J Grid Computing (2012) 10:69–84
DOI 10.1007/s10723-012-9209-4

VMMB: Virtual Machine Memory Balancing for Unmodified Operating Systems

Changwoo Min · Inhyeok Kim · Taehyoung Kim · Young Ik Eom

Received: 16 August 2011 / Accepted: 1 March 2012 / Published online: 28 March 2012

Abstract Virtualization technology has been widely adopted in Internet hosting centers and cloud-based computing services, since it reduces the total cost of ownership by sharing hardware resources among virtual machines (VMs). In a virtualized system, a virtual machine monitor (VMM) is responsible for allocating physical resources such as CPU and memory to individual VMs. Whereas CPU and I/O devices can be shared among VMs in a time sharing manner, main memory is not amendable to such multiplexing. Moreover, it is often the primary bottleneck in achieving higher degrees of consolidation. In this paper, we present VMMB (Virtual Machine Memory Balancer), a novel mechanism to dynamically monitor the memory demand and periodically re-balance the memory among the VMs. VMMB accurately measures the memory demand with low overhead and effectively allocates memory based on the memory demand and the QoS requirement of each VM. It is applicable even to guest OS whose source code is not available, since VMMB does not require modifying guest kernel. We implemented our mechanism on Linux and experimented on synthetic and realistic workloads. Our experiments show that VMMB can improve performance of VMs that suffers from insufficient memory allocation by up to 3.6 times with low performance overhead (below 1%) for monitoring memory demand.

Keywords Virtualization · Memory balancing · LRU histogram · Double paging

C. Min · I. Kim · T. Kim · Y. I. Eom (✉)
School of Information & Communication Engineering, Sungkyunkwan University, 300 Cheoncheon-Dong, Jangan-Gu, Suwon, Gyeonggi-Do 440-746, Korea
e-mail: yieom@ece.skku.ac.kr

C. Min
e-mail: multics69@ece.skku.ac.kr

I. Kim
e-mail: kkojiband@ece.skku.ac.kr

T. Kim
e-mail: kim15m@ece.skku.ac.kr

C. Min
Samsung Electronics, 416, Maetan-3Dong, Yeongtong-Gu, Suwon, Gyeonggi-Do, 443-742, Korea

1 Introduction

Virtualization technologies are becoming ubiquitous in various domains, such as data centers, web hosting and even in desktop computing. Moreover, in recent years, there are many efforts to utilize virtualization technologies in scientific computing [1–3]. The key benefit of virtualization is to improve the hardware resource utilization by running multiple VMs on a single physical

hardware. VMM controls all hardware resources while offering each VM an illusion of having dedicated raw machines by virtualizing the hardware. Though CPU and I/O devices can be efficiently shared by multiplexing, it is still challenging to share memory efficiently in accordance with memory needs [4–6]. Increasing a machine's physical memory is often difficult and expensive, because it is subject to the availability of extra memory slot and the support of higher-capacity memory module on the mother board. Moreover, main memory consumes up to 40% of server energy that is comparable to or slightly higher than the energy consumption of processors [7].

To efficiently share memory in a virtualized data center, there are two approaches in complementary relation. First, dynamic memory balancing approaches estimate the memory demand of each VM and then dynamically allocate the memory according to the demands [5, 8–11]. Second, VM migration approaches find overloaded VMs and relocate them to the idle physical machines [12–18]. During the migration, a VM is completely stopped and migration traffic negatively impacts to a data center. Since it is a relatively heavyweight solution, the number of total migration in a data center should be minimal. Therefore, the VM migration techniques are better suited to sustained overload. On the contrary, dynamic memory balancing techniques are better suited to transient workload, since it is more lightweight than the migration techniques. Williams et al. [19] analyze log data from the production data center to investigate characteristics of the memory overload. They show that the memory overload is largely transient, lasting for less than 2 min, and the average number of overloaded servers in a data center is quite small, 1.76% in their research. It implies that VMs sharing a physical machine are unlikely to experience correlated memory overload at the same time.

Considering the characteristics of the memory overload in a data center, dynamic memory balancing techniques have potential to alleviate the majority of the memory overload. However, they still have limitations. Statistical sampling [8] cannot estimate memory demand larger than the currently allocated memory size. Geiger [10] cannot estimate memory demand smaller than the current allocation, since it relies on buffer cache eviction. Moreover, because it relies on page fault and page table update information to trace buffer eviction, it is not compatible with hardware MMU virtualization support that eliminates traps to VMM when manipulating guest page table. Hypervisor exclusive cache [11] can deduce growing and shrinking of memory demand. However, it cannot be used for OS without source code because it requires modifying guest OS to monitor buffer cache eviction and promotion. Zhao et al. [5] proposed an LRU histogram [20] based approach for memory demand estimation. However it suffers from performance overhead for large working set, because the cost to calculate to update LRU histogram is linearly increasing to the working set size.

In this paper, we introduce VMMB (Virtual Machine Memory Balancer), a novel mechanism to dynamically re-balance the memory allocation among the VMs. VMM monitors the memory usage of each VM to estimate the memory demand, and then it periodically re-balances the memory based on the estimated memory demand. The distinguishing features of VMMB compared to prior work are as follows: First, VMMB can estimate the memory demand above and below the allocated size by using the LRU histogram without modifying the guest kernel. This is achieved by using the nested page faults for monitoring the memory access and a pseudo swap device for monitoring the guest swapping. We also present techniques to efficiently construct the LRU histogram. Second, we propose a LRU histogram guided memory allocation algorithm that has QoS support. Basically, the proposed algorithm determines the memory allocation size of each VM along with globally minimizing the page miss ratio. If a VM has a different QoS requirement, it reflects the QoS requirement on the memory allocation decision. Finally, we use a combination of ballooning [8] and VMM-level swapping in order to efficiently select the victim pages and to immediately allocate memory to a beneficiary VM. For an efficient VMM-level swapping, we introduce a technique to avoid double paging anomaly [21, 22] that can seriously degrade the performance when the same page is selected as a victim by the VMM and VM.

The remainder of this paper is organized as follows: Section 2 describes the detailed design and implementation of the system. In Section 3, we show the experimental results of VMMB. Section 4 presents the related work for the proposed scheme. Finally, in Section 5, we summarize our conclusions and suggest future directions.

2 Design of the System

In this section, we describe the design of the proposed system. We begin by providing the overall system architecture and continue by describing three design issues: (a) the efficient construction of the LRU histogram without modifying guest kernel, (b) re-balancing the memory dynamically based on the memory demand of each VM, and (c) efficient VMM-level swapping.

2.1 System Overview

As we illustrate in Fig. 1, VMMB consists of three parts. First, the VMM intercepts the memory access and the swapping operations from the VM and builds up an LRU histogram. We maintain a per-VM page list wherein the pages are ordered from the most to the least recently accessed. The LRU histograms are updated by calculating the stack distance [20] of the accessed page in the page list.

Second, the VMM periodically decides the memory allocation size for each VM to minimize the system-wide page miss ratio that is induced from the LRU histograms. If a VM has a different QoS requirement, it is also incorporated in the allocation size calculation. The new memory allocation sizes are enforced via the ballooning technique. When a VM reclaims memory, it reports the reclaimed pages back to the VMM so the VMM can allocate those pages to other VMs.

Finally, the VMM employs VMM-level swapping as a secondary memory reclamation mechanism from the VMs. The VMM preferentially uses ballooning to reclaim memory. However if the memory is not sufficiently reclaimed on time, the system falls back to a swapping mechanism. The VMM swaps out pages to the swap device of a victim VM. The swap device maintains which pages are swapped out by the VMM to avoid double paging.

