SEMMA: Secure Efficient Memory Management Approach in Virtual Environment

Xian Chen, Wenzhi Chen, Peng Long, Zhongyong Lu, Zonghui Wang
College of Computer Science and Technology
Zhejiang University, Hangzhou, China
Email: {chenxiancool, chenwz, longpeng, lzy6032, zhwang}@zju.edu.cn

***Abstract*—For the ability to explore the memory's fullest potential, memory de-duplication has been widely experimented with in current main stream virtualization platforms. A lot of work has been done to improve the efficiency of memory de-duplication, while too little attention was paid to the introduced security issues which have been proved by prior work. To deal with this security risk and efficiently manage the memory we start our work in this paper. We first conduct extensive experiments on different OS platforms to study the realistic situation of page sharing. By analyzing the results we find: 1) Page sharing is contributed mostly by self-sharing (nearly 90%), and the self-sharing rate varies significantly between Linux and Windows OS platforms. Compared with self-sharing the inter-VM sharing rate is extremely low, under 1% in most cases. 2) Page size has a larger influence on Linux platform than Windows platform, so sub-page de-duplication proposed in prior work may achieve better performance on Linux platform. On the basis of above findings we propose a group-based secure page sharing model (or GSKSM), in which we consider both VM processes and normal processes. We have successfully implemented it in Linux kernel 3.6.6, and the experiment results show it works well with negligible overhead. Finally based on GSKSM we further present an efficient memory management approach (or SEMMA), which combines GSKSM and balloon technique to efficiently manage the memory, and the preliminary experiment result is satisfactory.**

Keywords—virtual machine; KSM; Ballooning technique; memory de-duplicaiton

I. Introduction

Nowadays cloud computing is widely-known not only in academia and industry but also in our daily life. As the key technology of cloud computing, virtualization plays an import role in many fields for its ability to efficiently manage the underlying hardware resources and enable cloud service providers to service many tenants simultaneously. With virtualization one physical machine in cloud computing center could run several virtual machines which are rent to the tenants, thus the utilization of hardware resources can be significantly improved. But as the number of tenant increases, the workload on an individual physical machine becomes heavier, thus the performance of VMs running on it will be affected differently. One main cause is the memory capacity is not sufficient, which has triggered a lot of academic work [6][7][8][9] in this field to explore the memory's fullest potential. One direction is to de-duplicate the duplicate memory content to ease the physical machines' memory pressure. The representative technique is content-based page sharing (CBPS) [6], which is implemented in the hypervisor layer and has already been experimented with in current main stream virtualization platforms, e.g., XEN [11], KVM [12], and VMware ESX [21].

A lot of work has been done to improve the efficiency of CBPS, and it is adopted in many fields such as VM migration [13] and VM check pointing [14]. But too little attention was paid to its introduced security problems. As presented in [4], one VM user can use the difference in write access time to snoop the working set running on the other VM, which is unacceptable for the tenants whose applications running in their rented VMs are very important. On the other hand, work in [2] has proved that the inter-VM sharing rate is extremely low, which is also validated in our following experiments. So in theory, group mechanism is the best way to solve this security risk, because it isolates all the VMs into different groups, avoiding the malicious memory detection from the root cause. This is what we will present in section IV.

In addition to page sharing, there are also some other techniques to overcommit the memory [18], e.g., swapping mechanism, balloon technique [11], and memory compress. Swapping is controlled by the operating system, and may cause high performance cost, so it's usually the last choice. Balloon technique is a widely adopted skill, and the achieved performance is very good. On the other hand, much work has been done to appropriately use balloon technique to balance memory load in host machine, e.g., [19] and [20]. But all the prior work only considers balloon technique, lack of a reasonable management approach which can take all the memory overcommit techniques or several of them into consideration, and this is the motivation of our work in section V.

The rest of the paper is organized as follows. The related work is introduced in section II, section III presents our experimental study on page sharing, and then we detail our proposed group-based secure page sharing model (GSKSM) in section IV. Section V will describe our secure efficient memory management approach (SEMMA), which is based on the work in section IV. Section VI is about our conclusion and future work.

II. Related Work

Disco [5] is the first system to implement page sharing to improve memory utilization on a multiprocessor platform. It uses Copy-On-Write mechanism on disks. If the content that a

VM instance wants to read from a COW disk is already in the main memory, the VM instance directly maps the content page into its own address space, reducing the memory footprint and access time. However it needs to modify the guest operating system, which severely limits its implementation and usage.

Content-based page sharing is proposed in VMware ESX [6], it is implemented in the hypervisor layer, and does not need the assistance from the guest machine. The feature of transparent to the guest OS makes CPBS widely adopted in many virtualization systems. As one implementation of CBPS in Linux, KSM [8] could significantly improve the ability of KVM, and work in [16][17] does some optimization to make the native KSM work more efficiently. Potemkin [15] is another system based on the core idea of CBPS, in which new virtual machines can be created as clones of an existing VM image, thus the benefit of page sharing is greatly explored.

