Sha S, Hu JY, Luo YW *et al.* Huge page friendly virtualized memory management. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(2): 433–452 Mar. 2020. DOI 10.1007/s11390-020-9693-0

Huge Page Friendly Virtualized Memory Management

Sai Sha[1,2,3], *Member*, *ACM*, Jing-Yuan Hu[1], Ying-Wei Luo[1,2,3,*], *Member*, *CCF*, *ACM*
Xiao-Lin Wang[1,2,3], *Member*, *CCF*, *ACM*, and Zhenlin Wang[4], *Member*, *ACM*

[1] *School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China*

[2] *Peng Cheng Laboratory, Shenzhen 518052, China*

[3] *Shenzhen Key Laboratory for Cloud Computing Technology & Applications, School of Electronic and Computer Engineering, Peking University Shenzhen, Shenzhen 518000, China*

[4] *Department of Computer Science, Michigan Technological University, Michigan 49246, U.S.A.*

E-mail: {ss_boom, hujingyuan0303, lyw, wxl}@pku.edu.cn; zlwang@mtu.edu

Received May 7, 2019; revised October 14, 2019.

Abstract With the rapid increase of memory consumption by applications running on cloud data centers, we need more efficient memory management in a virtualized environment. Exploiting huge pages becomes more critical for a virtual machine's performance when it runs large working set size programs. Programs with large working set sizes are more sensitive to memory allocation, which requires us to quickly adjust the virtual machine's memory to accommodate memory phase changes. It would be much more efficient if we could adjust virtual machines' memory at the granularity of huge pages. However, existing virtual machine memory reallocation techniques, such as ballooning, do not support huge pages. In addition, in order to drive effective memory reallocation, we need to predict the actual memory demand of a virtual machine. We find that traditional memory demand estimation methods designed for regular pages cannot be simply ported to a system adopting huge pages. How to adjust the memory of virtual machines timely and effectively according to the periodic change of memory demand is another challenge we face. This paper proposes a dynamic huge page based memory balancing system (HPMBS) for efficient memory management in a virtualized environment. We first rebuild the ballooning mechanism in order to dispatch memory in the granularity of huge pages. We then design and implement a huge page working set size estimation mechanism which can accurately estimate a virtual machine's memory demand in huge pages environments. Combining these two mechanisms, we finally use an algorithm based on dynamic programming to achieve dynamic memory balancing. Experiments show that our system saves memory and improves overall system performance with low overhead.

Keywords virtualization, huge page, ballooning, memory balancing

1 Introduction

In this big data era, the memory demand of applications running on cloud data centers has been rapidly increasing, which puts more pressure on memory management especially in virtualized environments. One performance bottleneck due to a large dataset stems from a large number of translation lookaside buffer (TLB) misses. For memory-intensive applications with a large memory footprint, TLB cannot efficiently cache these translations due to its limited size. One solution to mitigate the overhead is to use huge pages [1], which can help significantly reduce TLB miss rate.

Transparent huge page (THP) is a simple adoption of huge pages in Linux kernel since version 2.6.38 [2]. The operating system (OS) automatically establishes a

Regular Paper
Special Section of ChinaSys 2019
The work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003604, the National Natural Science Foundation of China under Grant Nos. 61472008, 61672053 and U1611461, Shenzhen Key Research Project under Grant No. JCYJ20170412150946024, the National Science Foundation of USA under Grant No. CSR-1618384, and Beijing Technological Program under Grant No. Z181100008918015.

*Corresponding Author

huge page mapping by tagging the page table's page middle directory (PMD) entry as a huge page. This means that the virtual and real address conversion is reduced from four page tables to three, which improves the conversion efficiency and saves the amount of memory used by the page tables. Linux kernel uses, by default, a huge page size of 2 MB, which is 512 times as much as the regular page size. This means that the OS requires fewer address translation entries in TLB. The number of TLB misses is also greatly reduced. A study shows that up to 25% TLB miss reduction and 37.5% overall performance improvement can be achieved by using Linux THP for SPEC CPU2006 and PARSEC 3.0 workloads [3]. Especially for applications with working set size (WSS), the huge page mechanism can significantly improve system performance.

However, adopting huge pages brings new challenges in memory management for virtualized environments. Compared with the applications with small datasets, the performance of a large dataset application is highly relevant to the available memory size for a virtual machine (VM) where the application is running. For the applications with gigabytes of memory consumption, such as 631.deepsjeng_s, 649.fotonik3d_s in SPECCPU 2017①, and Graph in CloudSuite②, reducing a few megabytes of memory from the VM can degrade its performance several times as shown in Fig.1. This indicates that we need to adjust quickly once memory imbalance occurs in a multi-VM environment. Using huge pages allows a hypervisor to adjust memory quickly. Ballooning is an efficient method to adjust VMs' memory and has been used by many hypervisors such as KVM, Xen and VMware. However, previous ballooning mechanisms do not support huge pages. The ballooning process demotes the 2 MB huge pages to 4 KB regular pages for dispatching, which results in a significant performance loss. In addition, GB-level memory ballooning often takes several seconds to complete the dispatch. In order to solve this problem, we design and implement huge pages-based ballooning in KVM and QEMU.

Dynamic memory management needs to accurately predict the memory demand of each VM. OS uses a concept of WSS to represent memory demand. WSS was first defined by Denning [4], which represents the total number of memory pages accessed by a process during a certain period of time. We find that it becomes more challenging to estimate the WSS when huge pages are heavily used. To estimate the WSS, one common metric is page miss ratio curve (MRC), which maps memory size to page miss ratio. Mattson's stack algorithm [5] is usually used to construct an MRC. In the stack, each item records a page, and the stack distance (reuse distance) refers to the distance from an item to the top of the stack. The coverage of a huge page is much larger than that of a regular page. According to program locality, a sequence of memory instructions are more likely to access a same huge page than a same small page. Therefore, the proportion of small distance reuses will be very high in huge pages environments. Our experiments on SPEC CPU2017 workloads show that over 99.9% of reuse distances are smaller than 20. The reuse distance distribution in huge pages environments is more unbalanced. The unbalance leads to challenges in controlling memory tracking overhead and maintaining high miss ratio precision. We adopt a dynamic hot set to control overhead. Meanwhile, we analyze the impact of a hot set for MRC construction and propose a method to restore the accurate MRC based on a heuristic equation.

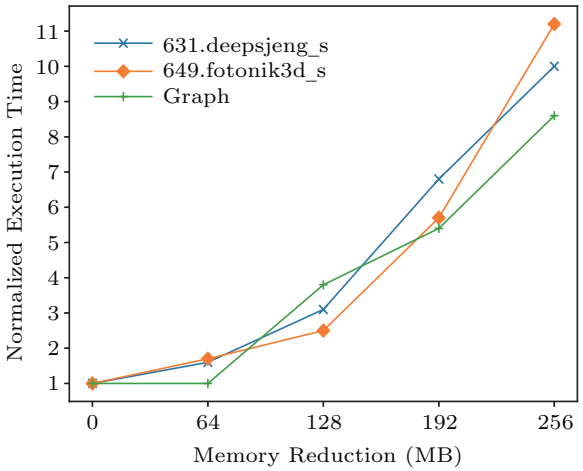


Fig.1. Performance degradation due to reduction of memory.

VM memory balancing based on regular 4 KB pages does not apply to programs with large WSSs. Combining the huge page ballooning and the huge page WSS estimation, we design an algorithm for memory reallocation based on dynamic programming. Our design goal is to adjust the memory of VMs in real time according to their memory phases. The algorithm attempts to ensure the performance while saving the overall memory as much as possible. In this paper, we propose a

①https://www.spec.org/cpu2017/index.html/, March 2020.
②http://cloudsuite.ch/, March 2020.

comprehensive huge page based virtual machine memory balancing mechanism in order to improve the memory utilization of multiple VMs running large working set programs. In particular, we make the following contributions:

• a huge pages based WSS estimation mechanism in a virtual execution environment, which guides VM memory balancing [5];

• an efficient ballooning mechanism based on huge pages which allows VM memory adjustment without huge pages demotion [6];

• a dynamic memory balancing scheme in a virtual execution environment, which integrates huge page based WSS estimation and huge pages based ballooning.

This paper is a summary and extension of [5] and [6]. Its main new contribution is to integrate the two technologies of [5, 6] and implement a memory balancing system with huge page in a virtualized environment.

The rest of the paper is organized as follows. In Section 2, we analyze WSS estimation based on huge pages and pay attention to swap handling. Section 3 details the ballooning mechanism and how we reconstruct it to support huge pages. Section 4 describes the architecture and algorithms of our huge pages based memory balancing system. Section 5 presents experimental results including the verification of the WSS estimation, huge page ballooning, and the whole memory balancing. Section 6 discusses related work and and Section 7 concludes the paper.

2 WSS Estimation Based on Huge Pages

We estimate the WSS of a VM by constructing an MRC. The MRC construction relies on accurately tracking page accesses of a workload or VM. However, most memory accesses from guest OSs are transparent to the hypervisor. Our approach to track memory accesses in a virtualized system is to modify the permission bits of the page table entry of a memory page. By revoking a page's access permission, next access to the page will trigger a page fault, which allows the hypervisor to extract access address and calculate reuse distance. Fig.2 shows an overview of LRU-based WSS estimation.

2.1 LRU-Based MRC Construction

We adopt Mattson's stack algorithm to construct an MRC at run time. The algorithm was initially proposed by Mattson *et al.* in 1970 [7] to reduce trace-driven processor cache simulation time. The main idea of Mattson *et al.*'s stack algorithm is to take advantage of the inclusion property in many cache/memory replacement algorithms such as the commonly used Least Recently Used (LRU) policy.

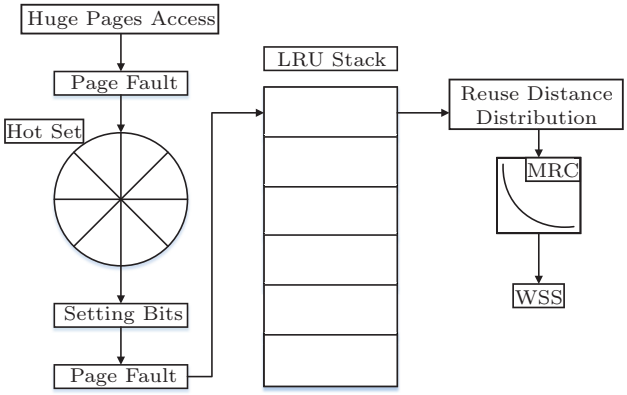


Fig.2. Overview of LRU-based WSS estimation.

