

iBalloon: Efficient VM Memory Balancing as a Service

Qi Zhang, Ling Liu, Jiangchun Ren
Georgia Institute of Technology

{qzhang90, lingliu}@cc.gatech.edu, jiangchunren2015@gmail.com

Gong Su, Arun Iyengar
IBM Research - T. J. Watson
{gongsu, aruni}@us.ibm.com

Abstract—Dynamic VM memory management via the balloon driver is a common strategy to manage the memory resources of VMs under changing workloads. However, current approaches rely on kernel instrumentation to estimate the VM working set size, which usually result in high run-time overhead. Thus system administrators have to tradeoff between the estimation accuracy and the system performance. This paper presents *iBalloon*, a light-weight, accurate and transparent prediction based mechanism to enable more customizable and efficient ballooning policies for rebalancing memory resources among VMs. Experiment results from well known benchmarks such as Dacapo and SPECjvm show that *iBalloon* is able to quickly react to the VM memory demands, provide up to 54% performance speedup for memory intensive applications running in the VMs, while incurring less than 5% CPU overhead on the host machine as well as the VMs.

I. INTRODUCTION

Cloud providers often face the challenges of both achieving high resource utilization in their data centers and at the same time allocating enough resources for individual VMs to guarantee their performance. Employing the widely adopted virtualization technology, cloud providers can multiplex a single set of hardware resources among multiple VMs, therefore increasing resource utilization by the means of overcommitting. In order to guarantee the performance of individual VMs, one simple approach is to allocate resources according to their peak demand. However, this can result in significant resource under utilization because VMs' peak demands for resources can be much higher than their average demands. Therefore, the fundamental challenge in achieving both high resource utilization and performance guarantee at the same time lies in the fact that the resources demands of VMs can vary significantly over time.

CPU and memory are the two hardware resources having the most significant impact on a VM's performance. Modern Virtual Machine Monitors (VMMs, also known as Hypervisors) typically already support dynamically allocating a pool of CPUs among different VMs. Therefore, lots of existing researches are focused on exploring how to dynamically adjust memory allocation to meet a VM's changing demand, as is the case with this paper. Drawing from the lessons and experiences of previous work, we believe that there are two common issues need to be addressed: (1) monitoring the VM resource demand at a low cost to decide when and where to move memory

among the VMs; (2) moving memory among the VMs with minimal impact on the performance of the VMs.

To address the first issue, existing researches [17] [14] [20] have proposed many methods to predict the VM memory utilization. However, an accurate prediction of VM memory working set size is still a difficult problem, especially under changing conditions [6]. Because of the fact that hypervisor lacks the knowledge of VM memory access pattern, virtualization environment makes this prediction even harder. Some researches try to break the *semantic gaps* [8] between VMs and the host by instrumenting their kernels, which brings non-negligible performance overhead [20].

The second issue has been partially achieved by the introduction of memory balloon driver [18], which allows memory to be moved among the co-located VMs and the host machine. However, balloon driver cannot work by itself. In other words, system administrators have to be involved to periodically check the memory utilization of each VM and make the decision of how to balance the memory around. There are actually some efforts to make it work automatically [1], but the system is still in its initial stage and there lacks extensive experiments to evaluate its performance. Although some researchers propose ideas to rebalance memory among VMs by using balloon driver [16] [20], they also require guest kernel modification and the overhead incurred by memory access interception in these approaches can be very high.

In this paper, we propose *iBalloon*, which is a low cost VM memory balancer with high accuracy and transparency. No modification is required for VMs or the hypervisor to deploy *iBalloon*, which makes it more acceptable in real cloud environment. *iBalloon* runs a light-weighted monitoring daemon in each VM, which gathers the information about memory utilization of that VM. At the same time, a balancer daemon is running in the host to collect information reported by the monitor, and automatically makes the decision about how to balance the memory around VMs. The balancer finally talks to the balloon driver in the host machine to actually move the memory around. We implement the prototype of *iBalloon* on a KVM platform and the evaluation results show that with less than 5% performance overhead. *iBalloon* is able to improve VM performance by up to 54%.

The rest of the paper is organized as following: Section II introduces the design details of *iBalloon*. Section III discusses its implementation on the KVM virtualization platform. We

present our experimental methodology and explain the evaluation results of *iBalloon* in section IV. The related work is discussed in section V and the paper is concluded in section VI.

II. IBALLOON DESIGN

The goal of *iBalloon* is to keep a balanced memory utilization among VMs running on the same host while avoiding any VM from being deprived of free memory, with low cost and high accuracy and transparency. As shown in Figure 1, *iBalloon* consists of a Per-VM Monitor and a Balancer. Both the Per-VM Monitor and the Balancer are user level daemons. The Per-VM Monitor, which runs in the user space of each VM, is responsible for collecting information about the memory utilization of this VM. The Balancer, which consists of three parts: VM Classifier, Memory Balancer, and Balloon Executor, executes in the user space of the host. By using the *Exponentially Weighted Moving Average (EWMA)* model, the Balancer reads the information collected by the Per-VM Monitor, predicts each VM's future memory utilization, and makes decisions about how to rebalance the memory among the VMs. The Balancer then contacts the balloon driver to actually move memory among the VMs. Communications between the Per-VM Monitor and the Balancer are via an in memory bitmap and shared files, which are located on host and exported to the VMs by the Network File System (NFS). Since disk I/O can become a bottleneck when multiple VMs simultaneously write to the NFS directory, we put this shared directory in a memory based filesystem - *tmpfs*.