2.2 Building the LRU Histogram

To construct the LRU histogram, we monitor the memory accesses and swapping operations from the VM. Monitoring the memory accesses is required to estimate a memory demand below the current allocation. We turn off the presence bits of the nested page table [23] to trap the memory access without modifying the guest kernel. Monitoring the swapping operations is needed to estimate memory demands above the current

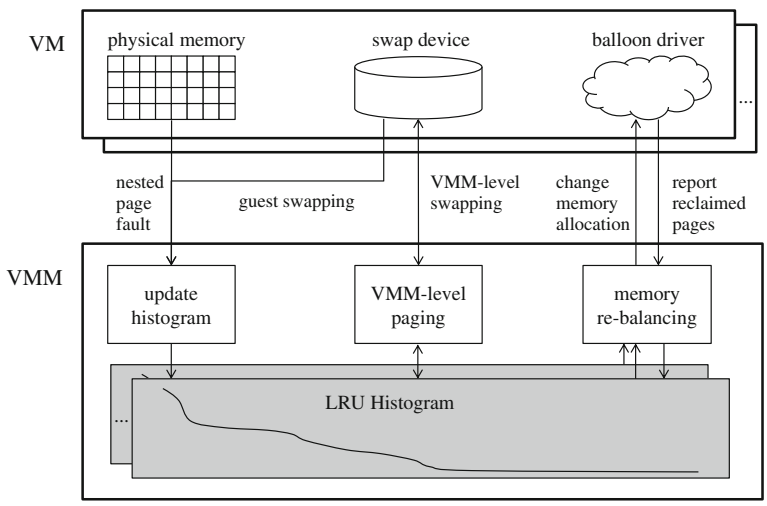


Fig. 1 System architecture

allocation. We propose a *virtualization aware swap device* (VSWAP) to monitor the guest swapping operations. VSWAP is composed of a front-end driver and a back-end driver. The front-end driver runs in the guest kernel and communicates with the corresponding back-end driver. Since in terms of the guest kernel, the front-end driver is the equivalent of a typical block device driver, it does not require modifying the guest kernel. The back-end driver performs the actual I/O operations and notifies the VMM of which guest physical pages are swapped in/out. The details of VSWAP will be discussed in Section 2.4.

Since the number of pages in the page list grows as a VM increases memory usage, the overhead of updating LRU histogram needs to be minimal. To minimize performance overhead, we design the page list structure as illustrated in Fig. 2. The VMM maintains the page list for each VM using three different groups: *hot list*, *warm list* and *cold list*. The hot list contains the most recently accessed pages. To reduce the monitoring overhead, the pages in the hot list are not monitored by turning on the presence bits. Our system dynamically changes the capacity of the hot list to keep a balance between the monitoring overhead and the accuracy of LRU histogram. The cold list holds the pages evicted by the VM or the VMM. Each list has a limited capacity for holding pages. A new page is added to the head of the hot list and the accessed page is moved to the head of hot list. If the addition of the page causes the hot list to go beyond its capacity, the tail of the hot list is moved to the head of the warm list. If the VM exceeds the available memory size of the VMM, the VMM evicts the tail of the warm list to the VSWAP, and then moves it to the head of the cold list. Similarly, when the VM evicts a page, it is moved to the head of the cold list. The size of the hot list and the warm list is controlled by an adaptive mechanism described in Section 2.3.2.

Algorithm 1: Calculating the distance

```
Input: weighted red-black tree T, node n
Output: distance d
begin
    d ← weight of n's left subtree +1
    p ← parent node of n
    while p is not nil do
        if n is in the right subtree of p then
            d ← d+ weight of p's left subtree +1
        end
        p ← parent node of p
    end
end
```

To update the LRU histogram, we must first find where the trapped page is in the page list. If it is in the page list, the stack distance is calculated, and the corresponding entry of the LRU histogram is incremented by one. After that, the page is moved to the head of the page list in order to maintain the order from the most to the least recently accessed. The most expensive operation in updating the LRU histogram is to calculate the stack distance of a page in the page list structure. A naive linear search from the head of the list takes the worst-case linear time. It is inefficient and not scalable. To calculate the stack distance efficiently, we propose a *weighted red-black tree*. A weighted red-black tree is an extension of red-black tree [24] whose node is annotated by *weight*. Weight represents the total number of nodes in its subtree. The distance can be efficiently calculated in the worst-case logarithmic time, as described in Algorithm 1. To correctly maintain the weights, we extend four red-black tree operations that

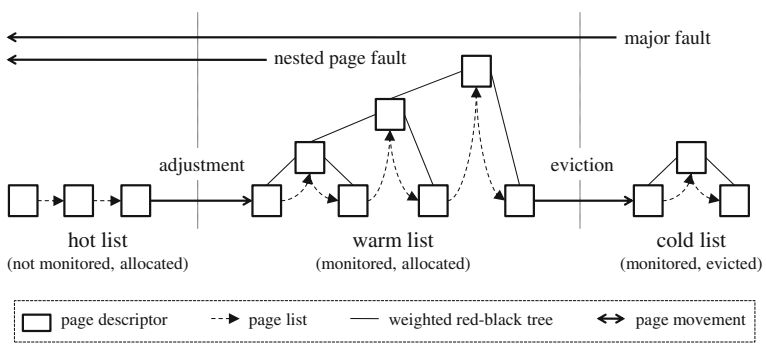


Fig. 2 The page list structure of a VM

change the tree height: insert, delete, left-rotate, and right-rotate. When a new node is inserted, its initial weight is assigned as 1 and the weights of its ancestors are increased by 1. Similarly, when a node is deleted, the weights of its ancestors are decremented by 1. For the left-rotate and right-rotate operations, we update the weights of the rotated nodes. Because our system does not monitor hot pages, the weighted red-black trees are attached to the only warm and cold lists.

2.3 Dynamic Memory Balancing

Based on the per-VM LRU histograms, the VMM periodically re-balances the memory allocation among the VMs. This process is composed of three steps: (a) calculating the suitable memory allocation size considering the overall system performance and the QoS requirement of each VM, (b) changing the capacity of the hot list of each VM in order to manage the monitoring overhead and the accuracy of the LRU histogram, and (c) imposing a new memory allocation size for each VM and reclaiming memory if required. In our experiments, we re-balance every 6 s.

2.3.1 QoS Aware Memory Allocation

When multiple VMs are competing for memory resources, it is generally desirable minimizing the system-wide page miss ratio by re-balancing the memory. However there are often conflicts with QoS requirements of the VMs that have different importance factors. For example, the importance can be defined by the penalty of failure, the required performance, or the service level agreement (SLA). Therefore, the memory allocation scheme in the virtual machine environment should consider both the global system performance and the QoS requirements of individual VMs.

To determine memory allocation for each VM, we propose a *QoS aware lookahead algorithm* derived from the lookahead algorithm of Qureshi et al. [25]. It minimizes the system-wide page miss ratio while considering the QoS requirements from the individual VMs. To control these conflicting goals, we provide two explicit parameters, M_i and w_i, that enables administrator to describe the QoS requirements. M_i is the minimum memory size of VM_i that should be allocated at any time. w_i is the relative importance of VM_i. We describe the details of the QoS aware lookahead algorithm in Algorithm 2. We first allocates M_i for VM_i in order to guarantee the minimum memory allocation. The remaining memory R is divided according to the effectiveness of additional memory allocation and QoS requirement of each VM. Assuming all remaining memory is allocated to a VM, the maximum delta is defined by the amount of decreasing page miss ratio per unit of additional memory allocation (`get_max_delta`). It represents the effectiveness of additional memory allocation for a VM. To consider the effectiveness and the importance of a VM together, we define Δ_i, which is calculated by multiplying the maximum delta and w_i. The VM with the greatest Δ_i gets the additional memory whose size is the minimum for achieving Δ_i. This process is repeated until all memory is assigned.