In the LRU stack, each item holds the physical frame number (PFN) of a huge page. The size of the LRU stack is the number of huge pages that the current guest OS can allocate. Reuse distance (LRU distance) is the distance from the top of the stack to each item in the LRU stack. We build a reuse distance histogram (RDH), $hist$, which shows reuse distance distribution (RDD). In our design, each item i in the stack has a property $hist(i)$. When a page is accessed, we first look up the item in the stack to find its reuse distance d, and $hist(d)$ increases by 1. If the page is not in the stack, it is a cold miss and $hist(\infty)$ increases by 1. Given a memory size of s huge pages, the page miss ratio is calculated as $MISS(s) = \frac{\sum_{j=s}^{\infty} hist(j)}{\sum_{i=0}^{\infty} hist(i)}$. In other words, when the guest OS has s huge pages, its page miss ratio is $MISS(s)$.

2.2 Hot Set

The LRU-based method provides an accurate MRC, but it requires a full trace of memory accesses. It is prohibitive to intercept every memory page access as it will cause a lot of system overhead. One solution is to apply memory address sampling [8], which only monitors a small subset of addresses. Unfortunately, this approach does not work for huge pages. Compared with regular pages RDD, huge pages RDD changes significantly for the same VM or application. Most reuse distances become small distances due to spatial locality brought by huge pages. However, due to the precision requirement,

we have to track a sufficient number of long-distance reuses. Page sampling may lose track of some large reuse distances, which may result in a significant error in the WSS estimation. We instead adopt a hot set [9] to solve this problem.

We divide the memory pages into a hot page set and a cold page set. When a new page table is populated (e.g., when creating a new process), all pages are cold. In subsequent fetches, the system captures a cold page, records its address, and moves it to a hot set. The hot set is an FIFO queue with a limited size, which stores PFNs. The purpose is to keep the pages that are used frequently in the hot set, and only track them when they exit the FIFO queue so as to avoid tracking a large number of short-distance reuses. When a page is dequeued from the hot set, we modify its permission bits in the extended page table (EPT) to trigger a page fault for its next access. For large WSS applications, we are less interested in small reuse distances as they only help provide the miss ratios for a memory allocation bounded by the hot set size. As long as we do not use a hot set that is too large, the resulting MRC is still effective for the WSS estimation, which is verified in Subsection 5.3.

2.3 Restoring RDD and Constructing MRC

With a hot set, only the pages dropped from the hot set will be tracked, and then rejoin the hot set after its next access when its reuse distance is recorded. Apparently the tracked memory accesses sequence with a hot set is different from the sequence without one. We need to examine its impact on RDD and MRC.

Although the sequence of memory accesses changes, the distribution of long reuse distances remains the same. Fig.3(a) and Fig.3(b) show examples of the distributions of reuse distance with and without a hot set. Fig.3(a) presents the distribution with a hot set of 500 MB (250 huge pages), while Fig.3(b) presents the distribution without a hot set. The workload is 605.mcf_s in SPEC CPU2017. We simulate the complete benchmark execution with the Intel Pin③ toolset to obtain the memory trace and implement an LRU stack to collect its reuse distance distribution. Although the hot set filters most of short reuse distances, the rest part shows almost the same shape as the lower one without a hot set. We run other workloads in SPEC CPU2017 with the memory consumption of more than 2.7 GB and reach the same conclusion. Our approach thus assumes that using a hot set will not change RDD for the distances larger than the hot set size.

Fig.3(c) shows three types of MRCs: MRC without hot set (MRC), MRC with hot set (MRC_HOT), and MRC with hot set with modification. We can observe that the shapes of the MRCs are very close but the

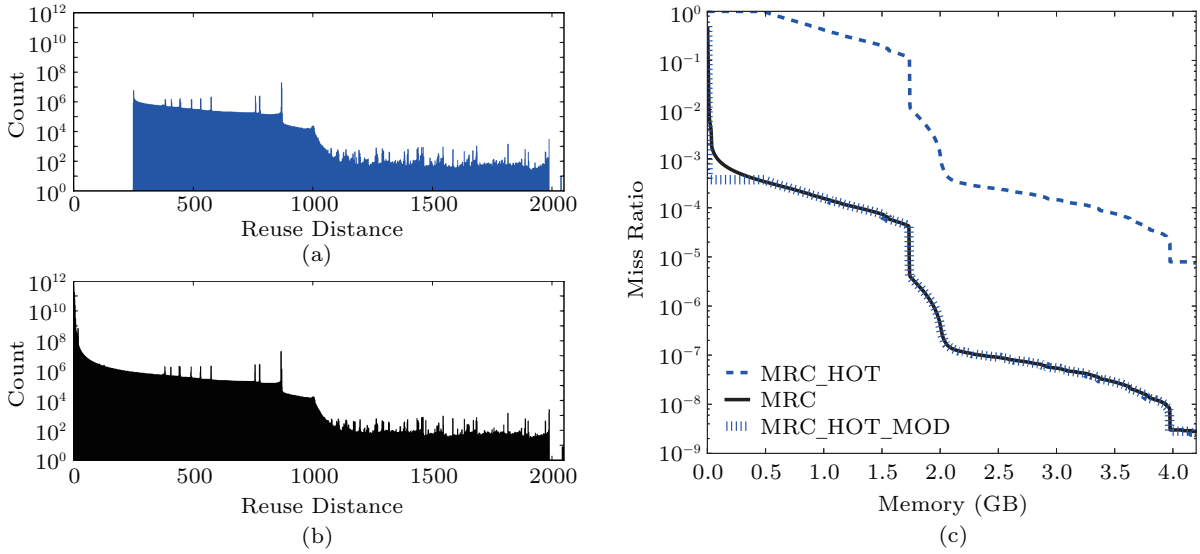


Fig.3. Comparison of reuse distance distributions and MRCs with and without a hot set using PIN simulation. (a) Reuse distance distribution with a hot set of 500 MB. (b) Reuse distance distribution without a hot set. (c) MRCs of 605.mcf_s. The y-axes in the figure are plotted in a logarithmic scale of 10, to present a complete view.

③https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, Feb. 2020.

MRC directly derived from the filtered reuse distance distribution overestimates the miss ratios. It is because the filtered reuse distances are short ones and typically result in cache hits. To deliver an acceptable MRC, we estimate the amount of filtered reuse distances and add them back to the reuse distance distribution.

Note that a memory access results in either a cold miss, if its location is first touched, or a reuse with a distance calculated through the LRU stack approach. The total number of reuse distances is $mem\_inst - cold\_miss$, where mem_inst is the total number of memory accesses and $cold\_miss$ is the number of cold misses. The cold miss count is the number of the distinct pages being tracked. Theoretically, the filtered memory access count, $filtered$, can be estimated as: $mem\_inst - mem\_tracked - cold\_miss$. $mem\_tracked$ is the memory accesses whose reuse distances we are able to track. We add the filtered part back to the reuse distance distribution. As we observe that 99.9% of huge pages reuse distances are less than 20, we make 99.9% filtered reuses evenly distributed from distance 1 to 20 and the rest 0.1% from 20 to the hot set size. Fig.3(c) shows the miss ratio curve, MRC_HOT_MOD, constructed from the restored reuse distance distribution. The new MRC is highly overlapped with the accurate MRC, expect for the memory size less than the hot set size. The less accurate portion of MRC_HOT_MOD will cause a minimal impact in practice as a system is rarely configured with an amount of memory less than the hot set size.

In our implementation in an Intel machine, the metric mem_inst can be determined using the Intel performance monitoring unit (PMU). We monitor the performance events MEM_INST_RETIRED.ALL_LOADS and MEM_INST_RETIRED.ALL_STORES and use the sum for mem_inst. Other two metrics are monitored by our memory tracking system.

2.4 Dynamic Hot Set (DHS)

The overhead of the memory tracking system is highly relevant to the hot set size, as it determines the number of page faults introduced by memory tracking. We find that using a fixed hot set size of 20 huge pages could generate page faults at 10-million level for each one hundred billion memory accesses. For the workloads in SPEC CPU2017 that are not sensitive to page faults, the total overhead is less than 5%. For the workloads sensitive to page faults, millions of page faults can cause a slowdown of 7 times. 631.deepsjeng_s in SPEC CPU2017 is one of this type. To further reduce the overhead, one solution is to adopt a dynamic hot set. During memory tracking, the hot set is re-sized depending on the number of memory instructions between two tracked adjacent page faults, denoted as Mem Tracked. Through experiments, we need to cap Mem Tracked to below a million to achieve an acceptable overhead.

Our DHS algorithm introduces two thresholds, $THmax$ and $THmin$, to help control the hot set. We set $THmax$ and $THmin$ to 10 million and 1 million, respectively.

These two thresholds are constantly modified in experiments with 649.McF_s, 619.Lbm_s, and 631.Deepsjeng_s programs separately. The reason for selecting these three programs is that their memory access characteristics contain many different features: 619.lbm_s is a single program with memory stages and good locality, 649.McF_s has poor locality and complex stages, and 631.Deepsjeng_s is a program with a single stage but poor locality.

The default hot set size is 500 MB (250 huge pages). During the memory tracking step, we record $Mem\_Tracked$ after the hot set is full. If the memory instruction count is larger than $THmax$, it means the time interval between two faults is too long, and the obtained reuse distances are insufficient and the hot set is oversized. To determine the degree of re-sizing, we calculate $Mem\_Tracked/THmax$. The hot set size is reduced by a certain percentage based on the quotient and we will drop the corresponding number of huge pages from the hot set. If the memory instruction count is smaller than $THmin$, we increase the hot set size using the same approach. We calculate $THmin/Mem\_Tracked$ and enlarge the hot set proportionally.

2.5 VM Memory Demand Calculation

The WSS derived from an MRC is based on the idea of saving memory by tolerating acceptable performance loss. The past experiences with a regular page MRC consider that the miss ratios are highly correlated to the performance [10]. A threshold of miss ratio is chosen to denote the acceptable performance loss. The memory size corresponding to the chosen miss ratio is then defined as the WSS. We observe that the precision of a regular page MRC is typically at 10^{-3}. The huge page miss ratio can reach a precision level of 10^{-8}. The miss ratio at this level of precision does not correlate well with performance gain or penalty when the memory allocation is changed. In fact, we can analyze the

distribution of reuse distance over a period of time to determine the actual memory demand of the application. We design experiments to explore the relationship between the reuse distance distribution and the actual memory demand of a program.

We select programs with memory demand over 1 GB from SPEC2017. We keep track of the program memory pages every 1 second, and record the distribution of reuse distance and the number of memory pages (2 MB huge pages). We define:

• Footprint: the size of memory accessed by a program every 1 second;

• 99%RD, 95%RD: the memory size covered by 99% and 95% reuse distance collected every 1 second after hot page set filtering respectively;

• Full_Reused: the memory size covered by 100% reuse distance;

• MAX: the memory size of all memory pages accessed by the program.