Based on CBPS, another type of page sharing is proposed in Different Engine [9], which is called sub-page sharing. Sub-page sharing not only explores the potential of the same pages but also the similar pages. For its excellent performance, Memory Buddies [10] has adopted it in VM migration [13] and VM consolidation [14].

Empirical studies of memory sharing in virtual environment have been done in [1] [2]. In [1], the authors conduct extensive experiments on the effectiveness of KSM on various kinds of workload. They find that for mixed CPU and I/O workload, KSM could achieve the most significant memory saving. But for CPU intensive applications, KSM does not have significant effect on dynamic sharing and it also causes higher runtime overhead.

Different from [1], [2] does not only focus on the output performance of benchmarks, they provide insight into the typical sources of sharing potential through an exploration and analysis of memory traces captured from real user machines and controlled virtual machines. They find that the absolute memory sharing rate is generally under 15%, contrasting with prior work, but partly in accordance with what we have got. Another important fact demonstrated in [2] is sharing within individual machines accounts for nearly all (>90%) of the sharing within a set of machines, also in accordance with our results.

The security issues introduced by KSM has been proved in [4], they propose a cross-VM attack method to snoop the working set running on the other VM with the help of the difference in write access time on de-duplicated memory pages. In their experiments, they have successfully detected the existence of sshd and apache2 on Linux, and IE6 and Firefox on Windows XP.

Work in [3] is similar to the second part of our work. They also propose a group-based approach to solve the secure risk introduced by page sharing. However, they split the global ksmd thread into per-group ksmds with a cgroup-based user interface. In our implementation we use only one ksmd thread, which can provide all the functions in [3], but with no additional cost of other ksmd threads. In addition, the normal processes are also taken into consideration in our work in this paper.

Ginkgo [20] provides a memory overcommit framework which regularly monitors application progress and incoming load for each VM. Then use this data to predict application performance under different VM memory sizes, and automatically adjust VMs' memory occupation. Ginkgo only considers balloon technique while in our work we also take page sharing into consideration.

III. EXPERIMENTAL STUDY ON PAGE SHARING

To better understand the real situation of page sharing in virtual environment, we conduct extensive experiments in several different scenarios. Table I shows the detailed configuration in our experimental environment.

TABLE I. EXPERIMENTAL ENVIRONMENT

Hardware (DELL OPTIPLEX 9010)	
Processor type	Intel I7-3770
Number of cores	4 cores (8 threads)
Clock frequency	3.4GHZ
Memory	8GB DDR3 1600
Disk	500GB SATA 7200rpm
Software Configuration	
Host OS	Ubuntu12.04 LTS (64 bits kernel3.6.6)
QEMU	1.4.0
VM	CPU:2 Memory:1GB IMG:10G/30G

We run Ubuntu12.04 (64 bits) on our hardware platform, and recompile kernel3.6.6 in the host OS for the need of following experiments. As to VMM, we chose Qemu1.4 for the front end of the KVM module, because this version has supported the stable KSM. In our tests, we configure each VM instance with 2 cores, 1GB memory and 10GB virtual disk (30 GB in kernel building experiments)

A. Data Collection

There are some ready-made tools for us to fetch the memory of VM instances e.g., pmemsave command in Qemu monitor and snapshot command in virsh monitor. In our experiments, we find that the dump files generated by the tools are not corresponding to the real situation of resident memory, the dump files are usually larger than the memory actually used. Then, we dive into this issue, finding out that all the tools generate a zero page when the page is not in physical memory or even does not have an entry in the page table, but those pages should not be contained in the dump files. This situation may be the cause of many zero pages in memory traces proved in prior work. So we make a tool to generate the dump files which only contain these pages resident in physical memory. The tool works intuitively: it scans all the VMAs (Virtual Machine Area) of a specific VM process, if one page in the VMA is resident in physical memory, then it will be put into the dump file. In addition, our tool can fetch the memory actually used by a normal process, thus we can analysis the page duplication rates of normal processes.

In order to fully study the situation of page sharing on different OS platforms, we configure 7 distinct VMs, each with 1GB physical memory:

- ***Linux:*** Centos 6.4 (no GUI) and Ubuntu 12.04 LTS. These distributions were chosen to be representative of typical desktop. we consider both 32-bit and 64-bit versions (4 VMs in total)

- ***Windows:*** both Windows 7 x86 and X64 versions, and also Windows XP (3 VMs in total)

We conduct our experiments in three scenarios as follows:

- ***No workloads:*** The VM is freshly booted, but not running any further applications. When the VM is running steadily, we stop the VM, and use our tool to fetch the VM's memory content. Three dump files each VM instance, 21 dump files in total.
- ***Normal workloads:*** In this scenario, the VM runs several normal applications, e.g., web browser, email client, office applications, media player. As the memory is not stable when the VM instance is running applications, we generate a dump file every 3 minutes, lasting for 30 minutes.
- ***Kernel build workload:*** We choose kernel build as the work load running in VM instances as [7], generating a dump file every 15 seconds, lasting for 30 minutes.

