Dynamic Memory Balancing for Virtualization

ZHIGANG WANG, XIAOLIN WANG, FANG HOU, and YINGWEI LUO, Peking University
ZHENLIN WANG, Michigan Technological University

2

Allocating memory dynamically for virtual machines (VMs) according to their demands provides significant benefits as well as great challenges. Efficient memory resource management requires knowledge of the memory demands of applications or systems at runtime. A widely proposed approach is to construct a miss ratio curve (MRC) for a VM, which not only summarizes the current working set size (WSS) of the VM but also models the relationship between its performance and the target memory allocation size. Unfortunately, the cost of monitoring and maintaining the MRC structures is nontrivial. This article first introduces a low-cost WSS tracking system with effective optimizations on data structures, as well as an efficient mechanism to decrease the frequency of monitoring. We also propose a Memory Balancer (MEB), which dynamically reallocates guest memory based on the predicted WSS. Our experimental results show that our prediction schemes yield a high accuracy of 95.2% and low overhead of 2%. Furthermore, the overall system throughput can be significantly improved with MEB, which brings a speedup up to 7.4 for two to four VMs and 4.54 for an overcommitted system with 16 VMs.

Categories and Subject Descriptors: D.4.2 [**Storage Management**]: Main Memory

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Virtualization, data center, memory, performance, working set size

---

The article contains material previously published in VEE'09 [Zhao et al. 2009] and USENIX ATC'11 [Zhao et al. 2011]. Compared to our conference articles, this article makes the following new contributions:

—We present a comprehensive and in-depth analysis and comparison of overhead-reduction techniques to track virtual machine working set size.
—We replace the heuristic, recursive memory balancing algorithm in VEE'09 with an efficient dynamic programming algorithm for memory balancing so it can work effectively for a large number of VMs.
—We implement the prototype system based on the latest version of Linux and Xen so our system can interact with the recent updates in Linux and Xen.
—We introduce scale-out data center applications into experimental evaluation and algorithm design and show that how applications with large working set size in modern data centers can affect our system design.
—We consider memory overcommit in our experimental design, in which we configure VMs' memory so the total memory demand far exceeds the available physical machine memory. We show that our memory balancing system can effectively adjust memory allocation to the VMs in order to deliver acceptable performance.

The research is supported in part by the 863 Program of China under Grant No. 2012AA010905, 2015AA015305; the National Science Foundation of China (Grants No. 61232008,61272158, No. 61328201, No. 61472008, and No. 61170055); and the Research Fund for the Doctoral Program of Higher Education of China under Grant No. 20110001110101. Zhenlin Wang is also supported by National Science Foundation CSR1422342.
Authors' addresses: Z. G. Wang, X. L. Wang, F. Hou, and Y. W. Luo, Computer Science Department, Peking University; emails: {pkuwzg, wxl, houfangcq, lyw}@pku.edu.cn; Z. L. Wang, Computer Science Department, Michigan Technological University; email: zlwang@mtu.edu. Corresponding author: X. L. Wang, Computer Science Department, Peking University, Beijing, China, 100871.

DOI: http://dx.doi.org/10.1145/2851501

ACM Reference Format:
Zhigang Wang, Xiaolin Wang, Fang Hou, Yingwei Luo, and Zhenlin Wang. 2016. Dynamic memory balancing for virtualization. ACM Trans. Archit. Code Optim. 13, 1, Article 2 (March 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/2851501

1. INTRODUCTION

Virtualization has become a common abstraction layer in modern data centers. Virtualization essentially enables multiple operating systems to run on one physical computer by multiplexing hardware resources and thus improves hardware resource utilization while maintaining reasonable quality of service. However, such a goal cannot be achieved without efficient resource management.

In a typical virtualized system, resources like processors and network interfaces can be assigned to a virtual machine when needed and given up when there is no demand. So these resources can be scheduled flexibly based on priority. However, the guest memory allocation is mostly static, as each virtual machine is assigned a fixed amount of machine memory in the beginning. Although Xen and VMware provide a ballooning driver to dynamically adjust guest memory allocation, it is a great challenge to tell when to reallocate and how much memory a virtual machine needs or is willing to give up to maintain its performance [Barham et al. 2003].

Modeling the relationship between memory allocation and performance is indispensable for optimizing memory resource management. Many studies have used Least Recently Used (LRU) stack distance-based profiling [Mattson et al. 1970; Zhou et al. 2004; Yang et al. 2006], with which they can construct a miss ratio curve (MRC) to describe the relationship between miss ratios and allocated memory sizes. These studies either require tracking virtual addresses, or need interaction between processes and the OS, and thus cannot be directly applied to virtual machines. In a virtualized environment, execution of a memory access instruction is usually handled by the hardware directly, bypassing the hypervisor or virtual machine monitor (VMM) unless it results in a page fault. Inspired by previous process-level memory predictor, we track memory accesses by revoking user access permission and trapping them into the hypervisor as minor page faults [Zhou et al. 2004]. As we are predicting memory demand of a VM, we build an LRU histogram based on host physical addresses (or machine addresses). As a result, the LRU-based predictor cannot provide the relationship between page miss and memory size beyond the size of machine memory allocation. When the memory of a VM is not enough, the miss ratio curve is unable to predict how much extra memory is needed. We monitor the swap space usage and use it as a guide to increase memory allocation for a virtual machine.

The key design choice of MRC construction is the tradeoff between accuracy and overhead. Because the complexity and data size of modern applications increase dramatically, the overhead of MRC tracking may overshadow its potential benefits. For example, for the benchmarks (SPEC CPU2006, DaCapo 2006, CloudSuite 2.0) used in the evaluation [Henning 2006; Blackburn et al. 2006; Ferdman et al. 2012], using a simple linked-list-based implementation, the overall execution time is increased by 758% on average. Some previous research optimizes MRC monitoring in terms of data structures. This article systematically integrates and evaluates the existing techniques to reduce overhead. Meanwhile, we introduce new techniques to further cut the overhead to an acceptable level.

To reduce data structure and algorithm complexity, we use an Georgy Adelson-Velsky and Evgenii Landis' tree (AVL)-based LRU implementation in the hypervisor. To decease page access trapping frequency, we introduce dynamic hot-set sizing (DHS). Experiments show that the combination of both brings down the overhead to 32% on average, which is still too high. By taking advantage of the phasing behavior of programs, we further design a novel technique, intermittent memory tracking (IMT),

to lower the overhead without a significant loss of accuracy. This idea is based on the fact that execution of a program can be divided into phases, within each of which, the memory demands are relatively stable [Denning 1968]. Thus, when the monitored system or process is predicted to stay within a phase, the memory tracking can be temporarily disabled to avoid tracking cost. Later, when a phase change is predicted to occur, the memory tracking is resumed to track the WSS of the new phase. Experimental results show that, only during an average of 14.3% of program execution time, memory tracking is turned on and the mean relative error of prediction is merely 4.2%. The average overhead is minimized to 2%.

We design a *Memory Balancer* (*MEB*), which takes advantage of the predicted memory demand and periodically reallocates guest memory to achieve high performance. Experimental results show that it is able to automatically balance the memory load and significantly increase the performance of the VMs which suffer from insufficient memory allocation with a slight overhead to the initially overallocated VMs.

The contributions of this article can be summarized as follows:

—**LRU-Based Low-Cost WSS Prediction**. We can accurately predict memory demand for VMs and reduce the overhead to an acceptable level. We also present a comprehensive and in-depth analysis and comparison of overhead-reduction techniques.
—**Dynamic Memory Balancing Mechanism**. Based on accurate and low-cost WSS prediction at runtime, we construct an efficient memory balancing strategy to dynamically adjust the amount of each VM's memory allocation to improve performance.

The remainder of this article is organized as follows. Section 2 presents an overview of MEB system. Section 3 details a series of optimization schemes to track working set with low cost. Section 4 describes our memory balance algorithm. Section 5 evaluates WSS tracking and MEB. Section 6 discusses the related work and Section 7 concludes.

2. OVERVIEW OF SYSTEM ARCHITECTURE

MEB consists of two parts: an predictor and a balancer. The predictor builds up an LRU histogram for each virtual machine and monitors the swap space usage of each guest OS. The balancer periodically adjusts memory allocation based on the information provided by the predictor. Figure 1 illustrates the system architecture.

—Memory accesses from the VMs are filtered by Dynamic Hot Set and then intercepted by the hypervisor. The intercepted pages are used to update the LRU histogram for each VM.
—CPU Events, such as DTLB miss, are monitored to detect the phase of a workload. The intermittent memory tracking mechanism will turn off the tracking system temporarily to decrease the overhead and turn it on if the phase of the workload changes.
—Each VM may have a background process, *Mon*, which retrieves the swap usage from OS performance statistics and posts the swap usage statistics to the central data store.
—The predictor reads the histograms and collects information from the central store periodically. It computes the WSS for each VM.
—The balancer arbitrates memory contention, if it exists, and sets the target memory size for each VM via ballooning.

3. LOW-COST WORKING SET SIZE TRACKING

This section describes how we estimate memory demand of a VM with low overhead while still maintaining enough accuracy. We adopt the idea of page permission revoking

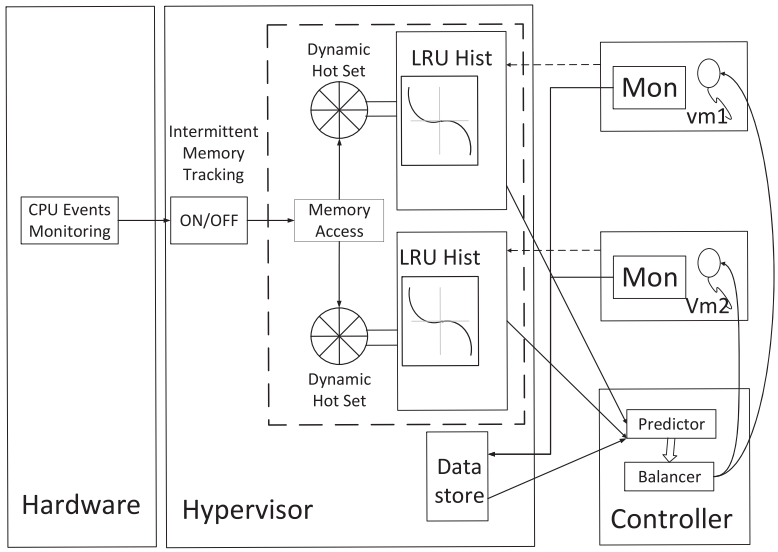

Fig. 1. System overview of MEB.

technique to intercept memory accesses and construct MRCs proposed by Zhou et al. [2004] and Yang et al. [2006]. However, the original idea, targeting the memory demand of a process or a Java virtual machine, is not designed for a VM. A straightforward implementation incurs high overhead, especially for programs with large memory demand or poor locality. This section starts with a simple implementation of VM-level WSS tracking and then solves the overhead problem by adopting three optimizing techniques.

