International Journal of Grid Distribution Computing
Vol.7, No.4 (2014), pp.65-80
http://dx.doi.org/10.14257/ijgdc.2014.7.4.07

Overcommitting Memory by Initiative Share from KVM Guests

Wan Jian, Du Wei, Jiang Cong-Feng and Xu Xiang-Hua

Department of Computer Science, Hangzhou Dianzi University, Hangzhou, 310018
wanjian@hdu.edu.cn, duwei_1987@sina.cn, cjiang@hdu.edu.cn, xhxu@hdu.edu.cn

Abstract

In virtualized environments, physical resources are abstracted as resource pool for provisioning, services consolidation, and service on demand. Traditional virtual machines are often over-provisioned to provide peak performance guarantees and thus waste a lot of memory resources. In this paper we propose a novel Initiatively Share Based Memory Overcommitting scheme, namely, IMR, to improve the memory utilization further. In IMR, guest virtual machine kernels are hacked with initiatively share functionality such that the guests can cooperatively share unused memory with the hypervisor, and consequently some of the possible page swapping and merging in such virtual machines are eliminated. The experimental results in shadow page tables and extended page tables scenarios show that our IMR approach not only improves the memory utilization but also has microsecond-level time overheads for shared memory reclaim, which outperforms the memory optimization schemes such as millisecond-level KSM (Kernel Same-page Merging) in best cases and millisecond-level page swapping at average.

Key words: *KVM; virtual machine; memory; overcommit; reclaim; initiative share*

1. Introduction

In order to improve the computational efficiency and facilitate system administrators, cloud computing has proposed a new service mode concept of IaaS [1](Infrastructure as a Service) that provides various computational resources to the end users via the internet on the basis of virtualization technology. By simulating hardware and isolating different virtual environments, virtualization technology enhances the compute density significantly for light load servers, whereas the resource overcommit strategies [2, 3] can further the enhancement.

There are many categories of virtualization, full virtualization simulates all the necessary hardware components with software, it runs unmodified guest operating systems, and switches the guests' privileged instructions into safe ones on the virtualization layer; paravirtualization [4] increases the performance by modifying the guest kernel or adding specific drivers, so that the guests are cooperative; the hardware assisted virtualization introduces hardware virtualization instructions to take the place of some time-consuming translations in full virtualization or paravirtualization; operating system level [5] virtualization utilizes the physical hardware by sharing the host kernel among all the guests, meanwhile limiting the guests in fixed operating systems.

For the memory aspect, besides the operating system level virtualization, the hypervisor must comprehend the detailed information for better overall memory overcommits efficiency. Because of this information gap, the coordination between the host and the guest is properly unsuitable, sacrificing either the host or the guest.

In this paper, a communication channel for guest memory information is established on the hypervisor, it handles the guests' memory demand selectively, reclaims unused guest memory transparently to the host for redistribution, and reduces some inappropriate disturbance to the

ISSN: 2005-4262 IJGDC

schedule of memory management unit on the host, as well as decreasing the overhead of swap and memory merging, thus reduces the valuable I/O [6, 7] and CPU usage.

Section 2 introduces the related works of memory virtualization. Section 3 shows the detailed design and the comparison to swap and KSM. Section 4 gives out the related experiments to illustrate the model described in section 2. Section 5 is the overall conclusion.

2. Related Works

2.1. Memory Management in KVM

KVM is a kernel module in Linux that offers core functions to guest operating systems such as memory management. It allocates a virtual address space for the guest without distributing actual physical memory, just like ordinary user processes. KVM platform allows memory overcommitting via the following strategies for better memory efficiency.

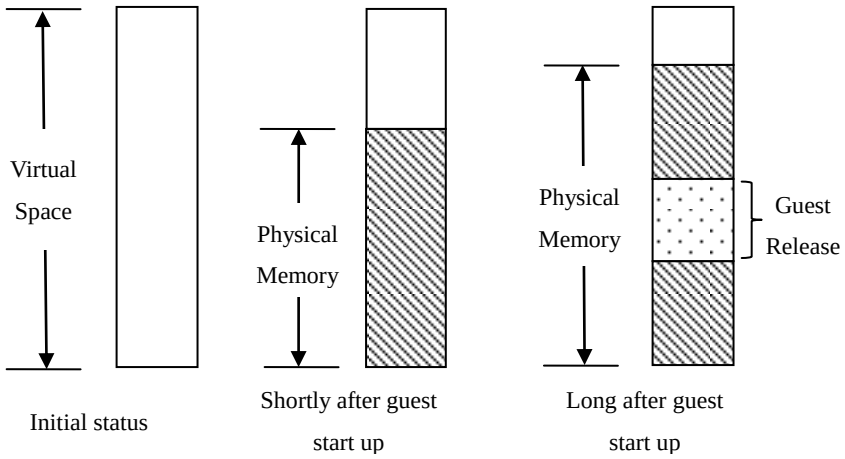


Figure 1. KVM Guest Memory Status without Overcommitting Strategies

As Figure 1 shows, the corresponding physical memory of guest processes will never decrease if none of these following strategies works.

(1) Swap: Triggered by memory pressure on the host. It swaps out the least recently used guest pages to disks. The memory pressure on the host will be relieved within a short period of time by occupying the I/O channel. The host may misunderstand the guest memory usage and get extra overhead as a consequence.