All the memory traces are generated when the VMs are stopped, so the VMs are not affected when tracing, and the logical time spent on tracing is zero.

B. *Calculating Sharing*

As described in [2], we separate sharing pages into two main categories: self-sharing (sharing within single VM) and inter-VM sharing (sharing between different VMs). To clearly depict what the two categories really mean, we use two figures to show how to calculate self-sharing rate and inter-VM sharing rate.

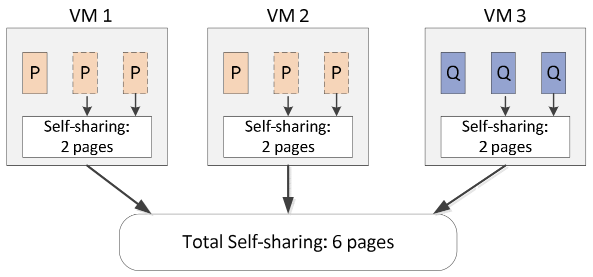

Figure 1. Self-sharing

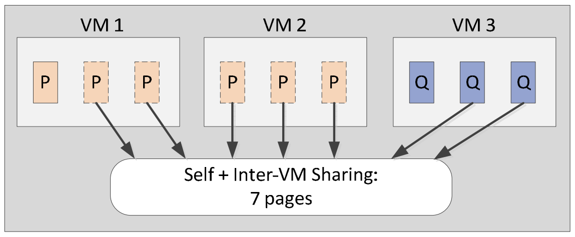

Figure 2. Inter-VM Sharing

In Figure 1, there are three VMs (VM1, VM2, VM3), each contains sharable memory pages: both VM1 and VM2 have three copies of page P, VM3 has three copies of page Q which is different from page P. If we only consider self-sharing, each VM can reduce its memory footprint to one third of the original space: only one copy of page P is reserved in VM1 and VM2, one copy of page Q in VM3. If we take inter-VM sharing into consideration, the remainder pages in VM1 and VM2 could be shared further as depicted in Figure 2. Then we can expand the benefit of page sharing, even though the inter-VM sharing only occupies a very small fraction. From the above analysis, we could get the self-sharing rate for each VM in Figure 1 is 67% (2/3), and the inter-VM sharing rate among three VMs in Figure 2 is 11.1% (1/9). In the following tests, we will use the same method to calculate the page sharing rates.

C. *Page Sharing In VMs*

To gain deeper insight into the page sharing rates in virtualization environment, we conduct several experiments both on different OS platforms and in different usage scenarios.

First we study the self-sharing on different platforms in no workload scenario. As depicted by Figure 3, there is a big gap between Linux and Windows platforms. The self-sharing rate is only less than 18% on Linux platforms, while it can reach as high as 87% within Windows XP. With further work, we find that this is caused by their different memory management mechanisms, for Windows OS it will occupy all the memory declared for it when the VM is created, while for Linux OS the memory is used on demand. In addition, we could find the majority of self-sharing on Windows platforms is occupied by zero pages, while little on Linux platforms.

