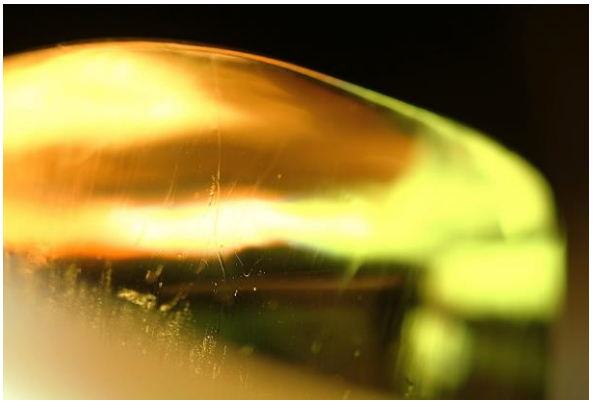

Memory Demand Projection for Guest Virtual Machines on IaaS Cloud Infrastructure

Yi-Yung Chen, Yu-Sung Wu, Meng-Ru Tsai
Dept. of Computer Science
National Chiao Tung University
Hsinchu City, Taiwan
chen.yi-yung@livemail.tw, hankwu@g2.nctu.edu,tw, vtxyer@gmail.com

***Abstract*—Virtualization technology has been widely adopted in IaaS cloud computing environment. Through virtualization, the processor, network, and storage resources can be transparently shared at fine granularity, but the memory still requires explicit coarse-grained provisioning in most cases. Yet it is not always clear how much memory should be provisioned for a virtual machine (VM). It depends on the application workload and characteristics of the underlying platform. We present NIMBLE, a novel system to project the memory demand of virtual machines in IaaS cloud environment. NIMBLE monitors the page swapping activities of a VM at runtime and project its memory demand by indicating the expected execution time of the application workload for each targeted guest physical memory size. This allows more intuitive and cost-effective memory resource provisioning for VMs. The experiment results indicate that NIMBLE can effectively project memory demand for selected benchmark workloads on both Linux and Windows guest VMs. The results also indicate that NIMBLE incurs negligible performance overhead.**

Keywords—virtual machine, memory demand, memory management, introspection, performance, cloud computing

I. INTRODUCTION

Platform virtualization enables consolidation of computation and storage resources in a datacenter environment. Resources are provisioned in the unit of virtual machines (VMs). Several VMs can be consolidated onto a host machine to improve the utilization of hardware resources and reduce cost. Still, a VM should have sufficient resource for its application workload to meet performance requirements. However, accurate provisioning of resources is difficult because in many cases the datacenter operator has few clues about the resource demand of the application workload running inside a guest VM. VM tenants, on the other hand, is not necessarily aware of the characteristics of the underlying platform either.

We develop NIMBLE, which stands for Non-Invasive Memory BottLeneck Estimator, to project the memory demand of guest VMs in IaaS cloud environment. A VM with insufficient memory resource may experience application performance degradation due to thrashing. On the other hand, over-provisioning of guest physical memory will limit the number of VMs that can be accommodated by a host machine. It is thus of great interests for both IaaS cloud service providers and VM tenants to know how much physical memory is actually required by a guest for its given application workload to attain cost-effective performance. Previous work[1-4] focus on estimating the page miss rates and dynamic memory rebalancing of VMs. However, not all application workload are as sensitive to page miss rates. It is not clear if the dynamic memory balancing will improve application performance cost-effectively. Jones et al.[5] proposed using the running time of workload as the metric for guiding the selection of proper physical memory size. However, their work was not targeted for the virtualization environment and it requires modification of the operating system kernel.

Distinct from previous work, NIMBLE can project the memory demand of a VM by indicating the expected application execution time for a targeted guest physical memory size. NIMBLE is designed to be non-invasive in that it does not require additional modifications to a guest VM other than preinstalled standard balloon drivers[6]. NIMBLE is OS agnostic and incurs minimal performance overhead. In the following, we will first have a brief overview of existing virtual machine memory management techniques in Sec. II. We will then introduce the issue of memory demand projection and the design of NIMBLE in Sec III. The evaluation and the conclusion are given in Sec. IV and Sec. V respectively.

II. VIRTUAL MACHINE MEMORY MANAGEMENT

The configuration of a system's physical memory (RAM) is traditionally considered as given and barely changes throughout the system's lifetime. Consequently, most existing memory management schemes are designed to make the best use of a given physical memory configuration. On an IaaS cloud, a hypervisor is used to provision the physical memory of a host machine to the VMs running on the host. The guest operating system of a VM will manage the provisioned memory, and the applications running on the guest systems may further employ additional layers of memory managements such as the garbage collection mechanism in JVM-based applications. Following is a summary of existing memory management schemes that have been proposed for virtual machines running in a hypervisor-based IaaS cloud environment.