3.1. LRU-Based Working Set Size Tracking

In order to estimate the working set size of a VM, it is necessary to infer its memory access behavior. Although most memory accesses from guest operating systems (OSs) are transparent to the hypervisor, page table operations such as creation, modification, and so on, are still visible to the hypervisor because it requires the hypervisor to map pseudophysical pages that guest OSs use to real machine pages. For a paravirtualized system, a guest OS has to make explicit calls, called *hypercalls*, to notify the hypervisor about these operations. We modify the hypervisor such that once a new page is installed, it will first perform the requested operations as usual and then revoke access permission from that page by setting the corresponding bit on the page table entry. As a result, an access to the page that corresponds to the modified page table entry will cause a page fault exception. In the context of that exception, both the virtual address and physical address of the access can be retrieved and tracked.

3.1.1. Linked-List Implementation. In order to maintain an LRU ordering of page level memory accesses, a traditional and natural design is to maintain a linked list of tags, each of which represents a page. To facilitate tag moving in the LRU list, the list is doubly linked. Once an access to one page is trapped, its corresponding tag is moved from its current location to the head (the LRU location) of the linked list. The positional order of each tag, or its LRU distance, is required to update the LRU histogram counters. The linked-list design enables $O(1)$ tag moving. However, the cost of locating a tag's position in the list requires linear search with a cost of $O(N)$ where N is the total number of tags. As a result, the overall time cost of LRU updating is in $O(N)$.

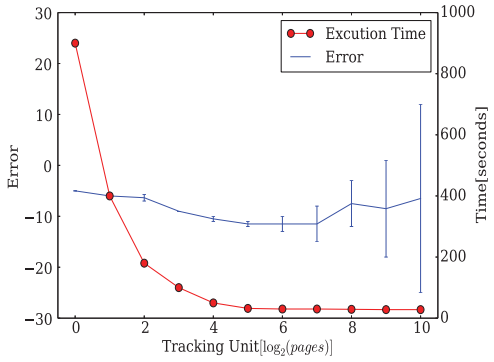

Fig. 2. Granularity and overhead on small WSS.

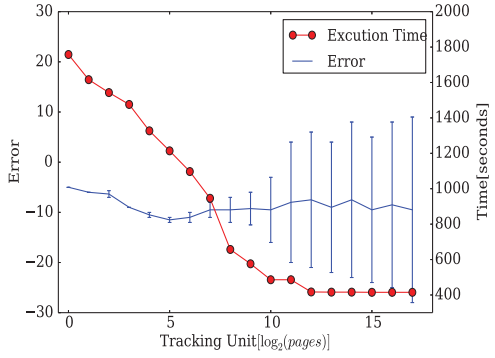

Fig. 3. Granularity and overhead on large WSS.

For a program with good locality, a target tag can usually be found near the beginning of the linked list. Unfortunately, for a program with poor locality and large memory footprint, the linear search cost can be prohibitive. We observe an average overhead of 758% in the evaluation.

3.1.2. Track Granularity. Maintaining the histogram at page level requires a large amount of space since each page needs a tag in the list and each tag requires a counter in the histogram. Bjornsson et al. [2013] devised a novel bucketing scheme for generating near-exact MRCs. Each bucket groups a set of items and can have different size. Applying a similar idea to reduce the space cost, we instead group every G consecutive pages as a unit and represent it by a tag. That is, given a page number p, its corresponding tag is p/G. As a result, it not only reduces the length of linked list but also shortens the average search time in finding LRU distances. We use a tracking granularity of G to construct the LRU histogram. The choice of tracking unit size is a tradeoff between estimation accuracy and monitoring overhead.

To find out an appropriate tracking unit size, we run two sets of experiments, targeting a VM with low and high memory demands, respectively. We first run a program which constantly visits 100 MB of memory space. Therefore, its WSS is known as 100 MB. We measure the execution time and estimation accuracy for various tracking granularity from 1 page to 1024 pages. During the program execution, estimations are reported periodically and the highest and lowest values are recorded. As illustrated by Figure 2, when G increases from 1 to 32, the overhead, which mainly comes from the histogram updating operations, drops dramatically. However, when G grows from 32 to 1024, there is no significant reduction in execution time while the estimation error begins to rise dramatically. We also conduct similar experiments on SPEC CPU2006, the average error is only 0.9% while tracking granularity is 32. Therefore, we use 32 pages as the LRU tracking granularity for a VM with relatively low memory demand.

For a large WSS workload, we intend to adopt a bigger granularity to further reduce overhead. As illustrated by Figure 3, we test the run time of Graph Analytics from CloudSuite with a 2GB workload. The overhead drops linearly while the granularity grows, and the curve becomes flat while $G > 2^9$. Therefore, we use 512 pages, which is also the size of a superpage [Intel 2015], as the tracking granularity for large WSS workloads.

In practice, G is set on the fly based on the VM memory demand. The initial value of G is set to 32, and it will be switched to 512 if the predicted WSS is bigger than 2GB.

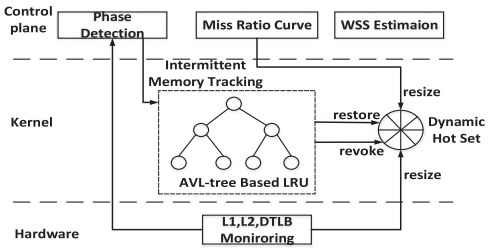

Fig. 4. Low-cost WSS tracking system.

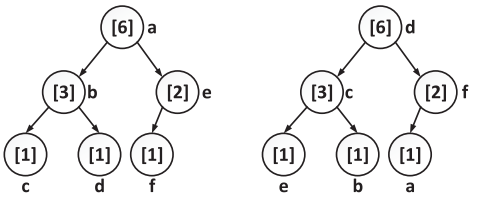

Fig. 5. An example of an AVL-based LRU list.

3.2. Overhead Analysis

Let I denote interception frequency, P the page fault time, and L the LRU structure updating time. The cost of the LRU-based memory tracking can be represented as $I*(P+L)$. As P is a constant value in a certain system, to lower the overhead, we can either decrease I or reduce L. We add three optimizations to decrease the overhead, and Figure 4 illustrates the system organization. First, we design an AVL-tree-based LRU list, which improves L asymptotically from $O(N)$ to $O(logN)$, where N is the length of the LRU list. Second, we use a dynamic hot set to decrease I. Third, to decrease I further, we introduce an intermittent memory tracking scheme, which can turn on/off memory tracking.

3.3. AVL-Tree-Based LRU (ABL)

To decrease the overhead from data structure, we use an $O(log(N))$-time AVL-tree-based organization for the tags. For any tag, its left child has a shorter LRU distance and its right child has a longer LRU distance. That is, an in-order traversal of the tree gives the same sequence as given by a linked-list traversal from the head. Figure 5 shows an example of a tree-based LRU list.

To facilitate locating the LRU distance of a tag, each tag has a field, *size*, which counts the number of nodes of the subtree rooted from itself. In Figure 5, the numbers in the square brackets are the values of the *size* fields of the nodes. For any tag x, its LRU distance (LD) is calculated recursively by:

$$LD(x)=\begin{cases}0 & x\ is\ nil\\ LD(ANC(x))+size(LC(x))+1 & x\ is\ not\ nil\end{cases},$$

in which $size(x)$ denotes the size of the subtree rooted as x, and $LC(x)$ is x's left child. For any tag x, $ANC(x)$ is the immediately previous node, based on in-order traversal, of the leftmost child of x. If the node does not exist, then $ANC(x)$ is nil. In algorithm implementation, $ANC(x)$ returns either *nil* or $x's$ nearest ancestor y, whose left child is neither x or $x's$ ancestor. For example, in Figure 5(a), $ANC(c)$ is *nil* and $ANC(f)$ is a. Before page e is visited, the LRU sequence is c, b, d, a, f, e. After that, e is moved to the leftmost position, and then the tree is rebalanced. In-order traversal gives e, c, b, d, a, f.

Since function *ANC* walks up along the tree towards the root, and the tree is balanced, the time cost of *LD* is $O(log(N))$. When an access to some physical page is intercepted, its corresponding tag's LRU distance is first computed and then the tag is removed and inserted as the leftmost leaf. During the insertion or removal, at most all the tag's ancestors' *size* fields need to be updated, which takes $O(logN)$ time, the same cost upper bound for tree balancing. As a result, the overall time cost is $O(log(N))$, while the space cost is still $O(N)$.

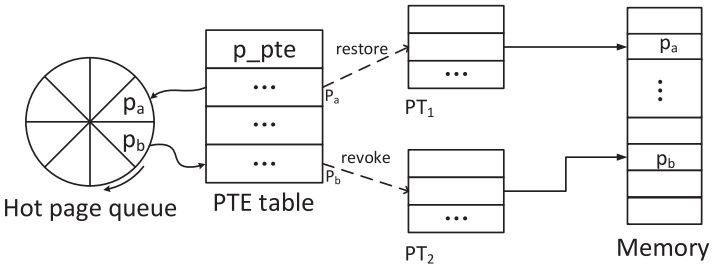

Fig. 6. Hot-set management.

3.4. Dynamic Hot Set (DHS)

The overhead of LRU monitoring is proportional to the number of intercepted memory accesses. It is prohibitive to intercept every memory page access. Since most programs show good locality, we logically divide all physical pages into two sets, a hot set and a cold set. Pages in the cold set are revoked access permission and thus will be trapped, while pages in the hot set have normal access permission. Initially, when a new page table is populated (e.g., when creating a new process), all pages are cold. Later, when an access to a cold page is trapped, after recording its address, its permission is restored and it is moved to the hot set. The concept of hot/cold pages is similar to the active/inactive page lists in Yang et al. [2006]. Our design decouples the page-level hot/cold control from the LRU list management.