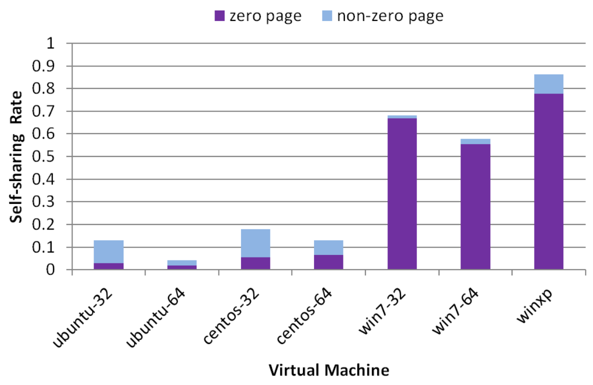

Figure 3. Self-sharing On Different OS Platforms Without Workload

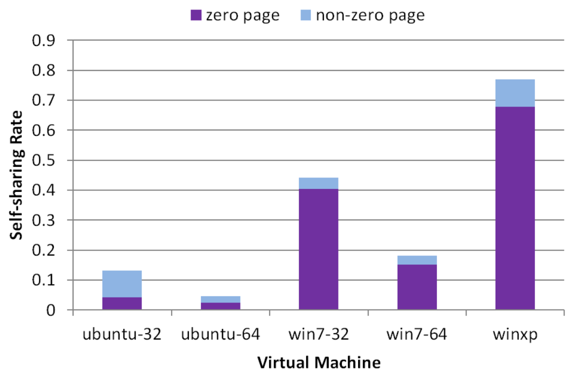

Figure 4. Self-sharing On Different OS Platforms With Normal Workloads

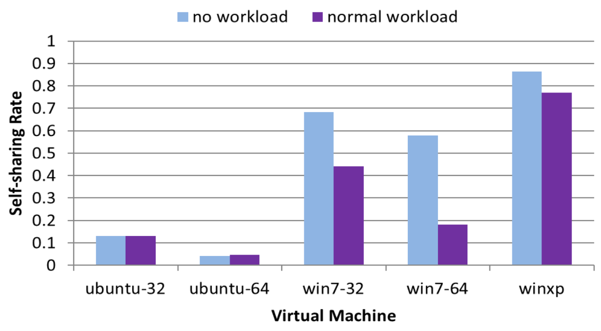

Figure 5. Comparison Between Non-workload And Normal Workloads

Figure 4 shows the self-sharing rates on different OS platforms with normal applications running in guests, it's similar to Figure 3, but the decrease in self-sharing rate on Windows platforms can be easily found, especially for window 7 (64 bits). Figure 5 can better depict this phenomenon, and we deduce this is because zero pages in self-sharing have been used by the normal applications. For Linux platforms, zero pages occupy very little part of self-sharing, so little influence on them.

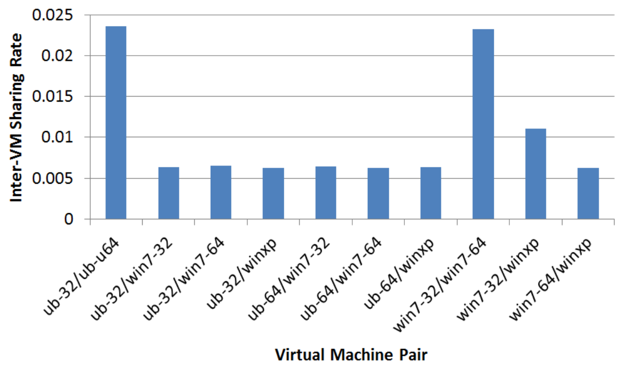

Figure 6. Inter-VM Sharing Between Virtual Machine Pairs

Second we study the inter-VM sharing between different virtual machines, and the result is presented in Figure 6. On the whole, inter-VM sharing rate is extremely low, occupying only 2.4% even in the best case and below 1% in most cases. If we dive into Figure 6, we can notice some interesting things: **1)** The inter-VM sharing rate is usually higher when the VMs have the same OS but different versions, e.g., 32-bit Ubuntu 12.04 and 64-bit Ubuntu 12.04. **2)** The inter-VM sharing rate between different OS platforms is usually lower (under 1%).

In our experiments the dump files of Linux are about 600 MB and 1G for Windows, then by combining Figure 5 and Figure 6 we could easily conclude that majority of page sharing between different OS platforms is contributed by the self-sharing within individual VM, while the inter-VM sharing only occupies about 10%.

Third we use the kernel build workload to test the volatility of page sharing, Figure 7 shows the result. From Figure 7 we could conclude that the page sharing rate varies with workload, so we should use dynamic mechanism to exploit the benefit of page sharing.

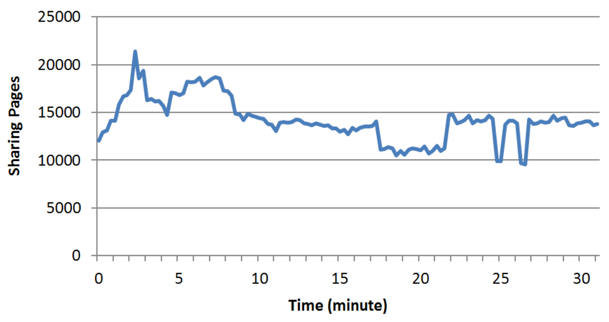

Figure 7. Sharing Pages On Kernel Build Workload

D. *Impact Of Page Size On Page Sharing*

Previous works in [9][10] have demonstrated that significant benefits can be achieved by sharing portions of similar, but not identical pages. In order to further study the real situation, we take a close look at our collected traces from the above experiments. This time we calculate the page sharing rate with different page size ranging from 1KB to 16KB, and the result is presented in Figure 8. The x-axis shows the page size in KB, the y-axis represents the page sharing rate which has been normalized to 4KB (normal page size). In Figure 8 we can see the influence of page size on Linux platform is larger than Windows platform, especially when page size is smaller than 4KB. So subpage level sharing proposed in prior work may bring more benefits for Linux platforms, and 4KB seems already OK for Windows platforms.

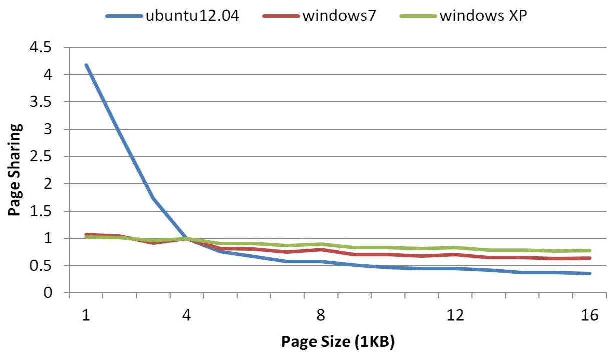

Figure 8. Impact of Page Size On Page Sharing Rate

IV. GROUP-BASED SECURE PAGE SHARING MODEL

As proved by [2] and our work in section III, the page sharing rate between different VMs is really low compared with self-sharing rate. On the other hand, the prior work in [4] has demonstrated that one VM user can use cross-VM attack to snoop the working set running on other VMs which are resident in the same physical machine. This is unacceptable for the tenants whose applications running in their rented VMs are very important. So we deem that a secure mechanism must be implemented to reduce the risk of information leakage. Group-based page sharing is the best way to address the security issue, as it isolates all VMs into different groups, thus one cannot snoop the memory information resident in other VMs by the shared pages.