The experimental results show that for most applications, the results of 99%RD and Full_Reused are close to Footprint which reflects the real memory demand of a program. Both of their average error are less than 1%, but the error of 95%RD is up to 10%. Therefore, 95%RD is not appropriate to reflect the actual memory demand of a program. For example, Fig.4 shows comparison of the memory demand curves of 605.mcf_s under different conditions. Fig.4(a) is comparison of memory changes at full execution time. Fig.4(b) and Fig.4(c) zoom in memory change curves by extracting a 30-second window under two different memory stages. 605.mcf_s has obvious memory stage and volatility. The memory demand increases sharply in a short time and then quickly returns to stability. In the steady state, the memory demand of 99%RD and Full_Reused are close (see Fig.4(c)). When the memory demand fluctuates, 99%RD and Full_Reused are significantly different (see Fig.4(b)).

Through experiments, we find that compared with Full_Reused, the 99%RD is more suited to reflect the actual memory demand of a program in a huge page environment than Full_Reused. For most programs, 99%RD typically reflects true memory demand as well as Footprint. In extreme cases, Full_Reused could reflect real memory requirements well, but it is likely to reflect the program's access for a few memory pages, and cannot reflect the actual memory demand. In addition, Full_Reused has to bring frequent memory adjustment, because it is sensitive. It is more appropriate to choose 99%RD that can reflect the memory access characteristics of the program stably.

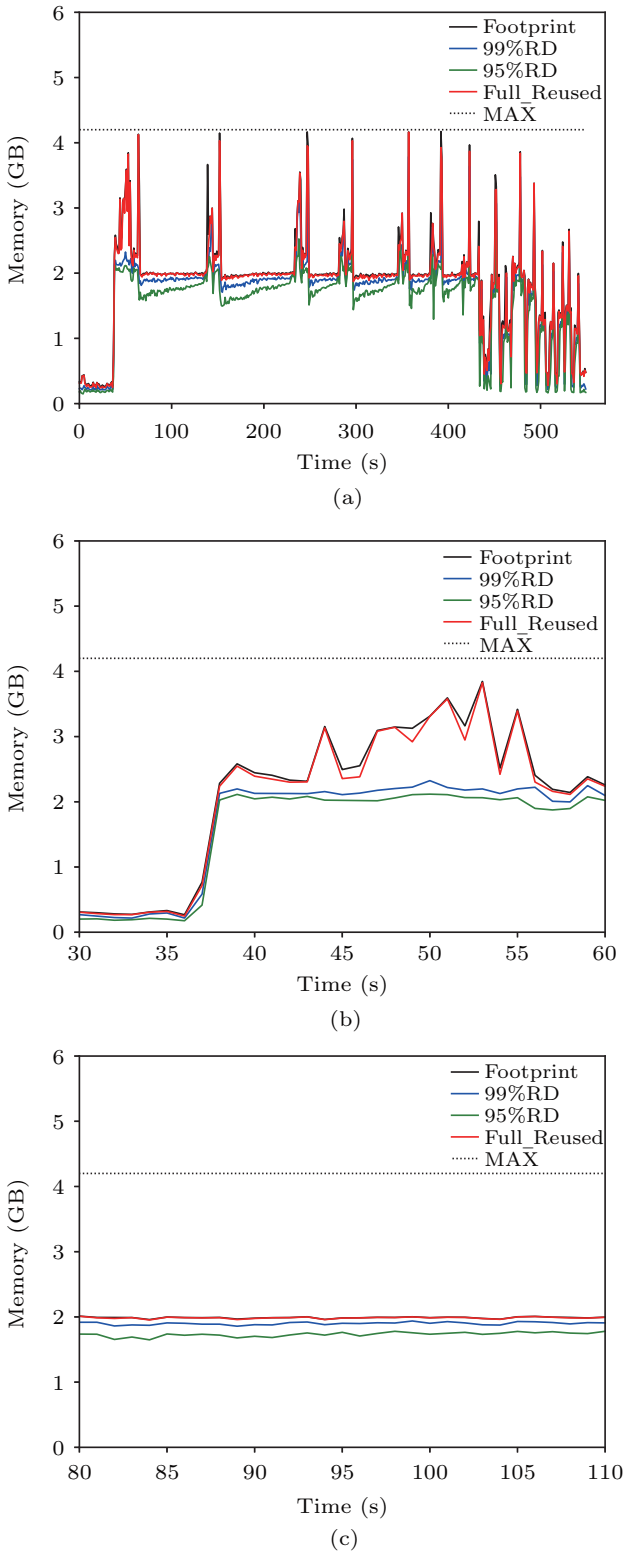


Fig.4. Comparison of memory demand of 605.mcf_s under different conditions. (a) Memory changes at full execution time. (b) (c) Memory changes under two different memory states with 30 s windows.

2.6 Dynamic Memory Growth

In fact, the above method can only estimate WSS no larger than the current memory size of the guest OS. When the guest OS encounters an out-of-memory condition, the constructed MRC can no longer predict memory requirement. A straightforward way to solve this problem is to track the record of the swap space. The size of the working set on the swap space is the extra memory required by the guest OS. Our system reads SwapTotal and SwapFree from /proc/memoryinfo, which is used to calculate the amount of swap that has been used. Linux kernel provides a lazy mechanism④ for swap processing: if the program does not access the data that exists in the swap area, the data will stay in the swap area even if additional free memory has been given to the OS. Therefore, we cannot determine whether a VM is swapping by the amount of swap that has been used. Instead, we save the latest s used swap values of each VM. If the last value is greater than the mean of the latest s values, the VM is considered as swapping. For a VM under swap process, the sum of the current memory and the swap amount is used as the predictive WSS.

3 Huge Page Ballooning

3.1 Ballooning Mechanism

Ballooning consists of two main phases: inflation and deflation. Fig.5 illustrates the inflation and the deflation phases of ballooning. Fig.5(a) and Fig.5(b) present the status of VM during the ballooning process. Inflation refers to the hypervisor sending a request to the guest OS to return a certain amount of memory to the hypervisor. After obtaining the target balloon size, the balloon driver allocates some guest physical pages inside the VM and pins them. Pinning is achieved through the guest operating system interface, which ensures that the pinned pages cannot be paged out to disk under any circumstances. Once the memory is allocated, the balloon driver notifies the hypervisor the page frame numbers of the pinned guest physical memory so that the hypervisor can reclaim the corresponding host physical pages [11]. Deflation means that the hypervisor returns a certain amount of memory to the guest OS and the principle is the same as inflation.

3.2 Problem Under KVM and QEMU

The WSS of many applications is fast growing in the big data era. Dozens of GB or even hundreds of GB memory requirements become a norm. This type of applications and small WSS applications often coexist; therefore most operating systems still run on regular pages mixed with huge pages. However, for large working set applications, memory performance is dragged by 4 KB pages more and more due to high TLB misses and page fault frequency. Hardware constraints prevent us from increasing TLB entries significantly. Large working sets yield a large number of TLB misses, which, in turn, incur substantial page walk overhead. In virtual execution environments, the performance loss is

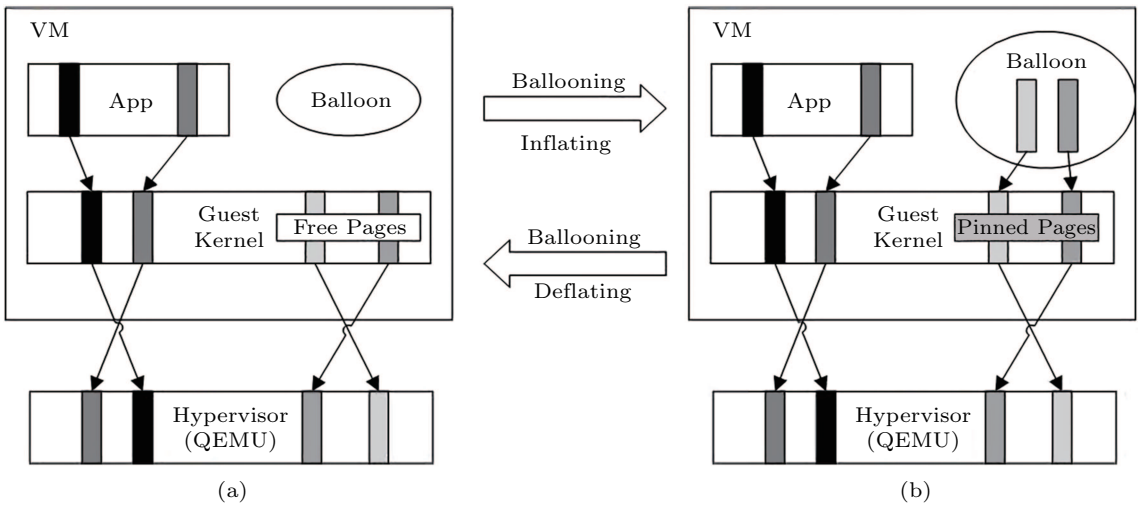


Fig.5. Ballooning mechanism.

④https://www.linuxquestions.org/questions/linux-general-1/reclaim-swap-memory-849992/, March 2020.

even worse as memory virtualization results in additional address translation. Huge pages can help relieve TLB pressure, but virtualization and huge pages do not work in concert currently.

In our experimental system with Linux Kernel 4.2.4, THP is enabled in both the host and the guests. However, we find that the ballooning process will lead to huge page demotion, which, in both the guests and the host, generates a large number of regular pages. This is because the current KVM ballooning mechanism, independent of the guest system and the host system's THP mechanisms, can only operate on the default 4 KB regular pages. Huge page demotions can thus degrade a virtual machine's performance, in particular, when ballooning is triggered. On the other hand, demotion also affects the efficiency of the ballooning process itself.

In the existing KVM and QEMU environments, pages exchanged in the ballooning process are all regular pages. When the balloon is deflated in QEMU, memory is returned to the guest as regular pages. This phenomenon can be verified by comparing the number of huge pages in the host before and after deflation. We observe that the proportion of huge pages in the QEMU process will be reduced due to deflation.

In KVM and QEMU, the hardware-managed extended page table (EPT) is used to complete the memory address translation of a virtual machine. EPT-assisted address translation is divided into two parts: from a VM virtual address to a VM physical address, and from a VM physical address to a host physical address. The memory address translation from the VM's physical addresses to the host's physical addresses will incur additional overhead. Compared with regular pages, huge pages can help reduce pressures on TLB, page walk cache, and the page walk penalty. As shown in Fig.6, the page walk for regular pages traverses four page table queries, while the page walk for huge pages touches only three.