A. Guest-Level Memory Managment

Guest operating systems commonly employ virtual memory management[7]. Memory pages that do not fit in the guest

physical memory are swapped to backing stores such as a disk. Virtual memory management will swap the memory pages back to the physical memory before access to the pages can be made. In general, when a guest is under memory pressure, an increase of page swapping activities can be expected. On the other hand, the execution of the application workload on the guest is very likely to be bound by the I/O due to the excessive page swapping activities. As a precondition, we may also expect high utilization of the guest physical memory. Otherwise, the virtual memory manager of the guest would have unnecessarily swapped out memory pages that could still be retained in the guest physical memory.

Most of the virtual memory managers used by the guest operating systems follow LRU-like page replacement algorithms[7]. Roughly speaking, they work by swapping less frequently used pages to the disk so that the physical memory space can be utilized efficiently for more performance critical workload such as the so-called working sets and the memory pages that are frequently accessed. To determine the memory usage of a guest, one may use VM introspection[8] to inspect the memory usage statistics as maintained by the guest virtual memory manager. For example, in the Linux kernel, the `totalram_pages` field tracks the total number of memory frames that can be used by the kernel. The `vm_stat` structure maintains records about the numbers of free memory pages and dirty memory pages. The `swap_info` structure maintains records of free swap size and total swap size. All these records are kept in the guest kernel memory space. Their memory offsets can be looked up through the kernel symbol `System.map`. However, it is not straightforward to determine the amount of memory as required by the guest by reading these usage statistics. It is neither clear how a change of the provisioned memory size would affect the performance of the guest. This is the memory demand projection problem we will introduce soon in Sec. III.

B. *Host-Level Memory Management*

The guest physical memory of a VM is provisioned by the hypervisor through software shadow page table or hardware-based nested paging such as AMD NPT[9] or Intel EPT[10]. At the most basic form, the provisioning simply maps a portion of the host physical memory to a guest VM, and the mapping will remain static throughout the lifetime of the VM. However, as it is not always possible to estimate the memory demand of a VM accurately beforehand, dynamic mapping mechanisms that can reclaim unused guest physical memory pages (and give them back to the guest on demand) have been proposed. *Memory ballooning* relies on a balloon driver[11] installed in the guest that steals unused memory from the guest when host physical memory is low. The process is referred to as the inflation of balloon. On the other hand, when host physical memory is abundant, space may be given back to the guest through deflation of the balloon, which is essentially a deallocation of the guest memory by the balloon driver. Another approach of reclaiming guest physical memory is *hypervisor swapping*[12]. In *Hypervisor swapping*, the hypervisor can swap out guest physical memory pages to a swap file. Should access to a swapped-out page is attempted by the guest, a page fault at either the shadow page table or the hardware NPT/EPT will be triggered. The hypervisor will then swap in the page back to the physical memory. Finally, there are also technologies such as *memory compression* and *transparent page sharing*[12] that aim at reducing redundant data in the physical memory.

III. Memory Demand Projection

Traditional memory management can be viewed as a reactive approach to virtual machine memory management in the sense that the software stack running the VM (and indirectly the VM tenant) focuses on the goal of making the best use of the provisioned guest physical memory. The VM tenant may request for more or less physical memory by changing the VM's configuration, and the software stack will re-adapt to the new configuration. However, it is not always obvious whether a VM should be given more or less physical memory. Due to inherent complexity of the software stack and its intertwined memory management mechanisms (Sec. II), even an experienced system administrator with comprehensive memory usage statistics logs at hands can have a hard time determining the right amount of physical memory as required by a guest VM. On the other hand, for a public IaaS cloud, the physical memory is still considered as valuable resource. It is generally not practical to assume a tenant can always begin with the maximum guest physical memory size setting and gradually reduce the size till the degradation of application performance becomes noticeable (as a practice for attaining cost-effective performance for the workload on the leased VM).