The worst case of this process is when all of the M_is are zero and so only one unit is assigned at every iteration. It takes $U + (U-1) + (U-2) + \ldots + 1 = \frac{U(U-1)}{2} \approx \frac{U^2}{2}$ where U is the number of units to allocate. In our experiments, we set the unit size to 1 MB.

Algorithm 2: QoS aware lookahead algorithm

```
Input: L_i, LRU histogram of each VM
Output: A_i, memory allocation size of each VM
begin
    A_i ← M_i for each VM i
    R ← T − Σ_{i=1}^{N} M
    while R do
        foreach VM i do
            Δ_i ← w_i · get_max_delta(i, R)
            B_i ← min memory to get Δ_i for i
        end
        s ← VM with maximum value of Δ
        A_i ← A_i + B_s
        R ← R − B_s
    end
end

get_max_delta(i, R):
begin
    Δ ← 0
    for j ← 1 to R do
        δ ← ( Σ_{k=A_i}^{A_i+j} L_i[k] ) / j
        if δ > Δ then Δ ← δ
    end
    return Δ
end
```

2.3.2 Adaptive Changes to the Hot List Capacity

A workload that heavily accesses the memory causes a large overhead due to too frequent histogram updates. To balance the monitoring overhead and the accuracy of the LRU histogram, we dynamically change the capacity of hot list, i.e. the number of non-monitored pages. The capacity of the hot list H_i is determined for monitoring only acceptable number of nested page faults for a balancing interval. Let E_i^t be the expected hot list capacity for an VM_i at the t-th interval. We estimate the expected capacity E_i^t, using the LRU histogram and the number of faults monitored in the previous interval. E_i^t is defined as follows:

$$E_i^t = \min\left(\max_{e \in E}(e),\ A_i\right) \tag{1}$$

$$E = \left\{ e \,\middle|\, \sum_{j=e}^{T} L_j \cdot \frac{F^{t-1}}{M^{t-1}} \geq F \right\} \tag{2}$$

where F is the target number of faults for an interval, F^{t-1} is the actual number of faults monitored during the previous interval, and M^{t-1} is the sum of the histogram entries for the warm and cold lists from the previous interval. Since it does not make sense for H_i to be larger than the memory allocation size A_i, E_i^t should be bounded by A_i. We finally determine H_i based on the predicted E_i^t. If the estimation from the histogram gives a new value (i.e. $E_i^t \neq E_i^{t-1}$), we select E_i^t as H_i. Otherwise, we adaptively change the capacity based on the gap between F^{t-1} and F as follows:

$$H_i^t = H_i^{t-1} \cdot \left(1 + \gamma \cdot \frac{F^{t-1} - F}{F}\right) \tag{3}$$

where γ is a positive coefficient smaller than 1. Because our approach estimates the capacity of the hot list directly from the LRU histogram, it becomes possible to quickly adapt. Therefore, our method performs with a small overhead even when memory demand changes rapidly. In our experiments, F is set to 16,384 pages (256 MB given the page size of 4 KB and page group size of 4) and γ is set to 5%.

2.3.3 Reclaiming the VM Memory

After the memory size A_i for a VM is decided, the VMM asks the VMs to set their memory usage to A_i. When a VM needs to reduce its memory usage, a balloon driver on the guest OS allocates the pinned memory. If the memory is available, the guest OS will allocate the memory from its free list. Otherwise, the guest OS will reclaim the least valuable memory. The balloon driver reports the allocated pages back to the VMM. The VMM deletes the reported pages in the page list in order to assign them to the other VM. When the VMM allows a VM to use more memory, the balloon driver frees the pinned memory for the guest OS to allocate when required.

2.4 VMM-level Swapping

In the ballooning technique, a victim VM should be scheduled to release its memory. Thus, there is a scheduling delay until the victim VM actually releases the memory. If a beneficiary VM asks additional memory before memory reclamation, a VMM swaps out a memory from the victim VM and immediately allocates the memory to the beneficiary VM. The VMM-level swapping can allocate memory without such scheduling delay. However, it is inefficient because it can result in a double paging anomaly when the VMM-level and the guest page replacement policies are aligned.

VSWAP that we presented in Section 2.2 has two purposes: monitoring the guest swapping without modifying the guest kernel, and performing VMM-level swapping without double paging. We illustrate the architecture of VSWAP back-end driver in Fig. 3. It has two external interfaces that connect to the guest OS and the VMM. Because it deals with both guest swapping and VMM-level swapping, it maintains the guest physical page frame number (GFN) to the physical sector number (PSN) mapping and the guest sector number (GSN) to PSN mapping information. GSN is the logical sector number of the VSWAP front-end device in the guest OS, and PSN is the physical sector number of the VSWAP back-end device in the VMM.

The VMM accesses the VSWAP back-end driver through GFN. When the VMM evicts a

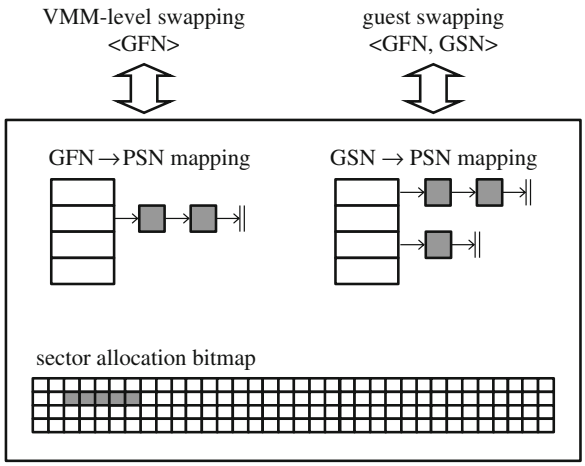


Fig. 3 Architecture of the VSWAP back-end driver

page, it asks to write a particular GFN. VSWAP first allocates new sectors, the PSN, for a page and then writes the contents of the GFN to the allocated sectors. After that, $< GFN, PSN >$ mapping is added to the mapping table. When the VMM swaps in a page, it asks to read a particular GFN. The VMM first finds the PSN corresponding to the GFN and reads the PSN into the GFN.

The VSWAP front-end driver in a guest OS accesses the VSWAP back-end driver through $< GFN, GSN >$. In case of a guest swap-out, the guest OS asks to write the particular GFN to the GSN. VSWAP first searches the GFN-to-PSN mapping table to check if the GFN has already been evicted by the VMM. If it has, VSWAP just adds the $< GSN, PSN >$ mapping to the mapping table and skips the actual write operations. Since our system omits the I/O operations for already evicted pages, it guarantees freedom from double paging. Otherwise, VSWAP allocates new sectors to the PSN, and writes the contents of the GFN to the PSN. After that, it adds the $< GSN, PSN >$ mapping to the mapping table. When the guest OS swaps in a page, it asks to read the GSN to the GFN. VMM translates the GSN to the PSN using the mapping table and reads the PSN to the GFN. The GSN-to-PSN and the GFN-to-PSN mapping tables are implemented by using a hash table. When double paging occurs, the same page is written twice and read once. Since our system avoids double paging, the page is always written only once. Therefore, the performance is significantly improved under heavy memory pressure. The probability of double paging is determined by how much the guest page replacement is synchronized with the VMM-level page replacement. If the two replacement policies are highly aligned, double paging occurs more likely. We will show how much they are synchronized in Section 3.3.

Typically, OS manages meta-data as well as swapped-out pages in a swap device. For the meta-data write, we do not need to maintain GSN-to-PSN mapping information. Linux uses the first sector to store the meta-data [26]. Therefore, our VSWAP back-end driver for Linux does not maintain the GSN-to-PSN mapping information for the first sector.