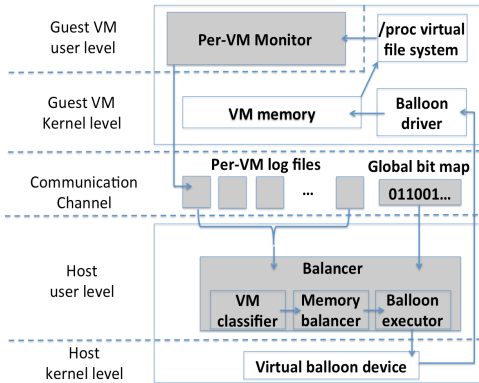


Fig. 1. *iBalloon* system overview

A. Per-VM Monitor

The Per-VM Monitor is a user level daemon running in each VM. It is responsible for periodically getting memory utilization statistics from the VM and writing them into a per-VM log file, which locates in a NFS directory provided by the host and shared by all the VMs running on this host. In our design, specifically, the monitor reads two metrics from the Linux virtual filesystem `/proc/meminfo`: *total memory* and *used memory*, and writes them into the log file. The log file can be maintained in either an appended only or an overwritten manner. The former method keeps adding the new

VM memory utilization statistics collected by the monitor to the end of the log file. The historical data in the log file may help the classifier in the host to better predict the VM's future memory usage, but the size of the log file will keep increasing with the execution of the VM, which may not be acceptable because of limited storage capacity. The latter approach always replaces the previous data in the log file with the newly collected information. In this case, the size of the log file will be small and constant, but the information provided by the file is limited. Therefore, in order to tradeoff between these two approaches, we design the per-VM log file in an overwritten manner, while keeping the aggregated VM historical memory utilization statistics in the VM Classifier running in the host. To be more specific, the per-VM log file only records two statistics: the VMs current *total memory* and *used memory*. The VM Classifier in the host will maintain a historical information indicator for each VM, and update this indicator periodically based on the current statistics read from the per-VM log file.

Three issues needs to be addressed in order to make *iBalloon* more scalable and accurate: updating interference, monitoring frequency, and transient outliers filtering.

Updating interference. The first issue is disk bandwidth interference incurred by the Per-VM Monitor to co-located VMs. Since the monitor in each VM periodically writes the VM memory statistics into its correspondent log file, it is quite possible that multiple VMs running on the same host are writing to their own log files simultaneously. Considering the fact that disk I/O requests from all these VMs have to go through the same host machine, it will lead to severe disk contention with the increasing number of VMs running with *iBalloon*, which could in turn degrade the performance of other I/O intensive VMs running on the host. Therefore, in our design, the NFS directory exported by the host is not created from disk, but from the memory. Considering the size of each VM log file is only two integers, the additional amount of memory taken by this design is negligible even if the number of VMs is very large.

Monitoring frequency. The second issue is the monitoring frequency. As the monitor in each VM periodically collects the VM's memory utilization statistics and communicates them to the *VM Classifier* in the host, the frequency of the monitor's execution greatly affects the scalability of the *iBalloon*, especially when there are large amount of VMs running on the same host. Accurate monitoring can be achieved by allowing a high monitoring frequency, however, it can lead to high computation and communication overheads. On the contrary, the data collected by a low frequency monitoring may not be accurate enough. Therefore, we employ an *adaptive frequency control* mechanism to build *iBalloon* scalable and accurate. Concretely, as described by algorithm 1, every monitor starts with a monitor interval τ . It checks whether the consecutive two monitored values vary within a pre-defined range λ . If yes, which means the memory utilization of this VM is in a relatively stable state, thus is it not necessary to update the current value in the log file. At the same time, the monitor

increases the value of current monitor interval by τ . Otherwise, the monitor updates the log file with the latest value and divides the value of *interval* by half. Note that there is an upper bound as well as a lower bound for the value of *interval*, in order to prevent Per-VM Monitor from being starved or executing too frequently. We empirically set n as 10 in our evaluation.

Algorithm 1 Per-VM Monitor

```

1: procedure VM_MEM_MONITOR
2:   while true do
3:     interval  $\leftarrow \tau$ 
4:     intervalmax  $\leftarrow n\tau$ 
5:     intervalmin  $\leftarrow \tau$ 
6:      $\gamma_{old} \leftarrow old\_vm\_mem\_util$  /*value from log file*/
7:      $\gamma_{new} \leftarrow get\_vm\_current\_mem\_util$ 
8:     if ( $|\gamma_{new} - \gamma_{old}| \geq \lambda$ ) then
9:       delay( $\beta$ )
10:       $\gamma_{delay} \leftarrow get\_vm\_current\_mem\_util$ 
11:      if ( $|\gamma_{delay} - \gamma_{old}| \geq \lambda$ ) then
12:        update_log_file
13:        interval  $\leftarrow MAX(interval_{min}, interval/2)$ 
14:      end if
15:      interval  $\leftarrow MIN(interval_{max}, interval + \tau)$ 
16:    end if
17:    sleep interval
18:  end while
19: end procedure

```

Transient outliers filtering. *iBalloon* should adjust a VM's memory when the VM is indeed short of memory, which is to guarantee the correctness and stability of the VM memory management. However, short term memory burst and transient outliers are often observed in a cloud environment. Therefore, in order to prevent a VM memory from going up and down dramatically, a delay is introduced in the *iBalloon* to filter these transient outliers. Concretely, as shown in algorithm 1, when the monitor detects that the difference between the old value and the latest value is obvious enough to issue an update to the log file, it delays for a β interval, and then checks the value for a second time. If the value is still satisfy the requirement of log file update, the monitor updates the log file. Otherwise, γ_{new} will be treated as a transient outlier.