(2) Memory ballooning [8]: The host instructs guest balloon driver to allocate guest memory for host release. This process is done manually because the amount of memory can hardly be decided automatically and precisely, excessive release may cause side-effects on the guest.

(3) KSM [9]: Scans guest memory for identical pages with a kernel thread, but it's time-costly jobs to do merge and recover on next write. Only those guests abundant in duplicated memory benefit obviously. The memory burden release result mostly depends on guest memory content.

2.2. Memory Management Status on Other Platforms

VMware ESX Server (vSphere) makes memory sampling for guests, and reclaims guests' unused memory by free memory tax algorithm.

IBM z/VM platform modifies guest kernel, records complex memory status and can also reclaim unused guest memory [10], but sometimes the hypervisor has to obey guest memory management schedule to some extent.

OpenVZ is an operating system level virtualization solution. It separates different virtual environments within a single host kernel, thus unifies memory management and achieves high efficiency.

2.3. Analysis on Memory Virtualization

Memory virtualization can be divided into two types according to how the hypervisor knows the guests: estimated or accurate. If the hypervisor cannot distinguish exact memory status on the guest and only knows the visit frequency of guest pages, it is the estimated type, otherwise it is the accurate type.

KVM and VMware ESX Server (vSphere) belong to the estimated type memory virtualization which knows little about guest memory page status, take KVM for example, it regards guests as normal processes [11], when some guest pages are released, yet the host still regard them as recently visited ones, some less frequently used pages may be swapped out before those unused pages. Even if the estimation is correct, the swap process of unused guest pages may still cause disk write operations.

z/VM and OpenVZ belong to the accurate type memory virtualization which knows accurate guest memory page status by modifying the guest kernel or share the host kernel to all guests. The disadvantage is sometimes the host has to listen to the guest, say if many guests use mlock system call to lock too much memory synchronously on the host, the total sharable memory decreases.

To keep away from those drawbacks mentioned above, a new strategy which can reclaim unused guest memory in KVM is designed in this paper. It has both accurate type and estimated type characteristics and handles only the unused memory pages accurately, leaving the rest guest memory in the original KVM memory management routine. This mechanism relieves the host memory burden by letting guests share unused memory initiatively, then reclaim them on the hypervisor when necessary to avoid swap and KSM, here we name it IMR (Initiatively-shared Memory Reclaim). In addition, the performance of being used guest memory won't be affected. The comparison between IMR and other existing strategies is shown in Table 1.

Table 1. Comparison about Memory Virtualization Features

Platform	How hypervisors know guest memory	How guests share memory
KVM, vSphere	Estimated	Passive
z/VM, OpenVZ	Accurate	Initiative share or occupy first, then passive
KVM with IMR	Half-estimated, half-accurate	Initiative share first, then passive

3. IMR Design

3.1. Design Principle and Framework

Figure 2 shows IMR guest memory status transition. The status name consists of two letters, the first letter 'Y' or 'N' represents whether a page is being used or not, the second letter 'h' or 'g' represents the view from host kernel or guest kernel. Label means the first time a memory page is being used, label means a guest memory page is released or reused in the guest, label and means to allocate guest physical pages in guest balloon driver, then release them in the hypervisor. Label ⑤ represents the way IMR free unused guest memory pages.

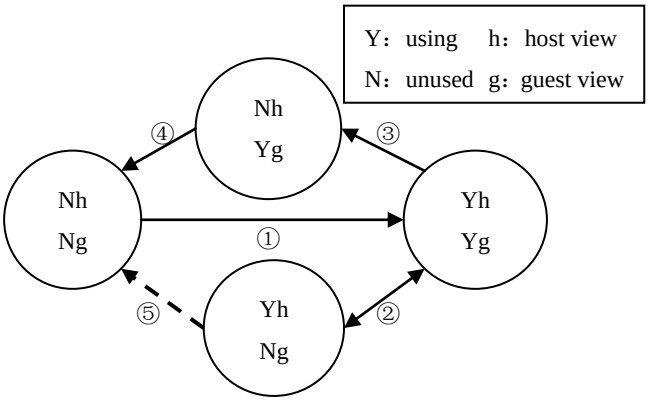


Figure 2. KVM Memory Status Transition

In this paper, IMR takes Linux and KVM as its foundation. Figure 3 shows the framework of IMR that the unused guest memory may go directly to the free memory list in host memory management unit without disturbing its swap or KSM algorithm. In this framework, the unused guest memory is separated from the rest currently being used memory for reclaim, as if the guest initiatively shares some memory with the host.

In order to realize the memory control flow described in Figure 3, some detailed work must be done to guarantee the reliability and performance:

(1) Guest kernel records the location of newly released physical memory.
(2) Take the execute efficiency into consideration, the guest only records buddy system [12, 13] memory blocks big enough for the handler in KVM. If a recorded block is reused, its previous record will be revoked.
(3) Mark after the guest releases physical memory; remove the mark before the guest uses physical memory in order to keep the consistency.
(4) Release the physical memory according to the record and then erase the record immediately.
(5) Continue memory slot loop so that the hot-plug feature of KVM memory is supported.
(6) If the host modification is also done on the guest, nested virtualization can be supported.