The hot set is implemented as an FIFO queue with limited size, which stores physical frame numbers (PFN). Once the queue is full and a new page needs to enter the queue, the page referred to by the head entry is dequeued and made cold again by revoking its access permission in the corresponding page table entry (PTE). To facilitate the PFN to PTE looking up, we introduce a PTE table, which stores the PFN to PTE mapping. The mapping is updated whenever an access is trapped. Occasionally, the PTE table may contain a stale entry (e.g., the PTE is updated), which can be detected by comparing the dequeued PFN and the actual PFN contained in the PTE. Figure 6 illustrates the workflow of a hot set. Suppose that an access to physical page p_a is trapped. Then the permission in the corresponding PTE in page table PT_1 is restored and the address of PTE is added to the PTE table. Next, p_a is enqueued and p_b is dequeued. Use p_b to index the PTE table and locate the PTE of page p_b. The permission in PTE of page p_b is revoked and page p_b becomes *cold*.

A page access is intercepted only when it is not in the hot set or, in other words, in the cold set. The hot-set size (HSS) can thus significantly affect the number of memory access interceptions as well as estimation accuracy. Even with a hot set, a program with poor locality can exhibit large overhead due to a great number of interceptions. To control the overhead, the idea is to dynamically adjust the hot-set size when the locality changes. The locality is evaluated by examining the miss ratio curve. Let $MRC^{-1}(x)$ be the inverse function of the miss ratio curve function, where x is miss ratio and $MRC^{-1}(x)$ is the corresponding memory size. We use $\alpha = \frac{MRC^{-1}(50\%)}{WSS/2}$ to quantify the locality. The smaller the value of α is, the better the locality is. When α is greater than an upper threshold, 1/3, we incrementally increase HSS by a factor of 1.2, until HSS reaches a preset upper bound. While α is smaller than a threshold, 1/5, we decrease HSS by a 10%, until HSS reaches a preset lower bound. An acceptable range for *HSS* change is 10–20%, as a bigger change may cause precision lost and a smaller change cannot deliver a fast enough response to system overhead. We select a bigger factor for increment, as we try to reach a stable state as soon as possible.

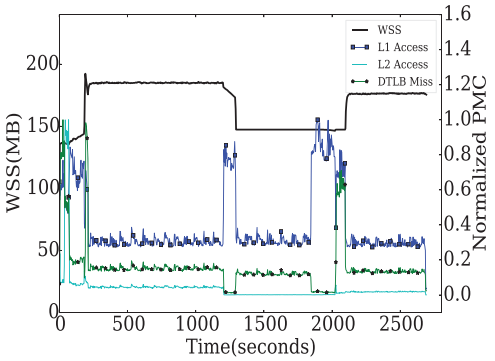

Fig. 7. Example of WSS and performance events.

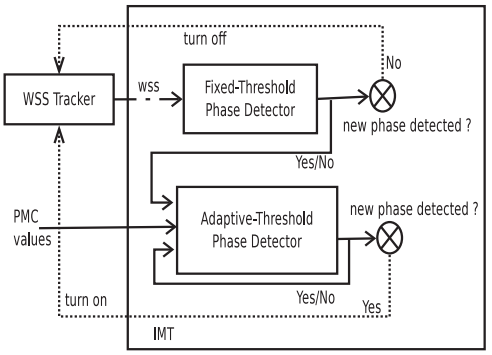

Fig. 8. Adaptive-threshold IMT.

In data centers, the WSS of a scale-out application can take several GBs of memory or more. If we use a small initial value for HSS, then it can take much time to reach the target size. We take a more aggressive strategy for workloads with large WSSs. *HSS* is initially set to a small value as we do not have enough information about the workload. After several sampling intervals, we get the predicted WSS, which is used for the adjustment of *HSS*. Through experiments, we find that $WSS/12$ is big enough for an initial *HSS* size while maintaining prediction accuracy, and thus we set $HSS = MAX(WSS/12,\ previous\ HSS * 1.2)$ when the locality is poor, as discussed above. The feedback of *WSS* to *HSS* helps us to find a proper *HSS* much faster.

3.5. Intermittent Memory Tracking (IMT)

Most programs show typical phasing behavior in terms of memory demands: Within a phase, the WSS remains nearly constant. This inspired us to temporarily disable memory tracking when the monitored program enters a stable phase and reenable it when a new phase is encountered. Through this approach, the overhead can be substantially lowered. When memory tracking is on, we can examine the predicted WSSs to decide if the VM has reached a stable state. We utilize hardware performance counters to detect phases when memory tracking is turned off.

3.5.1. Selection of Events. There exist numerous memory-related hardware events in modern processors. We observe that L1/L2 accesses/misses and DTLB misses correlate well with WSS. Figure 7 illustrates L1 accesses, L2 misses, and DTLB misses along with WSS for 473.astar in SPEC CPU2006, where the sampling interval is 5s. It shows that the three events are all closely correlated with WSS. The lines for L2 and DTLB misses show the same trend and the spikes of L1 accesses overlap with sudden changes of WSS as well. As a change of WSS implies changes in memory access patterns at the page level, it suggests that data DTLB misses would be the best event to detect phases for memory demands. We thus use DTLB misses in our experiments.

3.5.2. Phase Detection. Previous studies rely on sophisticated signal processing techniques such as Fourier transformation or wavelet analysis [Shen et al. 2004; Sherwood et al. 2001] to detect cache-level phases. Though these techniques are able to effectively filter out noises and identify phase changes in off-line analysis, their costs make them inappropriate for on-line phase detection.

We propose a simple yet effective algorithm to detect behavior changes for both memory demands and performance counters. First, a moving average filter is applied to smooth out the signals. Let v_i denote the sampled value (memory demand or the

number of occurrences of some hardware event) during i_{th} time interval. We pick $f(i) = (v_i + v_{i-1} + \cdots + v_{i-k+1})/k$ as the filtering function to smooth the sampled values, in which k is the filtering parameter, an empirical value. If the moving average filter has not been filled up with k data, then it means there is not enough information to make any decisions. So memory tracking is always turned on during this period. When enough data have been sampled, let v_j be the current sampled value and let $f_{mean} = mean(f(x)|x \in (j-k, j])$, $err_r = f(j)/f_{mean}$, and $err_a = |f(j) - f_{mean}|$. err_r is the relative difference between the current sampled value (smoothed) and the average of history data in the window and err_a is the absolute difference between the two. If $err_r \in [1-T, 1+T]$, where T a small threshold of choice discussed later, we assume the input signal is in a stable phase. Otherwise, we assume that a new phase is encountered. In this case, all the data in the moving average filter is cleared so the data that belong to the previous phase will not be used.

3.5.3. Fixed-Threshold Phase Detection. T is the key parameter in phase detection. We first propose a scheme that uses a fixed T. One phase detector, based on the past WSSs, checks if the memory demands reach a stable state so the WSS tracking can be turned off. The other detector uses values from performance monitoring counter (PMC) to check if a new phase is seen so the WSS tracking should be woken up.

For stability test of memory demands, T can be set to a small value (0.05 in our evaluation) to avoid accuracy loss. In addition, err_a can also be used to guide memory tracking. For example, if memory tracking is used for memory resource balancing at an MB granularity, then, as long as $err_a < 1MB$, WSS can still be assumed in a stable state even when $err_r > T$.

For phase detection of hardware performance events, an overstrict threshold may cause memory tracking to be enabled unnecessarily and thus undermine performance. On the other hand, if the threshold were too large, then WSS changes would not be detected, which leads to inaccurate tracking results. Our experiments show that, for a given hardware event, the appropriate T may vary between programs or even vary between phases for the same program. One solution to find the appropriate value of T is by means of experiments. By trying different T on an extensive set of programs, an empirical T can be found such that the average overhead can be lowered with a tolerable accuracy loss.

3.5.4. Adaptive-Threshold Phase Detection. To improve upon fixed-threshold phase detection, we propose a self-adaptive scheme which adjusts T dynamically to achieve better performance. The key is to feed the current stability of WSS back to the hardware performance phase detector to construct a closed-loop control system, as illustrated by Figure 8. Initially, the PMC-based phase detector can use the same threshold as used in fixed-threshold phase detection. When memory tracking is on, its current stability is computed and compared with the PMC-based phase detector's decision. If both of the results are consistent, then nothing will be changed. If the current memory demands are stable, while the PMC-based detector makes the opposite decision ($err_r > T$), then it implies that the current threshold is too tight. As a result, its T is relaxed to its current err_r. Next time, with increased T, the PMC-based detector will most likely find that the system enters a stable state and thus turn off memory tracking. On the contrary, if the current memory demands are unstable, while the PMC-based phase detector assumes stable PMC values, that is, $err_r < T$, then it implies an overrelaxed threshold. Thus, its current T is lowered to err_r. In short, when the WSS is stable and memory tracking is on, it is only because the PMC-based phase detector is oversensitive. As a result, T will be increased until PMC values are considered to be stable, too. Then, memory tracking will be turned off.

As long as WSS tracking is enabled, T is calibrated to make decisions that are consistent with the stability of current WSS. However, when memory tracking is off, this self-calibration is paused as well, which might miss the chance to tighten the threshold as it should had memory tracking been on. To solve this problem, we introduce a *periodic sampling* design. When memory tracking has been turned off for PS consecutive sampling intervals, it is woken up to check if T should be adjusted. If no adjustment is needed, then it will be turned off again until it reaches the next sampling point or meets a new phase. In an ideal case, memory tracking will be deactivated except for periodic sampling. The value of PS is adaptive. Initially, it is set to some predefined value PS_{init}. Afterwards, if no adjustment is made in the previous sampling point, then it can be increased by some amount, PS_{step}, until it reaches a maximum value PS_{max}. Whenever an adjustment is made, PS is restored to its initial value, PS_{init}. In the ideal case, the ratio of the time that memory tracking is on to the whole execution time, called *up ratio*, is near $1/PS_{max}$. In our evaluation, the initial value, PS_{init}, increment, PS_{step}, and the maximum value, PS_{max}, are set to 10, 5, and 20 sampling intervals, respectively.

3.6. Memory Growth

For process level WSS estimation, using tracked virtual addresses to construct LRU MRC is sufficient. Accesses in virtual address space represent the actual memory behaviors of a program. Thus, it can be used to guide both memory shrink and growth. Unfortunately, to estimate the memory demand of a whole guest OS, using virtual address to construct MRC causes an ambiguity problem because each process of the guest OS uses the same virtual address space. Our design tracks physical addresses. It can only estimate memory demand of no larger than its current memory allocation size, which is the maximum footprint of physical memory accesses. In other words, when a guest OS suffers from memory insufficiency, it is unable to tell how much more memory is needed and thus incapable of guiding memory growth.