3 Evaluation

In this section, we show the evaluation results of our system. The purposes of our experiments are as follows: validating how well the LRU histogram reflects the memory demand (Section 3.2), evaluating its overhead (Section 3.4), analyzing the double paging (Section 3.3), and demonstrating its effectiveness in balancing the multi-VM memory allocation (Section 3.5).

3.1 Environmental Setup

Our system is implemented using the `KVM` [27] virtual machine monitor for Intel processor. It uses a modified version of `QEMU` [28] for device emulation and Intel EPT [23] for MMU virtualization. In our prototype implementation, we used Linux kernel 2.6.33 and QEMU-KVM-0.12.3. All of the experiments described in this paper are performed on a PC using a 2.67 GHz Intel Core i5 quad-core processor with 4 GB of physical memory. The guest OS is Ubuntu 10.04 with Linux kernel 2.6.35.

To evaluate the various aspects of our system, we run two synthetic benchmarks and three realistic workloads. We implemented two synthetic benchmarks: `mono` and `random` designed by Zhao et al. [5]. In a given memory range $[low, high]$, `mono` sequentially scans though pages for a fixed number of iterations. During the first half of an

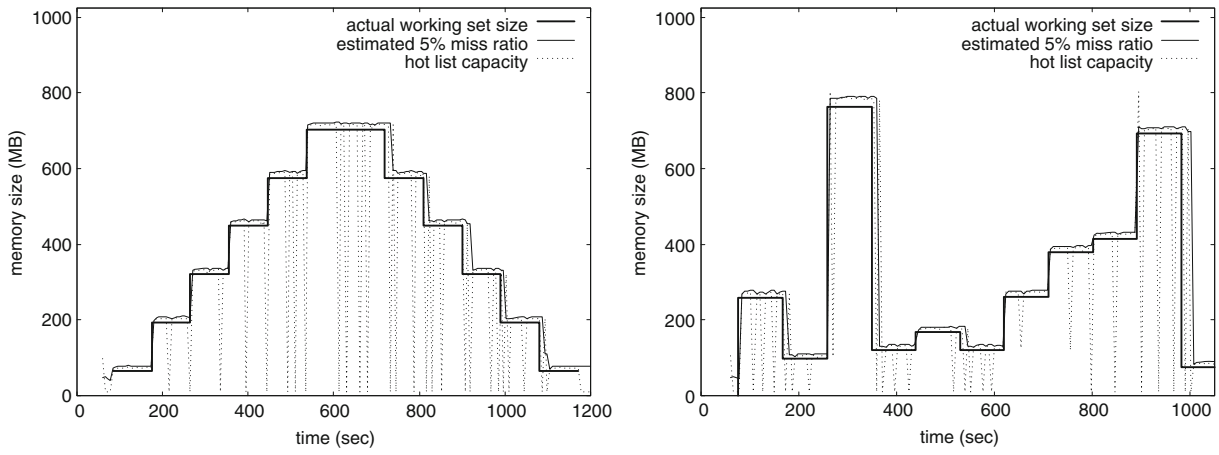
interval, it monotonically increases the number of scanned pages from *low* to *high*. For the second half of the interval, it decreases from *high* to *low*. `random` is similar to `mono` except that it scans a random number of pages for a fixed number of iterations. On the other hand, `random` scans a random number of pages for a fixed number of iterations. For three realistic workloads, we run one CPU intensive workload and two memory intensive workloads. SPEC CPU 2000 [29] is a CPU-intensive benchmark suite. SPECjbb2005 [30] evaluates the JVM performance by emulating a three-tier client/server system. JVM tends to allocate more memory to java heap in order to reduce garbage collection overhead. By default, the initial java heap size is 256 MB, and the maximum is 512 MB. A web server workload is also memory intensive, since OS tries to keep the accessed web pages in the page cache to reduce disk IO. We installed `Apache` [31] web server in each VM with approximately 512 MB web pages. To generate the workload, we used `http_load` [32] to make requests for randomly selected files for the web server.

We set T, the total memory size designated for the VMM, to 1,024 MB. M_i, the minimum memory size for each VM, is configured as 128 MB. Therefore, the amount of memory reserved for dynamic balancing is $(1{,}024 - 128 \cdot N)$ MB, where N is the number of VMs. By default, we set w_i, relative importance of each VM as 1. VSWAP size is configured as 512 MB for each VM.

3.2 Miss Ratio Estimation

To validate how well the LRU histogram reflects the memory demand of a VM, we first run `mono` and `random` under two different conditions. First, we run the benchmarks at a range of [64, 768] MB. No swapping occurs. We plot the memory size of 5% page miss while the benchmark is running. As Fig. 4 shows, our system closely follows the memory usage change. Second, we run the benchmarks at a range of [64, 1280] MB to show the behavior under guest swapping. In Fig. 5, the bars on the bottom represent the amount of guest level swapping in MB. The amount of guest swapping is about 1.1 GB for each. Since the guest swapping operations are incorporated with the LRU histogram, it is possible to estimate the page miss ratio using the LRU histogram.

Figures 4 and 5 also illustrate how the capacity of the hot list changes in MB. The proposed adaptive process for the hot list capacity can closely track the change of working set size. When the working set size of a benchmark increases, new pages are accessed and they are added to the head of the hot list. It makes the number of monitored faults from warm and cold lists temporarily decrease. Thus, the adaptive process shrinks the



(a) mono (b) random

Fig. 4 Miss ratio estimation without guest swapping

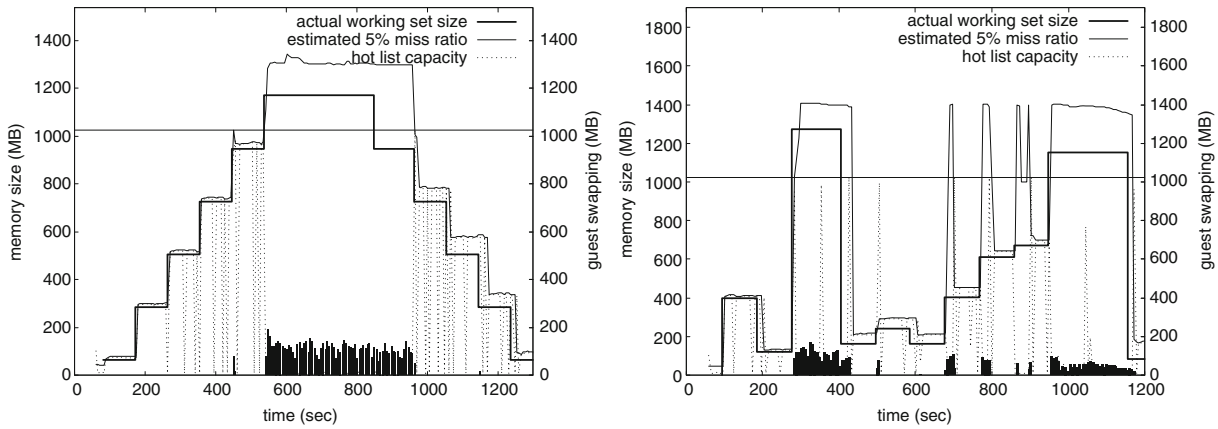


Fig. 5 Miss ratio estimation with guest swapping

capacity of the hot list to monitor more page faults. When the benchmark repeatedly accesses the changed working set, our system monitors sufficient page faults due to the small capacity. Then, our system increases the capacity of the hot list to control monitoring overhead. Similarly, when the benchmark shrink the working set, the estimation using (1) is not accurate and overestimates the capacity, since the LRU histogram does not reflect newly changed working set. Thus, our system decreases the capacity of the hot list by using (3). When the number of monitored faults becomes too many, the capacity is increased again. When the LRU histogram reflects the memory access pattern of the current working set through this process, the capacity of the hot list maintains in stable. In our experiment, the capacity of the hot list is decreased to 32 MB at minimum. Such adaptive process makes our system closely track the memory usage while keeping the overhead low.

