Working Set-based Physical Memory Ballooning

Jui-Hao Chiang
Stony Brook University

Han-Lin Li and Tzi-cker Chiueh
Industrial Technology Research Institute

Abstract

Minimizing the total amount of physical memory consumption of a set of virtual machines (VM) running on a physical machine is the key to improving a hypervisor's *consolidation ratio*, which is defined as the maximum number of VMs that can run on a server without any performance degradation. To give each VM just enough physical memory equal to its true working set (TWS), we propose a *TWS-based memory ballooning* mechanism that takes away all unneeded physical memory from a VM without affecting its performance. Compared with a state-of-the-art commercial hypervisor, this working set-based memory virtualization technique is able to produce noticeably more effective reduction in physical memory consumption under the same input workloads, and thus represent promising additions to the repertoire of hypervisor-level optimization technologies.

1 Introduction

Memory virtualization enables the hypervisor to allocate to each running VM just enough physical memory without performance degradation (memory ballooning) and consolidate physical memory pages with identical contents across VMs (memory deduplication [6, 10, 18, 16]). These optimization techniques make the best of the available physical memory on a virtualized server and maximize the number of VMs that could run on it, or the *consolidation ratio*. Because memory deduplication is an important technique used in both commercial and open-source hypervisors [21, 8] and has been extensively dealt with in a separate paper [13], this paper focuses only on memory ballooning.

When a VM is started, the amount of physical memory that the hypervisor gives to the VM is equivalent to that specified in its configuration file. However, in most cases VMs do not use up all the given memory because VMs tend to be provisioned conservatively. By definition, the amount of physical memory that a VM needs at any point in time is its working set size at that instant. Therefore, if there exists a way to accurately estimate a VM's working set size, the hypervisor could leverage this estimate to take away unneeded memory pages from the VM using the memory ballooning mechanism [21, 8, 20].

This paper describes the design, implementation and evaluation of an intelligent memory ballooning algorithm based on the working set size information of running VMs. To derive the working set size of a given VM, we exploit the page reclamation mechanism built into the guest OS by iteratively decreasing the VM's physical memory allocation until it starts swapping in pages. When we say a VM's current working set size is X, we meant the size of the memory pages the VM is going to access in the next observation window is X. In our design, the observation window is set to 1 second.

2 Working Set Estimation

The physical memory given to a VM on a virtualized server at the start-up time forms the VM's *guest physical address space*, which is mapped to the server's *machine physical address space* through a mapping table, the Extended Page Table (EPT) in the case of the X86 architecture. The working set of a VM is defined as the set of memory pages in the guest physical address space that are being actively used by the VM in the recent past [21]. If a VM's working set is a proper subset of the VM's guest physical address space, some physical memory pages allocated to the VM could be safely reclaimed. Even when a VM's exact working set is not available, being able to estimate the working set's size is still useful.

A naive way to determine a VM's working set is to intercept memory accesses made by the VM, for example, marking a VM's memory pages as not-present in the EPT so as to trap and record the number of accesses to each of its pages. The working set of a VM is the set of memory pages that have been accessed at least once in the observation window. However, this scheme is infeasible because the overhead of trapping every memory read/write is simply too prohibitive to be acceptable in practice. To get around this problem, VMware's ESX used a sampling approach to estimating the working set size of a VM. Periodically it marks a randomly sampled subset of the VM's guest physical pages as invalid, counts the number of pages in the subset that are accessed whenever a protection fault against any of these pages occurs, and uses the resulting count to infer the VM's working set size.

Another way to estimate a VM's working set size, used by the *self-ballooning* mechanism [15] in the Xen hypervisor, is to directly use the `Committed_AS` statistic maintained by the Linux kernel, which corresponds to the total number of *anonymous* memory pages consumed by all processes on a VM. For page reclamation, Linux maintains two LRU (Least Recently Used) lists, *Active* and *Inactive*, for each of the following two types of memory pages: (1) *Anonymous Memory*, which corresponds to the heaps and stacks of user processes, and (2) *Page Cache*, which corresponds to the kernel's memory to buffer and cache the payloads of disk reads and writes.