We define the memory demand of a guest virtual machine as the minimum amount of guest physical memory that is required for its application workload to reach an acceptable level of performance. Assuming the execution of application workload `W` on a virtual machine `V` that is equipped with `x` amount of guest physical memory, the time it takes for the execution of the workload to complete is defined as $\mathbf{T_V}(W, x)$, which is assumed to be a monotonically decreasing function with respect to `x`. We define the memory demand $\mathbf{MD_V}(W)$ as

$$\mathbf{MD_V}(W) := \inf\{x: \mathbf{T_V}(W, x) - \mathbf{T_V}(W, \infty) \leq T_{thres}\}$$

,where T_{thres} is a pre-determined positive constant

Note that memory demand is not necessarily the size of the guest virtual memory as allocated by the workload. One has to consider how often each memory page is accessed by the workload as well when reasoning about the execution time. On the other hand, the working set size of the workload could be a good approximation of the memory demand. However, the classical working set model and its estimation mechanism were designed for the case where the working set size is smaller than the physical memory size. Also, the working set model by itself lacks quantification of the performance impact on the application workload as a result of changing the guest physical memory size of a VM. Traditionally, all these issues do not cause much concern as a system's physical memory configuration is considered as given and will remain mostly fixed. However, with the advent of virtualization-based IaaS cloud service, the bargaining of physical memory resource (and CPU, network, storage, etc) is now at the core of business. In the following, we will introduce NIMBLE, a system that we built for projecting memory demand of guest VMs on IaaS cloud platform.

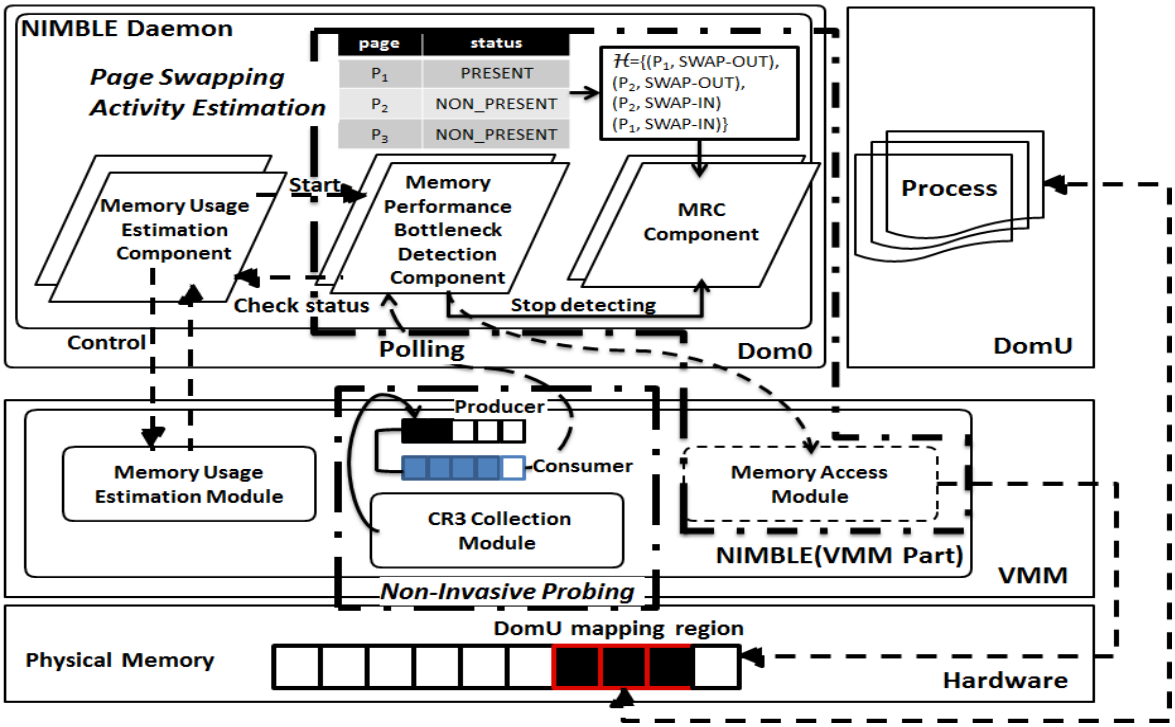

Fig. 1. NIMBLE Architecture

A. *Architecture of NIMBLE*

The architecture of NIMBLE is given in Fig. 1. NIMBLE consists of a daemon program in the management virtual machine Dom0[11] and a bunch of modules within the hypervisor. The hypervisor modules include the *memory usage estimation module*, the *CR3 collection module* and the *memory access module*. The *memory usage detection module* constantly monitors the memory utilization of a VM, the *CR3 collection module* periodically samples the values of the CR3 register, and the *memory access module* allows the NIMBLE daemon to introspect a DomU VM's memory space.

NIMBLE monitors a DomU VM for high memory utilization through the *memory usage estimation module*. The *memory usage estimation module* uses the same technique employed by Vmware[12]. During each operation cycle, it first disables the access permissions of some randomly selected pages in the EPT. When access to these pages are attempted, corresponding page fault events will be recorded, and the *memory usage estimation module* can derive the memory utilization level based on the frequencies of the recorded events. The technique is shown to incur minimal performance overhead. Only when a VM's memory utilization exceeds a threshold will NIMBLE proceed with the memory demand projection procedure, which we will discuss in the following sections.