3.3 Analysis on Double Paging

As we described in Section 2.4, double paging occurs more likely when the victim selection in the guest and the VMM are highly synchronized. Since our system uses the LRU replacement policy for VMM-level swapping, double paging occurs more likely when a page evicted by guest OS is located at the end of the VMM page list. In Fig. 6, we analyze distribution of victim selec-

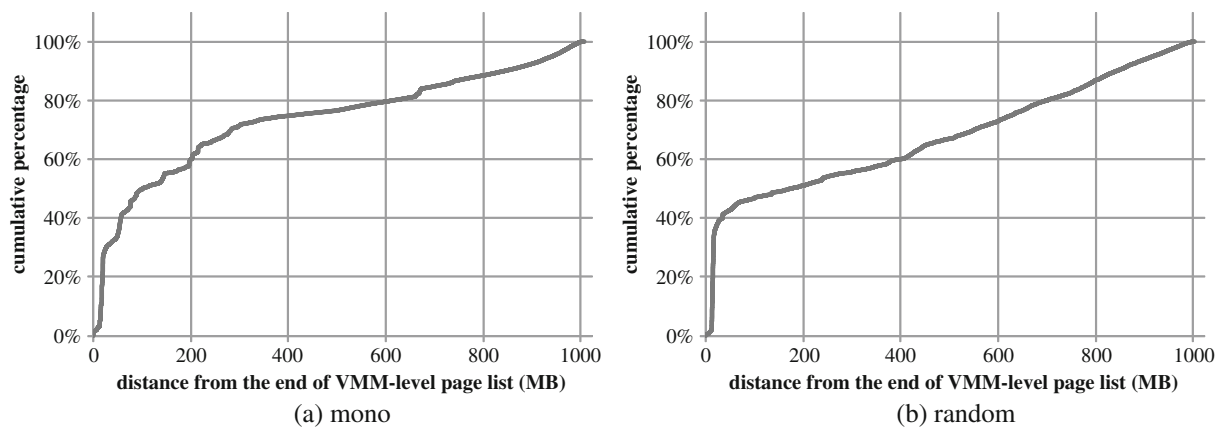


Fig. 6 Cumulative distribution of victim selection in Linux

tion in Linux guest for `mono` and `random`. The X-axis denotes the distance of a guest evicted page from the end of the VMM-level page list. The Y-axis is a cumulative percentage of the guest eviction by distance. We can see that the distribution is highly skewed. In case of `mono`, 40% of the guest victim selection is located in the last 5.4% (55.7 MB) of the VMM-level page list. Similarly, in `random`, the last 2.9% (29.6 MB) of the VMM-level page list covers 40% of the guest victim selection. This implies that the victim selection of Linux guest is highly synchronized with the VMM-level victim selection. Therefore, a small amount of VMM-level swapping can cause double paging, and VSWAP can significantly improve performance by avoiding double paging and reducing the disk I/O.

3.4 Performance Overhead

For our system to be practical, its performance overhead should be minimal. To measure the performance overhead, we run benchmarks on one VM. Figure 7 shows the monitoring overhead and how effective the proposed optimization techniques are. Performance with VMMB is tested using three different configurations. Without any optimization, the monitoring overhead is fairly large. Using the weighted red-black tree helps significantly, especially for workloads with large working set size; the normalized performance improves from 24.57 to 50.47% for SPEC2000, from 7.91 to 32.01% for SPECjbb2005, and from 28.2 to 70.79% for `http_load`. Adaptive resizing of the hot list further reduces the overhead. The normalized performances are very close to the non-monitored ones; the performances are 99.33% for SPEC2000, 99.56% for SPECjbb2005, and 99.53% for `http_load`.

The monitoring overhead is mainly composed of the LRU histogram updates and the other overheads including mode switching. Except for the LRU histogram update, the mode switching between the VM and the VMM is dominant, since this includes saving/loading status and TLB flushing. In Fig. 8, we classify the two overhead sources for each benchmarks. It shows that the histogram updating causes about 40% of the total overhead, regardless of the workload.

For SPEC2000 workloads, the performance overhead of the Zhao et al. [5] method is also small, 3% on average. However, the overhead of memory intensive workload with large working set such as `mcf` is not negligible, by up to 24%. On the contrary, for the same workloads, the average overhead of our system is even smaller, 0.67%, and more importantly, the largest overhead does not exceed 1.65%. It shows that our optimization techniques effectively control the overhead for the various workloads.