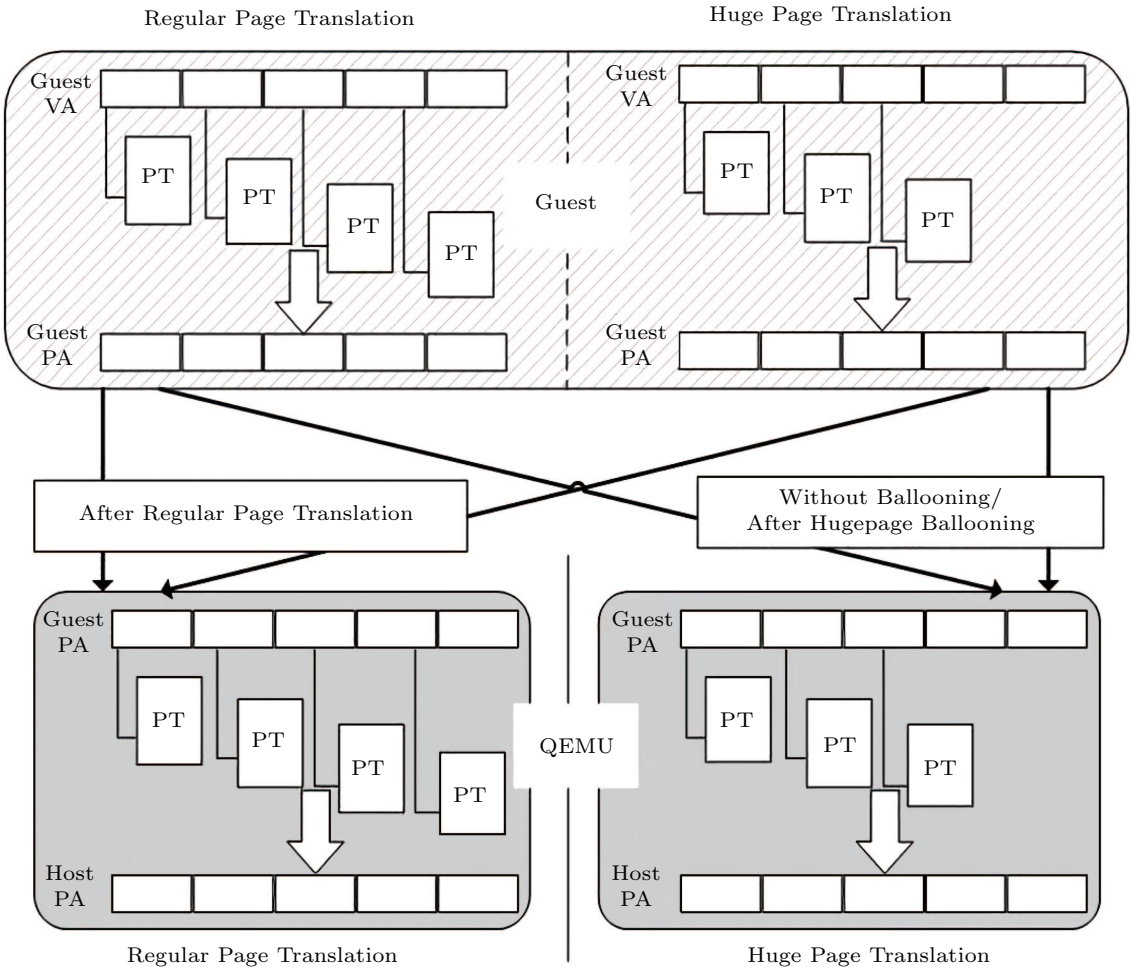


Fig.6. Page translation in host and guest OS.

When a huge page is demoted into regular pages in QEMU due to deflation, accesses to the corresponding pages in the guest, either the demoted regular pages or the huge page, can lead to address translation in QEMU and the host. This layer of translation becomes regular page translation which costs more. In short, the existing ballooning mechanism causes QEMU huge pages to downgrade, which can increase the pressure on TLB and increase page walk penalty.

3.3 Huge Page Support

We have modified the ballooning mechanism to support huge pages. Fig.7 shows the framework of huge page ballooning. The ballooning in QEMU-KVM consists of three parts: the balloon driver in the guest kernel, the communication between the guest kernel and the host QEMU, and the balloon module in the host QEMU. After receiving a ballooning instruction, QEMU computes the difference between the given balloon size and the existing ballooning size. It then invokes the balloon module to inflate or deflate balloon according to the difference. The pages exchanged between the guest balloon driver and the QEMU balloon module are handled in all three parts in sequence; thus all of them need to be adjusted to support huge pages.

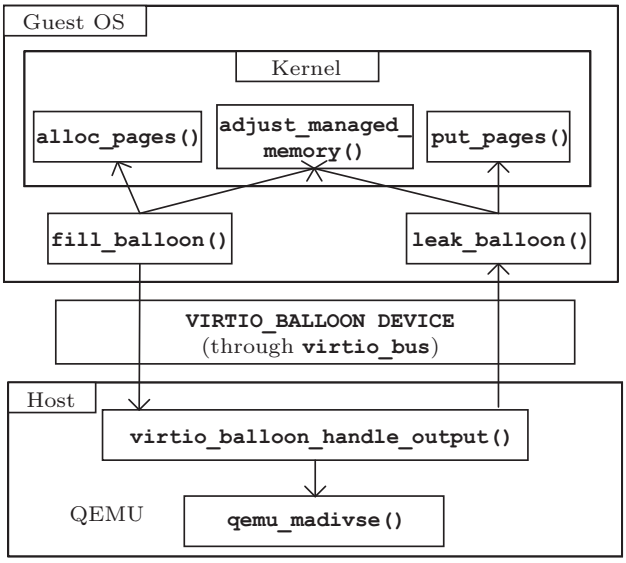


Fig.7. Framework of ballooning.

In QEMU, information is transmitted through *virtio_bus*, using a signal-slot mechanism similar to `Qt`, a cross-platform application framework and widget toolkit[5]. And the second memory allocation size is stored in the *config* file, and the page information of the specific deployment is saved in a queue. In our implementation, the page information is updated to describe a huge page. Related fields include page frame number, page number, $PAGE\_SHIFT$, and other information. `balloon_page()` is called in function `virtio_balloon_handle_output()` to handle the memory queue in *VIRTIO_BALLOON_DEVICE*. Function `balloon_page()` invokes `qemu_madvies()` to inflate or deflate memory. Here we adjust the page size to add support for huge pages, which means changing the page size from 4 K to 2 M.

In the guest OS, we mainly modify some parameters of the balloon driver. The structure *virtio_balloon* holds the main information of the entire balloon mechanism, including the page frame number list *pfns* passed to the hypervisor and *num_pfns* is used for counting. *VIRTIO_BALLOON_PAGES_PER_PAGE* indicates the granularity of the *num_pfns* count. It is defined as $PAGE\_SIZE >> VIRTIO\_BALLOON\_PFN\_SHIFT(VBPS)$, where $PAGE\_SIZE$ is 4K (2^{12}) and $VIRTIO\_BALLOON\_PFN\_SHIFT = 12$, which means $VIRTIO\_BALLOON\_PAGES\_PER\_PAGE = 1$. When we implement huge page ballooning, $PAGE\_SIZE$ has changed from 4K to 2M (2^{21}); therefore we set $VBPS$ to 21 to ensure the correctness of the *pfn* count.

Functions `fill_balloon()` and `leak_balloon()` perform the operations for allocating and returning memory respectively. After that, they call `adjust_managed_memory()` to update memory information. The `fill_balloon()` function requests the page by calling `balloon_page_enqueue()` and `leak_balloon()` releases the page via `balloon_page_dequeue()`. Allocating a regular page is different from allocating a huge page, using different allocation functions, different `GFP` (get free page) masks and an additional page number order. We change the page in `balloon_page_enqueue()` from a regular page to a composite page to support huge pages. The `adjust_managed_memory()` function accepts the changed number of pages as a parameter and completes the maintenance of memory information such as the number of pages, `MemFree`, and `MemAvailable`. We modify the changed number of pages in each update from 1 to 512 as one huge page equals 512 regular pages.

[5]https://https://www.qt.io/, March 2020.

4 Dynamic Memory Balancing

Fig.8 shows the framework of the proposed huge page based dynamic memory balancing system. We bind each VM to a host physical core. After starting, the system will monitor the memory access pattern of each VM and detect their memory phases using the Intel PMU. When detecting memory imbalance, WSS estimation is triggered to estimate memory demand of each VM. After that, Balancer makes memory reallocation strategy and triggers huge page based ballooning to implement the strategy.

4.1 Memory Phase Detection

An application typical shows different phases of memory consumption. To identify a proper metric for memory phase detection, we try several Intel PMU events and compare their variation trend with the memory demand. We confirm that *DTLB_MISS* is suitable as used by Zhao *et al.*[12] *DTLB_MISS* is the sum of *DTLB_LOAD_MISS.WALK_COMPLETED* and *DTLB_STORE_ MISS.WALK_COMPLETED*. The absolute value of *DTLB_MISS* can vary significantly in a short period of execution. We use a data smoothing technique to handle the TLB miss variation[12]. TLB holds translation tables of virtual addresses to physical addresses. When VM memory demand increases, a large number of new virtual/real conversion relationships are generated. And there are no corresponding entries in TLB; therefore *DTLB_MISS* will increase significantly. Memory phases can be defined by counting the number of TLB failures during the program's run time.

In our implementation, we monitor *DTLB_MISS* of each VM every 0.1 s. To avoid the interference of small TLB miss counts, we only consider *DTLB_MISS* larger than 5 000 within a monitoring window. By saving the average of the recent k *DTLB_MISS* counts for each VM, we compare the current average value with the mean of the previous k average values, noted as *ratio*. We define an upper and a lower threshold to determine memory phase. If the *ratio* of a VM lies out of the range, we consider that the VM enters a new phase and the system will check the memory usage of the VM by sending an SSH request. Once a VM has insufficient memory, the Controller will start memory tracking to estimate the WSSs of all VMs.

4.2 VM Memory Demand Prediction

In a huge page environment, the amount of memory dispatched can reach gigabyte level and the huge page swap of a VM can degrade the performance significantly. To eliminate memory imbalance between VMs, we firstly need to accurately predict the memory demand of each VM.

