Atom CPU n270 @1.6GHz machine with 2GB of RAM and Ubuntu 16.04 32 bit installed. The host machine is an Intel i7 machine with 8GB of RAM running Ubuntu 16.04. We used QEMU-KVM version 3.1 as our hypervisor and managed the virtual machine using virt-manager version 1.3.2.

We tested the proposed algorithm and compare its result to the algorithm proposed in [8] using the *sysbench* [13], [14] benchmark. In order to ensure that we obtain consistent results across different workloads, we tested the CPU, Memory and MySQL suites of *sysbench* v1.0.17. We record the number of page faults that occurred during the experiment and the estimated size of the working set.

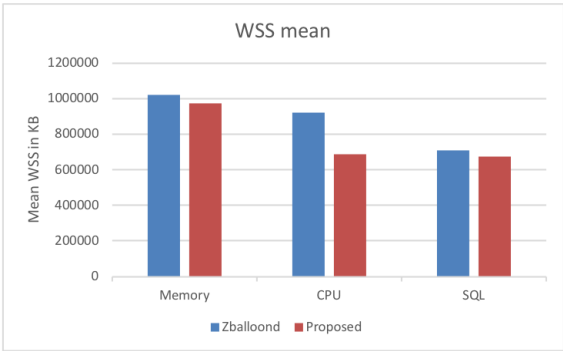

FIGURE 3. Working set size–average.

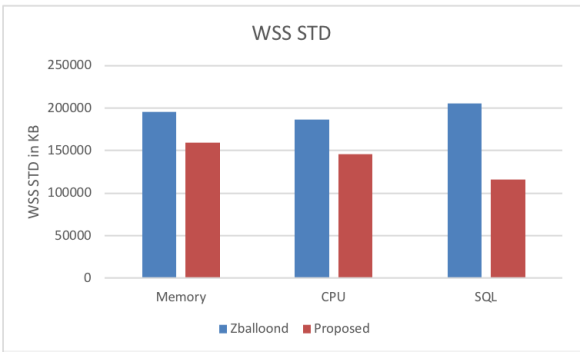

FIGURE 4. Working set size–standard deviation.

Each benchmark was run ten times and the values mentioned in the previous paragraph were collected. An important thing to note, is that to prevent continuous resets of the working set size by the algorithm described in [8], we only consider changes in *committed_AS* that are greater than 1%, reducing the threshold results in numerous resets in zballoond. The rest of this section contains a description of the results that we obtained.

A. SUMMARY STATISTICS

In this section of the paper, we present the mean and standard deviation of the number of page faults and the estimated working set size of both zballoond and the proposed algorithm. Recall that we run each benchmark ten times, thus for each algorithm we have thirty values for the number of page faults and the size of the working set, below are the charts that depict the result of this part of our work.

As can be seen, on average, the working set size estimated by the zballoond algorithm is larger than the one we propose in this paper. This is to be expected given the fact that we do not continuously reset the estimated working set size whenever there is a change in the value of *committed_AS* but attempt to incrementally change its value based on the change that occurred. This confirms claim 1.

This variability is also reflected in the standard deviation of the working set size of the two algorithms as depicted in Figure 4, and this confirms corollary 1. We now report the average and the standard deviation of the number of page faults reported by the two algorithms. The idea is to see if the tighter estimate of working set size exhibited by our algorithm will lead to a larger number of page faults. First, we present the average number of page faults of each algorithm.

As can be seen, there is one benchmark, the CPU benchmark, in which the proposed algorithm has a higher number of page faults than zballoond. But for the other two benchmarks, it has a lower number of page faults. In general, we believe that the tighter bound on working set size provided by our algorithm may result in a larger number of page faults as the system incrementally adjusts to a change in *committed_AS* as expressed in Claim 2.