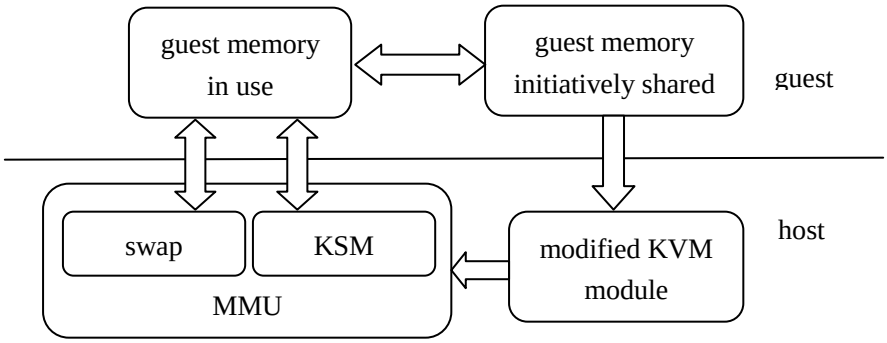


Figure 3. IMR Framework

Before the physical memory release, IMR should use down_write() to protect the memory range. For security concern, it should be zeroed before the release process. Then flush TLB (Translation Lookside Buffer), and finally use up_write() to exit the operation protection. These operations help ensure the compatibility with other strategies like swap and KSM.

Since these operations are made up of many synchronized instructions, adjacent memory blocks could be processed together to improve the performance. For 64-bit host operating systems, the max bit number that represents continuously-handled blocks is set to 64 for faster record analysis. The pseudo code is as follows.

```
start ← end ← flag ← 0            //Initialization
while start,end < total_length        //Set total guest memory limit
      for i ← 0 to 63          //Loop within 64 bits
            if flag = 1
                  end ← find next 0     //Find first bit 0 after flag set to 1
            else
                  start ← find next 1     //Find first bit 1(bit 1 for release)
                  end ← find next 0     //Find first bit 0
      if i < 63
            record(start,end)       //Locate continuous bit 1
            flag ← 0
      else          //Number of continuous bit 1may exceed 64
            flag ← 1
      get next 64 bits
if flag = 1
      record(start,end)       //Locate continuous bit 1 at the end
```

If the basic record block is small, for example, only 0.5MB, this algorithm may increase the analysis performance significantly.

After this procedure, it begins to release the corresponding guest memory with specific protection, and then it seeks for another memory slot and does the same operations.

Besides, the modified KVM module also has the right to decide the reclaim ratio according to the memory pressure on the host system.

There are totally 4 privileges on x86 architecture. For traditional operating systems, level 0 is used for kernel programs, level 3 is for user space programs. With virtualization technology, level 1 is developed for guest nowadays, it not only prevents data access from user space programs, but also allows the level 0 host kernel to visit the guest kernel easily. Because of this, the design of this paper establishes the communication channel with physical memory space allocation on the guest kernel, and it is the DMA zone area, to be exact.

3.2. IMR Performance Modeling

3.2.1. Comparison with Swap

Swap is a basic Linux memory schedule mechanism, it swaps out the overall least frequently used (LRU) memory pages to disks. Since KVM guests are already Linux user space processes, nothing needs to be changed in the KVM hypervisor for it to work. IMR may help reduce the swap pressure for directly releasing some unused guest pages ahead of the host reaches memory pressure.

Table 2 represents the comparison between swap and IMR. It is apparent that the IMR solution can help increase the overall performance for avoiding disk operations in swap mechanism.

Table 2. Comparison between Swap and IMR

Situation	Host operation	Guest page fault
Swap	Write to disk, release physical memory	Read from disk, establish page table
IMR	Release physical memory	Establish page table

Use $T_{freemem}$ to denote single physical memory page release time, T_{wmem} denotes single memory page write time, T_{rdisk} denotes disk read time for a single page at page fault, T_{wdisk} denotes disk write time for a single page. Use prefix T_e as the average extra time cost for generating a new free page, then the extra time cost for swap and IMR are $T_{e(swap)}$ and $T_{e(IMR)}$, page table creation at swap is $T_{e(paging)}$. Although IMR also establish page table, it's not the overhead because it doesn't belong to the previous pages. Then:

$$T_{e(swap)} = T_{wdisk} + T_{freemem} + T_{rdisk} + T_{e(paging)} \quad (1)$$

$$T_{e(IMR)} = T_{wmem} + T_{freemem} \quad (2)$$

T_{rdisk} and $T_{e(paging)}$ in formula 1 is subsequent cost of swap, which doesn't influence $T_{e(IMR)}$ in formula 2. Since disk operations far outweigh that of memory operation, disk read and write can be consolidated as T_{disk}. Now $T_{e(swap)}$ is:

$$T_{e(swap)} = 2T_{disk} + T_{freemem} + T_{e(paging)} \quad (3)$$

According to the design of modern operating systems, guests seldom visit unused raw physical memory that this case can be ignored here, even if this happens, it would be handled properly. IMR mostly only need to worry about page fault on write. For exact time cost in disk operation, memory write and physical memory release in Linux kernel, please refer to the data in the next chapter and the conclusion is $T_{e(swap)}$ is far more than $T_{e(IMR)}$ (including memory write time).

3.2.2. Comparison with KSM

KSM strategy utilizes a host thread to scan guest memory for same pages. No matter how much memory pressure it is on the host, the merging thread always run at certain CPU cost,

meanwhile wasting this overhead if guest memory doesn't have too much duplication or the merged area is soon written to produce extra COW(copy on write) operations. KSM takes more CPU cost than IMR and suits duplicated guest memory scenarios best. In a case in reference [9], it takes 10% of one CPU core. (It depends on settings like sleep and page_to_scan options, in this case, sleep=5000, pages_to_scan=60)

Table 3 shows their comparison.