We classify a VM's current memory phase as two states: stable and unstable. We use *DTLB_MISS* to de-

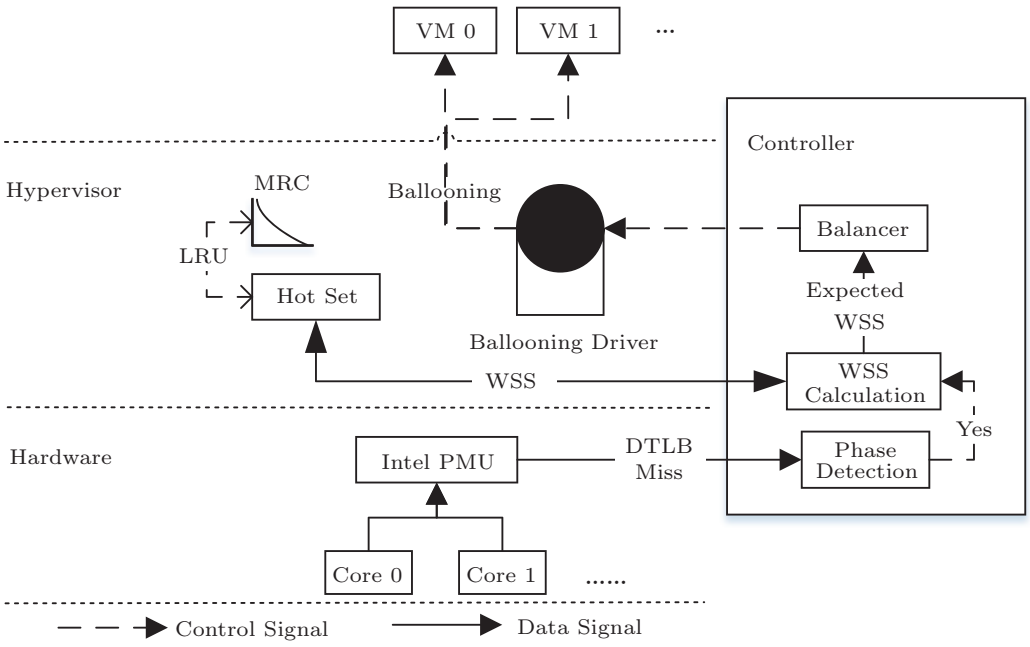


Fig.8. System overview of dynamic memory balancing.

termine the memory states. If the *ratio*s did not exceed the threshold in the last k memory phases as described in Subsection 4.1, the memory state is stable; otherwise, it is unstable. For each VM, we consider three cases. 1) A VM is swapping. 2) There is no swapping, the VM's memory state is stable, and the huge page ratio is over 90%. 3) There is no swapping, but the memory state is unstable or the huge page ratio is less than 90%. The pages in the swap area are regular pages; therefore once "swap in" occurs, regular pages will remain in the VM until they are released by the program. Experiments show that if the huge page ratio is less than 90%, the accuracy of huge page based WSS estimation decreases significantly.

In the first case, the memory of the VM is seriously insufficient and swap occurs; therefore the expected memory is the sum of the current VM's memory plus the amount of swapped out memory. In the second case, we construct an MRC to calculate WSS as described in Section 2. In the third case, the memory demand of VM is uncertain, but it is temporarily sufficient because no swap occurs. We predict that the VM does not need more memory and the memory demand remains unchanged.

4.3 Memory Balancing Algorithm

According to whether the total available memory of VMs is sufficient, there are two cases. If the sum of the memory expected by all VMs is less than or equal to total available memory, then the memory is allocated as expected. And if there is still surplus memory, it is evenly distributed to each VM in proportion to the expected memory. When the host machine cannot meet all VMs' memory demands, we use a dynamic programming algorithm to generate a memory allocation scheme that minimizes the overall number of page faults as used by Wang *et al.*[9]

The objective is to achieve the best overall performance. The state transfer function for the dynamic programming algorithm is as follows:

$$\begin{aligned} &Miss[i,j] \\ &= \min\{Miss[i-1,j-k] + PM_i[k] | L_i \leqslant k \leqslant H_i\}, \end{aligned}$$

where $Miss[i,j]$ is the minimum number of page faults when the first i VMs are allocated a total of j MB. L_i and H_i are the lower and the upper bounds of VM i's memory size, respectively. $PM_i[k]$ is the number of page faults when VM i obtains k MB, which can be derived from the VM's MRC. Note that we do not apply the dynamic programming algorithm to all VMs but only to the VMs that are not under swap process. The LRU-based WSS estimation is only triggered for the VMs that are not swapping.

5 Evaluation

5.1 Experimental Setup

Our experiments are conducted on an Intel® I7 6700 machine, with 32 GB of physical memory and four 3.4 GHz cores. Both the host and the guest OSs run CentOS 7.0. The kernel versions are 4.2.4 for the host and 4.18.12 for the guests, respectively. The hypervisor is QEMU-KVM version 2.4.9. All the experiments are conducted on the KVM, with hardware assisted mode enabled. Both the host and the guests enable transparent huge pages.

5.2 Benchmarks

As our huge page based memory management focuses on applications with large memory consumption, we choose a set of 15 benchmarks that require memory no less than 1 GB. Table 1 lists the details of our benchmarks, including their peak memory consumption and execution time when running on a single VM with sufficient memory. In an actual multi-virtual machine environment, the execution time will be longer due to interference among VMs.

Table 1. Benchmarks

Label	Benchmark	Memory (GB)	Time (s)
A	602.gcc_s	7.5	484
B	603.bwaves_s	11.6	7 902
C	605.mcf_s	3.8	522
D	607.cactusBSSN_s	6.9	1 911
E	619.lbm_s	3.3	945
F	631.deepsjeng_s	7.0	308
G	638.imagick_s	4.3	21 800
H	649.fotonik3d_s	9.8	878
I	654.roms_s	10.6	2 957
J	Graph-4	3.4	17
K	Graph-5	4.5	24
L	Memcached	4.1	23
M	Numeric_calculate	1.0	430
N	Sequential_read	5.0	300
O	Sequential_write	5.0	300
P	Random_write	5.0	300
Q	Random_read	5.0	300
R	Random_read_write	1.8	40

The benchmarks from A to I are picked from SPEC-CPU 2017. Graph and Memcached are chosen from CloudSuite. Graph's peak memory demand is based on the number of input nodes. J and K are set to 4 million and 5 million nodes, respectively. M to R are hand-coded micro-benchmarks. M is a CPU intensive program consisting of a series of numeric calculations. N to R are memory-intensive programs with different memory access characteristics.

5.3 Evaluation of WSS Estimation

5.3.1 MRC Accuracy

We first evaluate the accuracy of our MRC construction method. We use the Intel *Pin* to simulate benchmark execution. *Pin* is a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures that enables the creation of dynamic program analysis tools. We use *Pin* to simulate the execution of a benchmark and get the trace of the memory accesses, collect its RDD, and construct an MRC. We then compare it with the MRC generated in KVM through our MRC construction method. It takes the *Pin* tool hundreds to thousands hours to simulate a selected benchmark to completion. We instead only simulate 50 billion memory instructions of each chosen benchmark. We fast forward 10 billion memory instructions to bypass the warming-up phase. For fair comparison, we also track the first 10 to 60 billion instructions in KVM.

The experimental results are shown in Fig.9. The three miss ratio curves in each sub figure are the MRC constructed through *Pin* simulation (PIN), and the MRCs restored in KVM by the approach described in Subsection 2.3, with a fixed hot set (KVM_FHS_MOD) and with a dynamic hot set (KVM_DHS_MOD). The fixed hot set size is set to 40 MB, while the dynamic hot set is set to 500 MB to start. The results show that the restored MRCs highly overlap with the accurate MRCs.

It can be seen that the MRCs of the three sets of experiments are highly coincident when the memory size is large relatively. This suggests that our memory tracking system is able to track almost all the long reuse distances. The error of using DHS is a bit larger than that of using FHS, as DHS starts with a much larger hot set size. We also can observe a significant estimation error for memory sizes from 0 to about half of their dataset size. The error is mostly from small memory sizes as our approach approximates the distribution of small distances. The other error source is from the different testing environments between *Pin* and KVM. In *Pin*, we assume that all memory instructions will access huge pages. However, Linux THP cannot achieve 100% huge page utilization. We notice many regular pages accesses in KVM. The MRC obtained by *Pin* counts the reuse distances of those regular page accesses, which are not tracked in KVM. Another source of errors comes from extra huge pages monitored by KVM. Our memory tracking system monitors the memory accesses of the whole VM. As a result, the tracked huge pages include those from the OS and other processes running in the guest OS. In practice, the error for small memory sizes has little impact as we seldom allocate memory less than a half of an application's dataset size.

5.3.2 Overhead of Memory Tracking System

We evaluate the memory tracking system overhead with both a fixed-size hot set (FHS) and a dynamic hot set (DHS) and Table 2 presents the results. The second column WSS lists the WSS of each benchmark. "Runtime base" shows the baseline execution time without memory tracking, except for memcached where the average access latency is shown. "Runtime FHS" presents the execution time when memory tracking with a fixed-size hot set is turned on. We set the hot set size for each benchmark to half its WSS. "Overhead FHS" lists the percentage slowdown due to memory tracking with a fixed-size hot set size. "RD count FHS" is the total number of reuse distances tracked by memory tracking, which is also the page fault count introduced by memory tracking. The last three columns present the results of runtime, overhead, and reuse distance count respectively when applying a dynamic hot set in memory tracking.

The results show that the average overhead of our memory tracking method with FHS is 74.83% and that with DHS is 1.44%. In the FHS case, 631.deepsjeng_s stands out with an overhead of 729% and tracked reuse distances of 136 million, as most of its reuse distances are long ones. Except for this workload, the average overhead of the rest can reach 1.93%. By using a dynamic hot set, the page faults added by our memory tracking system are all reduced to a 100s of thousand level. Over half of the benchmarks show an overhead of within 1%. The performance of 631.deepsjeng_s is highly improved by DHS when compared with FHS, but it still has a slowdown of 7.2%. Most of its reuse distances appear at its peak memory usage time, and