A natural solution to this problem is to track swap space accesses and then construct a swap space-based LRU histogram to estimate the working set size on the swap space. The working set size on the swap space is the extra memory that the guest OS needs. Geiger [Jones et al. 2006] adopts this approach to infer the memory pressure and to estimate the amount of extra memory needed. A shortcoming of this approach is that it is insensitive to the decrease of memory demand. When a program's WSS shrinks, it stops making disk I/Os, which leaves the swap space-based MRC obsolete and thus overestimates the memory demand.

Alternatively, we take the swap space usage as the indicator for the amount of memory growth. On most modern OSs, swap space usage is a common performance statistic and available for user-level access. A simple user level process that runs on each guest OS can be designed to periodically report the value to the hypervisor. Though, compared with Geiger, the MRC-based estimation, this footprint-based estimation may overestimate the memory demand, it is actually close to the former one because the accesses to disk are filtered by in-memory references and therefore show weak locality. In other words, given a tolerable miss ratio, the memory demand indicated by the MRC is close to its footprint on the swap space. Moreover, compared with constructing an MRC, the overhead of this OS statistics-based estimation is almost negligible and it reflects the swap usage decrease more rapidly than the LRU-based solution.

4. DYNAMIC MEMORY BALANCING

With the LRU MRCs of all VMs on a physical machine, we can dynamically adjust each VM's memory demand. In this section, we present our memory resource balancing system. When there is no sufficient physical memory to meet all VMs' memory demand,

our arbitration algorithm is able to quickly find an allocation plan that aims for the overall performance.

In our implementation, the local balancer is written in Python and runs in domain 0 (dom0), a privileged guest domain. It communicates with the hypervisor via hypercalls to acquire LRU histograms and resize memory allocation. We set the balancing frequency as every 5s, an empirical value, which allows the WSS estimator to collect enough information but it is not too long to miss optimizing opportunities.

To resize the memory allocation of VMs, the balancer periodically collects each VM's WSS and dynamically adjust memory allocation by calling the ballooning module. As the ballooning driver needs time to inflate/deflate, and radical adjustment may cause performance fluctuation, it is also necessary to limit the extent of memory reclaiming. Reclaiming a significant amount of memory may disturb the target VM because the inactive pages may not be ready to be reclaimed instantly. Therefore, we introduce a pair of adjustment thresholds for each round of memory balancing, with a maximum shrink rate of 10% and a maximum growth rate of 30% as memory shortage is more urgent. So the low bound memory, L_i, of a VM at balance time is $\max(LowLimit_i, 0.9 * m_i)$ and the high bound, H_i, is $\min(HighLimit_i, 1.3 * m_i)$ where $LowLimit_i$ and $HighLimit_i$ are the minimum and the maximum memory of VM_i, and m_i is the current memory allocation for VM_i. To find a balance scheme in reasonable time, we choose an increment/decrement unit size S ($S = 8 \times G$), where G is the tracking unit size discussed in Section 3.1.2.

We implement a brute-force algorithm which enumerates all possible allocations to find one with the minimum page misses. This algorithm works well when the number of VMs is small. Note that a typical data center machine can host many VMs. When the number of VMs increases, the overhead of the algorithm can become prohibitive. Assume that M is the maximum-possible size of memory for one VM, S is the increment/decrement unit size, and the time complexity for N VMs is $O((\frac{M}{S})^N)$, which is exponential.

We apply dynamic programming to design a more efficient algorithm. Assume that $Miss(i, j)$ is the minimum page misses that first i VMs cause with the total memory size j. The dynamic programming algorithm needs to find $Miss(N, P)$, where N is the total number of VMs and P is the total memory size. $Miss(i, j)$ can be calculated following the recurrence equation below,

$$Miss(i, j) = min_{\{k|L_i \le k \le H_i \text{ with step } S)\}}(Miss(i-1, j-k) + MRC_i(k) * NR_i),$$

where $MRC_i(k)$ is the miss ratio of VM_i when allocating k memory and NR_i is the number of memory accesses in a recent epoch of VM_i. The number of page misses for VM_i is $MRC_i(m) * NR_i$. Algorithm 1 shows the pseudocode of our balancing algorithm.

Let $E_i = max(LowLimit_i, PredictedWSS_i)$ be the expected memory size of VM_i. When $P \ge \sum E_i$, it means that all VMs are satisfied and we use $P - \sum E_i$ as a bonus, which could be used flexibly. In our implementation, we aggressively allocate the bonus to each VM proportionally to E_i. The balancing algorithm finds A_i, the final target memory allocation size for VM_i.

If $P < \sum E_i$, then at least one VM cannot be satisfied. Here we assume all VMs have the same priority and the goal is to minimize system wide page misses. The balancer applies dynamic programming to calculate $Miss(i, j)$. The outermost loop searches all VMs, the inner j loop enumerates all possible memory sizes for current VMs, and the innermost loop goes through all possible memory allocation for the current VM. The time complexity is $O(N^2 * (\frac{M}{S})^2)$ compared with $O((\frac{M}{S})^N)$ for the brute-force solution. In our experiments that balance 4 VMs, the brute-force algorithm yields about 5%

overhead for each sampling period, but the overhead for the dynamic programming algorithm is negligible. Under the 16-VM setting, the brute-force algorithm could not finish in an acceptable amount of time while the dynamic programming algorithm continues to improve overall performance.

ALGORITHM 1: Algorithm of Memory Balancing Optimization

```
Require: P // host machine memory size
Require: {LowLimit_i} //low limit of VM memory
Require: {HighLimit_i} //high limit of VM memory
Require: {m_i} // actual memory of VM
Require: {PredictedWSS_i} //predicted working set size
Require: {MRC_i} //miss rate curve
Require: {NR_i} //number of memory accesses
  function ARBITRATE(V, P) ;
      for i ∈ V do
        L_i ← Max(0.9 * m_i, LowLimit_i) ;
        H_i ← Min(1.3 * m_i, HighLimit_i) ;
        E_i ← Max(LowLimit_i, PredictedWSS_i) ;
      end
      E_s ← Sum({E}) ;
      if P ≥ E_s then
        bonus ← P − E_s;
        for i ∈ V do
            A_i ← E_i + bonus * E_i/E_s
        end
        {BestA} ← {A};
      end
      else
        Low ← 0;
        High ← 0;
        for i ∈ V do
            Low ← Low + L_i;
            High ← High + H_i;
            for j ← Low → min(P, High) with step S do
                Miss[i][j] ← ∞;
                for k ← L_i → H_i with step S do
                    if j ≥ k then
                        if Miss[i][j] > Miss[i − 1][j − k] + MRC_i(k) * NR_i then
                            Miss[i][j] ← Miss[i − 1][j − k] + MRC_i(k) * NR_i;
                            Target[i][j] ← k;
                        end
                    end
                end
            end
        end
        T ← P;
        for i ← N → 1 do
            A_i ← Target[i][T];
            T ← T − Target[i][T];
        end
        {BestA} ← {A};
      end
      return {BestA}
  end function
```

5. EXPERIMENTAL EVALUATION

In this section, we first evaluate the overhead and accuracy of the WSS tracking system. Then we show the performance of memory balancing based on WSS prediction.

5.1. Experimental Setup

Our experiments are performed on an Intel CORE I7 machine with 16GB of physical memory and four 2.80GHz cores with support of Hyper-threading (HT). All of our experiments are carried out on a hypervisor based on Xen 4.2.1 and the Linux kernel version 3.10.12, while both dom0 and the guest systems are built on CentOS 6.4. To exclude the impact of CPU resource contention when multiple VMs are running, each VM is assigned a dedicated CPU core. Each VM is assigned with 4GB of memory in the WSS tracking evaluation, while the allocated memory for memory balancing will be specified for each experiment. In our experiments, WSS and PMC are sampled every 5s.

We use SPEC CPU2006 [Henning 2006], DaCapo [Blackburn et al. 2006], and CloudSuite 2.0 [Ferdman et al. 2012] for our evaluation. SPEC CPU2006 is chosen to represent the sequential workload. It contains integer and floating-point programs and benchmarks CPUs and memory systems. DaCapo is a widely adopted benchmark suite for Java applications, consisting of a set of open-source, real-world applications with nontrivial memory loads. CloudSuite is a benchmark suite for emerging scale-out applications for data centers. It contains eight benchmarks, which are based on real-world software stacks and represent real-world setups. In our evaluation, we select three applications from CloudSuite, Data Analytics, Web Serving, and Graph Analytics, whose performance relies heavily on memory.

5.2. Evaluation of Low-Cost WSS Tracking System

5.2.1. Accuracy of WSS Tracking System. We first evaluate the prediction accuracy using a synthetic benchmark, *RandomArray*, which visits a given mount of memory in a selected time window, so we can check the accuracy of our predictor. We then report the accuracy of a realistic workload using Pin [Luk et al. 2005], an instrumentation and profiling tool.

We first set *RandomArray* to access [40, 350] MB of memory on a VM with 400MB of physical memory. In this setting, no page swapping occurs, so WSS can be derived from the physical memory LRU histogram directly. For comparison purpose, we also implement the sampling-based estimation as used in the VMware ESX server [Waldspurger 2002]. When memory usage increases, our predictor follows closely. Due to the nature of the LRU histogram, it responds slowly to the decrease of memory demand. The average error of the LRU-based and sampling-based estimations is 13.46% and 74.36%, respectively. The LRU-based prediction is a clear winner.

Figure 9(b) shows the results when the WSS of *RandomArray* varies from 40MB to 700MB while the VM still has only 400MB of physical memeory. Now the WSS can be larger than the 400MB memory allocation. In this case, swap usage is involved in calculating the WSS. The sampling scheme cannot predict WSS beyond the current host memory allocation, while the combination of LRU histogram and OS swap usage tracks the WSS well.