But the results appear to show that this is not always the case. This may be due to the fact that changes in

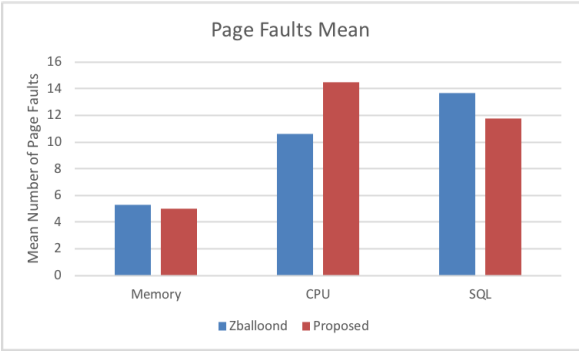

FIGURE 5. Page fault–average.

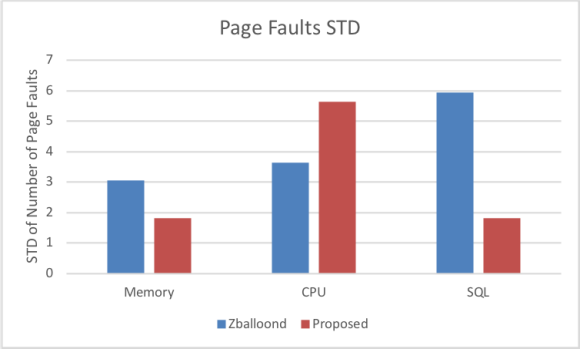

FIGURE 6. Page fault–standard deviation.

committed_AS may really be orthogonal to memory usage patterns in most cases. In any case, part of the results, the result for the CPU benchmark, confirm Claim 2, but the rest of the results appear to show that the proposed algorithm performs as well as, and sometimes better than, the algorithm proposed in [8].

We now present the standard deviation of the page faults of both algorithms to see whether ours exhibits greater variability.

The results follow the trend exposed by the average chart presented above, the CPU test presents the largest variability in the number of page faults in our algorithm. Similar to the discussion above pertaining to Claim 2, part of the results here appear to confirm corollary 2, while the rest of the results appear to show the proposed algorithm outperforming the algorithm described in [8].

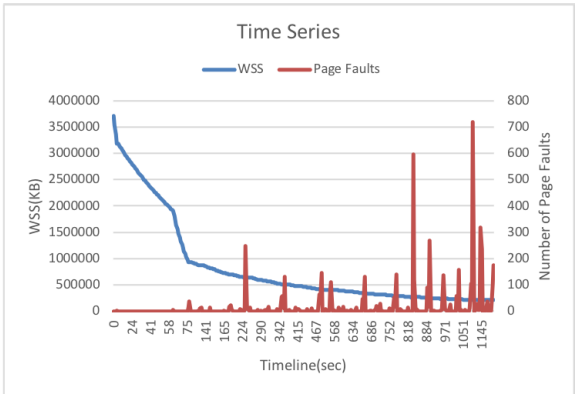

FIGURE 7. Proposed algorithm time series (CPU).

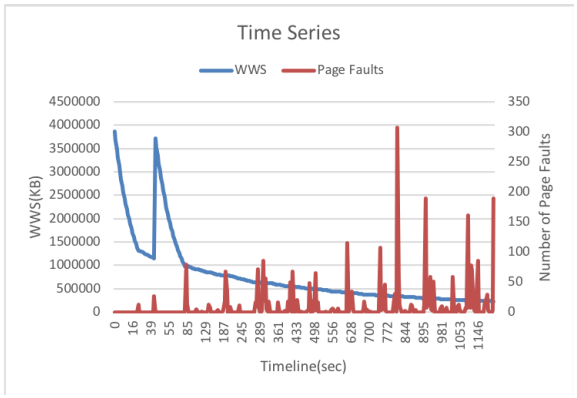

FIGURE 8. Zballoond time series (CPU).

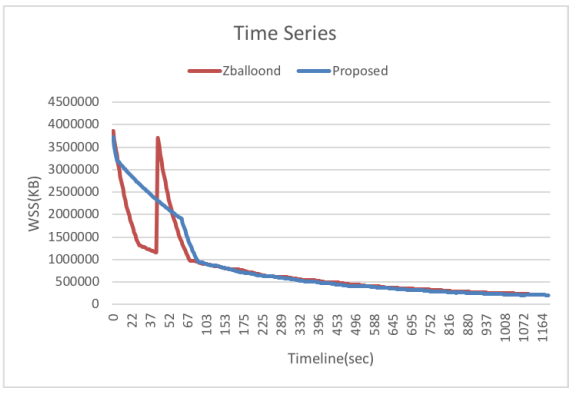

FIGURE 9. WSS combined (CPU).