Table 3. Comparison between Swap and IMR

Memory overcommit strategy	KSM	IMR
CPU cost	Big[9]	Tiny
Processing speed	Slow[9]	Very fast
For unused and duplicated	Merge, COW	Reclaim
For using and duplicated	Merge, COW	Idle
For unused and non- duplicated	Scan blindly	Reclaim
For using and non- duplicated	Scan blindly	Idle

Due to the different emphasis between IMR and KSM, the only comparable point occurs at dealing with unused duplicated guest memory.

To achieve similar guest memory reclaim amount, let the guest start with the same unused and duplicated pages for them. IMR's extra cost is still $T_{e(IMR)}$, $T_{e(ksm)}$ denotes that of KSM. Besides $T_{freemem}$, $T_{e(ksm)}$ must take merge and COW cost into account. T_{merge} denotes its merge time, and T_{cow} which happens with chance $P_{cow}(0 \leq P_{cow} \leq 1)$ denotes the copy on write overhead, so:

$$T_{e(ksm)} = T_{freemem} + T_{merge} + P_{cow}T_{cow} \quad (4)$$

T_{merge} is at least millisecond-level, the rest two are nonnegative, so the conclusion is even in the most duplicated guest memory environments that KSM prefers, $T_{e(ksm)}$ is still far more than $T_{e(IMR)}$, even though the performance in higher versions of KSM has improved. For detailed data, please refer to the next chapter.

3.2.3. Comparison with Memory Ballooning

Memory ballooning not only deploys its code in the hypervisor, but also need to install balloon driver on the guest. No matter how much memory pressure the guest suffers, it tries to release the user defined quantity of guest memory by the decision of guest kernel. Memory ballooning is less automatic and accurate than IMR, though it is perfect inside the guest.

Table 4 shows the comparison between memory ballooning and IMR.

Table 4. Comparison between Memory Ballooning and IMR

Memory overcommit strategy	Memory ballooning	IMR
Automation	Manual	Automatic
Accuracy	Precise inside guest, especially good for cache and buffer, but may lead to guest swap for over ballooning	Release only unused guest memory, never clear cache or buffer unless guest itself release them

4. Experiments and Analysis

As mainstream memory benchmarks care only in-use memory and macro scale swap experiment can be interfered easily, this paper mainly focus on the micro view of guest memory page activities, only section 4.2.6 performs macro-scale memory experiments. The following experiments mainly care about the micro variables defined in chapter 3.

4.1. Experiment Environment

Table 5 shows the basic hardware and software configuration for the experiment.

Table 5. Experiment Configuration

Virtual page table	Shadow page table	EPT
CPU	Intel E8400	Intel i5 3550
Memory	2GB DDR3-1066	16GB DDR3-1600
Disk	320GB 7200rpm(16MB cache)	1TB 7200rpm(64MB cache)
Guest memory	1GB	1GB/2GB
KVM version	0.14.1	1.0
Host OS	Ubuntu 11.10(kernel:3.5.3)	Linux Mint 13(kernel:3.4.2)
Guest OS	Ubuntu 11.10(kernel:3.6.1)	Debian 6(kernel:3.4.2)

(Note: The 2GB guest memory configuration is used for the macro scale experiment in section 4.2.6 only)

Shadow page table and EPT [14] (extended page table) environments are tested to demonstrate the difference could be ignored in comparisons with swap and KSM.

4.2. Experiment Scheme and Analysis

- Experiment 1: Anonymous memory usage analysis
  Goal:
  1. Anonymous memory usage in several mainstream open source software;
  2. Anonymous memory usage in calculation and database benchmarks;
  3. Analysis of the suitable situations of IMR.
- Experiment 2: Memory usage comparison with original KVM platform
  Goal:
  1. Extra time cost for establishing page tables on page fault;
  2. Extra time cost for in-memory data read and write operations.
- Experiment 3: Comparison with swap and KSM
  Goal:
  1. Time for disk read and write;
  2. Swap impact on CPU I/O wait;
  3. Time for page merge in KSM.
- Experiment 4: IMR performance test and analysis
  Goal:
  1. Extra mark time in guest kernel;
  2. Memory reclaim time of IMR;
  3: Write speed of host physical memory;
  4. Guest external memory fragment influence.
- Experiment 5: IMR's macro scale impact on memory
  Goal:
  1. Physical memory and swap space usage in multiple virtual machines' startup and dynamic memory utilization;

2. Proof of overall optimization with IMR.

4.2.1. Anonymous Memory Usage Analysis

Gedit, Firefox and LibreOffice Write are selected for this experiment as shown in table 6 without loading any file, although they are implemented in C, C++ and java respectively, and the underneath libraries are different, they share a common feature that large proportions of memory space are anonymous, though anonymous virtual space are not always fulfilled with data.

Table 6. Anonymous Memory Usage in Mainstream Applications

Application	Gedit	Firefox	LibreOffice Write
Anonymous virtual memory(KB)	30856	192312	81336
Total virtual memory(KB)	72864	275120	208728

The three applications' initial anonymous memory percentages are 42.35%, 69.90% and 38.98%. Anonymous memory is not involved in mmap system call that loads files, it is typically used as temporary storage for calculation. For example, an anonymous memory space is allocated by malloc(), it becomes useless after the compute procedure, then use free() to get released, what this paper emphasize is such cases on the guest. To better understand anonymous, buffer and cache memory usage, experiments via sysbench tool are given as below.