To get the exact WSS of a realistic workload, we use Pin to record every memory access and construct an accurate MRC accordingly. Pin is an offline profiling tool, which can incur huge overhead if used online. For example, the program 429.mcf from SPEC CPU2006 completes in less than 10min on a native machine but takes several hours under Pin. To measure accuracy, we compare the WSSs reported by Pin and the WSSs predicted by our dynamic tracking system. We keep track of instruction count to

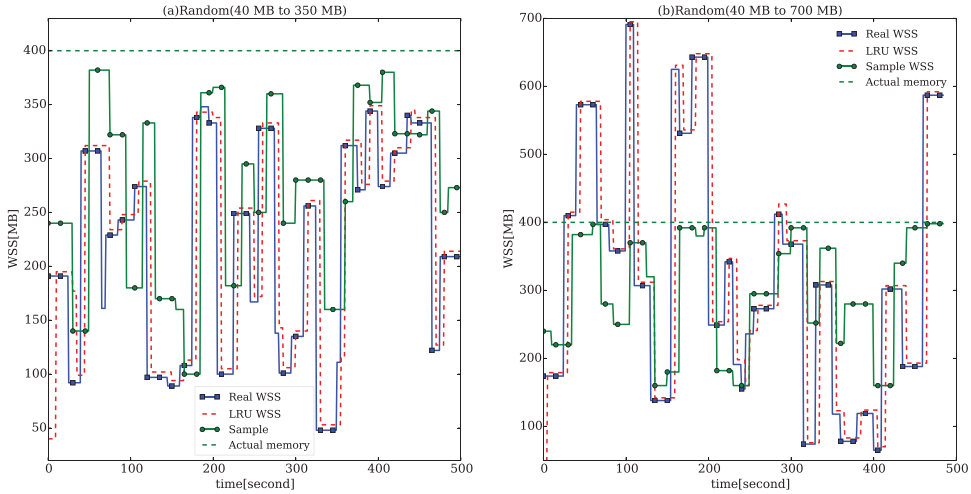

Fig. 9. WSS prediction on *RandomArray*.

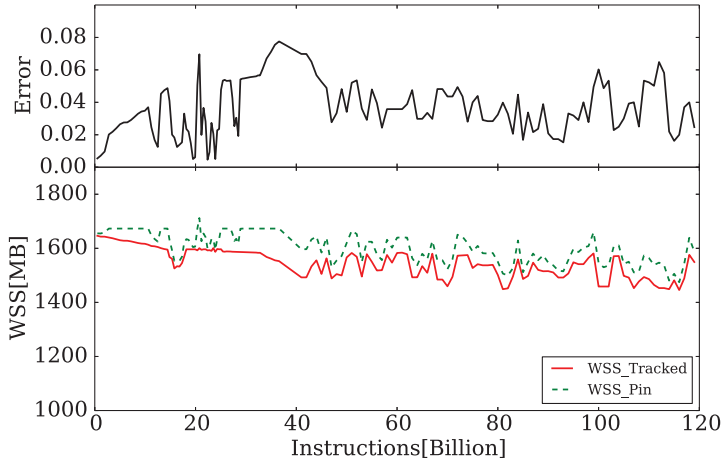

Fig. 10. Comparison between Pin and our Tracking System.

ensure the two runs are reporting the same periods of program execution. As Figure 10 shows, the trend of tracked WSS is close to the exact WSS reported through Pin, with an error range from 1% to 8%. The mean accuracy of SPEC CPU2006 is 95.2%.

5.2.2. Effectiveness of DHS and ABL. To measure the effects of various techniques on reducing overhead, we first measure the running time of SPEC CPU2006, DaCapo, and CloudSuite without memory tracking as the baseline performance. For SPEC CPU2006 and CloudSuite, each program is measured individually. While for DaCapo, each of them is run in alphabetic order and the total execution time is measured because some programs finish in several seconds, which is too short for IMT to start working. Then we measure the running time with memory tracking enabled, using the linked-list LRU implementation, DHS, ABL, and the combination of the latter two.

As shown in Table I, using DHS and ABL together, we can reduce the overhead for most programs significantly. Columns two to five of Table I list the normalized execution time against the baseline setting. For the benchmarks we selected, the linked-list

Table I. Normalized Execution Time of SPECCPU 2006 (L: Linked List; D: Dynamic Hot Set; A: AVL-Tree Based LRU; I(f): IMT Tracking with Fixed Threshold; I(a): IMT Tracking with Adaptive Threshold)

Program	L	D	A	D+A	I(f)	I(a)
400.perlbench	1.56	1.36	1.11	1.14	1.00	1.02
401.bzip2	1.11	1.12	1.05	1.04	1.01	1.01
403.gcc	1.98	1.29	1.48	1.11	1.02	1.03
410.bwaves	3.60	2.92	1.33	1.42	1.19	1.02
416.gamess	1.06	1.03	1.06	1.02	1.00	1.00
429.mcf	60.34	9.43	3.27	1.77	1.41	1.04
433.milc	13.34	8.87	3.45	2.79	2.46	1.05
434.zeusmp	2.74	1.45	1.23	1.22	1.06	1.06
435.gromacs	1.11	1.09	1.03	1.07	1.00	0.99
436.cactusADM	1.26	1.21	1.10	1.16	1.00	1.02
437.leslie3d	2.89	1.03	1.47	1.05	1.00	1.00
444.namd	1.09	1.11	1.05	1.08	1.00	1.00
445.gobmk	1.17	1.11	1.08	1.08	1.01	1.01
447.dealII	1.43	1.19	1.24	1.07	1.02	1.01
450.soplex	3.25	1.33	1.46	1.23	1.01	1.10
453.povray	1.07	1.08	1.06	1.07	1.00	1.00
454.calculix	1.05	1.02	1.09	1.06	1.00	1.00
456.hmmer	1.10	1.10	1.04	1.07	1.00	1.01
458.sjeng	9.93	1.16	1.90	1.12	1.01	1.01
459.GemsFDTD	7.06	4.34	3.51	3.00	0.99	0.99
462.libquantum	8.52	1.07	1.38	1.05	1.00	1.01
464.h264ref	1.12	1.07	1.08	1.05	1.01	1.00
465.tonto	1.08	1.07	1.09	1.09	1.00	1.01
470.lbm	4.70	2.50	1.83	1.81	1.01	1.00
471.omnetpp	42.36	1.17	4.74	1.07	1.00	1.04
473.astar	16.72	1.09	2.98	1.09	1.01	1.00
481.wrf	1.20	1.16	1.13	1.23	1.00	1.00
482.sphinx3	1.13	1.10	1.04	1.08	1.00	1.00
483.xalancbmk	8.04	1.40	2.33	1.12	1.02	1.00
Dacapo	1.45	1.14	1.22	1.16	1.04	1.01
Graph Analytics	56.73	15.87	5.33	2.35	1.56	1.04
Data Analytics	13.44	3.45	2.49	1.60	1.09	1.04
Mean	8.58	2.39	1.80	1.32	1.09	1.02
Overhead	7.58	1.39	0.80	0.32	0.09	0.02

design incurs a mean overhead of 758%. Using DHS and ABL separately, the mean overheads are lowered to 139% and 80%, respectively. Applying DHS and ABL together, the mean overhead is further reduced to 32%. When the WSS is small or the locality is good, the advantage of ABL and DHS over the regular linked-list implementation is not obvious. However, for benchmarks with large WSS, the overhead reduction by them is prominent. For example, in SPEC CPU2006, the top two programs with the largest WSSs are 459.GemsFDTD and 429.mcf, whose WSSs are 800MB and 1178MB, respectively. Using DHS and ABL together, the overhead against the linked-list setting is reduced by 56.65% and 97.06%, respectively. For 483.xalancbmk, although its WSS is merely 28MB, its poor locality leads to a 704% overhead under the linked-list design. Replacing the linked-list LRU with ABL and applying DHS, its overhead is cut to only 12%. However, even when both ABL and DHS are enabled, the mean overhead of 32% is still significant. IMT is able to further reduce the overhead as discussed next.

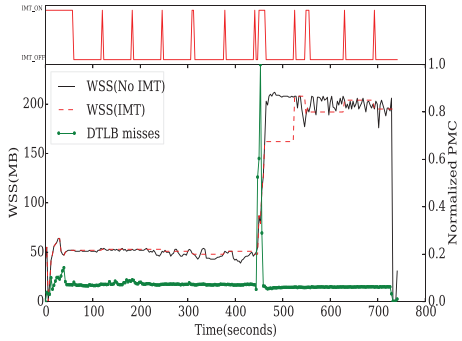

Fig. 11. Relation between DTLB miss and WSS in 450.soplex.

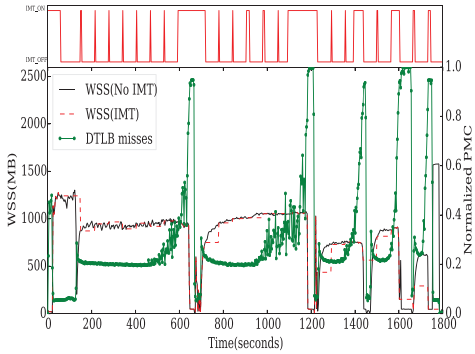

Fig. 12. Relation between DTLB miss and WSS in 429.mcf.

5.2.3. Evaluation of Intermittent Memory Tracking. The performance of intermittent memory tracking is evaluated by two metrics: (1) *up ratio*, which denotes the ratio of the execution time when memory tracking is on, and (2) *mean relative error*, which reports the accuracy loss due to temporary deactivation of memory tracking.

Since the actual memory demands may not be identical between runs, it is hard to precisely compare the accuracy loss caused by IMT with different fixed thresholds and adaptive threshold. Instead, we measure the two metrics by simulating the memory tracking using a single trace file. We first run each benchmark and sample the memory demands and hardware performance counters every 5s without intermittent memory tracking. Then we feed the trace to the intermittent memory tracking algorithm to simulate its operations. That is, given inputs $\{M_0, \ldots, M_i\}$ and $\{P_0, \ldots, P_i\}$, the intermittent memory tracking algorithm outputs m_i, in which M_i and P_i are the i-th memory demand and i-th PMC value sampled in the trace results, respectively, and m_i is the estimated memory demand. When the IMT algorithm indicates the activation of memory tracking, $m_i = M_i$, otherwise, $m_i = M_j$ where j is the last time when memory tracking is on. Given a trace with n samples, its mean relative error is computed as $MRE = (\sum_{i=1}^{n} \frac{|M_i - m_i|}{M_i})/n$, in which n is the number of samples.

To evaluate fixed-threshold IMT, we set T from 0.05 to 0.30, two extreme ends of the spectrum. Table II shows the details. Using fixed thresholds, when $T = 0.05$, memory tracking is off nearly two thirds of the time with an MRE of about 4.0%. When T is increased to 0.30, memory tracking is activated for only about one seventh of the time, while the MRE increases to 11.7%. With adaptive thresholds, its up ratio is nearly the same as that of $T = 0.30$, while its mean relative error is close to that of $T = 0.05$. Clearly, the adaptive-threshold algorithm outperforms the fixed-threshold algorithm. IMT reduces the average overhead to 9% with a fixed threshold of 0.2 and to 2% with adaptive thresholds as shown in Table I.

Figures 11 and 12 show the effects of two cases using adaptive-threshold IMT. The upper parts of each figure show the status of memory tracking: A high level means it is enabled and a low level means it is disabled. In the bottom parts, thick lines and thin lines plot the WSS and normalized data TLB misses from the traces (sampled without IMT), respectively. Dotted lines plot the WSS assuming IMT is enabled.