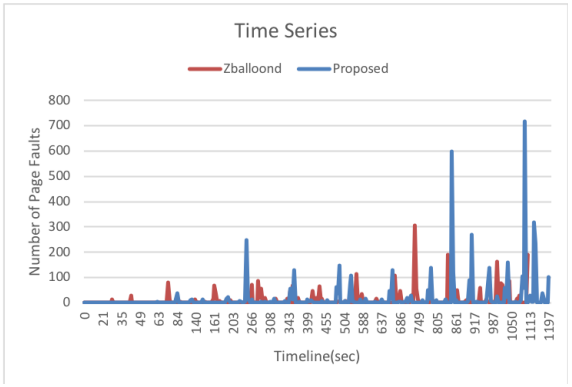

FIGURE 10. PF combined (CPU).

B. BEHAVIOR OF ALGORITHM OVER TIME

In this section of the paper, we discuss the behavior of the algorithm over time. In the last section, we summarized the behavior by calculating the average and standard deviation over time, in this section we provide the entire time series of the behavior of the algorithm in order to elucidate more information about the behavior of the algorithm.

As previously mentioned, we ran each benchmark ten times for a total of thirty results. To conserve space and to avoid unnecessary repetition, we present one representative result from each benchmark. The overall pattern of behavior of these algorithms is the same, so we do not find it necessary to include all results or to provide a time series of the average values at each point in time.

For each benchmark, we provide four time-series plots. The first two plots contain two time-series, one for the size of the working set, and the other for the number of page faults experienced over time for each of the algorithms individually. We then provide two plots that combine the time behavior of the working set size estimation and page faults of both algorithms on the same charts for comparison. We now present the result for the CPU benchmark.

Figure 7 depicts the results of the time series describing the behavior of the proposed algorithm, while figure 8 depicts the results of the time series describing the behavior of the algorithm in [8]. Figure 9 depicts the combined working set size estimation time series of both algorithms in the same plot, while figure 10 does the same for the number of page faults.

Note the spike in the working set size that occurs in figure 8, this corresponds to a period in time during which *committed_AS* has changed. Note also, that both algorithms exhibit higher page faults towards the end of the experiment, because at this time the size of the working set has become tight, and is driven to either increase or decrease based on the number of page faults – so it is expected that the number of page faults will increase at this time.

Another thing to notice, is that after the working set size is reset in the algorithm described in [8], the number of page faults abates for a bit. This occurs because the algorithm has over-estimated the effect of the change in *committed_AS* on the value of the working set size. Therefore, it resulted in a large increase in the value of the estimated working set size, this caused the number of page faults to drop. Notice that the page faults pick up again when the increased size of the working set is brought down again by the algorithm. Note also that in this particular case the value of *committed_AS* has actually decreased as can be seen in the fact that the spike that occurred with the reset is smaller in value than the value the algorithm started with.

The combined charts, figures 9 and 10, show the same trend described in the summary statistics section of the paper. Both algorithms converge to the same values of working set sizes at the end, but the proposed algorithm does this faster by

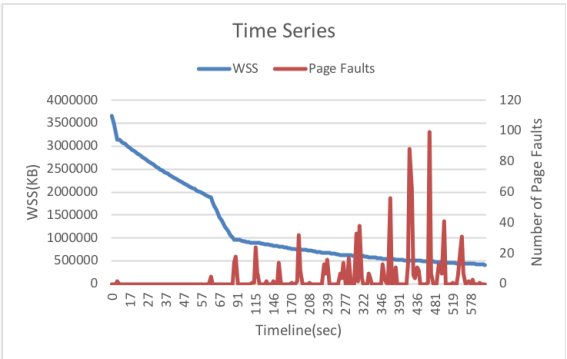

FIGURE 11. Proposed algorithm time series (memory).