B. *Non-Invasive Probing*

NIMBLE probes a VM to collect the performance readings needed for the projection of the memory demand of the VM. The probing mechanism assumes the VM is equipped with a balloon driver[6] (or equivalent mechanisms that can be used to dynamically adjust the amount of available guest physical memory). Other than the prerequisite balloon driver, the probing mechanism is OS agnostic and requires no additional modifications to guest VMs. We also designed the probe in such a way that it incurs minimal overhead on the guest VM.

The probing mechanism works by hooking Xen hypervisor's VCPU scheduler. The hook is referred to as the *CR3 collection module*. Right before each VCPU is about to be scheduled for running, the hook will read the CR3 register value from the VCPU register record and append the CR3 value to the tail of the *CR3 event queue* maintained by NIMBLE in the hypervisor. On x86, the CR3 register is used for storing the base address of the root page table. By collecting the CR3 register values, NIMBLE learns about the activations of the guest memory address spaces (i.e. each address space corresponds to a set of pages indexed by the page table pointed by CR3). The activation of memory space is an important clue for projecting memory demand as it indicates not only an extent of memory that is used by the guest but also how often the memory is being used.

The NIMBLE daemon running in Dom0 will read the CR3 values from the *CR3 event queue* (sequentially from head to tail). For each CR3 value read from the queue, the NIMBLE daemon will asynchronously traverse the page tables pointed by the CR3 value. The NIMBLE daemon will inspect each page table entry and determine the status of the memory page, which can be *present in memory* (`PRESENT`), *swapped out* (`NON_PRESENT`), or *unused* (`UNUSED`). For a memory page `P`, we use the notation `STATUS(P,i)` to denote `P`'s status as inspected by NIMBLE at the i^{th} time. We have the index $i \geq 1$.

C. *Page Swapping Activity Estimation*

Assuming a guest VM employs virtual memory management, when the guest is under memory stress, one can expect increased page swapping activities in the guest. The paging swapping activities will cause degradation of application performance, as the paging itself does not contribute to the progress of the application workload while it does eat up data bus and CPU bandwidths. A well-designed guest virtual memory management mechanism shall avoid unnecessary page swapping. Based on the assumption, we would like to project the memory demand of a guest by

estimating its impact on the page swapping activity of the guest. Specifically, we would like to determine the amount of extra physical memory that is needed for reducing the level of page swapping activity of a guest down below a target threshold.

In NIMBLE, we first build up a *page swapping activity history* $\mathcal{H}$ for the combination of a guest VM and a given application workload. A *page swapping activity history* $\mathcal{H}$:={S_1,S_2,..,S_k,S_{k+1},…,S_N} is a sequence of page swapping events sorted in ascending time order. A page swapping event S_k is a tuple (P, i), which corresponds to the swap-out of a memory page P followed by the swap-in of the same page P that coincides with the i^{th} time of P being inspected by NIMBLE. Formally speaking, that is

$S_k := (P, i) \rightarrow$ status(P,i-2)==PRESENT and
status(P,i-1)==NON_PRESENT and
status(P,i)==PRESENT

After a hypothetical change of the guest physical memory size and re-execution of the application workload, we may expect a different page swapping activity history $\mathcal{H}'$. Pertaining to the problem of memory demand projection, we are especially interested in knowing about the length $|\mathcal{H}'|$ of the hypothetical history. Specifically, we would like to minimize the length in a cost-effective manner.