B. VM Classifier

VM Classifier is one of the *iBalloon* components that running in the host user space. It is responsible to divide the VMs running on the host into three categories based on the their predicted memory utilization. The *Exponentially Weighted Moving Average (EWMA)* model is used in the VM Classifier for the prediction. Concretely, for a specific VM, the VM Classifier uses OFM_i to denote the VM's observed free memory, which is provided by the per-VM log file, in terms of percentage at time point i . At the same time, the VM Classifier maintains another variable PFM_i as the VM's historical information indicator. PFM_i represents the *EWMA* of the VM memory free memory from time point 0 to i . Then, according to *EWMA*, the predicted free memory in terms of percentage at time point i is based on the value of OFM_i and PFM_{i-1} . The predicted free memory for each VM is calculated as following:

$$PFM_1 = OFM_1$$

$$PFM_i = \alpha OFM_i + (1 - \alpha) PFM_{i-1}, i > 1$$

in which the value of α decides whether the prediction depends more on the current observed value OFM_i or the historical information PFM_{i-1} . We set the value of α as 0.125 in our evaluation.

After calculating the predicted free memory for each VM, the VM Classifier further divides the VMs into groups based on the prediction results. Three VM groups are defined as following:

$$VM = \begin{cases} Critical & \text{if } PFM_i \in [0\%, r_1) \\ Warn & \text{if } PFM_i \in [r_1, r_2) \\ Normal & \text{if } PFM_i \in [r_2, 100\%] \end{cases} \quad (1)$$

in which, r_1 and r_2 are two values between 0% and 100% and $r_1 \leq r_2$. We empirically set $r_1 = 15\%$ and $r_2 = 30\%$ in our evaluation. The VM groups created by the VM Classifier will be fed as the input to the Memory Balancer, which then, makes decisions about how to move memory around VMs accordingly. Another parameter passed from the VM Classifier to the Memory Balancer is a VM memory array *vm_mem_old[]*, which indicates the current memory utilization of each VM.

One thing needs to be mentioned is when to trigger the VM classification. In our design, the VM Classifier maintains a bitmap, which is shared among the VM Classifier and the Monitor. Each bit in the bitmap is correspondent to a specific VM. Whenever the Monitor updates the log file, it checks whether the state of the VM (i.e. *Normal*, *Warn*, *Critical*) is changed. If true, the Monitor will set its corresponding bit in the bitmap. On the VM Classifier side, it initializes the bitmap as all zero, and periodically checks the bitmap. The VM classification is triggered whenever the bitmap is non-zero, and the VM Classifier clears the bitmap to all zero again after the classification.

C. Memory Balancer

Based on the input provided by the VM Classifier, the Memory Balancer needs to decide how to move memory around VMs so that the memory utilization of each VM can be balanced. Algorithm 2 described how the Memory Balancer work. It firstly checks whether there exist any *Critical* VMs. On one hand, if *Critical* VMs exist, which indicates that these VMs urgently need more memory, then the Memory Balancer calculates δ , which represents the total amount of memory needed to bring these *Critical* VMs to a *Cushion* level, in which VM's free memory utilization reaches 20%. Then, the Memory Balancer follows a step by step manner to decide which VMs should sacrifice their memory and how much. The **first step** is the calculate δ_1 , which is the total amount of memory that can be taken from *Normal* VMs before making any of their free memory utilization drop to the *Warn* state. If δ_1 is already enough to satisfy all the *Critical* VMs, memory only needs to be proportionally moved from *Normal* VMs to *Critical* VMs. Otherwise, both *Normal* and *Warn* VMs need to scarifies their memory to help the *Critical* VMs, and

the Memory Balancer enters *step two*. After taking δ_1 from *Normal* VMs, all the *Normal* VMs will become *Warn* VMs. In step two, therefore, the Memory Balancer calculates δ_2 , which is the total amount of memory that can be taken from all the current *Warn* VMs before making any of them in *Critical* state. In this case, the Memory Balancer should guarantee that it will not turn any *non-Critical* VM into *Critical* VM after moving the memory. Therefore, if the total available memory from *non-Critical* VMs is not able to satisfy the demand of all *Critical* VMs, the Memory Balancer will issue a "short of physical memory" warning to the system administrator. On the other hand, if there exist both *Normal* and *Warn* VMs, but no *Critical* VMs, the Memory Balancer will move memory from *Normal* VMs to *Warn* VMs to balance the memory utilization between them. Otherwise, the Memory Balancer will stay idle if there exists only *Normal* or *Warn* VMs.