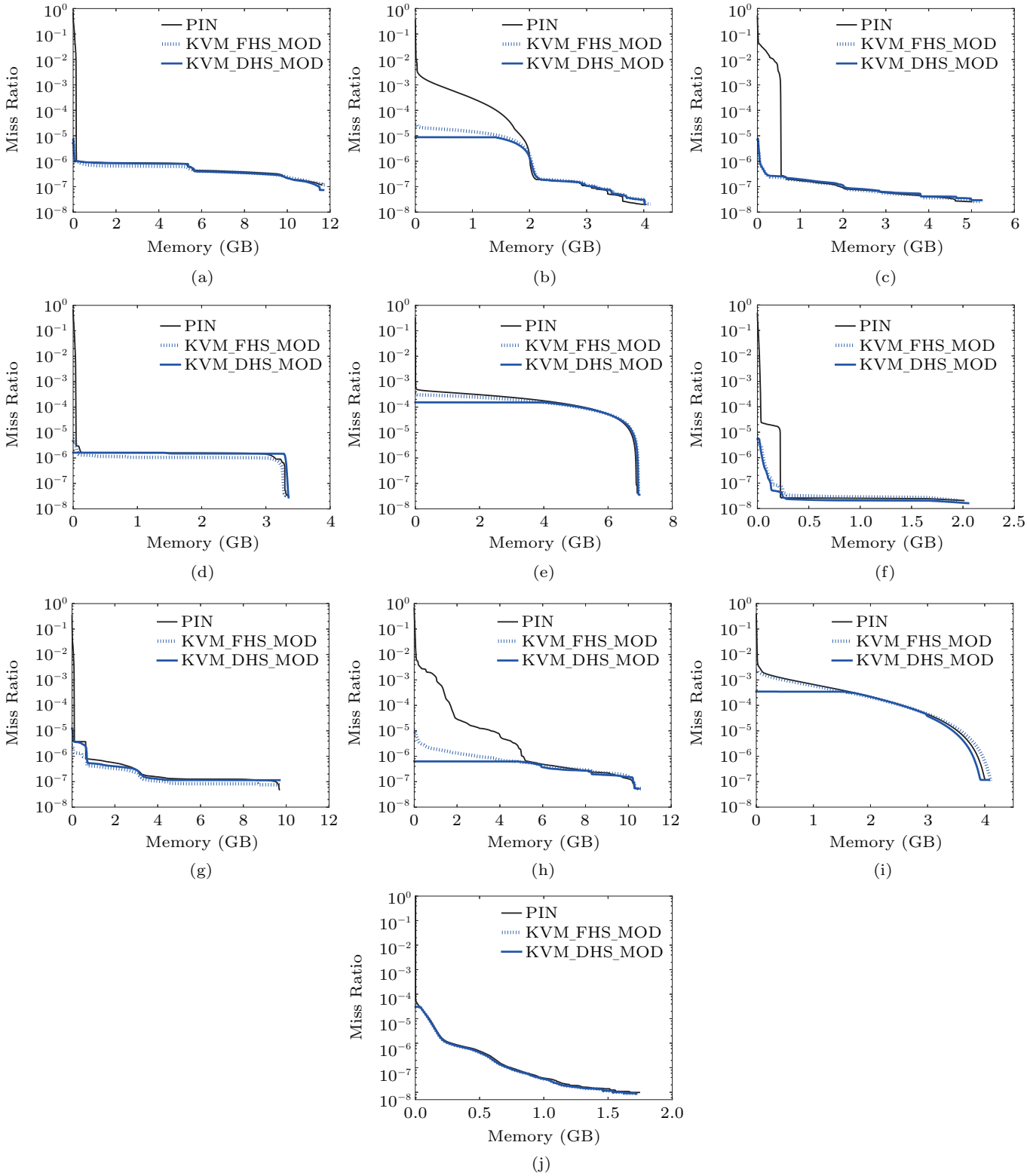


Fig.9. Comparison of MRC in PIN and KVM. The three curves in each sub-figure present the MRC in PIN, the MRC restored in KVM with a fixed 40 MB hot set, and that with a dynamic hot set. The x-axis presents the memory size in GB, and the y-axis presents the miss ratio in a logarithmic scale of 10. Benchmark graph only accesses 1.7 GB of memory in 50 billion memory instructions. (a) 603.bwaves_s. (b) 605.mcf_s. (c) 607.cactusBSSN_s. (d) 619.lbm_s. (e) 631.deepsjeng_s. (f) 638.imagick_s. (g) 649.fotonik3d_s. (h) 654.roms_s. (i) Memcached. (j) Graph.

Table 2. Memory Tracking Overhead

Benchmark	WSS (GB)	Runtime Base (s)	Runtime FHS (s)	Overhead FHS (%)	RD Count FHS	Runtime DHS (s)	Overhead DHS (%)	RD Count DHS
603.bwaves_s	11.6	7 902.00	8 716.00	10.30	395 751	7 957.00	0.70	203 287
605.mcf_s	3.8	522.00	529.00	1.42	518 476	527.00	1.07	376 805
607.cactusBSSN_s	6.9	1 912.00	1 923.00	0.60	834 170	1 916.00	0.21	33 184
619.lbm_s	3.3	945.00	961.00	1.74	1 834 941	956.00	1.29	702 675
631.deepsjeng_s	7.0	308.00	2 245.00	729.00	136 983 318	330.00	7.20	783 437
638.imagick_s	4.3	21 800.00	21 861.00	0.28	5 317 790	21 851.00	0.23	672 930
649.fotonik3d_s	9.8	878.00	897.00	2.17	494 227	885.00	0.83	202 348
654.roms_s	10.6	2 957.00	2 980.00	0.74	686 323	2 978.00	0.72	629 374
Graph	3.4	866.00	882.00	1.80	423 074	880.00	1.60	373 857
Memcached	4.1	22.96	23.02	0.26	723 826	23.00	0.20	213 465
Average overhead				74.83			1.44	

Note: For Graph, we run graph 50 times with four million nodes. For Memcached, we use average latency (ms) to evaluate overhead.

dynamically increasing the hot set size cannot reduce the reuse distance count efficiently.

5.4 Evaluation of Huge Page Ballooning

5.4.1 *Ballooning Overhead*

We compare the overhead of our proposed huge page ballooning (HPB) with the original regular page ballooning (RPB). We measure both execution time and TLB misses. Compared with RPB, HPB shows a great advantage in its execution time. We trigger ballooning when running the random write program in our micro-benchmark suite. We inflate the guest balloon by a fixed amount and then deflate the same amount. The result is shown in Fig.10. HPB takes 0.42 ms when inflating 256 MB of memory, and 46 ms for 10 GB. In contrast, RPB takes 92 ms to balloon 256 MB and 3 336 ms for 10 GB. The results show that HPB runs 70 times faster than RPB. It takes RPB seconds to balloon gigabytes of memory, while HPB only spends dozens of milliseconds. Fig.11 compares the TLB miss count. The miss counts caused by HPB are actually 3 to 4 times lower than RPB.

In the Linux kernel, the size of a huge page is 2 MB and a regular page is 4 KB. This means that the OS using regular pages needs 512 regular page TLB entries to cover a huge page. The OS needs to go through 512 TLB misses and 512 page faults to map 2 MB of application space to physical memory. By utilizing huge pages, the system will spend less time on page walk and also trigger less TLB misses.

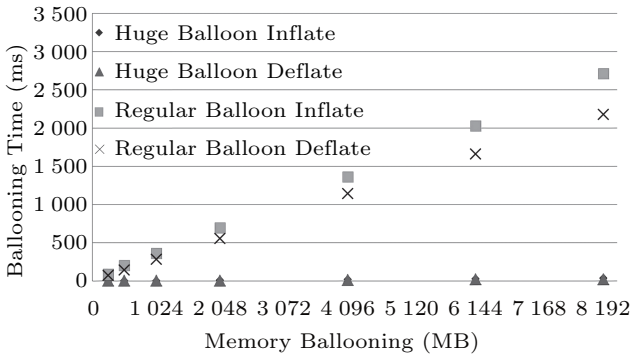


Fig.10. Ballooning execution time comparison.

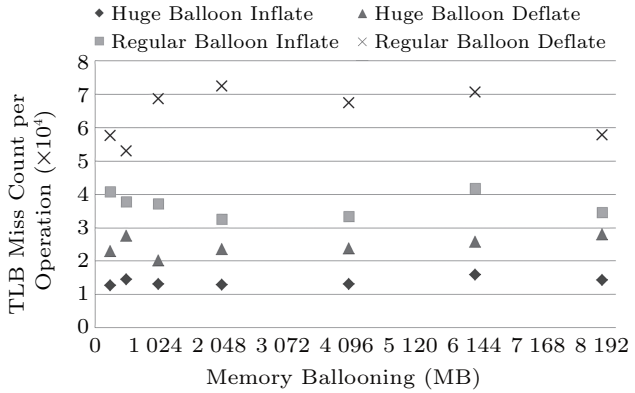


Fig.11. Ballooning TLB miss count comparison.

5.4.2 *Varying Workloads and Varying Phases*

To further evaluate the performance of HPB, we design two sets of experiments, varying workloads and varying phases. For varying workloads, we consider that different benchmarks with different working set sizes will run on the same VM. After the first benchmark finishes, the host will first adjust the guest mem-

ory to the minimum amount of 512 MB, and then restore the memory to the second benchmark's working set size. We use Intel PMU to collect the TLB miss count and other CPU events to monitor the second benchmark's performance. We also analyze the effect of ballooning on the guest memory by counting its huge page ratio. Since the first benchmark serves as a warming-up benchmark, we create a warming-up workload by mixing random reads and random writes. The new workload consumes a maximum of 5 GB of memory. For varying phases, we consider that a benchmark's memory demand varies during different phases of its execution. We detect phases and adjust the guest memory according to its current memory demand. This is the main objective of ballooning to improve memory utilization in VMs. Our micro-benchmark suite is designed to have clear phases and fits well for this set of experiments.

We run 15 benchmarks after executing the warming-up micro-benchmark. We measure their execution time under three settings: no ballooning, RPB, and HPB. "No ballooning" does not change memory allocation between the warming-up workload and the testing workload. Table 3 lists the TLB miss count increases and the percentages of runtime overhead when compared with no ballooning. The results show that these benchmarks suffer a significant number of TLB misses with RPB. HPB results in a minimal performance impact with a maximum overhead of 1%, while RPB can cause up to 9% slowdown. As discussed in Section 3, the disparity of TLB miss count between huge page and regular page ballooning comes from QEMU. We measure the huge page utilization of the host OS and find that after HPB, the QEMU process still uses huge pages. However, after regular page ballooning, the ballooned part of QEMU process' memory uses 100% regular pages.

Table 3. TLB Miss & Slowdown Compared with No Ballooning

Benchmark	TLB Miss Increment (%)		Slowdown (%)	
	RPB	HPB	RPB	HPB
A	6.45	2.03	1.93	0.01
B	1 839.88	19.42	0.06	0.03
C	4 420.49	70.49	8.51	0.02
D	1 497.48	3.39	7.13	0.27
E	21 733.39	7.25	1.87	0.02
F	52.02	0.26	6.02	0.05
G	667.01	19.49	0.12	0.06
H	365.43	6.71	0.41	0.04
I	911.97	2.64	0.91	0.05
J	2 654.40	8.39	1.89	0.33
L	53.41	0.65	0.59	0.09
N	17.80	0.30	1.51	1.21
O	31.57	0.01	2.26	0.11
P	528.07	15.44	1.21	1.08
Q	530.80	26.40	2.07	0.08

We find that most of the selected workloads do not show clear phases or their phases fluctuate frequently. We instead use the micro-benchmarks to evaluate the impact of ballooning on varying application phases. We inflate and deflate the balloon in the guest according to the actual demand of the micro-benchmarks in the current phase. The guest OS receives 5 GB of memory when the benchmark consumes 5 GB, and down to 1 GB when it only needs 1GB. We set the micro-benchmarks to alternate their memory demands 10 times during their execution. Fig.12 shows the normalized TLB misses and runtime slowdown over the "no ballooning" baseline. HPB causes minimal runtime and TLB overhead while regular page ballooning can cause over 150% slowdown. The results show that sequential benchmarks perform much better than random benchmarks, due to TLB prefetching and locality. It also shows that random_write performs worse than random_read, because write operation modifies TLB entry's dirty flag which leads to update of TLB. With HPB, the TLB miss count of random read and write is reduced by 89.6% and 62.7% respectively.