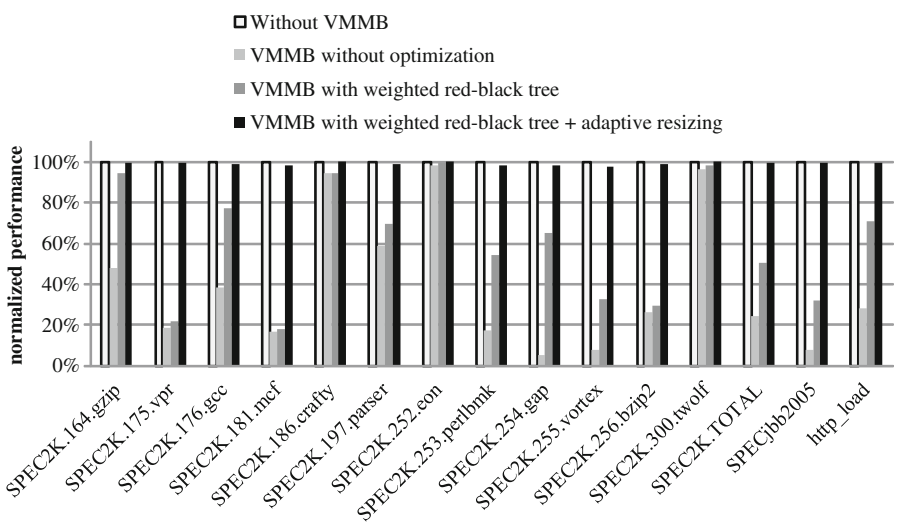


Fig. 7 Monitoring overhead

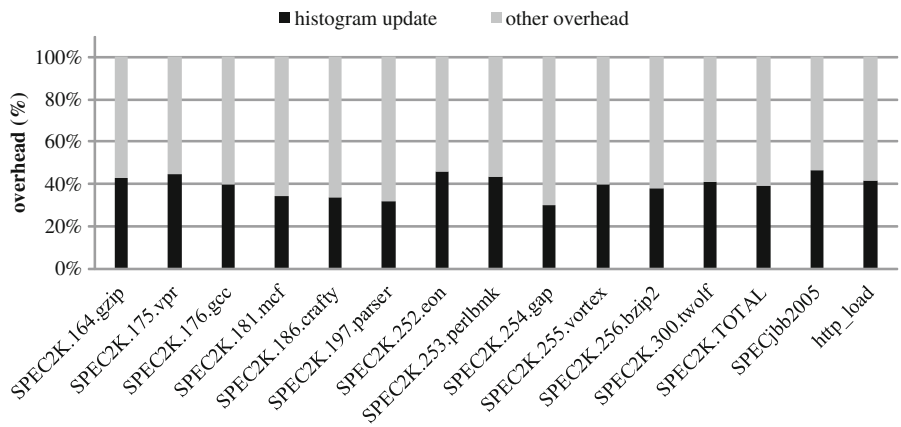


Fig. 8 Overhead breakdown

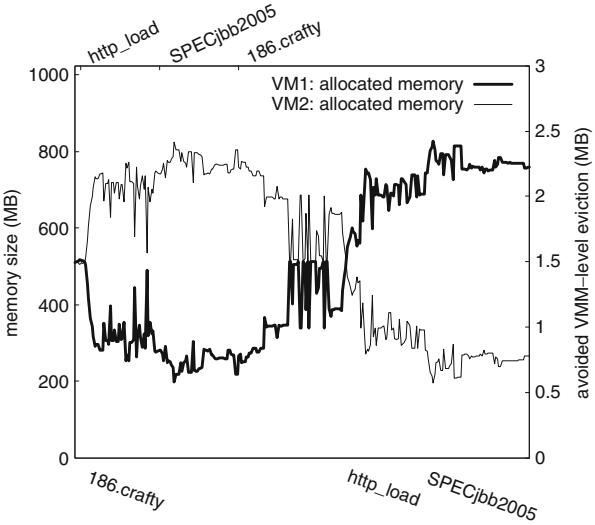
3.5 Effectiveness of Memory Balancing

In this section, we examine that our system allocates how much memory to each VM responding to changes in the memory demand and how this impacts performance.

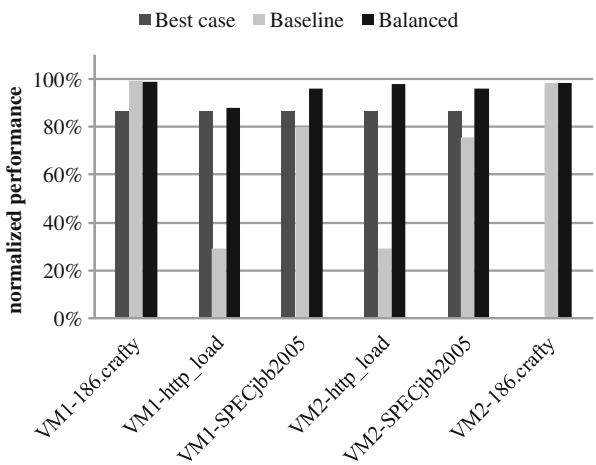
3.5.1 CPU Intensive + Memory Intensive Workloads

We start our evaluation with a simple scenario where the memory resource contention is rare. Two VMs run a CPU intensive workload and memory intensive workloads in a different order. For the CPU intensive but low memory demand workload, we run `186.crafty` for 20 iterations. For the memory intensive workload, we run `http_load` and SPECjbb2005 for 5 min respectively. VM1 runs the CPU intensive workload followed by the memory intensive workloads. VM2 runs the memory intensive workloads first, followed by the CPU intensive workload.

As Fig. 9a shows, our system initially gives more memory to VM2 that initially runs the memory intensive workloads and then gradually moves memory to VM1 over time. Figure 9b shows the performance comparison of the dynamic memory balancing. *Best case* is the performance when we



(a) memory allocation size



(b) performance

Fig. 9 CPU intensive + memory intensive workloads

run two VMs with 1,024 MB. This represents the theoretical maximum performance for comparison. *Baseline* is the performance using static partitioning that allocates 512 MB for each VM. Our system allocates less memory to `186.crafty`, however, its performance degradation is negligible; 1.0% overhead for VM1 and 1.4% overhead for VM2. On the contrary, the additional memory from the dynamic memory balancing is beneficial for the memory intensive workloads. For `http_load`, its performance improved from 27.0 to 97.9%. Similarly, the performance of SPECjbb2005 also improved from 75.8 to 96.0%. In this case, we can achieve a performance improvement for memory intensive workloads that matches closely to the best case performance with negligible performance overhead in CPU intensive workloads.

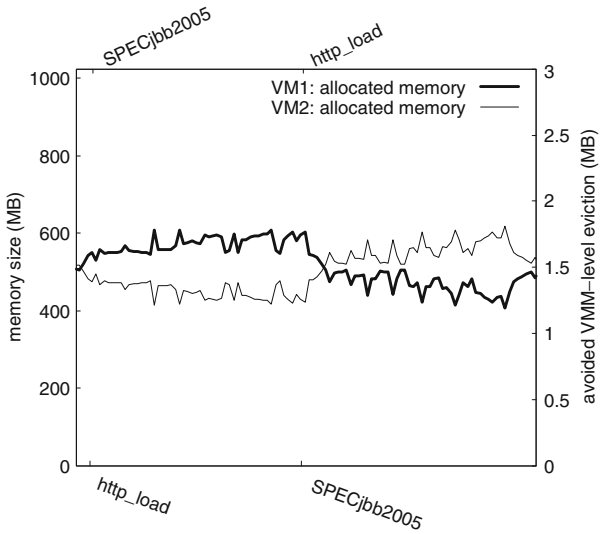
3.5.2 Memory Intensive + Memory Intensive Workloads

A more challenging case is found when running two memory intensive workloads simultaneously. VM1 first runs `http_load` for 5 min and then runs SPECjbb2005 for 5 min. VM2 runs them in reverse order. The QoS aware lookahead algorithm assesses the LRU histogram information and determines the memory allocation size that minimizes the global page miss ratio.

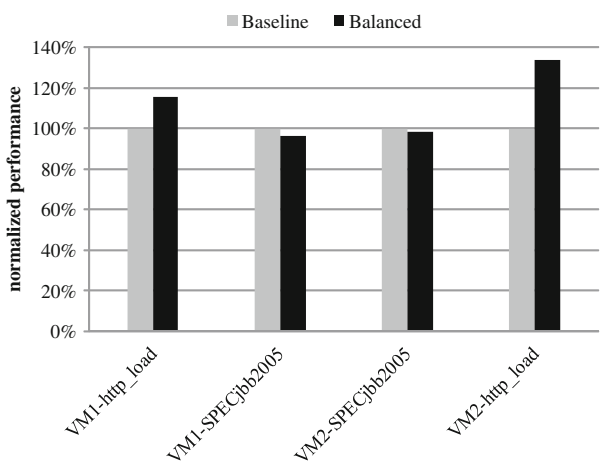
As shown in Fig. 10a, it gives more memory to `http_load` whose performance is more sensitive to the memory allocation size. Figure 10b shows the performance of the dynamic memory balancing. *Baseline* is the performance with static partitioning that allocates 512 MB for each VM. By virtue of the LRU histogram guided memory allocation minimizing the page miss ratio globally, the performance of `http_load` is improved by 15.3% for VM1 and 33.9% for VM2 whereas the performance degradation of SPECjbb2005 is in control; 3.8% for VM1 and 1.6% for VM2.

3.5.3 Mixed Workloads using Multiple VMs

To simulate a more realistic environment where multiple VMs are running various applications, we run three VMs simultaneously using different applications. VM1 runs `http_load` for 10 min, and VM2 runs `http_load` for 5 min and then runs `186.crafty` for 8 iterations. VM3 first runs `186.crafty` for 8 iterations and then runs SPECjbb2005. *Baseline* is the performance with static partitioning that allocates 341 MB for each VM.