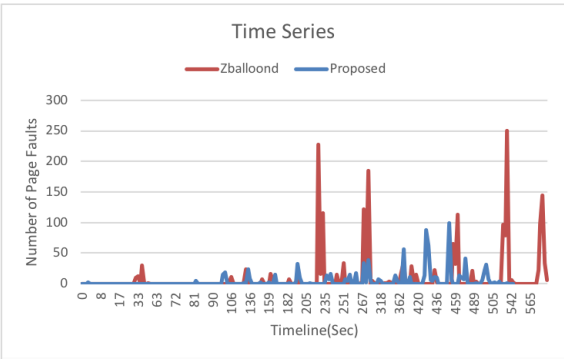

FIGURE 14. PF combined (memory).

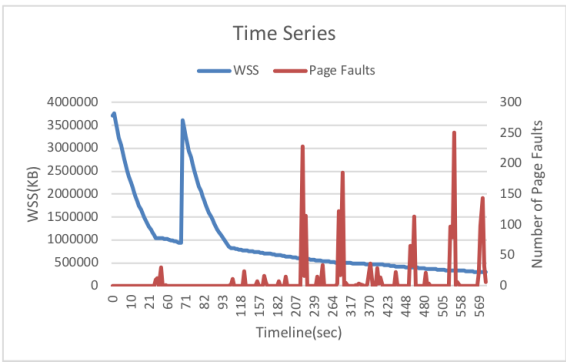

FIGURE 12. Zballoond time series (memory).

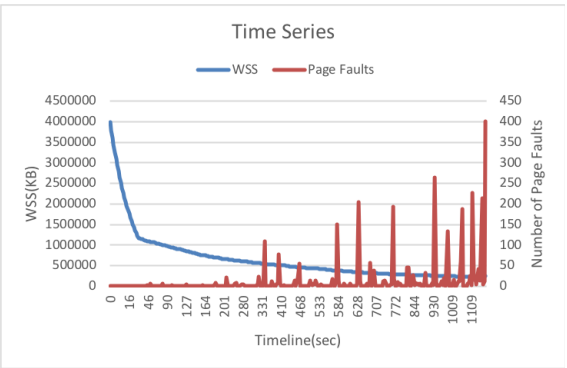

FIGURE 15. Proposed algorithm time series (SQL).

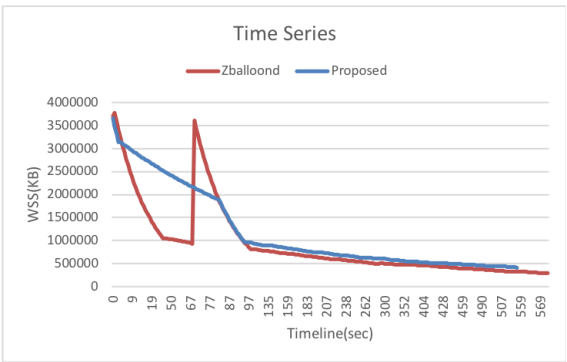

FIGURE 13. WSS combined (memory).

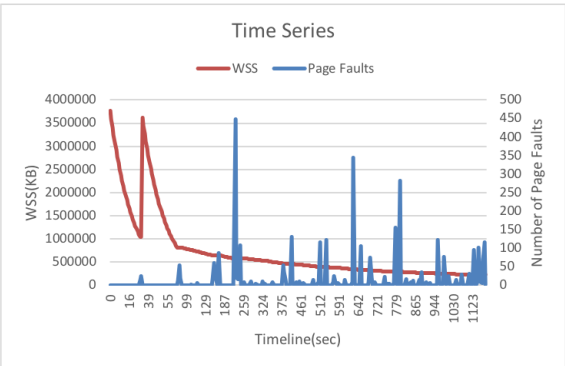

FIGURE 16. Zballoond time series (SQL).

not spiking every time the value of *committed_AS* changes. It also shows that, at least in this particular benchmark, the proposed algorithm has higher page faults. It should be noted, as explained earlier, the higher page faults are present only in the CPU benchmark, for the rest of the benchmarks the algorithms behavior similarly in terms of page faults – this can be seen in figures 14 and 18.