Algorithm 2 Memory Balancer

```

1:  $VMs[] \leftarrow vm\_mem\_old[]$ 
2:  $vm\_mem\_new[] \leftarrow NULL$ 
3:  $vm\_mem\_delta[] \leftarrow NULL$ 
4:  $\Delta_i, \Delta_j, \Delta_k, \eta \leftarrow 0$ 
5:  $\delta \leftarrow 0$  /*total memory needed by Critical VMs*/
6:  $\delta_1 \leftarrow 0$  /*maximum available memory from Normal VMs before any of them
dropping into Warn state*/
7:  $\delta_2 \leftarrow 0$  /*besides  $\delta_1$ , maximum available memory from non - Critical VMs
before any of them drop below Cushion state*/
8: procedure MEM_BALANCE
9:   if there exists any Critical VM in  $VMs[]$  then
10:    for each Critical VM  $VM_i$  do
11:       $\Delta_i \leftarrow mem\_needed\_by\_VM_i$ 
12:       $\delta \leftarrow \delta + \Delta_i$ 
13:    end for
14:    for each Normal VM  $VM_j$  do
15:       $\Delta_j \leftarrow mem\_available\_in\_VM_j$ 
16:       $\delta_1 \leftarrow \delta_1 + \Delta_j$ 
17:    end for
18:    if  $\delta_1 \geq \delta$  then
19:       $vm\_mem\_new = update(VMs[])$ 
20:      return
21:    else
22:       $set\_NormalVMs\_to\_WarnVMs$ 
23:       $\eta = \delta - \delta_1$ 
24:    end if
25:    for each Warn VM  $VM_k$  do
26:       $\Delta_k \leftarrow mem\_available\_in\_VM_k$ 
27:       $\delta_2 \leftarrow \delta_2 + \Delta_k$ 
28:    end for
29:    if  $\delta_2 \geq \eta$  then
30:       $vm\_mem\_new = update(VMs[])$ 
31:      return
32:    else
33:       $issue\_warning$ 
34:    end if
35:    else if there exists both Normal and Warn VMs in  $VMs[]$  then
36:       $average\_mem\_utilization$ 
37:    end if
38:     $vm\_mem\_delta[] = differ(vm\_mem\_new[], vm\_mem\_old[])$ 
39: end procedure

```

After all the calculations above, the Memory Balancer comes up with a new VM array $vm_mem_new[]$, which indicates the memory of each VM after the balancing. By comparing the $vm_mem_new[]$ with $vm_mem_old[]$, the Memory Balancer creates another array $vm_mem_delta[]$, which represents the memory movement that should be carried out by the Balloon Executor.

D. Balloon Executor

As a user level process running in the host, the Balloon Executor receives the $vm_mem_delta[]$ array from the Memory Balancer and invokes the balloon driver in the host to actually move memory around VMs. A positive number in $vm_mem_delta[]$ means the memory should be added to the correspondent VM, while a negative number means this VM needs to sacrifice its memory. Since the balloon driver itself does not support moving memory directly from one VM to another, the Balloon Executor should first take the memory from one VM to the host by inflating the balloon, and then move the memory from the host to the other VM by deflating.

In Balloon Executor, a straightforward method to invoke the balloon driver in the host kernel is using the $system()$ function to issue a shell command such as " $virsh qemu-monitor-command vm_id -hmp -cmd 'balloon target_mem_size'$ ". However, the overhead of $system()$ is high since it needs to fork a child thread in order to execute the shell command. Taking this into consideration, we created our own system call $vm_balloon(u64 vm_id, u64 target_mem)$ in the host to invoke the balloon driver with lower overhead.

III. IBALLOON IMPLEMENTATION

We have implemented an *iBalloon* prototype in KVM virtualization platform. The two main components of *iBalloon*, the Per-VM Monitor and the Balancer, are implemented in C as user space daemons. They communicate with each other through a memory based filesystem $tmpfs$.

In order to correctly reflect the memory pressure of each VM, the collector should distinguish the memory is actually used by the system from the cached/buffered memory. The Linux operating system usually uses the free memory as cache and buffers to reduce data access latency for applications and improve the disk I/O performance. Therefore, memory used as buffers and caches should not be counted as memory that is actually used. In other words, the large amount of memory used as buffers and caches does not mean that the system memory is under a high pressure.

Besides, the Linux operating system will start swapping pages out when there is still free memory available. From our observation, for example, there are usually 120MB free memory when a VM with 1GB memory starts swapping. The reason is that Linux kernel has set a watermark for each memory zone to guarantee that the free memory of each zone will not fall below the watermark. This OS-reserved free memory is used to deal with emergency memory allocation that can not fail. Therefore, the Per-VM Monitor should consider such memory as used memory in order to accurately reflect the memory pressure in the VM.

IV. EVALUATION

In this section, we present the evaluation of *iBalloon* prototype with several widely accepted benchmarks. We begin by introducing our experimental setup. Then, we measure the performance overhead of *iBalloon*, demonstrate how mixed

workloads can be benefited from *iBalloon*, and show the accuracy of *iBalloon* in terms of its VM memory prediction.