Figure 4 represents the memory usage in sysbench CPU test:

sysbench --test=cpu --cpu-max-prime=10000 run

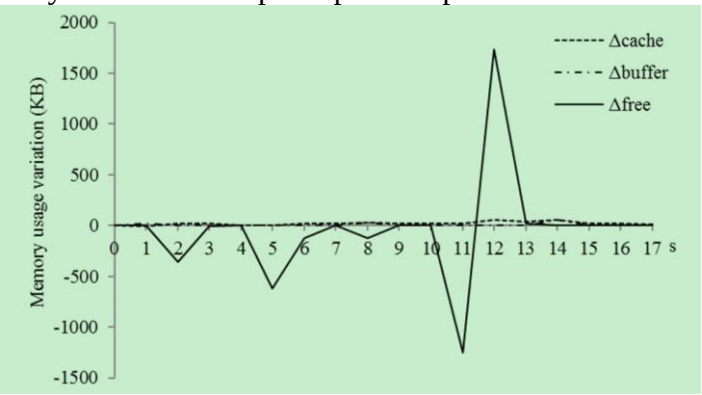


Figure 4. Memory Usage in Calculation

The actual calculation procedure ranges from 4th second to 12th second, the free available memory decreases for anonymous memory usage since Δfree is below zero and changes in Δcache and Δbuffer are little. Immediately after the experiment ends, free memory experiences a quick growth, this is achieved by the anonymous memory release. However, other experiments like the database case mainly focus on file retrieval, a lot of memory is used for buffer and cache to cooperate with file system.

Table 7 shows a typical memory status change among free, buffer and cache memory by the command:

sysbench --test=oltp --mysql-db=test --mysql-user=root --mysql-password=000000 --oltp-table-size=10000 run

From line 1 to line 2, buffer and cache increase 3600KB, while free memory increases 5680KB, it means 9280KB physical memory are released; from line 2 to line 3, free memory increases 4296KB, yet buffer and cache decrease 5080KB, it means during this second, 784KB physical memory are occupied after buffer and cache shrink.

Table 7. Memory Usage in Database OLTP Test

Time sequence(s)	Free(KB)	Buffer(KB)	Cache(KB)
n	85980	678804	893680
n+1	91660	680480	895604
n+2	95956	678868	892136
n+3	103100	679396	884992

4.2.2. Memory Usage Comparison with Original KVM latform: Write 10^8 random integers($4*10^8$ bytes) into guest virtual memory space, then sort them with sort() in C++ standard template library, record random number write to test establishing page tables on page fault and record sort time to test in-memory operations.

Table 8 shows the results, to divide the write time difference between reclaim and not reclaim by fault page number, the result is the extra time cost for establishing page tables on page fault.

Table 8. Basic Memory Operation Time

Memory test	Shadow page table			EPT		
	First time	Later		First time	Later	
		Original	IMR		Original	IMR
Random number write(s)	2.7	2.1	2.6	1.55	1.39	1.54
Establish page tables(μ s)	6.1	0	6.1	1.6	0	1.6
Sort(s)	24.56	24.57		21.47	21.48	

There is a big performance gap between shadow page table and EPT, nevertheless, they are microsecond-level operations and its impact can be neglected since swap and merge operations are millisecond-level, so the remaining experiments will use the shadow page table environment by default without special statement, and EPT environment is used in macro scale experiments. Besides, it can be concluded from sort experiment that IMR almost doesn't affect in-memory operations.

4.2.3. Comparison with Swap: Probably because of the hardware disk cache size, the average disk read time on page fault (T_{disk}) on shadow page table and EPT environment are 6 milliseconds and 2 milliseconds respectively. IMR may help reduce anonymous page swap and increase the compute density by avoiding too much I/O wait on CPU. Figure 5 is a typical swap's impact on CPU, the system begins to swap at 7th second and ends at 42nd second, during the swap process, "iowait" takes about half CPU time, which means a huge waste.

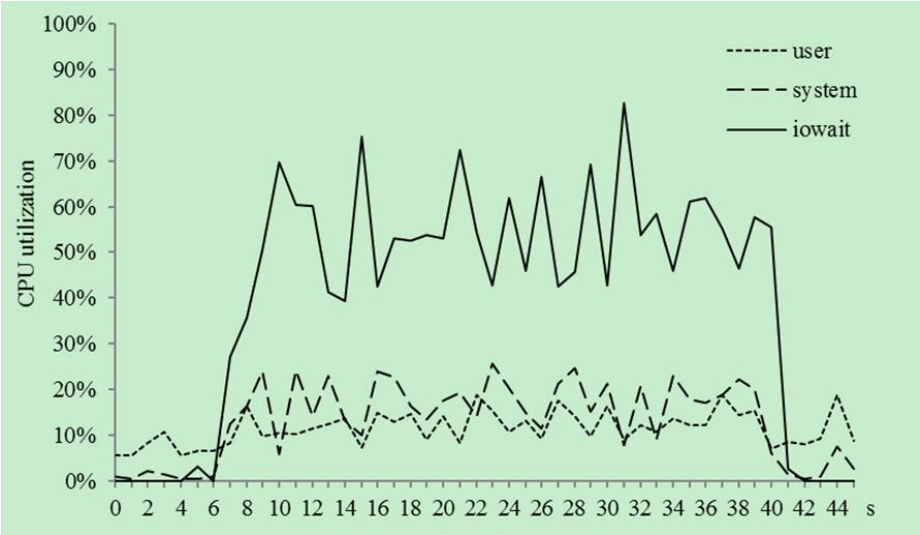


Figure 5. Swap Impact on CPU

4.2.4. Comparison with KSM: Fill the guest memory with completely duplicated or random data, though an ordinary application won't encounter such situations, it can be predicted to have some medium results.