Utilizing the hardware reference bit, Linux puts pages that are accessed more frequently into Active list and leave pages that are accessed less frequently in Inactive list. The page reclamation mechanism traverses the Inactive list to free its pages and possibly re-allocate them. If a reclaimed page belongs to anonymous memory, the kernel marks the page's page table entry as non-present, and swaps out the page's content to the swap disk. When the page is later accessed, a *swapin* event occurs and it is swapped in. If a reclaimed page belongs to page cache, the kernel flushes its content to disk if it has been dirtied. If the page is later accessed, a *refault* event occurs and it is brought back in.

When a VM's physical memory allocation is larger than or equal to its working set size, the number of swapin and refault events should be close to zero. This observation inspires the third way to estimate a VM's working set size: Gradually decreasing the balloon target of the balloon driver in the VM until the VM's swapin and refault counts start to become non-zero. The amount of physical memory allocated to the VM at that instant is the VM's working set size. More concretely, a 3-state finite state machine, as shown in Figure 1, is used to adaptively track a VM's working set size (WSS). Anytime the WSS changes, we adjust the VM's balloon target accordingly. The finite-state machine starts in the *FAST* state and initializes the VM's WSS to the VM's `Committed_AS`. While in the *FAST* state, the finite-state machine iteratively lowers the VM's WSS by 5% of the

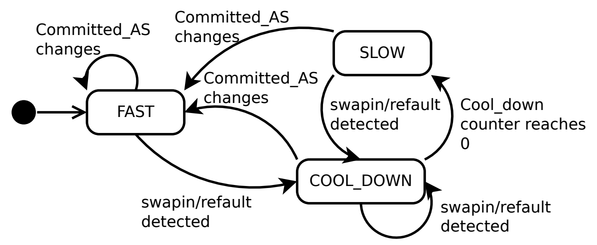

Figure 1: *The finite-state machine used to track a VM's working set size.*

current `Committed_AS` value at the end of every epoch (epoch size set to 1 second currently), until swapin or refault events occur within the current epoch, which suggests the finite-state machine may have overshot the WSS adjustment. As soon as swapin/refault events arise in an epoch, the finite-state machine raises the VM's current WSS estimate by the sum of the observed swapin and refault event counts, and enters the *COOL_DOWN* state, regardless of whether the finite-state machine was originally in the *FAST*, *COOL_DOWN* or *SLOW* state.

While in the *COOL_DOWN* state, the finite-state machine initializes a cool-down counter to a default timeout value (currently set at 8 seconds) and waits for it to expire, and resets the cool-down counter to the same default value if additional swapin/refault events arise. In the *SLOW* state, the finite-state-machine applies the same logic as in *FAST* state except that the VM's WSS is iteratively lowered by 1% of the current `Committed_AS` value in each epoch. Whenever the tracked VM's `Committed_AS` changes, the finite-state machine considers the VM's working set size has changed significantly, and resets itself by entering the *FAST* state and re-initializing the VM's WSS to the new `Committed_AS`.

3 TWS-based Memory Ballooning

Memory ballooning [21, 8] is a technique that reclaims physical memory from a VM by installing inside the VM a balloon driver that allocates memory pages from the VM's kernel via the standard APIs, pins them down, and returns them to the hypervisor. The balloon target of a balloon driver is the difference between the VM's configured memory requirement and the amount of memory it allocates from the VM.

How to correctly set a VM's balloon target is an important issue. When a balloon driver allocates more than the host VM's free memory pool, the VM OS's page reclamation mechanism is triggered to evict cold pages. The upper bound on a VM's balloon target is the VM's configured memory requirement, and the lower bound is the VM's minimum memory requirement that prevents Out-of-Memory exceptions. The optimal way to set a VM's balloon target is to set it to the VM's working set size, because this allows the hypervisor to reclaim the maximum amount of physical memory from a VM while reducing the performance impact on the VM to the minimum.

The self-ballooning mechanism in the Xen hypervisor sets a Linux VM's balloon target to its current `Committed_AS` value. This approach guarantees that applications consuming anonymous memory not suffer from any swap-in delay because all their stacks and heaps are likely to be memory-resident. However, compared with the working set-based approach to setting the balloon target, this approach has two deficiencies. First, `Committed_AS` does not factor the page cache into a VM's physical memory demand, and thus may cause substantial performance degradation for applications with intensive disk I/O activities, which could significantly benefit from the page cache. In contrast, the working set approach keeps a counter for refault events, and incorporates this counter into the calculation of a VM's working set size and thus balloon target. Second, `Committed_AS` captures only the pages that are allocated but not those that are actually used recently. More specifically, `Committed_AS` is incremented upon the first access to each newly allocated anonymous memory page and is decremented only when the owner process explicitly frees the page. For example, if a program allocates and accesses a memory page only once when the program starts but leaves it untouched until the program exits, the Linux kernel cannot exclude this cold page from a VM's `Committed_AS` even though it is clearly outside the VM's working set. In contrast, the working set approach actively forces the VM OS to invoke its page reclamation mechanism to pinpoint and evict cold pages.