A. Experiments Setup

We conducted all experiments on an Intel Xeon based server provisioned from a SoftLayer cloud [4] with two 6-core Intel Xeon-Westmere X5675 processors, 8GB DDR3 physical memory, 1.5 TB iSCSI hard disk, and 1Gbit Ethernet interface. The host machine runs Ubuntu 14.04 with kernel version 4.1, and uses KVM 1.2.0 with QEMU 2.0.0 as the virtualization platform. The guest VMs also run Ubuntu 14.04 with kernel version 4.1. We evaluate *iBalloon* using the following benchmarks and applications:

- **Dacapo.** [2] It is a benchmark suit consists of a set of open source, real world Java applications with non-trivial memory loads. For example, some of the applications are *h2*, which executes a JDBCbench-like in-memory benchmark for executing a number of transactions against a model of banking application; *eclipse*, which executes some of the (non-gui) jdt performance tests for Eclipse IDE, and *xalan*, which transforms XML documents into HTML, etc.
- **SPECJVM2008.** [5] It focuses on the performance of the Java runtime environment (JRE) executing a single application. The results reflect the performance of hardware processor and memory subsystem. It has low dependence on file I/O and includes no network I/O across machines. SPECJVM2008 includes real life applications such as *javac compiler* as well as area-focused benchmarks, such as *xml*, *crypto*
- **Himeno.** [3] It is developed to evaluate performance of incompressible fluid analysis code. This benchmark takes measurements to proceed major loops in solving the Poisson's equation using Jacobi iteration method. The performance of Himeno is especially affected by the performance of memory subsystem.
- **QuickSort.** This is a quick sort program we developed ourselves in C. We feed it with large data sets to make them memory intensive.

B. Performance Overhead

This set of experiments evaluate the performance overhead *iBalloon* incurs on VMs. In order to separate the performance overhead of *iBalloon* and that brought by the balloon driver, we disable the *Balloon Executor* in the *Balancer*, thus *iBalloon* in this set of experiments will run as usually but not actually move memory around. As we mentioned earlier, the Per-VM Monitor needs to periodically collect the memory utilization statistics from the VM's */proc* virtual file system and update its log file which locates in a memory based file system. Therefore, the performance overhead of the Per-VM Monitor could be incurred from two aspects: data collecting and log file updating. Intuitively, the higher frequently the Per-VM Monitor runs, the more CPU overhead it will incur. Although an adaptive frequency control mechanism is employed in the Per-VM Monitor, we use a fixed frequency in this set of

experiments by setting a constant value σ_1 as the monitor's execution interval. Note the actual performance overhead should be no larger than what we have measured. Since according to algorithm 1, the Per-VM Monitor's execution interval will not be short than τ .

Similarly, the Balancer running in the host could also introduce performance overhead. Because every time it runs, the Balancer has to first fetch the data from multiple VM log files, then classify the VM based on a prediction based algorithm, and finally invoke the balloon device to move memory around. The performance overhead of the Balancer is related with two factors: the number of VMs running on the host, and how frequently the Balancer runs. Although the execution of Balancer depends on whether the updated information from Per-VM Monitors indicates that the state of VM has been changed, we still set a fixed execution interval σ_2 for the Balancer in this set of experiments, to see the upper bound of the performance overhead.

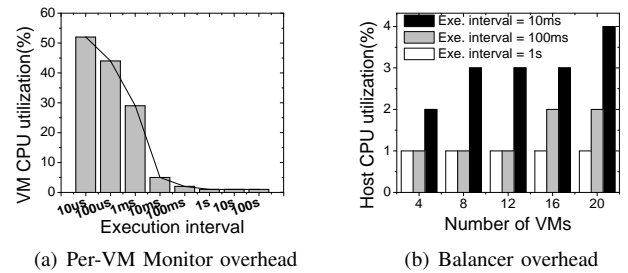


Fig. 2. Overhead of *iBalloon*

Figure 2(a) shows the VM's utilized CPU when the Per-VM Monitor is running with different frequencies. As we mentioned earlier, we vary the monitor's execution interval between different runs, but the interval is fixed for each single test. It shows that the VM's busy CPU stays as low as 1% when the Per-VM Monitor executes as frequent as every 1 second. The percentage of busy CPU begins to increase slight to 2% and 5% when the execution interval decreases to 100ms and 10ms. When the interval is shorter than 10ms, the percentage of busy CPU climbs up quickly, for example, 52% CPU is busy when the *Per-VM Monitor* execution interval decreases to 100us. From this set of experiments, we can tell that the performance overhead of Per-VM Monitor is negligible when it executes no less than every 1 second.

Figure 2(b) displays the overhead brought by the Balancer in terms of host CPU utilization. We vary the number of VMs from 4 to 20 and the execution interval of the *Balancer* from 10ms to 1s. The overhead of *Balancer* only slightly grows with the increase of its execution frequency and number of VMs. For example, when the execution interval is 10ms, the overhead increases from 2% to 4% when the number of VMs varies from 4 to 20. The overhead stays the same (1%), when the execution interval increases to 1 second. As we mentioned, since the overhead displayed in table 2(b) are supposed to be higher than that in the real case, these results indicate that the overhead of the Balancer is also negligible. Based on the

experimental results above, we set τ as 5 seconds in section `refmixworkloads`.

