IEICE Electronics Express, Vol.8, No.10, 748–754

Hardware assisted dynamic memory balancing in virtual machines

Changwoo Min[1,2a)], **Inhyuk Kim**[1b)], **Taehyoung Kim**[1c)], **and Young Ik Eom**[1d)]

[1] *School of Information & Communication Engineering, Sungkyunkwan University*
300 Cheoncheon-Dong, Jangan-Gu, Suwon, Gyeonggi-Do 440–746, Korea

[2] *Samsung Electronics, 416, Maetan-3Dong, Yeongtong-Gu, Suwon, Gyeonggi-Do, 443–742, Korea*

a) *multics69@ece.skku.ac.kr*
b) *kkojiband@ece.skku.ac.kr*
c) *kim15m@ece.skku.ac.kr*
d) *yieom@ece.skku.ac.kr*

Abstract: Virtualization technology can reduce the total cost of ownership by sharing resources with respect to the resource demand of each guest. Therefore, an efficient resource sharing mechanism is important for a virtual machine monitor (VMM). We introduce a hardware assisted dynamic memory balancing mechanism that balances memory among guests. Our proposed scheme estimates memory demand for each guest and periodically re-balances memory allocation based on this estimation. In order to estimate working set size (WSS), we use least recently used (LRU) histogram as a prediction model. Construction of LRU histogram is performed in a guest transparent way by using hardware memory management unit (MMU) virtualization support. Our experiments show that the proposed scheme accurately estimates the WSS with low overhead. They also show that it substantially improves performance over static memory allocation.

Keywords: virtualization, working set, nested paging
Classification: Science and engineering for electronics

References

[1] C. A. Waldspurger, "Memory resource management in VMware ESX server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, 2002.
[2] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: monitoring the buffer cache in a virtual machine environment," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 14–24, 2006.
[3] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache," *Proc. 2007 USENIX Annual Technical Conference*, Berkeley, CA, USA, pp. 1–15, 2007.
[4] W. Zhao, Z. Wang, and Y. Luo, "Dynamic Memory Balancing for Virtual Machines," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 37–47, 2009.

DOI: 10.1587/elex.8.748
Received March 29, 2011
Accepted April 18, 2011
Published May 25, 2011

[5] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM System Journal*, vol. 9, no. 2, pp. 456–789, 78–117, 1970.
[6] SPEC CPU2000. [Online] http://www.spec.org/cpu2000/
[7] SPEC CPU2006. [Online] http://www.spec.org/cpu2006/

1 Introduction

Efficient resource management is a key success factor in virtualization. However, allocating the proper amount of memory on demand is still challenging. Even though static memory allocation among guests is an easy solution, it can become a consolidation bottleneck.

In an effort to dynamically allocate memory in virtual machines (VMs), recent studies [1, 2, 3, 4] have introduced various approaches. However these studies still have limitations. Statistical sampling [1] and Geiger [2] cannot induce growth or shrinkage of WSS. Hypervisor exclusive cache [3] cannot be used for an OS without source code, since it requires modification of guest kernel. Zhao et al. [4] proposed an LRU histogram [5] based approach, however it has two major limitations. First, it suffers from performance overhead in large WSS, because monitoring overhead increases linearly with the number of pages. Second, the time needed to decide the allocation size grows non-linearly according to the number of VMs, thereby deteriorating the effectiveness of memory balancing. Since information needed to grow WSS is not incorporated into the LRU histogram, it needs to iteratively monitor the guest swapping frequency and the number of page fault to determine an allocation size larger than its current allocation.

We introduce a novel approach to dynamically balance memory among guests by using hardware MMU virtualization support. We use LRU histogram to estimate WSS for each guest; ballooning [1] is then used to reclaim memory from guests. Our contributions, compared to prior work, are as follows: First, it is applicable to commercial off-the-shelf OSs whose source code is not available, since our system does not require modification of guest kernel. This is achieved by using the nested page faults supported by H/W MMU virtualization in order to construct LRU histogram. Second, we present two techniques that efficiently build LRU histogram. *Weighted red-black tree* is proposed to calculate the stack distance in the worst case logarithmic time and *adaptive hot list resizing* is introduced to control monitoring overhead. Finally, we use *over allocation* technique to estimate WSS beyond the current memory allocation. In a virtualized system, it is difficult to estimate WSS larger than its current memory allocation, since LRU histogram is constructed using physical memory access. To estimate WSS beyond its current allocation, our system allows a guest to allocate more than the allocation size. The over-allocated memory is marked as VMM-level swappable; the access information is used to estimate WSS beyond its current allocation. Since LRU histogram contains the information needed for our system to shrink and

DOI: 10.1587/elex.8.748
Received March 29, 2011
Accepted April 18, 2011
Published May 25, 2011

grow WSS, the allocation size can be drawn directly from the LRU histogram in all cases.