The results for the memory and SQL benchmarks are similar, we include them here for completeness, but they do not offer any further insight into the relative behaviors of the algorithms.

As can be seen, when the proposed algorithm meets a change in *committed_AS*, it interprets it as an incremental change in memory behavior and responds correspondingly, whereas the algorithm proposed in [8] considers this a complete change in memory behavior and resets the working set size. This is the main difference between the two algorithms, and is what causes their different behavior. We believe the incremental approach will provide tighter values for the estimate of the working set size – we also believed, as stated in section IV, that this will come at the expense of an increase in page faults, however the experimental results appear to show that the algorithm is competitive with that proposed in [8] with respect to the number of page faults.

VI. ALGORITHM OVERHEADS

In this section of the paper, we discuss the overheads of the proposed algorithm. The overheads of the algorithm can be

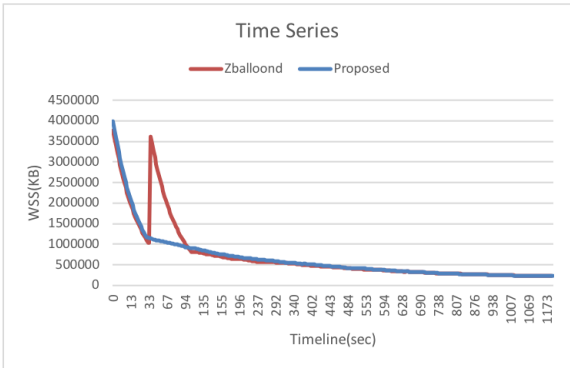

FIGURE 17. WSS combined (SQL).

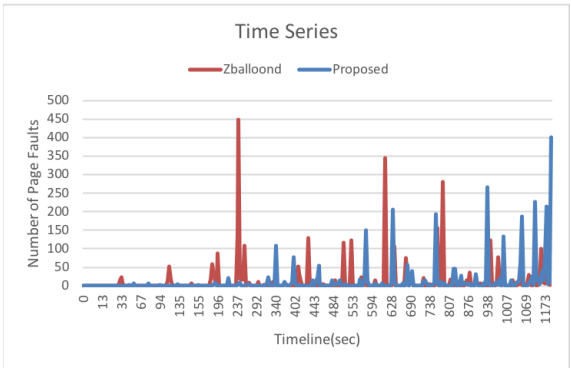

FIGURE 18. PF combined (SQL).

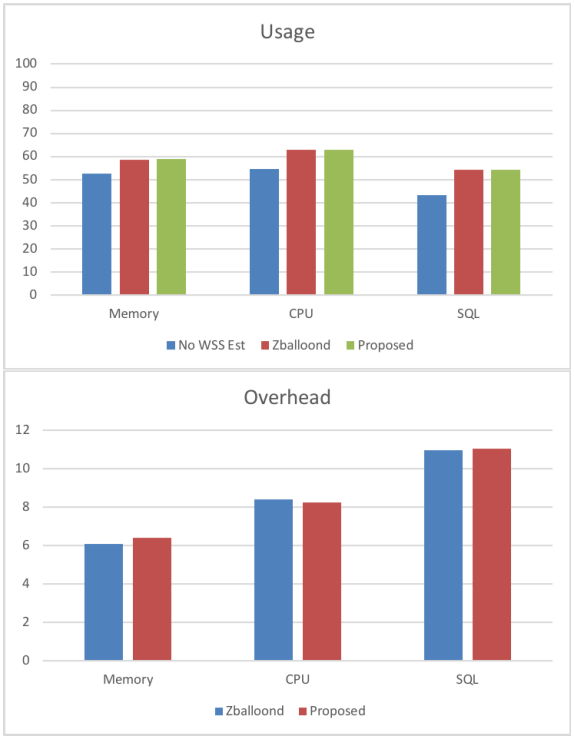

FIGURE 19. Overhead of algorithms.

divided into two categories, the first is the complexity of the code that decides on which states to transition to, and the second is the overhead associated with implementing the changes in the working set size using the balloon driver.