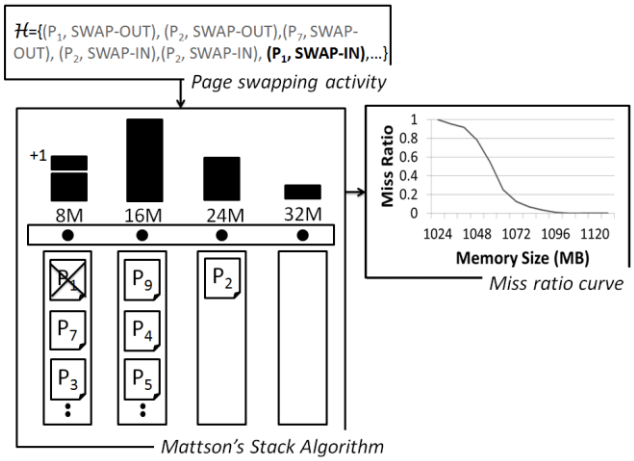

Fig. 2. Mattson's Stack Algorithm and Miss Ratio Curve

Assume that the guest virtual memory manager employs a LRU-like page swapping mechanism, the relation between available physical memory space and expected number of page swapping activities $|\mathcal{H}'|$ can be modeled by the miss ratio curve (MRC)[1] as shown in Fig. 2. The MRC curve can be built by using the Mattson's stack algorithm[1] with input of the page eviction events[13]. Following the algorithm, NIMBLE will sequentially scan trough the *paging activity history* $\mathcal{H}$ of a VM from head to tail. Each swapping-out event will be pushed to the stack. Later, when a swapping-in event is encountered, NIMBLE will look for the corresponding swapping-out event in the stack (it is the nearest swapping-out event for the same memory page that precedes the swapping-in event). If such a swapping-out event is found, its stack depth (distance between the last event pushed to the stack and the swapping-out event) will be calculated as in the Mattson's stack algorithm. In running the Mattson's stack algorithm, one may aggregate consecutive pages into a large pseudo page and run the algorithm with the large sized pseudo page to improve performance at the cost of prediction granularity.

D. Application Exeuction Time Prediction

While the MRC curve can predict the level of page swapping activity for a given combination of workload and a hypothetical guest physical memory size, it is not clear whether the reduced page swapping activity will have significant impact on the application performance. Generally speaking, the application performance can be quantified by the time it takes to complete the execution of the application workload on the given guest VM. In NIMBLE, we model the execution time of running application workload W on a guest VM V with x amount of physical memory by:

$$T_V(W, x) = T_V(W,\infty) + |\mathcal{H}| * T_S$$

, where T_S is the time cost of a page swapping action

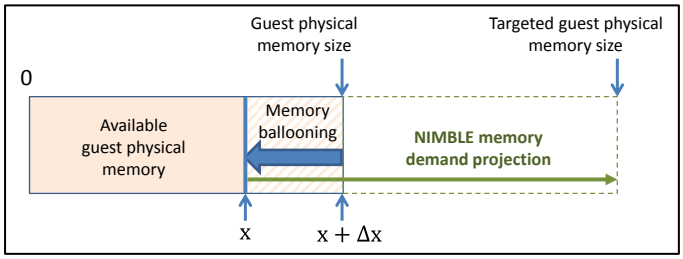

Fig. 3. Application Execution Time Prediction via Memory Ballooning

To determine the values of $T_V(W,\infty)$ and T_S, we may use the balloon driver to adjust the available guest physical memory size by a small Δx amount and rerun the workload to acquire a different page swapping activity history $\mathcal{H}'$ as shown in Fig. 3. We can then solve the following equations to determine the values of $T_V(W,\infty)$ and T_S.

$$\begin{cases} T_V(W,x) = T_V(W,\infty) + |H| * T_S \\ T_V(W,x+\Delta x) = T_V(W,\infty) + |H'| * T_S \end{cases}$$

If the application workload W is uniform, the measurement of $T_V(W,x+\Delta x)$ can be carried out following the measurement of $T_V(W,x)$. There will be no need for rerunning the application. Also, the Δx may be negative. It depends on whether the balloon driver was initially inflated or deflated.

IV. Evaluation

We built a NIMBLE prototype on top of Xen 4.2.1 for evaluation. The testbed environment consists of a server machine with Intel Xeon E5620 16-core processor, 64GB RAM and 1 TB disk. We use Fedora 18 Linux (kernel 3.6.10) and Windows Server 2008 R2 as the guest operating systems. Each DomU VM is given 4 VCPUs. We employed four benchmarks in the experiments. They are h2, tradebeans, y-cruncher[14] and 7zip compression[15]. Both the h2 and the tradebeans benchmarks are part of the Dacapo[16] benchmark suite, which is written in Java. The y-cruncher benchmark is a multi-threaded PI calculation program. The 7zip compression involves the compression of a 551MB rmvb file. The guest physical memory provisioned to the DomU VM is 1024MB for h2, tradebeans, and y-cruncher. For 7zip, we used a DomU VM with 2048MB memory.