C. Mixed Workloads

In this subsection, we demonstrate the effectiveness of *iBalloon* in an environment with mixed workloads by deploying it in a host with 4VMs running simultaneously. As illustrated in Figure 3, VM1 runs *Dacapo*, VM2 runs *Dacapo-plus*, VM3 runs *crypto.rsa* and *Himeno*, VM4 runs *QuickSort*. *Dacapo* includes both CPU intensive and memory intensive workloads, and they are executed sequentially in our experiments. The workloads in *Dacapo-plus* are the same as those in *Dacapo*, but are executed in a different order to create a mix of CPU and memory demand. *crypto.rsa* is a CPU intensive workload, while both *Himeno* and *QuickSort* are memory intensive. We evaluated and compare the performance of 3 cases: (1) Baseline, in which the VM memory is allocated statically; (2) Ramdisk Swap, in which a ramdisk is mounted to each VM as its swap area; (3) *iBalloon*, in which *iBalloon* is used to dynamically balance the VM memory. The VM swap area is hard disk is case the *iBalloon* case.

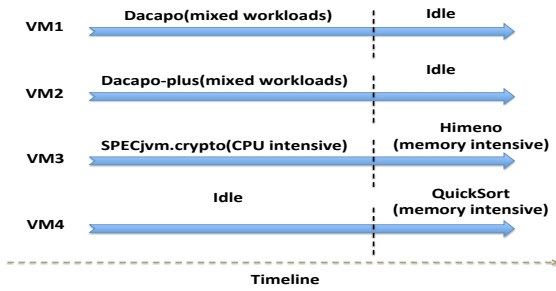


Fig. 3. Mixed workload experiments setup

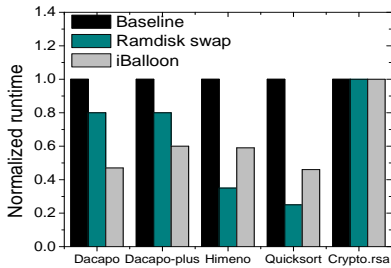


Fig. 4. Normalized performance of benchmarks

Figure 4 compares the total execution time of *Dacapo*, *Dacapo-plus*, *Himeno*, and *QuickSort*. It shows that first, with *iBalloon*, the total execution time of *Dacapo* and *Dacapo-plus* benchmarks are reduced by 52% and 50% respectively, while that of *Himeno* and *QuickSort* have been reduced by 40% and 54%, which demonstrates the effectiveness of *iBalloon* in terms of improving the performance of applications running inside VMs. Second, although using ramdisk as VM swap area can improve the application performance to some extent, it is

TABLE I
EXECUTION TIME OF REPRESENTATIVE WORKLOADS IN DACAPO AND DACAPO-PLUS (MS)

		Baseline	Ramdisk swap	iBalloon
VM1	eclipse	214,224	169,774	72,922
	h2	44,408	30,135	20,781
	jython	17,292	14,577	10,892
	fop	3,307	3,131	3,399
VM2	eclipse	168,505	124,871	93,785
	h2	36,390	26,919	13,990
	jython	12,842	12,140	11,898
	fop	3,168	3,132	3,341

still not as effective as using *iBalloon*. Taking the *Dacapo* benchmark for an example, its execution time in the *Ramdisk swap* cases is about 20% shorter than that in the *Baseline*, but still about 60% longer when compared with the *iBalloon* case. This is because even using ramdisk as the swap area is more efficient than using hard disk, each swap-in/swap-out operation from VM still needs to go through the block I/O layer of both the VM and the host, which will lead to much higher performance overhead compared with directly increasing the VM's memory capacity. Third, the different from *Dacapo* and *Dacapo-plus*, the performance of *Himeno* and *QuickSort* in the *Ramdisk swap* case is better than that in the *iBalloon* case. The reason is that the memory requirement of *Himeno* and *QuickSort* has exceeded the total amount of physical memory on the host, VM3 and VM4 have to swap their memory out even with *iBalloon*. Therefore swapping to the ramdisk will help more than using *iBalloon* and swapping to the hard disk.

Table I zooms into the execution of *Dacapo* and *Dacapo-plus*, and shows the execution time of some representative workloads. We find that first, *eclipse* and *h2* have the most obvious performance improvement among all the workloads in *Dacapo* or *Dacapo-plus*. For example, in VM1, the execution time of *eclipse* has been reduced from 214224ms to 72922ms, while that of *h2* has dropped from 44408ms to 24814ms, and similar trend can also be observed from VM2. The reason is that *eclipse* and *h2* are the most memory intensive workloads in the suit, which result in about 580MB and 650MB memory swapping without *iBalloon*. At the same time, the execution time of some other less memory intensive workloads is slightly reduced. For instance, the execution time of *jython* reduces by 39% from 17292ms to 10892ms. An interesting observation is that the execution time of *jython* has been slightly increased in VM2 when *iBalloon* is used. This is because before *eclipse* starts to execute in VM2, memory has been moved to other VMs by the *iBalloon* to satisfy their needs. So it takes time for *iBalloon* to move memory back to VM2 when *eclipse* needs it.