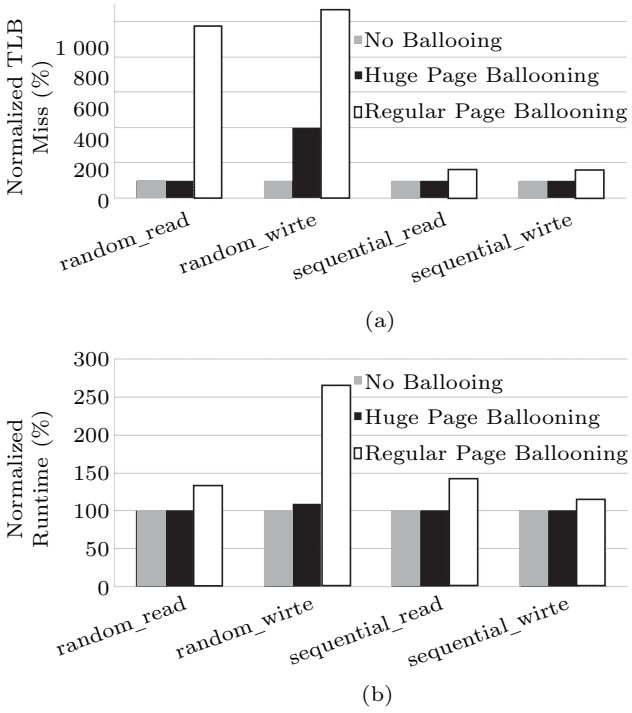


Fig.12. Normalized TLB misses and performance slowdown.

5.5 Evaluation of Memory Balancing

5.5.1 Experimental Design

The design goal of the memory balancing system includes: 1) if the total available memory of VMs is sufficient, the system can quickly find and adjust memory effectively when there is memory imbalance during the execution; 2) if insufficient, the system can ensure that the original memory-rich VMs still have sufficient memory, while free memory will be reclaimed and reallocated to out-of-memory VMs, to improve the overall memory utilization and performance.

We conduct eight sets of experiments. Table 4 shows the workloads and initial memory allocations of the VMs of each group. The benchmarks of each workload is desrcibed in Table 1. In order to demonstrate the effect of VM memory phases, we stitch multiple benchmarks to run on a VM in order. The overall experimental results of all groups are shown in Table 5.

Table 4. Experimental Configuration

Group	VM_ID	Benchmark	Memory (GB)	
			Best	B&D
1	VM1	NM	5.3	3.3
	VM2	MN		
2	VM1	EFCC	7.3	5.7
	VM2	FECF		
3	VM1	FCCC	7.3	4.9
	VM2	CFCC		
	VM3	CCFC		
	VM4	CCCF		
4	VM1	HEE	10.2	5.8
	VM2	EHE		
	VM3	EEH		
5	VM1	ACCC	7.8	6.4
	VM2	CCAC	7.8	
	VM3	DC	7.2	
6	VM1	ECF	7.0	6.3
	VM2	EFE		
	VM3	EFE		
7	VM1	J	3.7	2.7
	VM2	R		
	VM3	R		
8	VM1	R	4.8	3.0
	VM2	K		
	VM3	R		

Note: The benchmark is from Table 1. B&D: baseline and dynamic balancing.

For each set of experiments, there are two control groups: best and baseline, and one experimental group named as balancing. "Best" presents the results when each VM is provided with enough memory. In "baseline", the memory provided remains unchanged during the entire execution process. In "balancing", we use our dynamic huge pages memory balancing system to adjust VMs' memory sizes. In our experiments, the initial memory size for each VM in "best" is set to its peak memory, as shown in Table 1. In "baseline" and "balancing", the initial memory size (IMS) is set to the same value, which is calculated as: $IMS = \max\{M_i | 0 \leqslant i \leqslant k\}/n + C$, where n is the number of VMs, k is the execution time, and M_i is the sum of used memory of all VMs at moment i. In addition, the VM itself needs a certain amount of memory to start and it also reserves some free memory. We represent those two parts of memory as a constant C. According to the experience, we set C to 300 MB in our experiments.

Table 5. Memory Balancing Slowdown

Group	AvgSlowdown	FairSlowdown	MaxSlowdown
1	1.08	1.08	1.09
2	1.04	1.04	1.04
3	1.03	1.02	1.04
4	1.03	1.03	1.04
5	1.05	1.04	1.06
6	1.05	1.04	1.07
7	1.41	1.34	3.44
8	1.86	1.35	3.52

5.5.2 Experimental Results with Multiple VMs

In the first six experiments, the total memory of VMs is sufficient, but memory imbalance occurred during the execution. We take the first set of experiments as an example.

We track the changes in available memory (total_memory) and used memory (used_memory) for each VM as shown in Fig.13. Initially, both VMs run without out-of-memory. As the memory requirements of the two VMs continue to increase, VM2 has sufficient memory but VM1 begins to swap. Memory balancing is triggered. According to our memory balancing strategy, the expected memory size of VM1 is the current memory and the amount of swap. VM2's expected memory is measured using huge page WSS estimation. Then, ballooning based on huge pages is triggered.

During the balancing process, VMs may experience multiple rounds of ballooning, because the VM that lacks memory may not get enough memory in the first round. The system will detect memory imbalance again

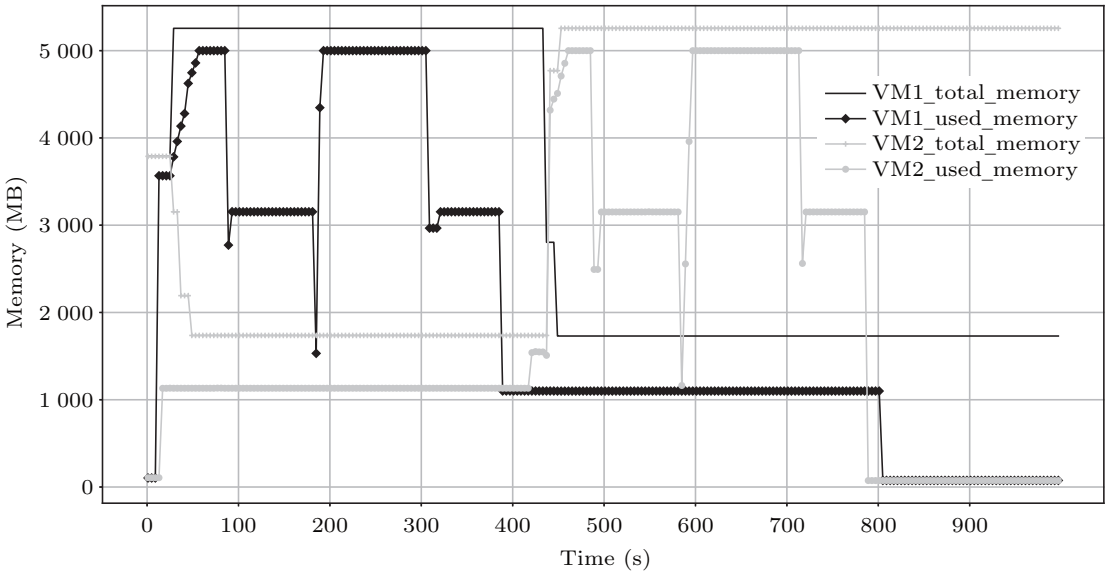


Fig.13. Memory balancing between two VMs.

and continue to replenish the memory for the VM until the memory is sufficient. As shown in Fig.13, at about half way of execution cycle, the second memory imbalance occurs, and the balancing process is roughly the same as the first time.

Compared with "baseline", the execution time of VM1 and VM2 in memory balancing, is reduced by 97.5% and 97.4%, respectively. Compared with best, the average slowdown of balancing is only 8%. The average slowdown (AvgSlowdown) for the other groups is shown in Table 5.

5.5.3 *Memory Overcommitment*

In an actual multi-VM system, each VM is independent, and it is difficult to ensure that there will be no shortage of total physical memory due to a sudden increase in memory requirement of an individual VM. In this case, memory balancing follows the principle below: we try to claim memory from other VMs to the VM short of memory, while we ensure that the performance of other VMs does not drop significantly.

We conduct two sets of experiments (groups 7 and 8) and use the group 7 as an example. As shown in Fig.14, as the memory demand grows, VM1 runs out of memory and begins to swap. The memory balancing system firstly estimates memory demand of each VM and finds memory overcommitment. Under the condition of the current total memory shortage, memory balancing uses the dynamic programming algorithm to calculate the optimal memory allocation scheme. Memory balancing recycles as much memory as possible from VM2 and

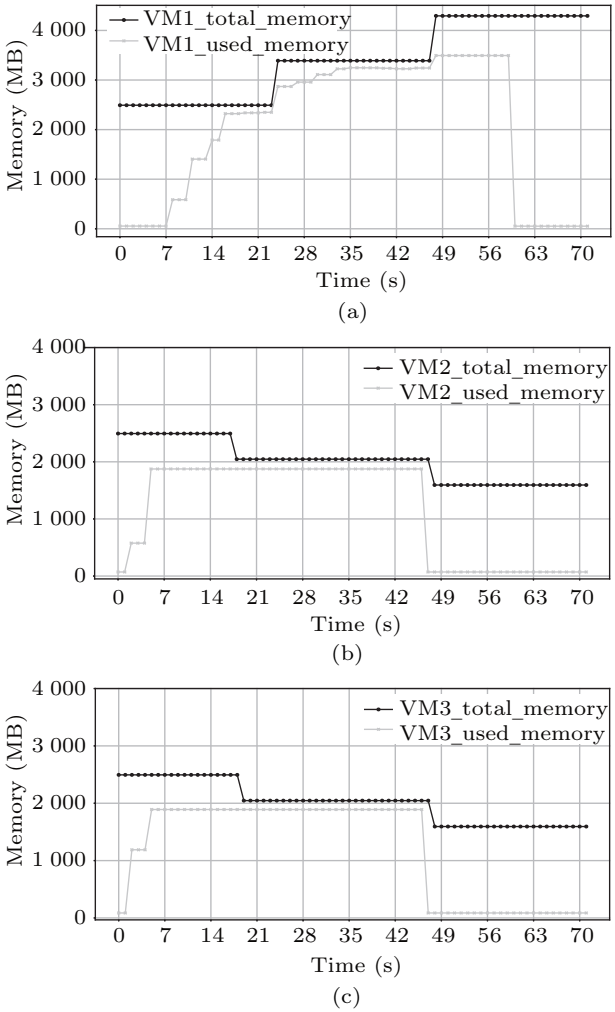


Fig.14. Multi-VM memory balancing with memory overcommitment. (a) VM1. (b) VM2. (c) VM3.