2 Hardware assisted dynamic memory balancing

Our dynamic memory balancing mechanism consists of three parts: (a) monitoring memory access and constructing an LRU histogram for each guest, (b) periodically determining the memory allocation size and resizing the number of unmonitored pages to control the overhead, (c) enforcing the new allocation sizes by reclaiming the memory from the guests.

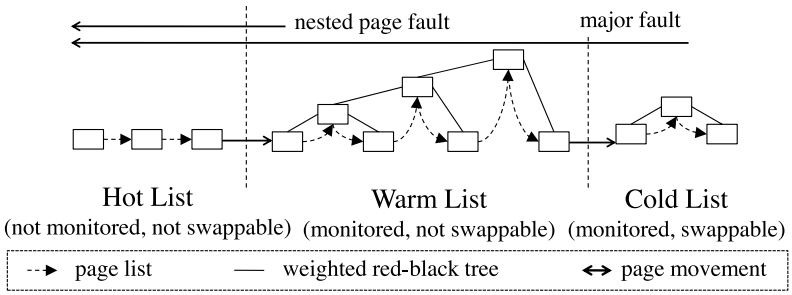


Fig. 1. Page list structure of a guest.

2.1 Memory access monitoring

Our system maintains a page list for each guest and intercept memory access. To monitor the memory access of a guest, we turn off the presence bits in the nested page table to trap the memory access in a guest transparently. The trapped page's stack distance is calculated and the corresponding LRU histogram is incremented by one. The page is moved to the head of the page list to maintain the order in the page list from the most to the least recently accessed.

To minimize performance overhead, we design the page list structure as illustrated in Fig. 1. Our system manages the page lists using three groups: *hot list*, *warm list*, and *cold list*. To reduce the monitoring overhead, the pages in the hot list are not monitored by turning on the presence bits. Hot list size is dynamically changed to keep a balance between the monitoring overhead and the accuracy of LRU histogram. On the contrary to the others, cold pages are the over-allocated pages marked as VMM-level swappable. When a VMM is under memory pressure, these pages will be selected as victims. The histogram data of the cold list is used to capture WSS larger than its current memory allocation. We will explain how to determine the size of each list in Section 2.2.

When updating the LRU histogram, the most expensive operation is calculating the stack distance. Because naive linear search takes the worst-case linear time, it is inefficient and not scalable. For an efficient calculation, we propose a *weighted red-black tree*. It is a kind of red-black tree whose node is annotated by weight. *Weight* represents the total number of nodes in its

DOI: 10.1587/elex.8.748
Received March 29, 2011
Accepted April 18, 2011
Published May 25, 2011

subtree. Since the nodes in a tree are ordered, the weight of left child is the number of predecessor of a node and the weight of right child is the number of successor of a node. For a node n, let P_s denote the set of ancestors whose successors include n. The distance of n is the sum of predecessors of P_s and size of P_s. It takes the worst case logarithmic time.

2.2 Dynamic memory balancing

Our system periodically re-calculates the proper memory allocation size of each guest for balancing. The time interval for this process is six seconds.

Let A_i denote the size of the memory allocation for i-th guest, and B_i represent the memory size for the i-th guest to allow allocating. By the definition of each page list described in the previous subsection, $A_i = H_i + W_i$ and $B_i = A_i + C_i$, where H_i, W_i and C_i are the sizes of hot, warm and cold list respectively. We define the WSS as the smallest memory allocation that yields a page miss rate no larger than δ. WSS of the i-th guest, WSS_i, can be taken from LRU histogram. For QoS purposes, our system guarantees the allocation of a minimum memory size M for each guest. The rest of the memory is allocated proportionally by each guest's WSS. Since the LRU histogram in our system also has memory access information that is larger than the current allocation, we can calculate the memory allocation size directly from the LRU histogram even in case of WSS growth. A_i is determined by:

$$A_i = \frac{T - M \times N}{\sum_{j=1}^{N} WSS_j} \times WSS_i + M \tag{1}$$

where T is the total memory size designated for the VMM and N is the number of running guests. In all of our experiments, we set T to 768 MB and M to 128 MB. The over-allocation size, C_i is calculated from B_i by definition. C_i needs to be large enough to cope with WSS growth and small enough to unload the idle memory by ballooning. We define $B_i = max(\beta \times BS_i, A_i)$, where BS_i is the smallest memory size that yields a page miss ratio no larger than τ, and β is the ratio of the extra margin. In our experiments, a small τ is effective in getting rid of outliers in the histogram. We set τ to 2% for cutting off the outliers of histogram and set β to 120% for monitoring the WSS growth.