Besides memory intensive benchmarks, a CPU intensive benchmark is also running simultaneously in VM3. Figure 4 shows the normalized throughput of *crypto.rsa*, which is the CPU intensive benchmark running in VM3 while *Dacapo* is running in VM1 and *Dacapo-plus* is running in VM2. It shows that the variation among the execution time of *crypto.rsa* in

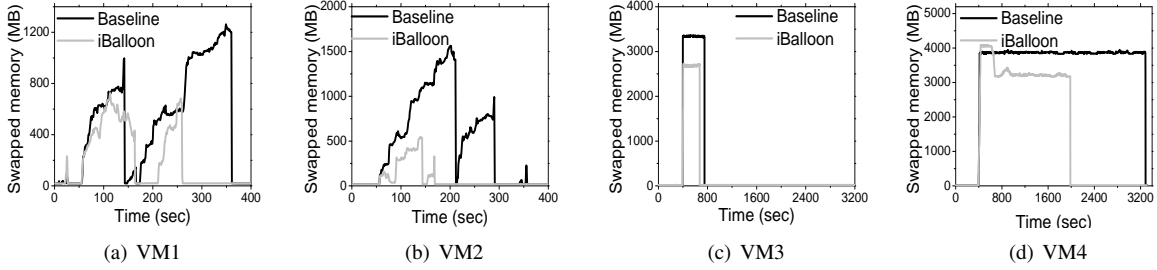


Fig. 5. Swapped memory in VMs

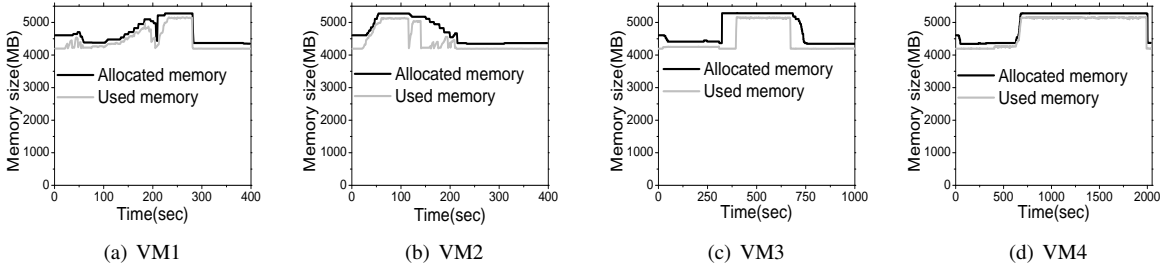


Fig. 6. Allocated memory vs. used memory in VMs working with *iBalloon*

all cases is within 4%. This demonstrates that *iBalloon* is able to improve the performance of memory intensive applications, while having a very low impact on that of CPU intensive ones.

Figure 5 further illustrates the amount of swapped memory in the 4 VMs at different time point during the experiment. In each sub-figure, the amount of swapped memory in cases with and without *iBalloon* are compared. It shows that the *iBalloon* significantly reduces the amount of memory pages that need to be swapped out. Take VM2 for example, the memory swapping demand lasts about 285 seconds with peak value above 1500MB in baseline. While in the *iBalloon* case, VM2’s memory swapping stops at about 175 seconds, and the maximum required swapping space is between 500-600MB. For VM3 and VM4, their memory intensive workloads did not start until 400th second. *iBalloon* moves other VMs’ free memory to VM3 first to satisfy its memory intensive benchmark *Himeno*. After *Himeno* finishes at around 700th second, *iBalloon* then moves the free memory to VM4 to help the execution of *QuickSort*.

Figure 6 compares the allocated memory with the amount of memory that is actually used in each VM when working with *iBalloon*. We find that *iBalloon* is able to appropriately adjust the memory size of each VM based on its workload demands, which prevents the VMs from wasting their memory resources. An interesting observation is that the allocated memory jumps up before the used memory in VM3 at 287th second. This is because right after *Dacapo* and *Dacapo-plus* finish execution, most of the memory has been moved from VM3 and VM4 to VM1 and VM2. At this point of time, *iBalloon* takes place to rebalance the memory among the 4 VMs before *Himeno* and *QuickSort* starts to run in VM3 and VM4.

V. RELATED WORK

VM working set size estimation. Dynamic VM memory allocation and VM memory deduplication are the two major mechanisms that are proposed to increase the memory utilization in virtualized environment. Accurate VM memory working set size estimation is essential to the performance of dynamic memory management mechanisms.

Pin *et al.* [21] proposed using page miss ratio as a guidance of VM memory allocation. However, the tracking of page miss ratio is implemented through using a specific hardware, which is not easy to accomplish, or modifying the OS-kernel, which can result in unacceptable performance overhead.

Zhao *et al.* [20] proposed using LRU histogram to estimate the VM memory working set size. In their method, memory accesses from each VMs are intercepted by the hypervisor to build and update the LRU histograms. They introduced the concept of *hot pages* and *cold pages* to alleviate the performance cost incurred by memory access interception. But according to the evaluation result in their paper, there is still considerable performance overhead. Besides the performance overhead, accurate VM working set size prediction is difficult under chaining conditions [10] [13] [9]. Therefore, we design *iBalloon* which estimates the VM working set size via lightweight daemons, and more importantly, makes efforts to guarantee the VM performance even if the estimation is not accurate by using shared memory swapping.

Balloon Driver vs. Memory Hotplug. In order to handle the dynamic VM memory demands and increase the memory utilization in virtualized environment, balloon driver[18] was proposed in 2002 and has been widely adopted in mainstream virtualization platforms such as KVM[11], Xen[7], VMware[15], and etc. Similarly, memory hotplug[12][16] is

another technique aiming at reducing wasted memory by enabling memory to be dynamically added to and removed from VMs. Some researchers [19] explored using shared memory to increase the physical memory utilization while maintaining good VM performance. Also, there are several other works focusing on comparing balloon driver with memory hotplug from in terms of their performance and functionality.