4 Performance Evaluation

In this paper, we report the results of a performance evaluation study of TWS-based memory ballooning. The test machine used in this study contains an Intel Core i7 quad-core processor with VT and EPT enabled and 16 GB physical memory, and runs Xen-4.1 with 64-bit vanilla Linux 3.2.6 as the *Dom0* kernel. All the VMs in this study are configured with 1 virtual CPU and 2GB memory, and run Linux 3.2.6 64-bit kernel with the our developed *zballoond* kernel module for memory ballooning. *Zballoond* is a kernel thread that wakes up every second to collect relevant information, such as *Committed_AS*, swapin count and refault count, and make adjustments to the balloon target.

To verify the effectiveness of these TWS-based ballooning algorithm, we first compared it with self-ballooning mechanism in the Xen hypervisor. Then we compared it with the latest VMware ESXi 5.0 server.[1]

Benchmark Used	**TWS Ballooning**		**Self Ballooning**	
	Degradation	**Target**	**Degradation**	**Target**
SPECweb	0%	263.3MB	0%	263.3MB
SPECcpu	3.08%	783.6MB	4.11%	922.6MB
OLTP	3.31%	350.8MB	17.99%	328.8MB

Table 1: *Comparison between TWS-based ballooning and self ballooning in terms of performance degradation and balloon target for the three benchmarks, SPECweb Banking, SPEC CPU 401 and OTLP. The performance degradation is calculated based on a comparison with the performance of the same VM that is configured with 2GB memory.*

In this comparison, we used two identical test machines where one runs the Xen hypervisor with the TWS-based memory virtualization optimizations and the other runs the ESXi server. The memory given to each VM does not include anything owned by the hypervisor.

4.1 Effectiveness of TWS-based Ballooning

We evaluate the effectiveness of TWS-based ballooning by comparing the performance degradation and balloon target of a VM running a set of benchmark programs when TWS-based ballooning is used with those when Xen's self-ballooning is used. The balloon target of a VM is the amount of physical memory that a memory ballooning scheme allocates to the VM. The performance degradation of a memory ballooning scheme is the performance difference between a benchmark program running in a VM whose physical memory allocation is controlled by the ballooning scheme in question and the same benchmark program running in a VM that is configured with and indeed given 2GB memory, or the *Baseline* configuration. The following three benchmark programs are used: *SPECweb Banking* [3] running against Apache [1], *SPEC CPU*, and *OLTP* from the Sysbench suite [4] running against MySQL [2].

Table 1 shows the performance degradation and balloon target comparison between TWS-based ballooning and self-ballooning for the three benchmark programs. The memory requirement of SPECweb Banking benchmark is smaller than the minimum physical memory allocation to the test VM, 263.3MB. As a result, both TWS-based ballooning and self-ballooning produce the same balloon target, which is the same as the minimum physical memory allocation, and the benchmark program does not experience any performance degradation under TWS-based ballooning and under self-ballooning, when compared with the Baseline configuration. For the SPEC CPU 401 benchmark, the average balloon target of TWS-based ballooning is 15.07% (783.6MB vs. 922.6MB)

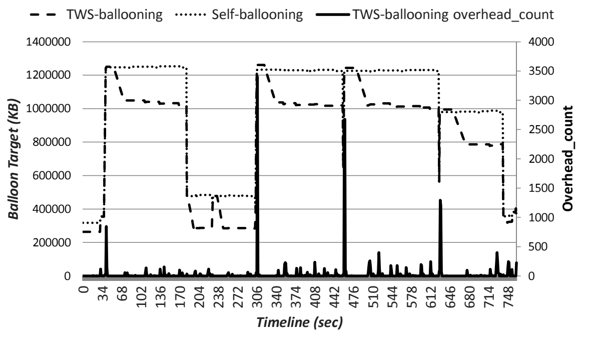

Figure 2: *The balloon targets produced by TWS-based ballooning and self-ballooning over time, and the resulting combined swapin and refault count over time under TWS-based ballooning, when the SPEC CPU 401 benchmark is used as the test workload.*