The transition code, the code that decides which state the algorithm should go to, is a series of if statements. Thus, the complexity of this code is *O(1)*. We, therefore, do not expect that the code that determines the transition paths in the algorithm will have significant overhead.

On the other hand, the part of the proposed system that enforces the estimated working set size on the guest OS is expected to have measurable overhead. The reason for this is twofold, first, the estimation of the working set size occurs on the guest OSes, and this needs to be communicated to the host OS since the host OS is the entity that triggers the balloon driver. Second, the balloon driver itself will introduce overhead while allocating memory on the guest for reclamation.

Naturally, the values of these overheads will depend on the method used to communicate between the host and guest OS – for example, using shared memory to communicate between the host and guest would be very fast. However, this method would be disruptive in terms of modifications to the code base to allow guest and host OSes to see the same memory locations. Therefore, we decided to communicate between the guest and host OS using socket programming. This causes minimum disruption to the code base, maintains memory isolation and leaves the balloon driver unmodified, but is expected to be more time consuming. Please note that the complexity of this communication is still *O(1)*, but the constant term is not negligible.

A final overhead that needs to be considered is that which occurs due to the increase in page faults as the working set size shrinks and the algorithm becomes driven by page fault events. As the working set size becomes tighter, more page faults will occur and thus more CPU time will be consumed to handle them.

To measure the effect of all these overheads combined, we measured the CPU consumed when the proposed method and zballoond are run and compare them to the CPU consumed when no working set size estimation algorithm is used using the three benchmarks mentioned in section V. Figure 19 depicts the results of this experiment.

Figure 19 contains two subplots, the top subplot depicts the total CPU usage for the proposed algorithm, zballoond, and the no working set estimation algorithm (No WSS Est) case. As can be seen, the proposed algorithm and zballoond behave similarly, they both have higher overheads than the baseline case.

It should also be noted that they differ most from the baseline case in the CPU and SQL benchmarks. A quick revisit of section V will show that these are the two benchmarks in which the most number of page faults occur – thus, a large part of this CPU overhead is probably attributable to the page fault management code in the OS.

The second subplot further clarifies this by depicting the difference between the working set size estimation algorithms' CPU usage and the baseline's – i.e., the CPU usage of the baseline is subtracted from the CPU usage of each of

the algorithms and the result is plotted in this graph. As can be seen, in the best case, the algorithms consume 6% more CPU than the baseline, and at worst 11% more. It should also be noted that the proposed algorithm has less than a single percentage point more overhead than zballoond – they are virtually identical in terms of overhead.

VII. CONCLUSION AND FUTURE WORK

In this paper, we designed an algorithm that attempts to more accurately estimate the size of the working set of virtual machines. The algorithm does this by incrementally decreasing the estimated size of the working set size according to a finite state machine, and enforcing this working set size by inflating and deflating a balloon device in the guest OS. We attempt to improve on existing techniques by not resetting the working set size every time the memory consumption of the machine changes.

Our results indicate that the algorithm provides a tighter bound on the working set size while not suffering any significant performance penalty in terms of number of page faults incurred. The overhead of the proposed algorithm is also very similar to the state of the art. To summarize:

- The proposed algorithm provides better estimates of working set size
- The proposed algorithm has similar behavior in terms of page faults as the state of the art
- The proposed algorithm has similar overheads as the state of the art

In the future, we propose to build on this algorithm in order to minimize the downtime of live migration of virtual machines by only migrating the hot pages in the working set size and offloading cold pages to a system-wide virtual memory system (e.g., memcached [15]), as well as studying the effect of the algorithm on the consolidation ratios in data centers. We will also attempt to improve the algorithm to further optimize its behavior. We would also like to study in greater detail the behavior of page faults under tighter estimates of working set size.

REFERENCES

[1] Geeta and S. Prakash, "Role of virtualization techniques in cloud computing environment," in *Advances in Computer Communication and Computational Sciences*, S. K. Bhatia, S. Tiwari, K. K. Mishra, and M. C. Trivedi, Eds. Singapore: Springer, 2019, pp. 439–450.