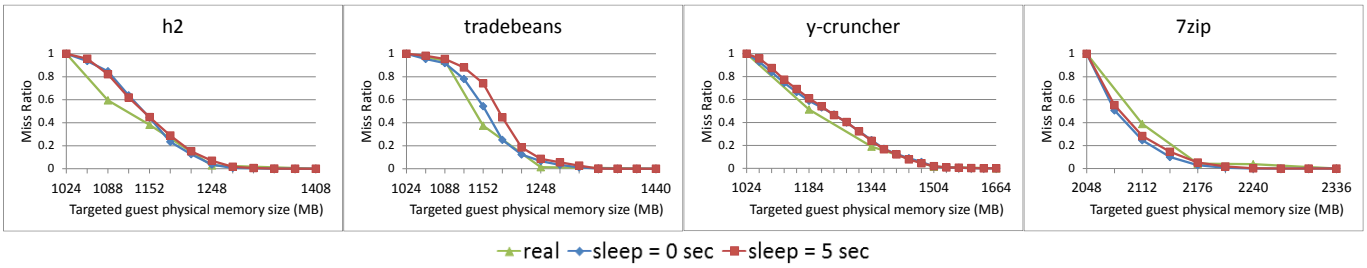

Fig. 4. MRC curve estimation for Fedora 18 guests

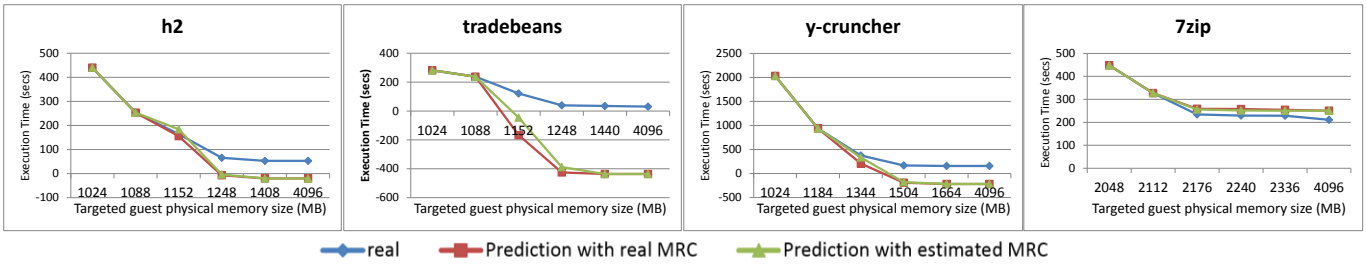

Fig. 5. Execution time prediction for Fedora 18 guests

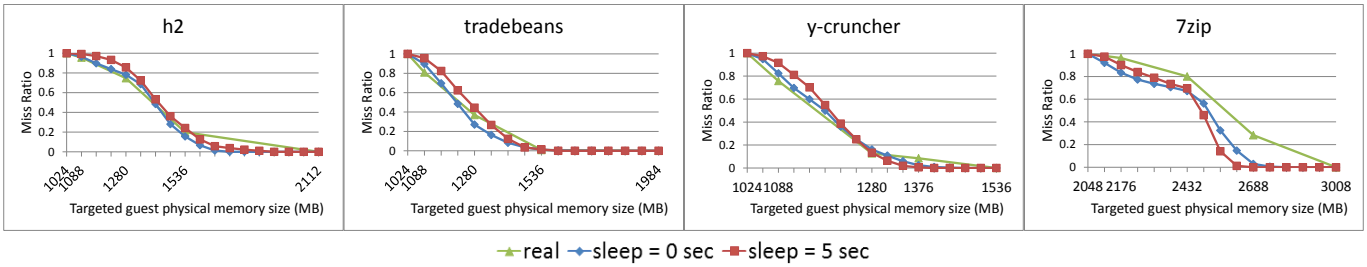

Fig. 6. MRC curve estimation for Windows Server 2008 guests

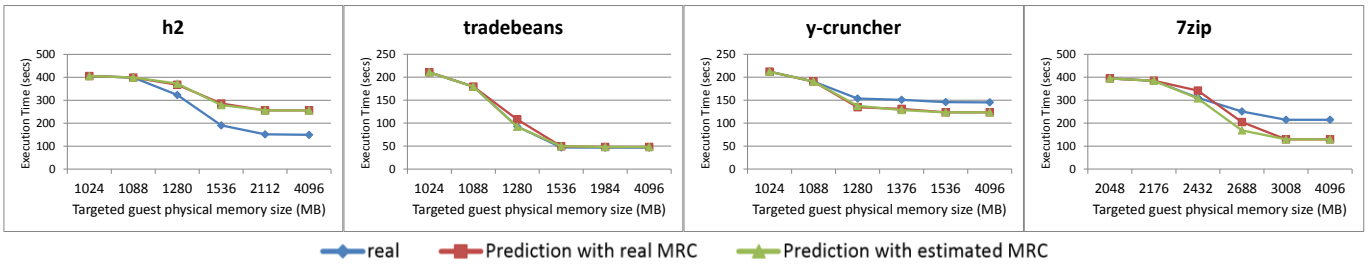

Fig. 7. Execution time prediction for Windows Server 2008 guests

A. *Demand Projection for Linux Guests*

We first look at the MRC curves built from the non-invasive probing and Mattson's stack algorithm. By design, the NIMBLE daemon polls the *CR3 event queue* in the hypervisor for CR3 sample data. When a new CR3 sample data is available, the NIMBLE daemon will immediately scan (`sleep`=0) the page tables and update the *page swapping activity history*. Here we also conducted an experiment with 5 seconds sleep delay (`sleep`=5) between the scans. The estimated MRC curves for Fedora 18 Linux guests are shown in Fig. 4. Overall, we can see that in both cases the estimated MRC curves are pretty close to the real MRC curves. This implies that NIMBLE can accurately project the level of page swapping activities for different guest physical memory sizes.

Fig. 5 presents the application execution time predicted by NIMBLE for Fedora 18 guests. For comparison, we also present the time prediction based on the real MRC curve from Fig. 4. Overall, we can see that the time prediction is pretty accurate for the `7zip` benchmark. Noticeable prediction errors were observed for `h2`, `y-cruncher`, and `tradebeans` as the targeted guest physical memory size goes farther away from the provisioned size. Nevertheless, the trends are still accurate enough for assisting the VM tenants in the selection of VM memory size. On the other hand, the swap cost `Ts` may not be stable enough to be treated as a constant (Sec. III.D). We believe the errors were largely due to the simplified model. However, from a practical point of view, we also notice that the errors in Fig. 5 can be amended by capping the predicted time at 0 (i.e. do not allow negative execution time). This