Liu, *et al.*[12] conducted a comparative study between balloon driver and memory hotplug. They mentioned that the implementation of balloon driver is far more straightforward than memory hotplug. Since balloon driver is able to directly use the native MMU of the guest. However, balloon driver cannot enlarge the memory size of a VM beyond its *cap*, which is a preset parameter associated with each VM. Memory hogplug can go beyond the *cap*. Another finding from their work is memory hotplug should have a better scalability than balloon driver. Since balloon driver relies on the buddy system of guest MMU, which results in memory fragmentation problems. But memory hogplug can avoid this problem by adding or removing memory by a whole section.

Schopp, *et al.* [16] concisely explained how balloon driver and memory hotplug work and compared their advantages and disadvantages respectively. For example, memory hotplug allows adding memory that was not present at boot time to scale Linux up in response to changing resources, and there is no *cap* for memory hotplug to add memory. But memory hotplug has limitations on not being able to remove memory containing certain kinds of allocations. Balloon driver is able to directly use the native memory management in VM, but it could fragments the pseudophysical memory map of the guest VM.

VI. CONCLUSIONS AND FUTURE WORK

We have proposed *iBalloon*, a lighted-weighted, high accurate and transparent VM memory balancing service. *iBalloon* consists of two major components: the Per-VM Monitor and a global Balancer. *iBalloon* predicts the VM memory utilization based on Exponentionally Weighted Moving Average (EWMA) model and dynamically adjust the VM memory accordingly. We evaluate the performance of *iBalloon* by using various widely accepted benchmarks and applications in a complex environment where multiple VMs running simultaneously. The results show that, with only up to 5% performance overhead, *iBalloon* is able to accurately adjust VM memory based on its real-time requirement, and greatly improve performance of applications running in the VMs by up to 54%. There are a number of extension for *iBalloon* we are considering in the future. For example, the performance of balloon driver can to be further improved. Based on our investigation, current balloon driver moves memory in a page by page manner, which may not be optimal. Batching operation could be applied to achieve better performance.

VII. ACKNOWLEDGMENTS

The first two authors performed this work under the partial support by the National Science Foundation under Grants IIS-

0905493, CNS-1115375, NSF 1547102, SaTC 1564097, and Intel ISTC on Cloud Computing. The third author is a visiting scholar while collaborating on this project, funded by China with the School of Computer, NUDT, China.

REFERENCES

- [1] Auto Ballooning. <http://www.linux-kvm.org/page/Projects/auto-ballooning>.
- [2] Dacapo. <http://www.dacapobench.org>.
- [3] Himeno. <http://accr.riken.jp/2444.htm>.
- [4] Softlayer. <http://www.softlayer.com>.
- [5] SPECjvm2008. <http://www.spec.org/jvm2008>.
- [6] AMIT, N., TSAFRIR, D., AND SCHUSTER, A. Vswapper: A memory swapper for virtualized environments. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014), ACM, pp. 349–366.
- [7] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 164–177.
- [8] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real [operating system relocation to virtual machines]. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on* (2001), IEEE, pp. 133–138.
- [9] HINES, M. R., GORDON, A., SILVA, M., DA SILVA, D., RYU, K. D., AND BEN-YEHUDA, M. Applications know best: Performance-driven memory overcommit with ginkgo. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on* (2011), IEEE, pp. 130–137.
- [10] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. In *ACM SIGOPS Operating Systems Review* (2006), vol. 40, ACM, pp. 14–24.
- [11] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.
- [12] LIU, H., JIN, H., LIAO, X., DENG, W., HE, B., AND XU, C.-Z. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines.
- [13] LU, P., AND SHEN, K. Virtual machine memory access tracing with hypervisor exclusive cache. In *Usenix Annual Technical Conference* (2007), pp. 29–43.
- [14] MOLTÓ, G., CABALLER, M., ROMERO, E., AND DE ALFONSO, C. Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements. *Procedia Computer Science* 18 (2013), 159–168.
- [15] ROSENBLUM, M. Vmware’s virtual platform. In *Proceedings of hot chips* (1999), vol. 1999, pp. 185–196.
- [16] SCHOPP, J. H., FRASER, K., AND SILBERMANN, M. J. Resizing memory with balloons and hotplug. In *Proceedings of the Linux Symposium* (2006), vol. 2, p. 313319.
- [17] TASOULAS, E., HAUGERUND, H., AND BEGNUM, K. Baylocator: a proactive system to predict server utilization and dynamically allocate memory resources using bayesian networks and ballooning. In *Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques* (2012), USENIX Association, pp. 111–122.
- [18] WALDSPURGER, C. A. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 181–194.
- [19] ZHANG, Q., AND LIU, L. Shared memory optimization in virtualized cloud. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on* (2015), IEEE, pp. 261–268.
- [20] ZHAO, W., WANG, Z., AND LUO, Y. Dynamic memory balancing for virtual machines. *ACM SIGOPS Operating Systems Review* 43, 3 (2009), 37–47.
- [21] ZHOU, P., PANDEY, V., SUNDARESAN, J., RAGHURAMAN, A., ZHOU, Y., AND KUMAR, S. Dynamic tracking of page miss ratio curve for memory management. In *ACM SIGOPS Operating Systems Review* (2004), vol. 38, ACM, pp. 177–188.