(a) memory allocation size



(b) performance

Fig. 10 Memory intensive + memory intensive workloads

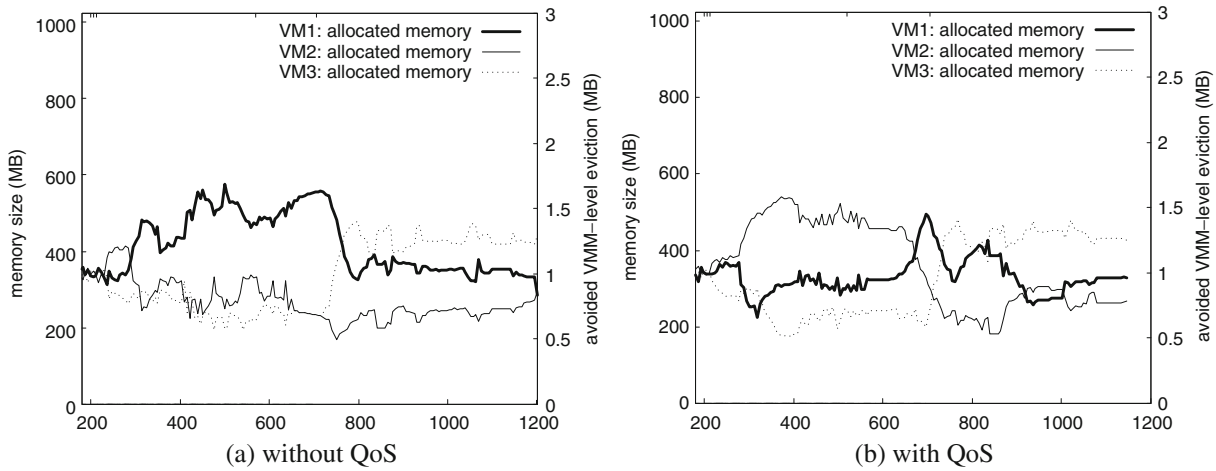


Fig. 11 Memory allocation of the mixed workloads

Figure 11a shows the memory allocation of the three VMs when we set the weights of all three VMs 1. In this case, the memory allocation size is determined by considering only the global page miss ratio. Our system allocates the least memory to `186.crafty` that is the least memory demanding workload and gives more memory to `http_load` and SPECjbb2005. Compared to the baseline, it improves the performance of `http_load` and SPECjbb2005 to 66.7 and 89.6% respectively, whereas the performance overhead for the rest is maintained between 1.5 and 4.2% as seen in Fig. 12.

To verify how effectively our proposed QoS aware lookahead algorithm works, we set the weight of VM2 to 10 while maintaining the others at 1. This improves the performance of VM2 from 98.4 to 148.3% for `http_load` and from 96.3 to 98.5% for `186.crafty`. Although we set the weight of VM2 ten times greater than the others, the performances of `http_load` in VM1 and SPECjbb2005 in VM2 are still better than the baseline performance; 108.3 and 163.4% respectively.

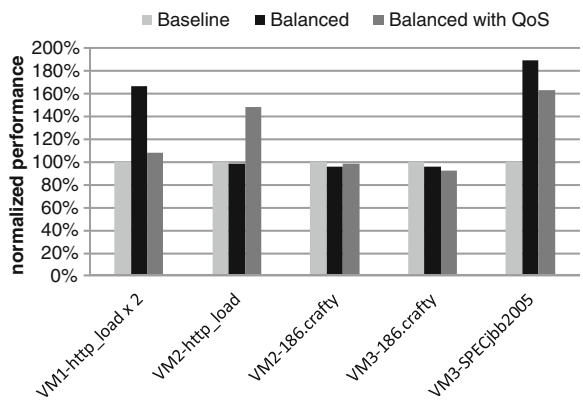


Fig. 12 Performance of the mixed workloads

4 Related Work

Our system achieves high server utilization by dynamically balancing memory among VM. In this section, we describe some related work on memory balancing and VM migration in a virtualized environment to achieve high resource utilization in a data center.

4.1 Dynamic Memory Balancing

In an effort to dynamically allocate memory in a virtual machine environment, previous studies have sought to estimate the memory demand for each VM and then allocate the memory.

Waldspurger [8] presents several mechanisms used in VMWare ESX server to dynamically balance the memory needed by VMs. First, he presents a statistical sampling method used to estimate the memory demand without any VM involvement. At each sampling period, a small

number of randomly selected pages are monitored to see whether they are accessed or not. The fraction of the accessed pages over the selected pages is considered to be the memory demand of a VM. However this method cannot estimate working set size larger than the current memory allocation. If a VM begins to trash, it simply reports its working set size as 100% of the allocated memory. In this case, the VMM can give some added memory in order to keep its measured working set size below 100%. However this is possible only if there is available physical memory. Second, he proposed a ballooning technique used to reclaim memory from a VM as a result of dynamic memory balancing. To reclaim memory, the balloon driver in a guest OS allocates some guest physical pages as victims and then reports them back to the VMM for future re-assignment to other VMs. The victim selection of ballooning technique is effective, because the determination of which pages are least valuable is known only by the guest OS. However there are several limitations. First, the memory reclamation can happen only after the victim VM is scheduled. Such scheduler-induced delays can deteriorate the effectiveness of memory balancing [33]. Second, it is not possible to reclaim memory when the balloon driver fails to allocate pinned memory under heavy memory pressure.

To cope with such drawbacks of ballooning technique, some studies [6, 8] use VMM-level swapping. In VMM-level swapping, the VMM performs page replacement on the guest physical memory. However there could be a pathological performance degradation known as the double paging anomaly [21, 22]. When the guest tries to swap out a page that is already swapped out by the VMM, the page needed to be swapped in by the VMM. Collaborative memory management (CMM) [9] addresses this problem by using a page hinting technique. In CMM, the VM shares its page usage information with the VMM, that is, what pages are being used and what pages may be evicted with little penalty. Milos et al. [34] also address the issue in page sharing context. Their system manages a repayment FIFO, a list of pages for a guest to give up without prior notification. However, page hinting and repayment FIFO are not applicable to a commodity OS, since they require modification of the guest kernel.

Ghost buffer technique [10, 11, 35, 36] can be used to predict the page miss ratio of a VM larger than its current allocation. Jones et al. [10] proposed techniques used to monitor the buffer cache in a virtual machine environment. They monitor the buffer cache eviction and promotion by intercepting all I/O operations at the VMM level. The memory demand is estimated from an LRU histogram of the buffer cache. However, it is impossible to predict a miss ratio smaller than its current allocation, since the memory accesses that hit the VM memory do not incur a buffer cache eviction. Furthermore, this method is not compatible with the hardware MMU virtualization support that eliminates traps to the VMM when manipulating guest page tables, because it relies on the page fault and the page table update information to trace the buffer cache eviction. To predict a miss ratio below the current memory allocation, Lu and Shen [11] presented a hypervisor exclusive cache scheme. In their approach, the VMM manages a part of the VM memory and uses it as VMM-level exclusive cache. Memory balancing among the VMs is achieved by allocating a different size VMM-level cache for each VM. They estimate memory demand from an LRU histogram that is constructed from the hypervisor cache access information. By extending the memory managed by the VMM, it can determine the growth and the shrinkage needed by the working set size. However, the coverage of the prediction and the amount of balancing are limited by the size of the VMM-level cache. In addition, it requires modification of guest kernel in order to monitor the buffer cache eviction.

Finally, Zhao et al. [5] proposed the LRU histogram based approach for dynamic memory balancing. However it suffers from performance overhead for workloads with large working set size, since the cost incurred to calculate an LRU histogram linearly increases according to the number of pages. In addition, since the memory demand information larger than the current allocation is not incorporated into the LRU histogram, the memory allocation of their methods is suboptimal.

4.2 Virtual Machine Migration

VM migration has become very popular in a data center for free of residual dependency and dynamic load balancing [37]. Since stop-and-copy migration technique [38] imposes significant downtime for VMs, the recent live migration techniques aim to minimize VM downtime. Pre-copy based live migration techniques are the most popular: implementations include `Xen` [12], `KVM` [27] and VMWare's vMotion [13]. In order to reduce the downtime further, post-copy based live migration technique [14] has been proposed.