amendment heuristic will effectively address the excessive time prediction error for the `tradebeans` benchmark and will also improve the prediction results of `h2` and `y-cruncher` as well. We could possibly explore using more complex models (e.g. one with varying swap cost) in the future, but we suspect a complex model will require more comprehensive observations, some of which may require invasive probing and involve high overhead.

B. Demand Projection for Windows Guests

NIMBLE is designed to be OS agnostic, so we also tested the prototype against Windows Server 2008 guests. The estimated MRC curves for the four benchmarks are shown in Fig. 6. The predicted application execution times are shown in Fig. 7. Overall, we see that NIMBLE memory demand projection is effective for Windows Server 2008 guests as well. There is noticeable but relatively minor error in the MRC curve estimation for `7zip`. There are errors in the execution time prediction for `h2` and `7zip`, but none of them are as excessive as the `tradebeans` benchmark on Fedora 18 and the trends are all accurate. On the other hand, we also notice that the heuristic of capping execution time at 0, which we proposed in the evaluation for Fedora 18 guests, will not work here, as all the time predictions are greater than 0.

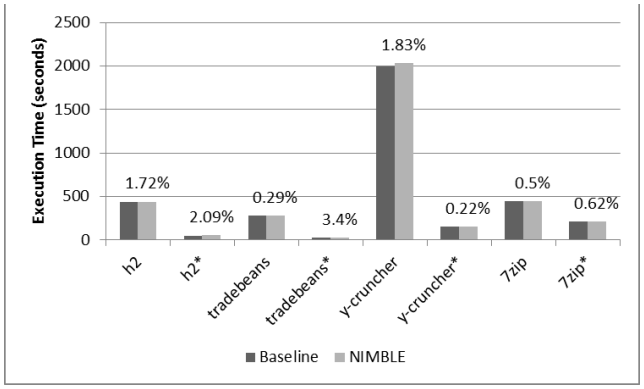

Fig. 8. Performance overhead (Fedora 18)

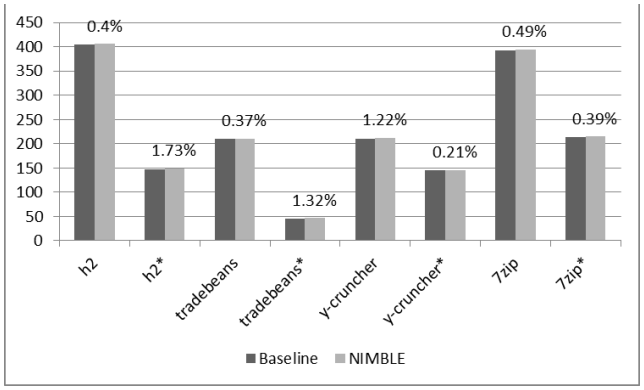

Fig. 9. Performance overhead (Windows Server 2008)

C. Performance Overhead of NIMBLE

The performance overheads incurred by NIMBLE are negligible as shown in Fig. 8 and Fig. 9. As the performance overhead is sensitive to the guest physical memory size, we also conducted a set of experiments, in which the guest VMs were given sufficient memory (4096MB). The corresponding results are superscripted by star signs.