Figure 11 shows a simple case where both WSS and DTLB misses are stable during the whole execution except for a spike. Hence, most of the time, memory tracking is turned off except for periodic sampling. Figure 12 shows a typical case where there are multiple phases in terms of WSS and DTLB misses.

Table II. Up Ratios and MREs of Various Thresholds (T: Threshold; U: Up Ratio of IMT; M: MRE)

	T=0.05		T=0.10		T=0.20		T=0.30		Adaptive	
Program	U	M	U	M	U	M	U	M	U	M
400.perlbench	50.3	8.7	36.0	13.5	20.4	19.1	21.0	19.0	30.5	3.4
401.bzip2	89.7	0.3	83.8	0.8	66.6	1.7	55.7	4.4	24.0	3.9
403.gcc	98.0	0.2	87.1	0.9	69.9	1.8	53.6	2.1	24.2	4.0
410.bwaves	50.2	2.2	47.6	2.5	45.9	1.3	36.4	2.2	13.9	5.2
416.gamess	30.9	1.9	2.1	4.9	2.2	4.8	2.3	5.3	6.7	1.0
429.mcf	57.8	2.7	41.0	4.5	34.6	16.6	26.7	73.1	27.1	40.3
433.milc	23.8	6.0	6.9	11.7	5.4	15.1	1.1	20.0	9.0	8.3
434.zeusmp	93.9	0.4	60.1	1.0	3.4	1.9	3.3	2.0	12.6	1.9
435.gromacs	15.3	0.5	4.8	1.2	4.8	1.3	4.2	1.2	9.3	0.2
436.cactusADM	8.0	12.7	5.4	10.8	3.3	8.6	3.4	9.3	8.3	1.8
437.leslie3d	5.2	1.0	2.3	1.9	2.4	1.7	2.4	1.8	6.3	0.3
444.namd	10.9	0.2	5.5	0.3	4.4	0.3	4.5	0.3	9.7	0.1
445.gobmk	66.3	0.4	15.0	1.3	5.7	2.3	4.6	2.3	9.5	0.9
447.dealII	98.3	0.0	97.6	0.0	96.1	0.0	90.4	1.1	53.6	4.7
450.soplex	18.9	4.7	20.2	4.5	14.3	8.9	13.8	8.9	18.1	4.1
453.povray	18.2	7.0	18.5	7.0	17.7	7.7	16.0	6.8	17.3	1.3
454.calculix	14.7	1.7	2.2	6.8	1.1	7.2	1.1	7.1	5.4	0.6
456.hmmer	24.9	6.1	2.2	84.7	2.1	79.2	2.2	80.7	14.5	3.9
458.sjeng	9.5	3.0	6.3	3.6	2.4	6.5	2.1	6.4	10.0	2.9
459.GemsFDTD	3.4	0.4	1.2	0.6	1.1	0.6	1.2	0.6	5.4	0.7
462.libquantum	4.6	0.0	3.4	0.2	3.2	0.2	3.3	0.4	8.0	0.2
464.h264ref	65.6	0.2	61.2	0.7	6.7	2.4	7.1	2.3	7.6	0.6
465.tonto	93.9	0.1	31.7	1.9	15.9	12.2	16.3	11.9	35.3	4.1
470.lbm	5.5	2.8	5.7	2.7	4.2	2.8	4.3	2.8	10.3	0.7
471.omnetpp	2.2	45.2	2.3	49.2	2.3	47.4	2.2	46.1	7.8	4.3
473.astar	17.7	0.5	13.9	0.4	11.6	1.0	11.7	1.9	9.2	1.1
481.wrf	26.7	2.0	18.5	2.0	5.6	3.1	5.5	1.3	8.2	0.5
482.sphinx3	33.2	0.4	4.1	5.7	4.8	5.6	3.2	5.3	9.2	0.6
483.xalancbmk	18.5	7.9	14.3	12.1	9.5	11.1	9.7	12.3	15.0	3.4
DaCapo	52.6	2.1	39.7	6.5	31.5	12.0	31.9	10.9	9.0	12.1
Graph	39.3	4.0	25.1	5.8	17.2	6.7	11.7	9.3	11.0	7.1
Data	46.1	4.4	37.6	6.7	28.2	10.0	20.0	13.9	12.1	9.0
Mean	37.3	4.0	25.1	8.0	17.0	9.4	14.8	11.7	14.3	4.2

With adaptive-threshold IMT, the mean MRE of 429.mcf, DaCapo, and Data Analytics is still large. For 429.mcf, as Figure 12 shows, most of the time, the WSS estimation using IMT follows the one without using IMT. The high MRE is because its WSS changes dramatically up to 9 times at the borders of phase transitions for DaCapo, as it contains several subbenchmarks, which have different WSSs and phases. Borders between benchmarks are the main reason of high MRE. Data Analytics experiences high MRE also because of frequent change of phases. Though after a short delay, IMT detects the phase change and wakes up memory tracking, and those exceptionally high relative errors lead to a large MRE. More specifically, we observe that, during 70% of its execution time, the relative errors are below 4.2%, and during 82% of the time, the relative errors remain within 11%.

5.3. Evaluation of Memory Balancing

We evaluate memory balancing (MEB) on different types of workloads and VM settings. We report three sets of results for each configuration, *Baseline*, *Balancing*, and *Best*.

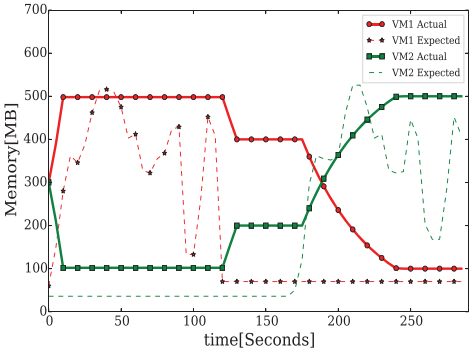

Fig. 13. Memory competing between VMs running DaCapo and 186.crafty.

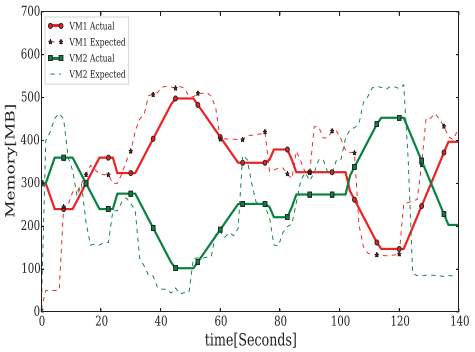

Fig. 14. Memory competing between DaCapo and DaCapo′.

Table III. MEB Between 186.Crafty and DaCapo

	Baseline		Balancing		Best	
	VM_1	VM_2	VM_1	VM_2	VM_1	VM_2
Major page faults	84226	81635	6908	9465	1688	1572
Time (DaCapo)	155.3	153.1	115.2	120.5	113.2	114.3
Time (186.crafty)	40.7	40.4	40.4	40.6	40.4	40.6

Baseline reports the case without MEB while each VM receives a fixed amount of physical memory. *Balancing* shows the results when MEB is turned on. *Best* presents the results when each VM is allocated enough memory.

5.3.1. CPU Intensive vs Memory Intensive. Our evaluation starts with a simple scenario where memory resource contention is rare. The workloads include the DaCapo benchmark suite and 186.crafty, a program with intensive CPU usage but low memory load. On VM_1, 186.crafty runs four iterations (we report the average time) followed by the DaCapo suite. Meanwhile, on VM_2, the DaCapo suite runs first, followed by four iterations of 186.crafty. Initial memory for each VM is 300 MB, and the upper limit is 500 MB while the lower limit is 100 MB. To show the performance that an ideal memory balancer can deliver, we measure the best-case performance on two VMs, each with 500 MB fixed memory, the peak memory allocation that a VM could own during balancing.

Figure 13 displays the actual allocated memory size and expected memory size on both VMs respectively. Note that the VM running 186.crafty gradually gives up its memory to the other VM. When both VMs are running 186.crafty, bonus is gradually allocated to the two VMs. Table III lists the number of major page faults and execution time for both VMs. With memory balancing, the number of major faults is reduced to 8.2% and 11.6%, respectively, while the performance of DaCapo increases by 34.8% and 27.1%, respectively.

5.3.2. Memory Intensive vs Memory Intensive. The challenging cases for memory balancing are the ones with frequent memory contention. We run the DaCapo benchmark suite on two VMs at the same time but the programs are executed in different orders: VM_1 runs programs in alphabetical order while VM_2 runs them in the reversed order (denoted as DaCapo′). Note that Eclipse and Xalan require about 500MB memory at peak times and Eclipse takes about half of the total execution time. When the execution of two occurrences of Eclipse overlaps, memory resource contention happens. The initial memory for each VM is 300MB and the upper limit is 500MB while the lower limit is

Table IV. MEB Between DaCapo and DaCapo'

	Baseline		Balancing		Best	
	PF	time	PF	time	PF	time
DaCapo	390092	397.5	48726	131.6	1324	113.7
DaCapo'	612371	455.2	64092	142.6	1527	114.2

Table V. MEB Between Graph and Data

	Baseline		Balancing		Best	
	PF	time	PF	time	PF	time
Graph	4032651	4837.0	109287	520.7	12457	510.0
Data	8985	181.4	15258	188.4	7176	179.8

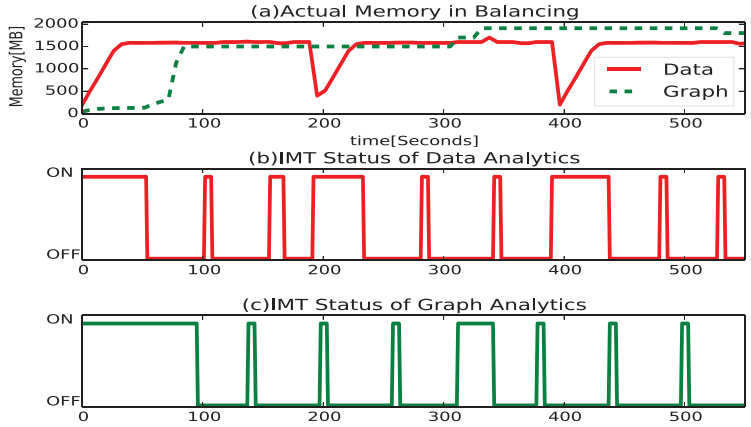

Fig. 15. MEB between Graph and Data.