VM3, and then allocates it to VM1 at 23 s. At 45 s, the programs on VM2 and VM3 finish, and the two VMs have more free memory. At this time, part of the memory of the two VMs is reallocated to VM1, which further alleviates the memory shortage of VM1.

Obviously, the performance degradation caused by the lack of total physical memory cannot be completely solved, but the loss is controlled through dynamic memory balancing. Our experiments show that compared with "best", the average performance loss of "baseline" is 23% higher than memory balancing, which means dynamic huge page memory balancing reduces the performance loss significantly.

5.5.4 *Evaluation of Fairness and Quality of Service*

We use two metrics, FairSlowdown and MaxSlowdown, to evaluate the fairness and the quality of service of memory balancing respectively. FairSlowdown is the harmonic average slowdown of memory balancing compared with "best" of all VMs, which balances both fairness and performance between different VMs. MaxSlowdown emphasizes QoS and performance, which represents the upper bound of the performance losses of the balancing programs. It is the maximum slowdown of each VM with memory balancing when compared with "best".

As shown in Table 5, the FairSlowdown and MaxSlowdown of each group from 1 to 6 are less than 1.1. This indicates that the dynamic allocation achieves balanced and good performance of each VM when the total available memory is sufficient. However, the two indicators in group 7 and group 8 are very high. The high FairSlowdown indicates that the performance of each VM is not balanced. In order to achieve the overall optimal performance, there will be some performance loss for part of VMs when the total memory is insufficient. This can also be demonstrated by MaxSlowdown.

6 Related Work

WSS estimation is a core part for a dynamic memory balancing system. In a regular page environment, the typical method of estimating an application's WSS is to construct an MRC by measuring reuse distances and construct an LRU histogram [7, 8, 13–16]. Another method is to derive an MRC using a reuse time histogram with the footprint theory [17] and the AET theory [18, 19]. Compared with these two technologies, reuse distance based methods provide a more accurate MRC with a larger runtime overhead.

With respect to memory balancing, the most closely related work is that focusing on dynamic memory management for virtualized systems using ballooning techniques [9, 10, 20–24]. They are all designed for memory management via 4 KB regular pages, while our work focuses on applications with large memory consumption that can benefit from huge pages instead of regular pages. For example, memory balancer (MEB) proposed by Wang *et al.* [9], dynamically adjusts virtual machines' memory based on ballooning in Xen. MEB detects memory phases, predicts the memory need of each VM via a least recently used (LRU) histogram, and uses a dynamic programming algorithm to adjust memory.

In commercial virtual machines, there also exist corresponding memory allocation mechanisms. For example, in VMware's memory management mechanism [11], according to the available memory margin, host memory has four states, high, soft, hard and low. When the available memory reaches the high state, the ballooning or Swap mechanism will not be started. When it is between the high state and the soft state, the ballooning mechanism is activated. The host uses the drivers in VMware tools to scan a VM's memory usage regularly to see how much active memory it has. The host will calculate the shares-per-page ratio of each VM, and start the balloon driver for the virtual machine with the smallest share of shares-per-page ratio. VMware's ballooning mechanism is mainly to ensure that the host has enough free memory to facilitate timely allocation. However, its ballooning-based recycling process is extremely slow and usually takes a few minutes. This does not meet the timeliness requirement of virtual machine memory provisioning, especially for VMs running large working set programs.

7 Conclusions

With the continuous development of virtualization, the memory management becomes more important for applications with a large amount of memory consumption. This paper proposed an efficient memory balancing system for large working set applications in a multi-virtual machine environment. The results showed that our system can effectively reallocate memory among virtual machines while maintaining the overall VMs' performance with low overheads.

In the future, we plan to implement a memory pool, which helps reserve memory for future bursty virtual machine memory demand. As long as a swap process is

detected for a VM, the reserved memory will be given to the VM before the normal memory balancing procedure starts in order to mitigate the swap overhead. The existing swap mechanism does not fully support huge pages. We intend to implement a more huge page-friendly swap mechanism to further improve the system performance.

References

[1] Khalidi Y A, Talluri M, Nelson M N, Williams D. Virtual memory support for multiple pages. Technical Report, Sun Microsystems Laboratories, Inc., 1993. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=AA2CA3D6205E02FDA1FC545D691C5C20?doi=10.1.1.32.6162&rep=rep1&type=pdf, Sept. 2019.

[2] Arcangeli A. Transparent hugepage support. In *Proc. the 2010 KVM Forum*, August 2010.

[3] Wang X, Luo T, Hu J, Wang Z, Luo Y. Evaluating the impacts of hugepage on virtual machines. *Science China Information Sciences*, 2017, 60(1): Article No. 12103.

[4] Denning P J. The working set model for program behavior. In *Proc. the 1st ACM Symposium on Operating System Principles*, October 1967, Article No. 15.

[5] Hu J, Bai X, Sha S *et al*. Working set size estimation with hugepages in virtualization. In *Proc. the 2018 IEEE ISPA/IUCC/BDCloud/SocialCom/SustainCom*, Dec. 2018, pp.501-508.

[6] Hu J, Bai X, Sha S *et al*. HUB: Hugepage ballooning in kernel-based virtual machines. In *Proc. International Symposium on Memory Systems*, Oct. 2018, pp.31-37.

[7] Mattson R L, Gecsei J, Slutz D R, Traiger I L. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 1970, 9(2): 78-117.

[8] Waldspurger C A, Park N, Garthwaite A T, Ahmad I. Efficient MRC construction with SHARDS. In *Proc. the 13th USENIX Conference on File and Storage Technologies*, February 2015, pp.95-110.

[9] Wang Z, Wang X, Hou F, Luo Y, Wang Z. Dynamic memory balancing for virtualization. *ACM Transactions on Architecture and Code Optimization*, 2016, 13(1): Article No. 2.

[10] Zhao W, Wang Z, Luo Y. Dynamic memory balancing for virtual machines. *ACM SIGOPS Operating Systems Review*, 2009, 43(3): 37-47.

[11] Waldspurger C A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 2002, 36(5): 181-194.

[12] Zhao W, Jin X, Wang Z, Wang X, Luo Y, Li X. Low cost working set size tracking. In *Proc. the 2011 USENIX Annual Technical Conference*, June 2011, Article No. 14.

[13] Zhou P, Pandey V, Sundaresan J, Raghuraman A, Zhou Y, Kumar S. Dynamic tracking of page miss ratio curve for memory management. *ACM SIGOPS Operating Systems Review*, 2004, 38(5): 177-188.

[14] Wires J, Ingram S, Drudi Z, Harvey N J, Warfield A, Data C. Characterizing storage workloads with counter stacks. In *Proc. the 11th USENIX Symposium on Operating Systems Design and Implementation*, October 2014, pp.335-349.

[15] Niu Q, Dinan J, Lu Q, Sadayappan P. PARDA: A fast parallel reuse distance analysis algorithm. In *Proc. the 26th International Parallel and Distributed Processing Symposium*, May 2012, pp.1284-1294.

[16] Tam D K, Azimi R, Soares L B, Stumm M. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proc. the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2009, pp.121-132.

[17] Xiang X, Bao B, Ding C, Gao Y. Linear-time modeling of program working set in shared cache. In *Proc. the 2011 International Conference on Parallel Architectures and Compilation Techniques*, October 2011, pp.350-360.

[18] Hu X, Wang X, Zhou L, Luo Y, Ding C, Wang Z. Kinetic modeling of data eviction in cache. In *Proc. the 2016 USENIX Annual Technical Conference*, June 2016, pp.351-364.

[19] Hu X, Wang X, Zhou L, Luo Y, Wang Z, Ding C, Ye C. Fast miss ratio curve modeling for storage cache. *ACM Transactions on Storage*, 2018, 14(2): Article No. 12.

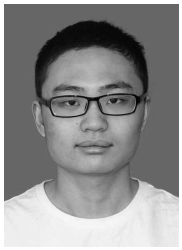
[20] Xiao Z, Song W, Chen Q. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 2012, 24(6): 1107-1117.

[21] Tasoulas E, Haugerund H, Begnum K. Bayllocator: A proactive system to predict server utilization and dynamically allocate memory resources using Bayesian networks and ballooning. In *Proc. the 26th Large Installation System Administration Conference on Strategies, Tools, and Techniques*, December 2012, pp.111-122.

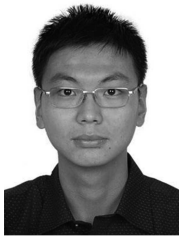
[22] Gordon A, Hines M, Silva D, Ben-Yehuda M, Silva M, Lizarraga G. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *Proc. the 2011 Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments*, May 2011.

[23] Nitu V, Kocharyan A, Yaya H, Tchana A, Hagimont D, Astsatryan H. Working set size estimation techniques in virtualized environments: One size does not fit all. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018, 2(1): Article No. 19.

[24] Liu H, Jin H, Liao X, Deng W, He B, Xu C. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *IEEE Transactions on Parallel and Distributed Systems*, 2015, 26(5): 1350-1363.



Sai Sha is a Ph.D. candidate in the School of Electronics Engineering and Computer Science, Peking University, Beijing. Before that, he received his Bachelor's degree in computer science from Beijing Institute of Technology, Beijing, in 2018. His research interests include system software, virtualization, domestic operating system, and deep learning.



Jing-Yuan Hu is an assistant researcher at the Institute of Information Engineering, Chinese Academy of Sciences, Beijing. He got his Ph.D. degree in computer science from Peking University, Beijing, in 2019. His current research interests include system virtualization, memory management, data mining and information security.

Ying-Wei Luo received his Ph.D. degree in computer science from Peking University, Beijing, in 1999. He is a full professor of computer science in the School of Electronics Engineering and Computer Science (EECS) in Peking University, Beijing. His research interests include operating system, system virtualization, and cloud computing.

Xiao-Lin Wang received his Ph.D. degree in computer science from Peking University, Beijing, in 2001. He is now a full professor of School of Electronics Engineering and Computer Sciences, Peking University, Beijing. His research interests include system software, system virtualization, and cloud computing.

Zhenlin Wang received his Ph.D. degree in computer science in 2004 from the University of Massachusetts, Amherst. He is now a full professor of Department of Computer Science, Michigan Technological University, Michigan, USA. His research interests are broadly in the areas of compilers, operating systems and computer architecture with a focus on memory system optimization and system virtualization. He is a recipient of NSF (National Science Foundation) Career Award.