In this section we propose a group-based secure page sharing model, in which we assign all VMs into different groups as [3] according to the needs of tenants, but we don't split the global ksmd (server thread) into per-group ksmds, which we think will introduce more cost on memory space and executing time. In addition, our model can automatically group the normal processes into a private group which does not contain any VM process. We have successfully implemented the model in kernel 3.6.6, and test results show it can work well with negligible overhead.

A. Architecture

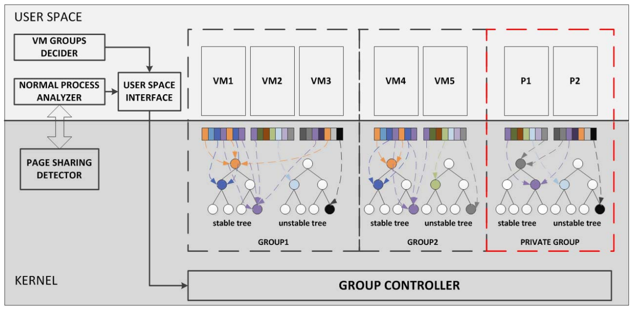

Figure 9. Architecture Of Group-based Secure Page Sharing Model

Figure 9 presents the overall architecture of our proposed group-based secure page sharing model. As the model demonstrates, group controller can construct two kinds of groups: VM group and private group (denoted by the red dashed line box). VM group only contains VM processes of different tenants, one group is corresponding to a tenant. The private group is used only for the normal processes running in the host OS, through this group we could de-duplicate the duplicate memory content in normal processes, thus expanding the benefit of page sharing in host machine.

The construction and destruction of each group is controlled by the group controller module which also exposes an interface to the user space. In addition, we can also tune the merging speed of every group respectively. If the workload is too heavy in one group, we can let the merging service thread occupy less CPU time by lowering the merging speed and vice versa.

The normal process analyzer is responsible for selecting appropriate normal processes which should have large memory footprint and the page sharing rate is relatively high. Once the normal processes are determined, this module puts them into the private group through user space interface. In the process of selecting appropriate candidates, the normal process analyzer module first sorts all the processes (except for VM processes) running in the host OS by their memory footprint, then select those with enough memory footprint and estimate their page sharing rates with the help of page sharing detector module. Eventually, the processes with large memory footprint and high page sharing rate will be selected and put into the private group.

From a certain meaning, the VM groups decider module in Figure 9 refers to the cloud service provider who should isolate all the VMs into different merging groups in accordance to the will of the tenants.

B. Implementation

Our proposed page sharing model is implemented based on KSM [8] in kernel3.6.6. As presented in Figure 9, each group has two red-black trees, one is the stable tree and the other is the unstable tree. The stable tree contains all the already shared and not changing KSM generated pages while the unstable tree is used to maintain the pages which are not shared yet but still tracked by KSM. The detailed operations on the two trees are same as those implemented in KSM.

To reduce the cost on memory space and CPU time, we only construct a main merging thread to manage all the groups which is different from [3]. The main merging thread (i.e. ksmd) repeatedly scans all the memory content in each group, and merges these pages with the same content. Here we use the weight mechanism to treat every group fairly, in which every group has a weight value corresponding to its workload, and the weight value ranges from 0 to 10. If a group wants to get more benefits from page sharing, so it should have a larger weight value, then the main thread will spend more time on merging same content pages in this group. In our implementation we calculate the number of pages which the main thread should scan for each group by the following formula.

$$\boldsymbol{GroupPages = N \times MaxPages \times \frac{GroupWeight}{TotalWeight}}$$

In the above formula, GroupPages represents the number of pages which the main thread should scan for a group, N is the number of existing groups, MaxPages is the number of pages the main thread should scan for a group if all the groups have the same weight value. GroupWeight represents the group's weight value and the TotalWeight is the sum of all the groups' weight values.

As to normal processes, we implemented a daemon to repeatedly analyze the memory footprint and page sharing rate of the applications running on the host machine. In the process of analysis, the daemon first sorts all the normal server processes by their memory footprint, then select those with enough memory occupation. Once the candidates are determined, the daemon will use page sharing detector kernel module to determine whether the page sharing rate is enough high, if all conditions are appropriate, the process will be selected and put into the private group to make its contribution to the benefit of page sharing.

C. Evaluation

To evaluate the performance of our proposed group-based secure page sharing model, we select three VMs (32 bits Ubuntu 12.04, 32 bits Windows 7 and Windows XP) and group them with different combinations as follows:

- ***One Group:*** we put the three VMs into one group, and calculate the total sharing pages every 2 seconds, lasting for about 220 seconds.