There are many migration based approaches to decide when what VMs are migrated to where according to various placement objectives. Sandpiper [16] automates VM migration in a data center by detecting the sustained hot-spots and migrating to the idle machines. Entropy [15] uses constraint programming to find mappings of VMs to physical machines in order to minimize the number of migrations. Stage and Setzer [17] propose a network topology aware migration scheduling to avoid network and CPU overhead of migration. Andreolini et al. [18] propose an algorithm to identify the sustained hot-spot, by considering the load profile of hosts and the load trend behavior of the guest. VM migration based approaches aim to alleviate hot-spot and increase the utilization of resource in a data center wide. However they are focused on long term sustained hot-spot, because migration overhead is not negligible.

5 Conclusion and Future Work

We present VMMB, a novel mechanism used to dynamically balance the memory used among VMs based on the memory demand and the QoS requirements of each VM. Since our system is designed not to modify guest kernel in any manner, it is more widely applicable than other methods. In exchange for a small monitoring overhead (below 1%), VMMB can dramatically improve performance of workload that suffers from memory. Because VMMB decides the memory allocation size to globally minimize the page miss ratio and provide QoS support for individual VMs, our experiments show performance improvement in over-committed environments.

As future work, we plan to extend our technique to be integrated with VM migration. We expect that such integration can handle transient and sustained memory overload efficiently, thereby increasing the degree of consolidation while reducing total migration cost in a data center.

Acknowledgements This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (2011-0020520).

References

1. Simons, J., Buell, J.: Virtualizing high performance computing. SIGOPS Oper. Syst. Rev. **44**(4), 136–145 (2010)
2. Lange, J., Pedretti, K., Dinda, P., Bae, C., Bridges, P., Soltero, P., Merritt, A.: Minimal-overhead virtualization of a large scale supercomputer. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 169–180 (2011)
3. Iosup, A., Ostermann, S., Yigitbasi, M.N., Prodan, R., Fahringer, T., Epema, D.H.J.: Performance analysis of cloud computing services for many-tasks scientific computing. IEEE T Parall Distr **22**(6), 931–945 (2010)
4. Magenheimer, D.: Memory Overcommit Without the Commitment. Extended Abstract at the Xen Summit Boston 2008 (2008)
5. Zhao, W., Wang, Z., Luo, Y.: Dynamic memory balancing for virtual machines. SIGOPS Oper. Syst. Rev. **43**(3), 37–47 (2009)
6. Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A.C., Varghese, G., Voelker, G.M., Vahdat, A.: Difference engine: harnessing memory redundancy in virtual machines. Commun. ACM **53**(10), 85–93 (2008)
7. Barroso, L.A., Holzle, U.: The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan and Claypool Publishers (2009)
8. Waldspurger, C.A.: Memory resource management in VMware ESX server. SIGOPS Oper. Syst. Rev. **36**(SI), 181–194 (2002)
9. Scwidefsky, M., Franke, H., Mansell, R., Raj, H., Osisek, D., Choi, J.: Collaborative memory management in hosted linux environments. In: Proceedings of the Linux Symposium, vol. 2. Ottawa, Canada (2006)
10. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Geiger: monitoring the buffer cache in a virtual

machine environment. SIGOPS Oper. Syst. Rev. **40**(5), 14–24 (2006)
11. Lu, P., Shen, K.: Virtual machine memory access tracing with hypervisor exclusive cache. In: Proceeding of the 2007 USENIX Annual Technical Conference, pp. 1–15. Berkeley, CA (2007)
12. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines, In: Proceedings of the 2nd Symposium on Networked Systems Design & Implementation, pp. 273–286 (2005)
13. Nelson, M., Lim, B.-H., Hutchins, G.: Fast transparent migration for virtual machines. In: Proceedings of USENIX Annual Technical Conference (2005)
14. Hines, M.R., Gopalan, K.: Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In: Proceedings of the International Conference on Virtual Execution Environments, pp. 51–60 (2009)
15. Hermenier, F., Lorca, X., Menaud, J.-M., Muller, G., Lawall, J.: Entropy: a ConsolidationManager for Clusters. In: Proceedings of the International Conference on Virtual Execution Environments, pp. 41–50 (2009)
16. Wood, T., Shenoy, P., Venkataramani, A.: Black-box and gray-box strategies for virtual machine migration, In: Proceedings of the 4th Symposium on Networked Systems Design & Implementation, pp. 229–242 (2007)
17. Stage, A., Setzer, T.: Network-aware migration control and scheduling of differentiated virtual machine workloads. In: Proceedings of ICSE Workshop on Software Engineering Challenges of Cloud Computing, pp. 9–14 (2009)
18. Andreolini, M., Casolari, S., Colajanni, M., Messori, M.: Dynamic Load Management of Virtual Machines in Cloud Architectures. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering **34**(6), 201–214 (2010)
19. Williams, D., Weatherspoon, H., Jamjoom, H., Liu, Y.: Overdriver: handling memory overload in an oversubscribed cloud. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 205–216 (2011)
20. Mattson, R.L., Gecsei, J., Slutz, D., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM Syst. J. **9**(2), 456–789 (1970)
21. Goldberg, R.P., Hassinger, R.: The double paging anomaly. In: Proceedings of the National Computer Conference and Exposition, pp. 195–199 (1974)
22. Sherman, S.W., Brice, R.S.: Performance of a database manager in a virtual memory system. ACM Trans. Database Syst. **1**(4), 317–343 (1976)
23. Intel 64 and IA-32 Architectures Software Developers Manual vol. 3B: System Programming Guide, Part 2. Copyright 1997–2009 Intel Corporation.
24. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn, pp. 308–329. The MIT Press (2009)
25. Qureshi, M.K., Patt, Y.N.: Utility-based Cache Partitioning: A Low-overhead, High-performance, Runtime Mechanism to Partition Shared Caches. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 423–432 (2006)
26. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly (2005)
27. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: kvm: the Linux virtual machine monitor. In: Proceedings of the 2007 Ottawa Linux Symposium, pp. 225–230 (2007)
28. QEMU: http://wiki.qemu.org/ (2011). Accessed 15 August 2011
29. SPEC CPU2000: http://www.spec.org/cpu2000/ (2011). Accessed 15 August 2011
30. SPECjbb2005: http://www.spec.org/jbb2005/ (2011). Accessed 15 August 2011
31. The Apache Software Foundation: http://www.apache.org/ (2011). Accessed 15 August 2011
32. http_load: http://www.acme.com/software/http_load (2011). Accessed 15 August 2011
33. Hwang, W., Roh, Y., Park, Y., KPark, W., Park, K.H.: HyperDealer: reference-pattern-aware instant memory balancing for consolidated virtual machines. In: Proceeding of the 2010 IEEE 3rd International Conference on Cloud Computing, pp. 426–434 (2010)
34. Milos, G., Myrray, D.G., Hand, S., Fetterman, M.A.: Satori: enlightened page sharing. In: Proceedings of the 2009 Conference on USENIX Annual Technical Conference (2009)
35. Wong, T.M., Wilkes, J.: My cache or yours? Making storage more exclusive. In: Proceedings of the USENIX Annual Technical Conference, pp. 161–175. Monterey, CA (2002)
36. Chen, Z., Zhou, Y., Li, K.: Eviction based placement for storage caches. In: Proceedings of the USENIX Annual Technical Conference, pp. 269–282. San Antonio, TX (2003)
37. Milojicic, D.S., Douglis, F., Paindaveine, Y., Zhou, S.: Process Migration, ACM Comput. Surv. **32**(3), 241–299 (2010)
38. Kozuch, M., Satyanarayanan, M.: Internet suspend/resume. In: Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications, pp. 40–46 (2002)