The amount of test memory is still set to $4*10^8$ bytes (about 381.5MB). To ensure KSM merge not affected by other existing guest memory, all the address space are written with character 'a' first to let the merge thread work and shrink all guest memory duplications, when the merge process ends, refill the $4*10^8$ bytes space with another character 'b' to make the reliable merge.

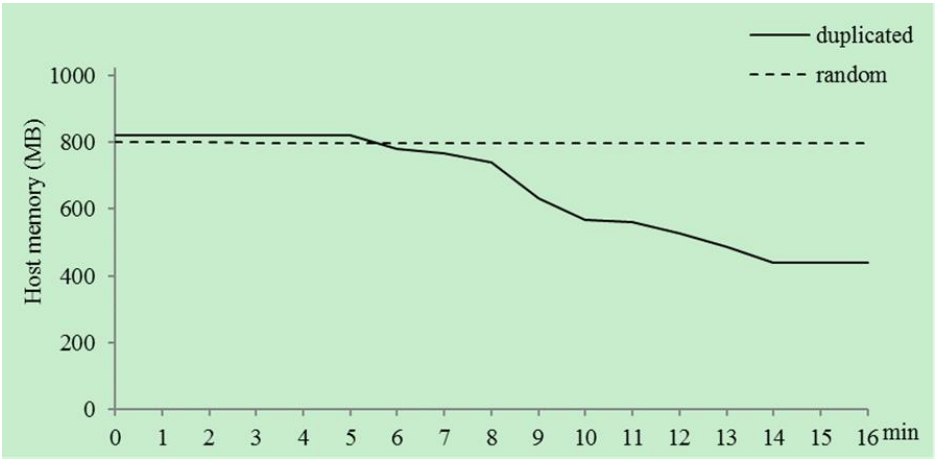


Figure 6. KSM Merge Effect

As shown in Figure 6, in the ideal case where all test data are duplicated, KSM merges 380.9MB (99.85%) by 14 minutes, so the average merge time (T_{merge}) is 8.6 milliseconds, whereas the random data case merges almost nothing (0.07%). Although KSM has improved greatly, notable CPU cost and failures in merging random data are always inevitable because KSM cannot predict if the data can be merged until it reads the last byte of every memory

page, in this test, the average CPU cost is 1% under default settings(sleep=20000, pages_to_scan=100).

4.2.5. IMR Performance Test and Analysis

The amount of duplicated and random test memory is still $4*10^8$ bytes, and the control block is set to 4MB. The extra mark time in guest kernel is tested to be less than 10 nanoseconds for each memory block record and could be ignored directly in this experiment. The test result is shown in Table 9. In the most duplicated and random case, IMR host reclaims 89% and 92% of guest-release memory. Divide the duration time by total reclaim page number, $T_{e(IMR)}$ is 0.56 microseconds. The host physical memory write speed is tested by sysbench as 7218.89 MB/s, then T_{wmem} for each page (4KB) is 0.54 microsecond, so the rest $T_{freemem}$ is about 0.02 microsecond.

Table 9. IMR Reclaim Efficiency

Data type	Duplicated data	Random data
Memory usage before release(MB)	865	882
Memory usage after release(MB)	525	530
Duration(s)	0.049	0.051

Table 10. Impact of Memory Control Block Size

Memory control block size(MB)	4	2	1	0.5
Before reclaim(MB)	882	879	876	887
After reclaim(MB)	530	511	501	509
Reclaim percentage	92.3%	96.5%	98.3%	99.1%

Because of Linux external memory fragment, IMR is unable to reclaim all guest-release memory. On the one hand, smaller memory control block can efficiently reduce its impact like the results shown in table 10; on the other hand, small blocks of guest memory will likely be used later.

4.2.6. IMR's Macro Scale Impact on Memory

To better understand how IMR helps reduce the overhead, the following experiment steps are designed to demonstrate IMR's advantage:

(1) Start up 15 KVM guests with the second configuration in Table 5 (EPT, 2GB) to login screen one by one.
(2) Then allocate and fill each guest with 1GB random data and release the memory inside guests immediately.

The results can be seen from Figure 7 and Figure 8 that KVM with IMR reduces about 25% physical memory usage in the guest system startup case (without IMR, a guest uses about 120MB memory; with IMR, it costs about only 90MB), and reclaims almost all anonymous free guest memory pages in dynamic guest memory usage experiment. The memory growth may not be linear because of KSM's effect.