[2] E. Jacob, J. Astorga, J. J. Unzilla, M. Huarte, D. García, and L. N. López-De-Lacalle, "Towards a 5G compliant and flexible connected manufacturing facility," *Dyna, Spain*, vol. 93, no. 6, pp. 656–662, 2018.

[3] U. Deshpande, D. Chan, T.-Y. Guh, J. Edouard, K. Gopalan, and N. Bila, "Agile live migration of virtual machines," in *Proc. IEEE 30th Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 1061–1070.

[4] J. H. Schopp, K. Fraser, and M. J. Silbermann, "Resizing memory with balloons and hotplug," in *Proc. Linux Symp.*, 2006, pp. 313–319.

[5] *Xen 3.3 Feature: Memory Overcommit—Xen Project*. Accessed: May 18, 2019. [Online]. Available: https://xenproject.org/2008/08/27/xen-33-feature-memory-overcommit

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. 19th ACM Symp. Oper. Syst. Principles (SOSP)*. New York, NY, USA: ACM, 2003, p. 164. [Online]. Available: http://portal.acm.org/citation.cfm?doid=945445.945462

[7] C. A. Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, 2002.

[8] J.-H. Chiang, H.-L. Li, and T.-C. Chiueh, "Working set-based physical memory ballooning," in *Proc. 10th Int. Conf. Autom. Comput. (ICAC)*, 2013, pp. 95–99.

[9] J.-H. Chiang, T.-C. Chiueh, and H.-L. Li, "Memory reclamation and compression using accurate working set size estimation," in *Proc. IEEE 8th Int. Conf. Cloud Comput. (CLOUD)*, Jun./Jul. 2015, pp. 187–194.

[10] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the buffer cache in a virtual machine environment," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 5, pp. 14–24, Oct. 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1168857.1168861

[11] L. Pin and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache*," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2007, pp. 29–43.

[12] V. Nitu, A. Kocharyan, H. Yaya, A. Tchana, D. Hagimont, and H. Astsatryan, "Working set size estimation techniques in virtualized environments: One size does not fit all," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 46, no. 1, pp. 62–63, 2018.

[13] *Ubuntu Manpage: SysBench—A Modular, Cross-Platform and Multi-Threaded Benchmark Tool*. Accessed: Apr. 26, 2019. [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html

[14] *How to Benchmark Your System (CPU, File IO, MySQL) with Sysbench*. Accessed: May 25, 2019. [Online]. Available: https://www.howtoforge.com/how-to-benchmark-your-system-cpu-file-io-mysql-with-sysbench

[15] V. Chidambaram and D. Ramamurthi, "Performance analysis of memcached," Univ. Wisconsin-Madison, Madison, WI, USA, Tech. Rep., 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.409.411

AHMED A. HARBY was born in Egypt, in 1988. He received the B.Sc. degree from the Computer Engineering Department, Arab Academy for Science, and Technology and Maritime Transport (AAST), Egypt, in 2015, where he is currently a Teaching Assistant. He has been with the AAST for four years. His main areas of research interest include virtualization and operating systems.

SHERIF F. FAHMY was born in Egypt, in 1981. He received the B.Sc. degree and the M.Sc. degree from the Computer Engineering Department, Arab Academy for Science, and Technology and Maritime Transport, Cairo, Egypt, in 2002 and 2005, respectively, and the Ph.D. degree from Virginia Tech., USA, in 2010. He is currently the Department Chair of the Computer Engineering Department, Arab Academy for Science, and Technology and Maritime Transport. He has been in academia for 17 years. His main areas of research interest include concurrency control mechanisms, virtualization, and distributed systems.

AHMED F. AMIN received the Ph.D. degree in electronics and communications from Cairo University, in 1981, and the Ph.D. degree in computer engineering from NPS, CA, USA, in 1987. He was the Head of the Department of Engineering, Air Force Academy, Egypt, from 1987 to 1989. From 1989 to 1993, he was a faculty member in Saudi Arabia. Since 1993, he has been with Arab Academy for Science, and Technology and Maritime Transport. During this time, he was the Head of the Department of Computer Engineering with the Qatar and Egypt branches of the AAST, until 2015. His research interests include microprocessor design, operating systems, VHDL design, and embedded systems.