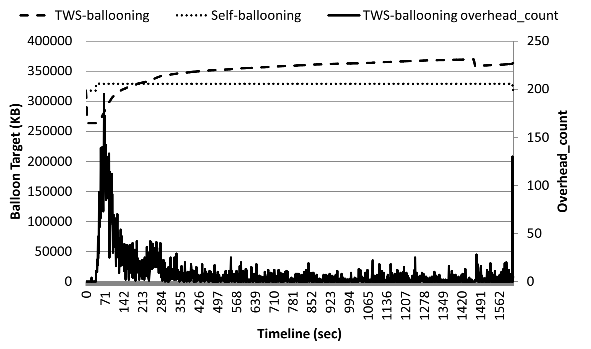

Figure 3: *The balloon targets produced by TWS-based ballooning and self-ballooning over time, and the resulting combined swapin and refault count over time under TWS-based ballooning, when the Sysbench OLTP benchmark is used as the test workload.*

smaller than that of self-ballooning, and yet the performance degradation of TWS-based ballooning is smaller than that of self-ballooning (3.08% vs. 4.11%).

The superiority of TWS-based ballooning comes from the fact that the working set size it produces effectively removes pages that are allocated but unused, as shown by the gap between the two balloon target curves in Figure 2. However, despite allocating a smaller amount of physical memory to the test VM, the performance degradation of TWS-based ballooning is smaller than self-ballooning, because it reacts faster to the sudden change in the VM's demand, e.g. at time points 320 seconds, 460 seconds, and 630 seconds of Figure 2. During these transitions, TWS-based ballooning is able to allocate more physical memory than Committed_AS, and thus cuts down unnecessary swapin and refault events.

Because the OLTP benchmark performs intensive disk I/O accesses and thus requires a larger page cache, Committed_AS is not an accurate estimate of the benchmark's working set as it does not take into account page cache. As a result, the average balloon target produced by TWS-based ballooning is 6.70% higher than self-ballooning, and justifiably so, because the performance degradation of TWS-based ballooning is only 3.31%, which is significantly smaller than that of self-ballooning, or 17.99%. As shown in Figure 3, TWS-based ballooning detects refault events and increases the test VM's balloon target accordingly, and as a result produces a balloon target that is more in line with the VM's working set size and more capable of reducing the performance overhead of memory ballooning to the minimum.

We also run two VMs, one with a constant working set size of 300MB and the other with a constant working set size of 1200MB, on the Xen hypervisor with TWS-based ballooning and on VMware's ESXi 5.0. Each VM is configured with 2 GB memory but given only 263.3MB at the start-up time. After these two VMs start to run, it takes TWS-based ballooning 10 seconds to reach the ideal physical memory allocation, i.e., giving 300MB to the 300MB VM and giving 1200MB to the 1200MB VM. However, for the same set-up, it takes VMware ESXi 136 seconds to reach the same ideal physical memory allocation. The reason that VMware ESXi takes longer to accomplish the same is because it uses a sampling approach to probe a VM's working set size.

5 Related Work

Standard operating systems estimate the active portion of buffer cache or page cache by maintaining LRU-like statistics [19, 12, 5] to implement page replacement logic. Lu et al. [14] proposed to allocate a small portion of memory to each VM while leaving the remaining memory as an exclusive cache is managed by the hypervisor. Thus, the memory accesses of VMs can be intercepted within the exclusive cache, and the LRU miss ratio curve [5] is derived to measure the working set size. Zhao et al. [24, 23] track the memory access of VMs by changing the user/supervisor privilege bit of guest page table entries to supervisor mode so that all memory access of VM will be trapped because the VM runs in user mode. Similarly, the LRU miss ratio curve is also derived for working set size prediction.

To reduce the overhead from trapping memory access, the VMware ESX server [21] uses sampling based mechanism to predict the working set size of VMs. To perform the sampling, the ESX server randomly chooses a few hundreds memory pages periodically, e.g., the default setting is to choose 100 pages per 60-second for each VM. However, this mechanism only gives a rough estimation of the VM working set size, and it can not reflect the working set size exceeding the current allocated memory.