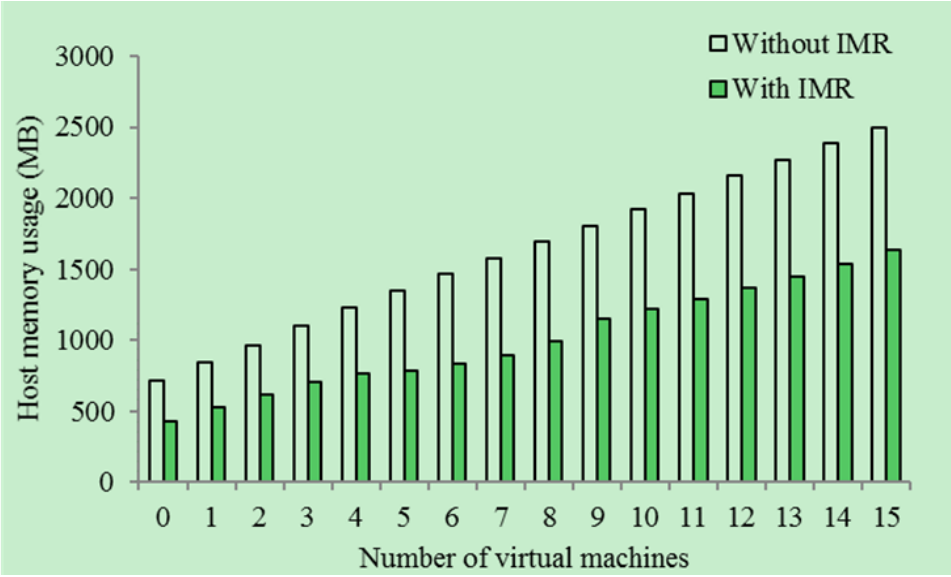


Figure 7. Memory Usage in System Startup

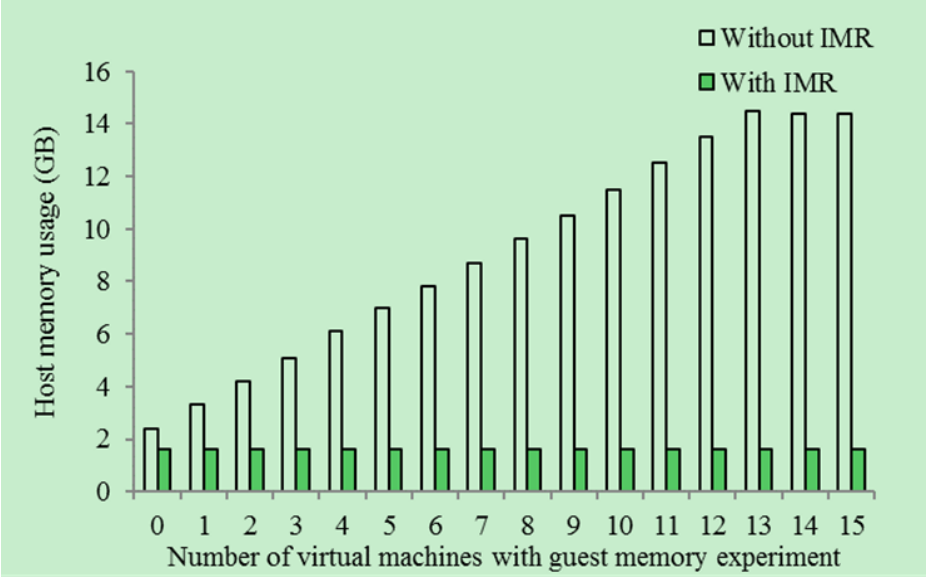


Figure 8. Dynamic Guest Memory Usage

In the case without IMR, the last two allocations are achieved after the virtual machine manager shuts off the third and the sixth virtual machines automatically. The gap between physical memory growth and the guest allocation is covered by swap mechanism. Detailed statistics is shown in Figure 9 that within 487MB swap space, IMR can effectively reclaim unused guest memory without using any swap space, so the data of the "with IMR" category is always 0 for too low memory pressure.

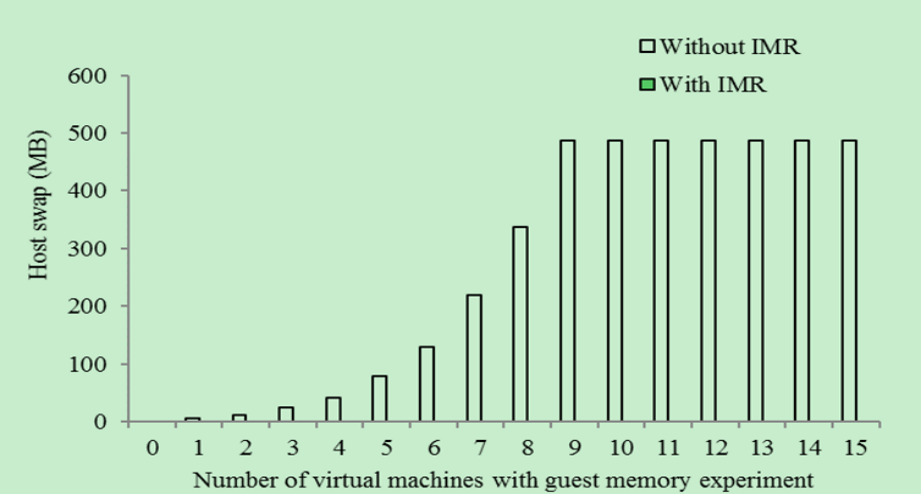


Figure 9. Swap Situation in Dynamic Guest Memory Usage

5. Conclusions

Currently there are some solutions for memory management, especially for memory operation efficiency improvement, such as page swapping, KSM, memory ballooning, *etc*. However, these solutions also have some weaknesses in various aspects.