- ***Two Groups:*** the VMs running Ubuntu and Windows 7 are in the same group, while Windows XP in another group, then calculate the total sharing pages as above.
- ***Three Groups:*** one VM occupies one group, the calculation method is the same as the above.

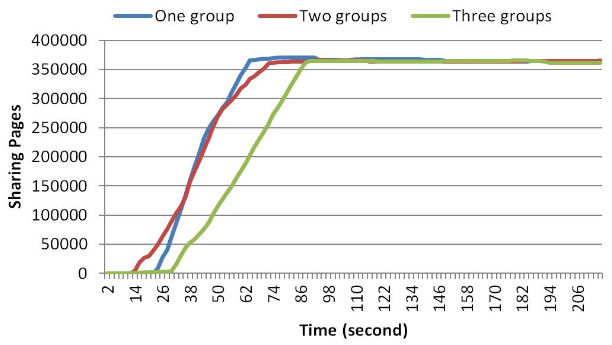

Figure 10. Total Sharing Pages Using Different Grouping Methods

In order to get rid of the influence of normal processes, we disable the normal process analyzer module in our tests. The experiment result is depicted by Figure 10. Because it's hard to make all the test scenarios completely consistent, so the time intervals between the start moment and the moment all the VMs are running smoothly differ from each other, but this will not influence the conclusion. As depicted in Figure 10, once the state is stable, the number of total sharing pages with different grouping methods is very close, which indicates that the overhead of our grouping mechanism on the page sharing is low.

In addition, we also conduct another contrast experiment to evaluate the efficiency of the normal process analyzer module (NPA). In this experiment we just construct one VM group containing three VMs running Ubuntu 12.04, Windows 7 and Windows XP. In the host we start Firefox browser to browse some web sites and open the Office tools to edit several texts, then we calculate the total sharing pages achieved by GSKSM when NPA is enabled and disabled, the result is shown in Figure 11.

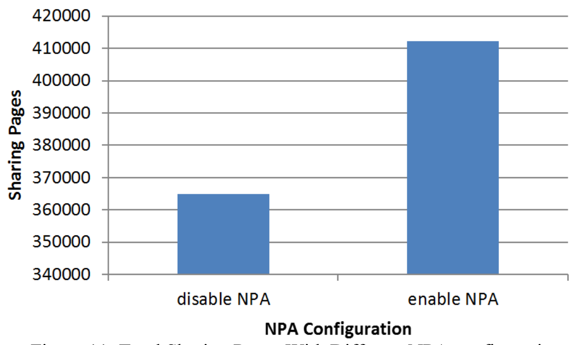

Figure 11. Total Sharing Pages With Different NPA configuration

If we enable the normal process analyzer module, we can get 47436 more sharing pages (about 185MB), and the performance is improved by about 13%. These added pages are mainly contributed by the Firefox browser and Office tools.

V. SECURE EFFICIENT MEMORY MANAEMENT APPROACH

Based on our work in section IV, we further propose a novel memory management approach which we named SEMMA. As we all know there are several methods to allow a virtual machine to overcommit its memory, e.g., swapping mechanism, balloon technique and page sharing. The swapping mechanism is automatically managed by OS, and may cause high performance cost, so not suitable for user to use. Balloon technique is a widely used skill in current main stream virtualization platforms, and it is implemented as a virtual device. When it inflates, it will take some memory from the guest, and give the reclaimed memory to the host, this situation is also called guest shrinking. When the guest does not have enough memory for its applications, it will deflate the balloon, thus some memory will be back to the guest. In the actual production environment balloon technique is really a powerful tool for cloud system administrator to balance the memory load while providing stable performance. However, there still lack of a reasonable memory management approach which can consider all the memory overcommit techniques or several of them, and this is the motivation of SEMMA.

A. Premise Explanation

SEMMA is based on the group-based secure page sharing model proposed in section IV, and now under implementation. To clearly describe its working mechanism, we make the following definition.

- ***MUV:*** memory used in virtual machines.
- ***MUP:*** memory actually used by the VM process.
- ***UR:*** calculated by dividing MUV by MUP. It indicates the actually memory used percentage by the guest.
- ***MUH:*** memory utilization in host machine.

To efficiently use the memory in host, we always maintain UR between the lower bound (70% in current implementation) and the upper bound (80%), if UR is higher than the upper bound, the guest has little free memory for its applications, and the performance may be influenced, so the host should return the occupied memory to the guest by deflating the guest's balloon driver until UR is lower than the upper bound. If UR is lower than the lower bound, the memory occupied by the guest is more than its actually need, so we should reclaim the unused memory by inflating the balloon driver, then the host will have more free memory and can host more virtual machines.