To control monitoring overhead while maintaining a reasonable histogram accuracy, we determine the size of hot list, H_i to be able to monitor a sufficient number of nested page faults for an interval. To reduce the overhead efficiently, H_i needs to be adapted quickly according to the workload transition. For a fast adaption, we estimate H_i directly from the LRU histogram. We define E_i^t as the estimated hot list size at the t-th interval for the i-th guest to monitor F number of page accesses. It is determined from the LRU histogram, L by:

$$E_i^t = \min\left(\max_{e \in E}(e), A_i\right) \tag{2}$$

$$E = \left\{ e \middle| \sum_{j=e}^{T} L_j \times \frac{F^{t-1}}{M^{t-1}} \geq F \right\} \tag{3}$$

DOI: 10.1587/elex.8.748
Received March 29, 2011
Accepted April 18, 2011
Published May 25, 2011

where F^{t-1} is the number of monitored page accesses and M^{t-1} is the sum of the histogram entries for warm and cold list for the previous interval. If the estimation from the histogram gives a new value (i.e. $E_i^t \neq E_i^{t-1}$), we select E_i^t as H_i. Otherwise, H_i is adjusted adaptively by $\gamma \times H_{i-1}$ considering if F^{t-1} is larger than F. In our experiments, F is set to 32,768 pages and γ is set to 5%.

2.3 Reclaiming guest memory

After the B_i for each guest is determined, our system asks guests to change their memory usage to B_i by using ballooning. The reclaimed pages are reported back to the VMM, and they are then assigned to the other guests whose allocation sizes have increased.

3 Evaluation

Our system is implemented as an extension of KVM, an open source virtual machine monitor, and uses Intel EPT for the hardware MMU virtualization. Experiments are performed on a PC with a 2.67 GHz Intel i5 processor and 4 GB of physical memory. We use 12 benchmarks from SPEC CINT 2000 [6] and 11 benchmarks from SPEC CINT 2006 [7] on Ubuntu 10.04.

3.1 System overhead

Fig. 2 (a) shows the normalized performance and the effectiveness of the proposed optimization techniques. Without any optimization, the performance overhead is fairly large. Using the weighted red-black tree helps performance significantly, especially where WSS is large; the normalized performance improves from 25% to 50%. Adaptive hot list resizing further reduces the overhead, especially in workloads that heavily access memory; the performance improves very closely to an unmonitored case, 98.38%. Even in the worst case, *vortex*, the performance is reasonable, 96.24%. In Fig. 2 (b), we break down the performance overhead. Around 40% of overhead comes from the histogram update regardless of the workloads.

Although the performance overhead of the Zhao et al. method [4] was at up to 24% in SPEC CINT 2000, the performance overhead of our system was 1.76% in total and 3.76% at maximum. It shows that our optimization techniques effectively control the overhead regardless of the workloads.

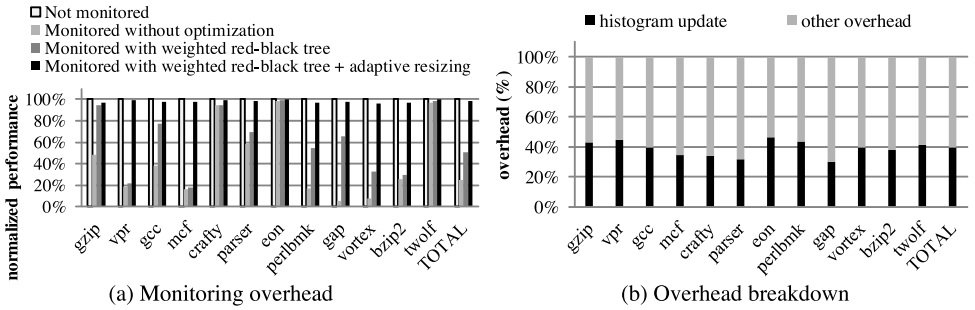
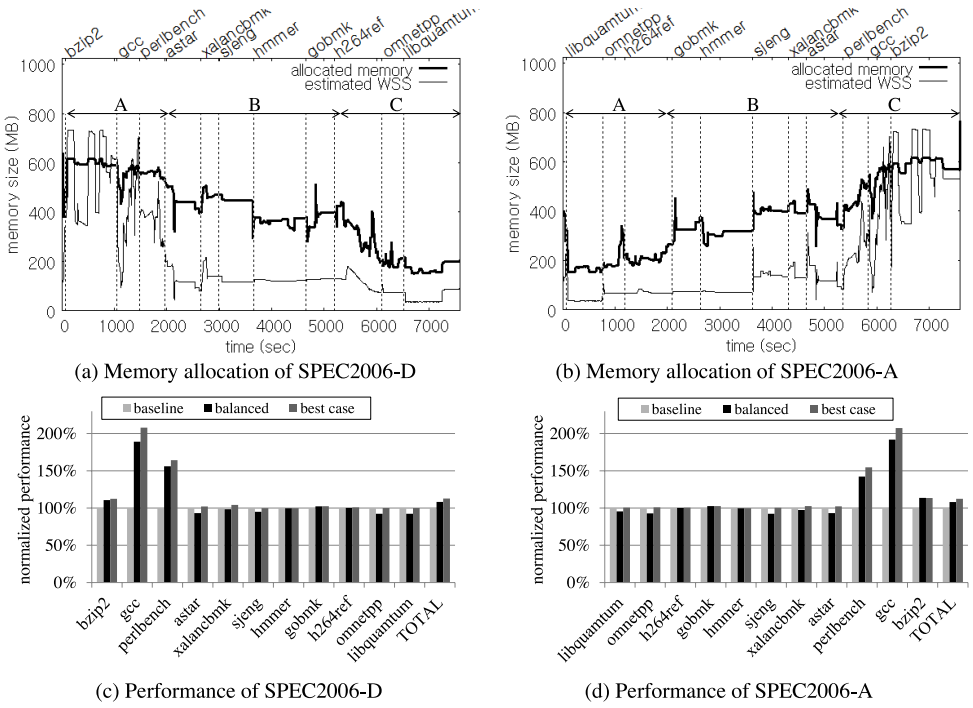