For example, page swapping introduces additional I/O overheads and is not suitable for highly contended virtualized environments. Memory ballooning are neither automatic nor accurate and can also result in page swapping in guest virtual machines. KSM utilizes notable CPU resources and is time-consuming. Its performance in memory pressure decrease depends on the quantity of same pages in guests. In summary, the existing solutions do not consider the total memory efficiency on the host.

In this paper we propose a novel Initiatively Share Based Memory Overcommitting scheme, namely IMR, to improve the memory utilization further. In IMR, guest virtual machine kernels are hacked with initiatively share functionality such that the guests can distinguish the unused physical memory and cooperatively share unused memory with the hypervisor, and consequently a number of the possible page swapping and merging in such virtual machines are eliminated. The experimental results in shadow page table and EPT scenarios show that our IMR approach not only improves the memory utilization but also has microsecond-level time overhead for shared memory reclaim, which outperforms the memory optimization schemes such as millisecond-level KSM in best cases and millisecond-level page swapping at average. The reclaim of unused memory from virtual machines helps enhance the compute density and reduce the energy cost of the whole system.

The proposed IMR is more efficient, automatic and can be implemented both in cooperative and non-cooperative virtualized environments. Its compatibility is guaranteed with locks of certain memory regions before trying to release the related physical memory, and the performance sacrifice is tolerable. The IMR design also takes the memory hot-plug and nested virtualization scenarios into account for better availability.

Acknowledgements

The funding supports of this work by Natural Science Fund of China (Grant No. 61003077), Natural Science Fund of Zhejiang Province (Grant No. Y1101092, LY12E09009), Research Fund of Department of Education of Zhejiang Province (Grant No. GK100800010),

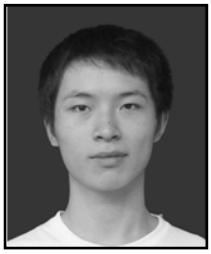
and Technology Research and Development Program of Zhejiang Province (Grant No. 2013C31115), are greatly appreciated.

References

[1] S. Bhardwaj, L. Jain and S. Jain, "International Journal of engineering and information Technology", vol. 1, no. 2, **(2010)**.
[2] J. Heo, X. Zhu, P. Padala and Z. Wang, "Memory overbooking and dynamic control of Xen virtual machines in consolidated environments", Proceedings of IFIP/IEEE Symposium on Integrated Network Management (IM'09), **(2009)** June 1-5, New York, USA.
[3] P. Padala, K. Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal and A. Merchant, "Automated control of multiple virtualized resources", Proceedings of the 4th ACM European conference on Computer systems. ACM, **(2009)** March 30-April 3, Nuremberg, Germany.
[4] X. L. Wang, Y. F. Sun, Y. W. Luo, Z. L. Wang, Y. Li, B. B. Zhang, H. G. Chena and X. M. Li, "Science in China Series F: Information Sciences, vol. 1, no. 53, **(2010)**.
[5] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors", ACM SIGOPS Operating Systems Review. ACM, **(2007)** March 21-23, Lisbon, Portugal.
[6] W. Zhao, Z. Wang, Y. Luo, "ACM SIGOPS Operating Systems Review", vol. 3, no. 43, **(2009)**.
[7] B. B. Zhang, X. L. Wang, L. Yang, R. F. Lai, Z. L. Wang, Y. W. Luo and X. M. Li, "Jisuanji Xuebao (Chinese Journal of Computers)", vol. 12, no. 33, **(2010)**.
[8] C. A. Waldspurger, "Memory resource management in VMware ESX server", ACM SIGOPS Operating Systems Review. SI, 36 **(2002)** December 9-11, Boston, USA.
[9] A. Arcangeli, I. Eidusa and C. Wright, "Increasing memory density by using KSM", Proceedings of the linux symposium, **(2009)** July 13-17, Montreal, Canada.
[10] M. Schwidefsky, H. Franke, R. Mansell, H. Raj, D. Osisek and J. Choi, "Collaborative memory management in hosted linux environments", Proceedings of the Linux Symposium, **(2006)** July 19-22, Ottawa, Canada.
[11] I. Habib, "Linux Journal", vol. 166, **(2008)**.
[12] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Ceccheta and M. D. Corner, "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)** March 11-13, Washington, D.C., USA.
[13] M. Gorman, "Understanding the Linux virtual memory manager", Prentice Hall, **(2004)**.
[14] Advanced Micro Devices, Inc. AMD-V Nested Paging, White paper, http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf, **(2008)**.

Authors

Wan Jian, he received his Ph.D. Degree in Computer Science from Zhejiang University, China, in 1996. He is now a professor and dean of School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China. His research interest includes virtualization, grid computing, services computing, and wireless sensor networks.



Du Wei, he is now a master student in Hangzhou Dianzi University, Hangzhou, China, his major is computer science, and his current research interest is system virtualization.

Jiang Cong-Feng, he received his Ph.D. Degree from the Engineering Computation and Simulation Institute (ECSI), Huazhong University of Science and Technology (HUST), Wuhan, China, in 2007. He is now an associate professor in School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China. He is with the Grid and Services Computing Lab. His current research interests include virtualization, grid computing, and power aware computing system.

Xu Xiang-Hua, he received his Ph.D. Degree in Computer Science from Zhejiang University in 2005. He is now a professor in School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, China. His research interest includes virtualization, grid computing, services computing, and wireless sensor networks.