100MB. As Figure 14 shows, sometimes the total expected memory exceeds the total available memory. The allocated memory by the balancing algorithm follows the trend of expected memory. Table IV shows that, without balancing, the number of major page faults of the two VMs is 390,092 and 612,371, respectively. After applying balancing, it is lowered to 48,726 and 64,092, respectively, which leads to a speedup of 3.02× and 3.19×, respectively.

5.3.3. Workload with Large WSS. To evaluate the overhead of WSS tracking and performance impact of memory balancing for workload with large WSS, we select two benchmarks from CloudSuite, Graph Analytics, and Data Analytics. We set the number of nodes of Graph Analytics so its peak WSS is 1900 MB. We observe that Data Analytics has a WSS of 1600 MB. As Data Analytics consists of lots of phases which have almost the same memory demand curve, we report mean execution time of one phase.

Initially, each VM is allocated with 1800 MB of memory. VM_1 runs Graph Analytics, and VM_2 runs Data Analytics at the same time. Figure 15 shows the WSS and WSS tracking status along the execution of the two programs. The IMT mechanism detects the phases well and the WSS tracking is turned off most time. Table V shows the execution times and page faults with and without balancing. MEB is able to achieve the performance close to when each VM is given sufficient memory. The VM with Graph Analytics shows a 9× speedup compared to when there is no balancing, while Data Analytics slows down by about 4% because of memory tracking overhead.

Figure 16 shows memory balancing when both VMs run an instance of Graph Analytics, while Graph Analytics on VM2 is started 70s later than that on VM1. In this evaluation, the initial memory is 1800MB. Note that when both VMs reach a peak memory demand of 1900MB, MEB sacrifices VM2 to satisfy the need of VM1 in order to minimize the overall page faults. As Table VI shows, performance of VM1 is close to the case when there is sufficient memory with an 18% slowdown to VM2. This is a significant improvement over the baseline where both VMs stick to 1800MB allocation.

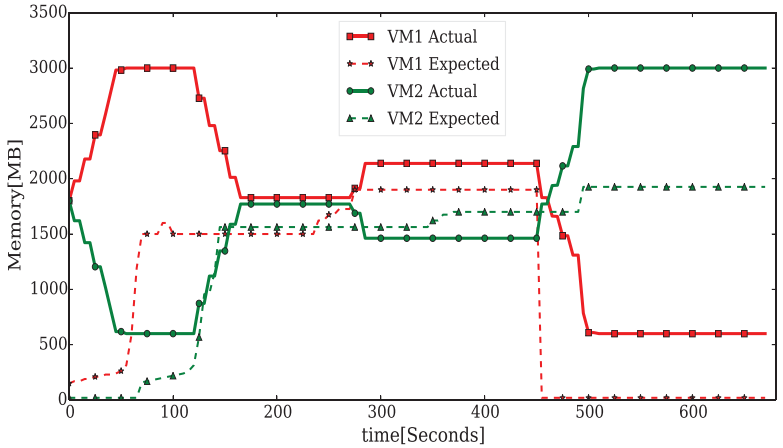

Fig. 16. MEB between VMs running Graph Analytics.

Table VI. Memory Balancing Between Graph Analytics

	Baseline		Balancing		Best	
	PF	time	PF	time	PF	time
VM1	4056276	4890.1	89571	518.2	12457	510.0
VM2	4112084	4873.2	2036128	605.0	12303	512.3

Table VII. Performance Comparison of 4 VMs

Program	Baseline	Balanced	Best
Graph Analytics	11058.4	546.5	510.3
DaCapo	115.3	117.2	114.6
429.mcf	4302.1	572.4	540.7
186.crafty	40.4	40.5	40.0

5.3.4. Mixed Workloads of Four VMs. To simulate a more realistic setting in which multiple VMs are hosted and diverse applications are deployed, four VMs are initialized with 1280MB of memory each, and different workloads are assigned to each VM. VM_1 runs the Graph Analytics with WSS of 1900 MB, VM_2 runs the DaCapo suite, VM_3 runs 429.mcf, and VM_4 runs 186.crafty. To make sure there exists a workload in every VM at any time, we run the benchmarks repeatedly and report the mean execution time. As shown in Table VII, with memory balancing, the performance of Graph Analytics is boosted by a factor of 20.2 while 429.mcf gets a speedup of 7.5. The performance of 186.crafty and DaCapo is slightly degraded by around 1%. The overall mean speedup of using memory balancing is 7.4. Compared with the two VM scenarios, the performance of Graph Analytics in the four VM setting is significantly enhanced due to a larger memory resource pool which allows the arbitrator to allocate more memory to these memory-hog programs.

5.3.5. Memory Overcommit. The evaluation in this section is performed on an Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz with 16GB of physical memory. We evaluate our balancing system with 16 VMs under a 16GB memory pool. We choose different workloads from SPEC CPU, DaCapo, and CloudSuite to accommodate workloads of different WSSs. We configure the number of nodes of Graph Analytics to generate different workloads.

In a commercial data center, users may estimate the resource demands of their workloads first and then purchase a VM which provides a sufficient amount of memory for their workloads. Suppose that the service provider only offers memory configurations with an increment of 512MB. In our evaluations, we define the *HighLimit* of every VM as $\lceil \frac{1.1*WSS}{512} \rceil * 512\text{MB}$. *HighLimit* is thus the maximum amount of physical memory a virtual machine can receive or, in other words, a user would like to subscribe. To

Table VIII. Memory Overcommit in 16 VMs (*HighLimit*: The Maximum Memory a VM Can Receive; *Init*: Initial VM Memory Allocation; *Best*: Execution Time with Enough Memory in a Single VM; *Baseline*: Execution Time Without Balancing; *Balanced*: Execution Time with Balancing; *SpeedUp*: *Baseline*/*Balanced*; *QOS*: *Best*/*Balanced*, and s is Short for Seconds in the Table)

Workload	WSS(MB)	HighLimit(MB)	Init(MB)	Best(s)	BaseLine(s)	Balanced(s)	SpeedUp	QOS
GraphAnalytics	1250	1536	1024	356.10	695.22	372.53	1.87	0.96
GraphAnalytics	1500	1536	1024	428.13	1362.17	446.23	3.05	0.96
GraphAnalytics	1750	2048	1536	499.58	1876.33	523.99	3.58	0.95
GraphAnalytics	2000	2560	1536	572.72	12192.51	586.99	20.77	0.98
GraphAnalytics	2250	2560	2048	637.37	9806.77	841.21	11.66	0.76
GraphAnalytics	2500	3072	2048	676.26	16276.40	994.24	16.37	0.68
WebServing	230	512	512	450.04	452.11	451.15	1.00	1.00
DataAnalytics	1600	2048	1024	199.03	1095.40	236.01	4.64	0.84
Dacapo	487	1024	512	114.22	115.15	118.45	0.97	0.96
186.crafty	77	512	512	40.29	40.32	40.95	0.98	0.98
429.MCF	1680	2048	1536	345.10	557.26	402.21	1.39	0.86
410.bwaves	672	1024	512	470.75	603.32	512.37	1.18	0.92
433.milc	655	1024	512	376.33	522.08	400.76	1.30	0.94
434.zeusmp	726	1024	512	409.11	625.26	417.60	1.50	0.98
436.cactusADM	698	1024	512	715.04	981.12	724.37	1.39	0.99
459.GemsFDTD	820	1024	1024	380.72	385.02	381.93	1.01	1.00
Total/Average	18895	24576	16384				4.54	0.92

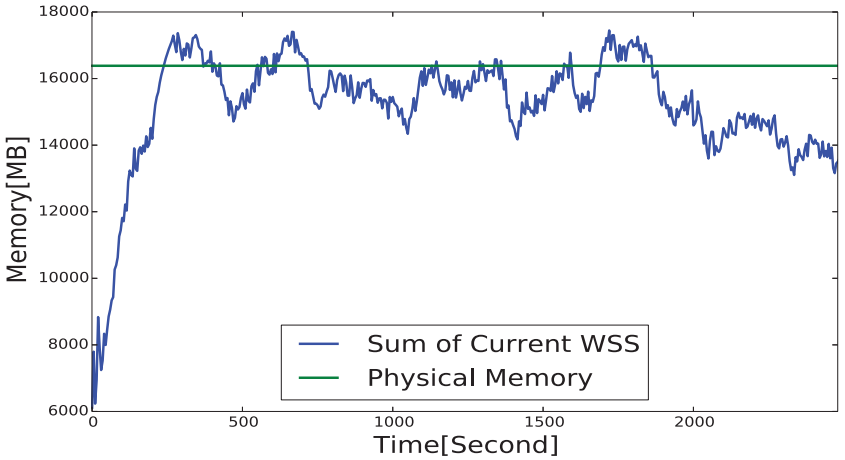

Fig. 17. Memory status of 16 VMs.

ensure all VMs are busy, for every VM, we run the workload repeatedly and calculate the average executing time for every round.

As shown in Table VIII, the sum of WSSs is 18895 MB, which is larger than the total machine memory 16384MB. Indeed, from the user point of view, the 16 VMs provides the amount of physical memory equals to the sum of *HighLimit*, which amounts to 24576MB. The system is apparently overcommitted. Figure 17 shows the total memory demands of the 16 VMs along execution. We observe that for about 15% of time, the system is short of memory and needs to compromise certain VMs to minimize total page misses. Memory balancing delivers an impressive speedup of 4.54 compared to the baseline. For most VMs, the balanced system is able to maintain an acceptable performance within 15% of the best , that is, the performance of VM always with the

HighLimit amount of memory can deliver. Specifically, quality of service is important for web-based applications. The throughput of the Web Serving benchmark stays at 21 operations/second on average with balancing, nearly the same as its best case.

6. RELATED WORK

Denning [1968] first defined WSS as the set of memory pages referenced by a process during a time interval. The same idea can be extended to VMs. WSS estimation provides a necessary metric that ensures the system to reach the maximum performance as expected.

The MRC-based WSS estimation is a widely proposed technique [Mattson et al. 1970; Chandra et al. 2005; Zhou et al. 2004; Yang et al. 2006; Jones et al. 2006; Sugumar and Abraham 1993; Saemundsson et al. 2014]. A page MRC plots the page miss ratios against various amounts of memory allocation. With an MRC, we can redefine WSS as the size of memory that results in less than a predefined tolerable page miss rate. Since an MRC models the performance and memory allocation size, it is especially suitable for memory resource arbitration. Unfortunately, exact implementation is too heavy for practical online use in production systems. Wires et al. [2014] presents a novel data structure called *counter stack*, which can produce approximate MRCs using sublinear space. Waldspurger et al. [2015] introduces an approximation algorithm that employs uniform randomized spatial sampling which further reduces the space overhead of MRC construction to constant. Our optimization instead focuses on phasing behavior and locality properties of applications.