V. Conclusion and Future Work

The use of virtualization technology in IaaS cloud environment enables the provisioning of hardware resources and consolidation of application workload. However, it is not always clear how much physical memory should be provisioned for a VM to achieve cost-effective performance for its application workload. We develop a system called NIMBLE to project the memory demand of a virtual machine on IaaS cloud environment. NIMBLE predicts the application execution time for each targeted guest physical memory size. This allows VM tenants to know if a leased VM requires more memory resource and importantly how much performance improvement is expected.

The experiment results indicate that NIMBLE can effectively project memory demand for selected benchmark workloads on both Linux and Windows guest VMs. The results also indicate that NIMBLE incurs negligible performance overhead. We do notice that there are some cases of noticeable prediction errors. A possible explanation is that the MRC model may not sufficiently capture the behavior of the guest virtual memory management, notably the combination of the `7zip` benchmark and the Windows Server 2008 guest. On the other hand, the swap cost T_s may not remain constant through the timespan of an application workload. We plan to explore these further in the future work.

Acknowledgment

This study is supported by the Cloud Computing Center for Mobile Application of the Industrial Technology Research Institute. The study is also partially supported by the Ministry of Science and Technology of the Republic of China under grant number 101-2221-E-009-076-MY3.

References

[1] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in The 11th ACM International Conference on Architectural Support For Programming Languages and Operating Systems, 2004, pp. 177-188.

[2] W. Zhao, Z. Wang, and Y. Luo, "Dynamic memory balancing for virtual machines," ACM SIGOPS Operating Systems Review, vol. 43, pp. 37-47, 2009.

[3] Y. Niu, C. Yang, and X. Cheng, "Dynamic Memory Demand Estimating Based on the Guest Operating System Behaviors for Virtual Machines," in The 9th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2011, pp. 81-86.

[4] C. Min, I. Kim, T. Kim, and Y. I. Eom, "VMMB: virtual machine memory balancing for unmodified operating systems," Journal of Grid Computing, vol. 10, pp. 69-84, 2012.

[5] S. T. Jones, A. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "MemRx: What-If Performance Prediction for Varying Memory Size," Technical Report 1573, Department of Computer Sciences, University of Wisconsin-Madison2006.

[6] XenServer.org. (2014-9-2). XCP FAQ: Dynamic Memory Control. Available: http://wiki.xenproject.org/wiki/XCP_FAQ_Dynamic_Memory_Control

[7] A. S. Tanenbaum and H. Bos, Modern Operating Systems, 4 ed.: Prentice Hall, 2014.
[8] K. Nance, M. Bishop, and B. Hay, "Virtual machine introspection: Observation or interference?," IEEE Security & Privacy, vol. 6, pp. 32-37, 2008.
[9] AMD, "AMD-V Nested Paging," 2012.
[10] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," 2014.
[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, et al., "Xen and the art of virtualization," in ACM Symposium on Operating Systems Principles, 2003, pp. 164-177.
[12] I. VMWare, "Understanding Memory Resource Management in VMware ESX 4.1," 2010.
[13] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: monitoring the buffer cache in a virtual machine environment," in The 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems 2006, pp. 14-24.
[14] A. J. Yee. (2010). y-cruncher - A Multi-Threaded Pi-Program. Available: http://www.numberworld.org/y-cruncher/
[15] I. Pavlov. 7-Zip. Available: http://www.7-zip.org/
[16] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, et al., "The DaCapo benchmarks: Java benchmarking development and analysis," in ACM Sigplan Notices, 2006, pp. 169-190.

Yi-Yung Chen *received the B.S and M.S degrees in Computer Science from National Chiao Tung University, Taiwan in 2012 and 2014. His research interests include virtualization technology and operating systems.*

Yu-Sung Wu *received the B.S. degree in Electrical Engineering from National Tsing Hua University, Taiwan in 2002, and the Ph.D. degree in Electrical and Computer Engineering from Purdue University, West Lafayette, Indiana in 2009. He is an assistant professor in the Department of Computer Science, National Chiao Tung University, Taiwan, where he leads the Laboratory of Security and Systems. His research interests include security, dependability, and systems.*

Menq-Ru Tsai *received the B.S. and M.S. degrees in Computer Science from National Chiao Tung University, Taiwan in 2011 and 2013 respectively. His research interests include security, system and front end technology.*