As to page sharing, we adjust the scan speed of each group according to the host's memory load. Here we also define two thresholds: the lower bound (60% in current implementation) and the upper bound (80%). If the MUH is higher than the upper bound, which indicates the host is suffering higher memory load, then we speed up GSKSM scanning to generate more usable memory by merging more pages. When MUH is lower than the lower bound, the host has enough free memory for all the VMs running on it, and then we should slow down the scanning to reduce the overhead of GSKSM. More detailed mechanism is presented in Algorithm 1.

Algorithm 1 Core processing in Decision Maker

```
for all VMs in host
  if UR > UR_UPPER_BOUND && guest does not get all its memory then
      deflate the balloon driver to return the occupied memory to guest
  else if UR < UR_LOWER_BOUND then
      inflate the balloon driver to reclaim the unused memory from guest
    end if
  end if
  if MUH > MUH_UPPER_BOUND then
      speed up GSKSM scanning to generate more usable memory
  else if MUH < MUH_LOWER_BOUND
      slow down GSKSM scanning to reduce the overhead of GSKSM
    end if
  end if
end for
```

B. *Architecture and Implementation*

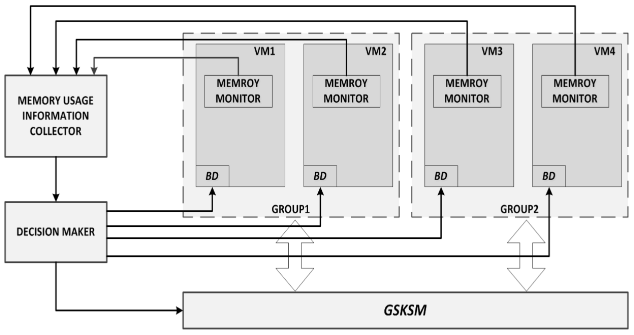

Figure 12. Architecture Of SEMMA

As depicted by Figure 12, SEMMA contains three main subsystems: memory usage information collector (or MUIC), decision maker and GSKSM (proposed in section 4). MUIC periodically collects three types of memory utilization, i.e. MUV, MUP and MUH. It's easy to collect MUP and MUH, just by executing top command in the host, while some more work is needed to get the memory usage information from the guests, but still there are many ways to finish this work. In current implementation, we use a network application for the guests running Windows OS, while just executing the scp command for the guests running Linux OS.

According to value of MUV, MUP and MUH, decision maker will take corresponding actions for each VM as presented in Algorithm 1. If it wants to reclaim/return memory from/to a guest, it needs to use the interfaces provided by Qemu monitor to inflate or deflate the balloon driver (or BD). If the host is suffering from heavier memory load, the GSKSM module will be notified to speed up the scanning to generate more free pages.

C. *Evaluation*

We conduct an experiment to evaluate the efficiency of SEMMA, in which we create a VM instance when the memory utilization in the host is about 75%, and use it to browse some web sites and edit some documents. At the same time we record the memory utilization in the host machine with three different configurations: disable KSM and SEMMA, only enable KSM and only enable SEMMA.

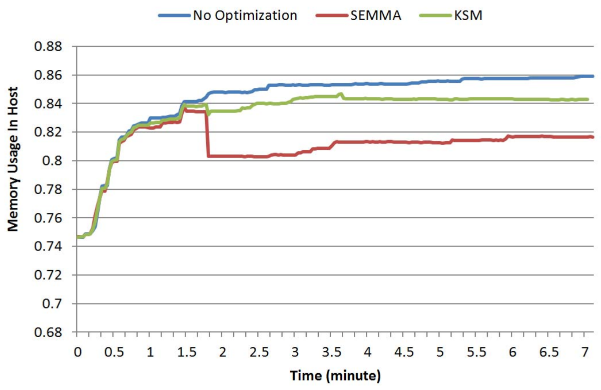

Figure 13. Memory Usage In Host With Different Configurations

As shown in Figure 13, in the beginning the memory utilization is about 75% resulting from other workloads running in the same host machine, then it increases gradually when the VM instance is being created. When the creating is done the memory usage is about 83%, if without any optimization it will increase by about 3% for the following operations in VM. If SEMMA is enabled, there will be a sharp down in the memory footprint. This is because the balloon technique is triggered when the UR is lower than the lower bound (70%), which indicates there are many unused memory in guest, so the balloon driver is inflated to reclaim some memory from the guest, maintaining the UR always between the lower bound and the upper bound. For native KSM, the result is also not bad, but still about 2.5% (205MB) higher than SEMMA.

It should be noted that: in our experiment we only construct one VM instance running Linux OS, so the benefit is not very obvious. If more VMs are taken into consideration or the VM is running Windows OS, the result will be better. SEMMA is still under implementation, many optimizations can be added to it, although the preliminary test result is a bit satisfactory.