When it comes to reclamation mechanism, the Clock algorithm [9] is commonly used in guest OSs and several research efforts [17, 22, 7, 11] aimed to estimate the working set size by monitoring the changes of access bit on the hardware page table. This approach requires modifications to the guest OS. In contrast, our approach leverages the guest OS's page reclamation mechanism and does not require any guest OS modifications.

6 Conclusion

Making efficient utilization of the physical memory available on a virtualized server is a key technical challenge for modern hypervisors. Possible solutions include *memory de-duplication*, which allows different VMs to share common pages, and *memory ballooning*, which reclaims unused pages from a VM when its physical memory allocation is larger than its working set size. This paper describes and evaluates techniques that exploit the knowledge of each VM's working set to deliver more efficient memory ballooning. More concretely, the specific research contributions of this work are

- A low-overhead active probing mechanism that could accurately sense the working set of each VM and track it dynamically,
- An intelligent memory ballooning algorithm that could detect allocated but unused pages and reclaim them, and

Compared with VMware's ESXi, which is a state-of-the-art hypervisor, the proposed working set estimation scheme is more accurate and more responsive to working set changes, but incurs a slight probing overhead, the proposed memory ballooning algorithm is able to quickly reclaim more memory pages without incurring additional performance penalty.

References

[1] Apache http server project. http://httpd.apache.org/.

[2] Mysql: open source database server. http://www.mysql.com/.

[3] Specweb2009. http://www.spec.org/web2009/.

[4] Sysbench: a system performance benchmark. http://sysbench.sourceforge.net/.

[5] ALMÁSI, G., CAŞCAVAL, C., AND PADUA, D. A. Calculating stack distances efficiently. MSP '02, ACM, pp. 37–43.

[6] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. *Increasing memory density by using KSM.* Linux Symposium, 2009, pp. 19–28.

[7] BANSAL, S., AND MODHA, D. S. Car: Clock with adaptive replacement. FAST '04, USENIX Association, pp. 187–200.

[8] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. *Xen and the art of virtualization*, vol. 37. ACM, 2003, pp. 164–177.

[9] CORBATO, F. J. A paging experiment with the multics system. In *In Honor of P.M* (1969), Morse, MIT Press, pp. 217–228.

[10] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. OSDI '08.

[11] JIANG, S., CHEN, F., AND ZHANG, X. Clock-pro: an effective improvement of the clock replacement. ATEC '05, USENIX Association, pp. 35–35.

[12] JIANG, S., AND ZHANG, X. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. SIGMETRICS '02, ACM, pp. 31–42.

[13] JUI-HAO CHIANG, HAN-LIN LI, T.-C. C. Introspection-based memory de-duplication and migration. VEE '13.

[14] LU, P., AND SHEN, K. Virtual machine memory access tracing with hypervisor exclusive cache. USENIX ATC'07, USENIX Association, pp. 3:1–3:15.

[15] MAGENHEIMER, D. Add self-ballooning to balloon driver. discussion on xen development mailing list and personal communication, april 2008.

[16] MAGENHEIMER, D. *Transcendent Memory on Xen.* XenSummit, February 2009, p. 3.

[17] MAUERER, W. *Professional Linux Kernel Architecture.* Wrox Press Ltd., Birmingham, UK, UK, 2008.

[18] MURRAY, D. G., H, S., AND FETTERMAN, M. A. Satori: Enlightened page sharing. ATEC '09.

[19] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. The lru-k page replacement algorithm for database disk buffering. *SIGMOD Rec. 22*, 2 (June 1993), 297–306.

[20] SCHOPP, J. H., FRASER, K., AND SILBERMANN, M. J. Resizing memory with balloons and hotplug. *Linux Symposium 2* (2006), 313319.

[21] WALDSPURGER, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36* (December 2002), 181–194.

[22] ZHANG, I., GARTHWAITE, A., BASKAKOV, Y., AND BARR, K. C. Fast restore of checkpointed memory using working set estimation. *SIGPLAN Not. 46*, 7 (Mar. 2011), 87–98.

[23] ZHAO, W., JIN, X., WANG, Z., WANG, X., LUO, Y., AND LI, X. Low cost working set size tracking. USENIX ATC'11, USENIX Association, pp. 17–17.

[24] ZHAO, W., AND WANG, Z. Dynamic memory balancing for virtual machines. In *VEE '09* (2009).

Notes

[1]VMware ESXi 5.0.0 build-623860.