Fig. 2. Monitoring overhead of SPEC2000.

DOI: 10.1587/elex.8.748
Received March 29, 2011
Accepted April 18, 2011
Published May 25, 2011



(a) Memory allocation of SPEC2006-D

(b) Memory allocation of SPEC2006-A

(c) Performance of SPEC2006-D

(d) Performance of SPEC2006-A

Fig. 3. SPEC2006-D + SPEC2006-A.

3.2 Performance of multiple workloads

We examine how our memory balancing scheme interacts each other and impacts on overall performance. We assign different workloads to two guests. For test workloads, we run SPEC CINT 2006 benchmarks in different orders. On $guest_1$, we run the benchmarks in the descending order of WSS (CINT2006-D). On $guest_2$, we run them in the ascending order (CINT2006-A). Our system initially gives more memory to $guest_1$ and gradually moves memory to $guest_2$ over time. We evaluate the performance impact of memory balancing scheme using three different settings: *baseline*, *balanced* and *best case*. *Baseline* is the performance that occurs when the total memory is equally divided in two. To show the maximum performance gain of an ideal memory balancing scheme, we measure the *best case* performance for two guests assigning the total memory size respectively. The gap between the *baseline* and the *best case* is the maximum theoretical gain.

Fig. 3 (a) and Fig. 3 (b) show the estimated WSS and the actual allocated memory size over time, respectively. The normalized performances are shown in Fig. 3 (c) and Fig. 3 (d). In the early stage (*stage A*), the three benchmarks with the largest WSS run on $guest_1$ and the other three benchmarks with the smallest WSS run on $guest_2$. Our system gives approximately 580 MB memory to $guest_1$ and 180 MB to $guest_2$ with respect to the estimated WSS. Execution time of *bzip2*, *gcc* and *perlbench* are significantly reduced by giving more memory. The performance gains over the baseline are close to the maximum performance gains: 9.6%, 47.1% and 35.9% respectively. Conversely, the amount of memory given to the three benchmarks running on $guest_2$ is smaller than the amount of baseline. However the performance degradations are negligible. The performance degradation of *libquamtum*, *omnetpp* and

DOI: 10.1587/elex.8.748
Received March 29, 2011
Accepted April 18, 2011
Published May 25, 2011

h264ref are 4.7%, 7.8% and 0% respectively. In *stage B*, our system almost equally divides the memory for two guests because their estimated WSSs are similar. Under these circumstances, the monitoring overhead degrades the performance. However, the total performance degradation for this stage is small at 2.8%. *Stage C* has the opposite situation of *stage A*.

In terms of overall performance, our system achieves 7.5% performance improvement including 2.8% performance degradation found in *stage B*. Considering the performance gain of *best case* is 11.1%, its improvement is quite impressive.

4 Conclusion

We introduced a hardware assisted dynamic memory balancing scheme among guests. It can deduce the WSS growth and shrinkage in a guest transparent way. We also present techniques that reduces the monitoring overhead and estimates WSS larger than its current memory allocation. Performance evaluations show that the proposed scheme significantly improves the performance of tasks that suffer from insufficient memory with a low performance overhead.

Acknowledgments

This research was supported by Future-based Technology Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0020730).

DOI: 10.1587/elex.8.748
Received March 29, 2011
Accepted April 18, 2011
Published May 25, 2011