VI. CONCLUSION AND FUTURE WORK

In this paper we first conduct extensive experiments on different OS platforms to study the real situation of page sharing, and get several conclusions: 1) Self-sharing occupies majority of the total page sharing, nearly 90%, and the self-sharing rate is very different between Linux and Windows platforms. Compared with self-sharing the inter-VM sharing rate is extremely low, under 1% in most cases. 2) Page size has a larger influence on Linux platform than Windows platform, so sub-page de-duplication may achieve better performance on Linux platform. On the basis of the above findings we propose a group-based secure page sharing model (or GSKSM), which not only considers the VM processes but also the normal processes. We have successfully implemented it in Linux kernel 3.6.6, and the results show it works well with negligible overhead. Based on GSKSM, we further present a secure efficient memory management approach (SEMMA), which combines GSKSM and balloon technique to efficiently manage the memory. Even though the preliminary experiment result is satisfactory, SEMMA is still under implementation and many parts need to be optimized.

In future, we will continue our work in section V. First we will try to design a better algorithm to predict the memory usage, thus we could eliminate the instantaneous fluctuation of the sampled data. Second we will try more benchmarks on SEMMA to select a better set of bound values. Third we will pay more attention to the stability of SEMMA, making it a usable memory management system.

ACKNOWLEDGMENT

We would like to thank all the members of Linux Interest Group of ARC Lab at Zhejiang University for their fruitful discussions that motivate this work. We also wish to thank the anonymous reviewers for their hard work for this conference. This work is supported by the key state science and technology project (2013ZX03003010-002).

REFERENCES

[1] Chang Chao-Rui, Jan-Jan Wu, and Pangfeng Liu. "An empirical study on memory sharing of virtual machines for server consolidation." Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on. IEEE, 2011.

[2] Sean Barker, Timothy Wood, Prashant Shenoy, Ramesh Sitaraman. "An empirical study of memory sharing in virtual machines." Usenix ATC. 2012.

[3] Sangwook Kim, Hwanju Kim, Joonwon Lee. "Group-Based memory deduplication for virtualized clouds." Euro-Par 2011: Parallel Processing Workshops. Springer Berlin Heidelberg, 2012.

[4] Kuniyasu Suzaki, Kengo lijima, Toshiki Yagi, Cyrille Artho. "Memory deduplication as a threat to the guest OS." Proceedings of the Fourth European Workshop on System Security. ACM, 2011.

[5] Edouard Bugnion, Scott Devine, Kinshuk Govil, Mendel Rosenblum. "Disco: Running commodity operating systems on scalable multiprocessors." ACM Transactions on Computer Systems (TOCS) 15.4 (1997): 412-447.

[6] Waldspurger, Carl A. "Memory resource management in VMware ESX server." ACM SIGOPS Operating Systems Review 36.SI (2002): 181-194.

[7] Grzegorz Miłós, Derek G.Murray, Steven Hand, Michael A. Fetterman. "Satori: Enlightened page sharing." Proceedings of the 2009 conference on USENIX Annual technical conference. USENIX Association, 2009.

[8] Arcangeli, Andrea, Izik Eidus, and Chris Wright. "Increasing memory density by using KSM." Proceedings of the linux symposium. 2009.

[9] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese. "Difference engine: Harnessing memory redundancy in virtual machines." Communications of the ACM 53.10 (2010): 85-93.

[10] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy. "Memory buddies: exploiting page sharing for smart colocation in virtualized data centers." Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. ACM, 2009.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris. "Xen and the art of virtualization." ACM SIGOPS Operating Systems Review 37.5 (2003): 164-177.

[12] Avi Kivity, Yaniv Kamay, Dor Laor. "kvm: the Linux virtual machine monitor." Proceedings of the Linux Symposium. Vol. 1. 2007.

[13] Christopher Clark, Keir Fraser,Steven Hand, Jacob Gorm Hansen, Eric Jul. "Live migration of virtual machines." Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2. USENIX Association, 2005.

[14] Saurabh Agarwal. Rahul Garg, Meeta S. Gupta, Jose E. Moreira. "Adaptive incremental checkpointing for massively parallel systems." Proceedings of the 18th annual international conference on Supercomputing. ACM, 2004.

[15] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft. "Scalability, fidelity, and containment in the potemkin virtual honeyfarm." ACM SIGOPS Operating Systems Review. Vol. 39. No. 5. ACM, 2005.

[16] Sharma, Prateek, and Purushottam Kulkarni. "Singleton: system-wide page deduplication in virtual environments." Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing. ACM, 2012.

[17] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand. "KSM++: Using I/O-based hints to make memory-deduplication scanners more efficient." RESoLVE.2012

[18] Ishan Banerjee, Fei Guo, Kiran Tati, Rajesh Venkatasubramaninan. "Memory Overcommitment in the ESX Server." VMWARE TECHNICAL JOURNAL: 2, 2013

[19] Chiang, Jui-Hao, Han-Lin Li, and Tzi-cker Chiueh. "Working Set-based Physical Memory Ballooning."

[20] Abel Gordon, Michael R. Hines, Dilma da Silva, Muli Ben-Yehuda, Gabriel Lizarraga. "Ginkgo: Automated, application-driven memory overcommitment for cloud computing." Proc. RESoLVE (2011).

[21] Al Muller, Seburn Wilson. "Virtualization with VMware ESX server." Rockland, Ma: Syngress Publ. Inc, 2005.