The VMware ESX server [Waldspurger 2002] uses a share-based allocation scheme and a sampling method to determine memory utilization. One drawback of sampling is that it only works for the VMs that have free memory. Furthermore, it cannot tell the potential performance loss if more memory than what is currently idle is reclaimed.

Zhou et al. [2004] propose two approaches to capture memory accesses and construct page MRC for an application. The hardware approach has little overhead, but the hardware is not available on current commodity processors; the OS approach can acquire enough memory accesses with low cost via partitioning physical pages into two groups: frequently accessed pages and infrequently accessed pages. Our hot-set approach derives from this scheme. Yang et al. present an MRC-based virtual memory manager that tracks WSS of a garbage-collected application and computes the appropriate heap size to improve the application's performance [Yang et al. 2004, 2006]. RapidMRC [Tam et al. 2009] collects memory accesses by using performance monitoring units of PowerPC processors and builds an MRC for an L2 data cache. Magenheimer [Magenheimer 2008] uses an operating system's own performance statistics to guide memory resource management. However, the memory usage reported by most modern operating systems is larger than its WSS. Geiger [Jones et al. 2006] infers information about page admissions or evictions of a guest OS's buffer cache by observing page faults and intercepting disk I/Os. For each intercepted disk I/O, Geiger tracks the disk location and the associated memory page and infers if a page is newly allocated or evicted from the buffer to construct an MRC. However, it is unable to detect overallocated memory as there is no page eviction.

Instead of using a page-protection-based technique to intercept memory accesses, the hypervisor exclusive cache [Lu and Shen 2007] intercepts memory accesses from a VM by capturing its disk I/Os, but it introduces an additional layer of memory management. Moreover, since the cache is exclusive, a VM's memory states spread across its direct memory and the hypervisor, which breaks the semantics of hypervisor and complicates VM migration. In addition, it requires modification to the guest OS to notify the hypervisor of page release. Another approach is by periodically checking and scanning the access bit of a process's page table [Intel 2015]. The access frequency of

each page can be estimated and used to construct MRC, but sequentially scanning the whole page table is expensive. Zhang et al. [2009] present a locality jumping scheme that utilizes the spatial locality of a program to skip checking many nonaccessed pages. However, in a virtualized environment, setting and clearing the access bit in the hypervisor may disturb a VM's memory management because its page replacement policy usually relies on the access bits to infer page usage.

Program Phases. Prior studies [Denning 1980; Shen et al. 2004] have shown that an executing program usually exhibits phase behaviors. Some metrics, such as memory access patterns, are relatively stable within some temporal intervals while there exist abrupt changes. Sherwood et al. [2001] first apply Fourier analysis on basic block execution frequency to identify recurring phases and then find representative phases of whole program to accelerate architecture simulation. They further show that, by analyzing historic phases, future phase behaviors can be forecast and used to guide dynamic cache size configuration and processor width adaption [Sherwood et al. 2003]. For accurate phase detection, prior approaches use offline profiling because it provides a whole view of program characteristics and allows for expensive analysis such as the wavelet analysis used in Shen et al. [2004]. However, in some cases such as capturing the phase behaviors of a VM that may run any program instead of a particular one, on-line phase detection is desirable. Dhodapkar and Smith [2002] show an on-line phase detection scheme to guide the dynamic tuning for multiple configurable hardware units such as caches and TLBs, but it requires hardware support to collect every committed instruction. Nagpurkar et al. [2006] present a framework for on-line phase detection.

Memory Balancing. Memory balancing between VMs is widely deployed for performance optimization [Zhou et al. 2010], power saving [Liao et al. 2015], and memory overcommitment in data centers [Agmon Ben-Yehuda et al. 2014]. Mortar [Hwang et al. 2014] builds a shared memory pool, in which the VMM manages a volatile data store to increase memory utilization in a data center. VSwapper [Amit et al. 2014] systematically explores the behavior of uncooperative hypervisor swapping and implements an improved swapping subsystem for KVM. Ginkgo [Gordon et al. 2011] presents an application-driven approach to explore how memory can be shared effectively. XHive [Kim et al. 2011] introduces an efficient cooperative caching scheme for VMs, which manages buffer caches of consolidated VMs in order to accommodate a shared working set in machine memory. Rao et al. [2011] propose a distributed learning mechanism as well as self-adaptive capacity management to deploy VMs to proper nodes, but they admit that they have not found a good way to estimate the VM's WSS. Ginseng [Agmon Ben-Yehuda et al. 2014] focuses on a market-driven method and considers more human decisions on the balancing strategies.

7. CONCLUSION AND FUTURE WORK

LRU-based WSS estimation is an effective technique to support memory resource management. This article makes this technique more applicable by explaining how to significantly reduce its overhead. We improve previous achievements on overhead control and present a novel intermittent memory tracking scheme. Experimental evaluation shows that our solution is capable of reducing overhead while maintaining prediction accuracy. In an application scenario of balancing memory resources for VMs, our solution boosts the overall performance. In the future, we plan to develop theoretical models that verify the correlations among various memory events. Moreover, we expect to develop a system to derive a cache-level MRC with the assistance of a page MRC or vice versa. We also plan to investigate how the size of the hot set affects WSS prediction accuracy.

REFERENCES

Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. 2014. Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, New York, NY, 41–52.

Nadav Amit, Dan Tsafrir, and Assaf Schuster. 2014. VSwapper: A memory swapper for virtualized environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 349–366.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS Operat. Syst. Rev.* 37, 5 (2003), 164–177.

Hjortur Bjornsson, Gregory Chockler, Trausti Saemundsson, and Ymir Vigfusson. 2013. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. ACM, New York, NY, 59.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas Van- Drunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. ACM, New York, NY, USA, 169–190.

Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*. IEEE, 340–351.

Peter J. Denning. 1968. The working set model for program behavior. *Commun. ACM* 11, 5 (1968), 323–333.

Peter J. Denning. 1980. Working sets past and present. *IEEE Trans. Software Eng.* 1 (1980), 64–84.

Ashutosh S. Dhodapkar and James E. Smith. 2002. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings. 29th Annual International Symposium on Computer Architecture*. IEEE, 233–244.

Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, New York, NY, 37–48. DOI:http://dx.doi.org/10.1145/2150976.2150982

Abel Gordon, Michael Hines, Dilma Da Silva, Muli Ben-Yehuda, Marcio Silva, and Gabriel Lizarraga. 2011. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. *Proc. RESoLVE* (2011).

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Comput. Arch. News* 34, 4 (2006), 1–17.

Jinho Hwang, Ahsen Uppal, Timothy Wood, and Howie Huang. 2014. Mortar: Filling the gaps in data center memory. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, New York, NY, 53–64.

Intel. 2015. Intel 64 and IA-32 architectures software developers manual. *Volume 3A: System Programming Guide, Part* 1 (2015). http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2006. Geiger: Monitoring the buffer cache in a virtual machine environment. In *ACM SIGOPS Operating Systems Review*, Vol. 40. ACM, 14–24.

Hwanju Kim, Heeseung Jo, and Joonwon Lee. 2011. XHive: Efficient cooperative caching for virtual machines. *IEEE Trans. Comput.* 60, 1 (2011), 106–119.

Xiaofei Liao, Hai Jin, Shizhan Yu, and Yu Zhang. 2015. A novel memory allocation scheme for memory energy reduction in virtualization environment. *J. Comput. System Sci.* 81, 1 (2015), 3–15.

Pin Lu and Kai Shen. 2007. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proceedings of the USENIX Annual Technical Conference*. 29–43.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Not.* 40, 6 (2005), 190–200.

Dan Magenheimer. 2008. Memory overcommit... without the commitment. *Xen Summit* (2008), 1–3.

Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J.* 9, 2 (1970), 78–117.

Priya Nagpurkar, Michael Hind, Chandra Krintz, Peter F. Sweeney, and V. T. Rajan. 2006. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*. IEEE.

Jia Rao, Xiangping Bu, Kun Wang, and Cheng-Zhong Xu. 2011. Self-adaptive provisioning of virtualized resources in cloud computing. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, New York, NY, 129–130.

Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. 2014. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, New York, NY, 1–14.

Xipeng Shen, Yutao Zhong, and Chen Ding. 2004. Locality phase prediction. In *ACM SIGOPS Operating Systems Review*, Vol. 38. ACM, New York, NY, 165–176.

Timothy Sherwood, Erez Perelman, and Brad Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*. IEEE, Washington, DC, 3–14.

Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase tracking and prediction. In *ACM SIGARCH Computer Architecture News*, Vol. 31. ACM, New York, NY, 336–349.

Rabin A. Sugumar and Santosh G. Abraham. 1993. *Efficient Simulation of Caches Under Optimal Replacement with Applications to Miss Characterization*. Vol. 21. ACM, New York, NY.

David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. 2009. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, New York, NY, 121–132.

Carl A. Waldspurger. 2002. Memory resource management in VMware ESX server. *ACM SIGOPS Operat. Syst. Rev.* 36, SI (2002), 181–194.

Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, 95–110.

Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, Andrew Warfield, and Coho Data. 2014. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

Ting Yang, Emery D. Berger, Scott F. Kaplan, J. Eliot, and B. Moss. 2006. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 103–116.

Ting Yang, Matthew Hertz, Emery D. Berger, Scott F. Kaplan, J Eliot, and B. Moss. 2004. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 4th International Symposium on Memory Management*. ACM, New York, NY, 61–72.

Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*. ACM, New York, NY, 89–102.

Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Xiaoming Li. 2011. Low cost working set size tracking. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA, USA, 11–17.

Weiming Zhao, Zhenlin Wang, and Yingwei Luo. 2009. Dynamic memory balancing for virtual machines. *ACM SIGOPS Operat. Syst. Rev.* 43, 3 (2009), 37–47.

Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic tracking of page miss ratio curve for memory management. In *ACM SIGOPS Operating Systems Review*, Vol. 38. ACM, New York, NY, 177–188.

Wenyu Zhou, Shoubao Yang, Jun Fang, Xianlong Niu, and Hu Song. 2010. Vmctune: A load balancing scheme for virtual machine cluster using dynamic resource allocation. In *Proceedings of the 2010 9th International Conference on Grid and Cooperative Computing*. IEEE, 81–86.

Received April 2015; revised September 2015; accepted November 2015