Dynamic Memory Balancing for Virtual Machines

Weiming Zhao[†] Zhenlin Wang[†‡] Yingwei Luo[‡]

[†] Department of Computer Science, Michigan Technological University,
and [‡]Department of Computer Science and Technology, Peking University
wezhao@mtu.edu zlwang@mtu.edu lyw@pku.edu.cn

Abstract

Virtualization essentially enables multiple operating systems and applications to run on one physical computer by multiplexing hardware resources. A key motivation for applying virtualization is to improve hardware resource utilization while maintaining reasonable quality of service. However, such a goal cannot be achieved without efficient resource management. Though most physical resources, such as processor cores and I/O devices, are shared among virtual machines using time slicing and can be scheduled flexibly based on priority, allocating an appropriate amount of main memory to virtual machines is more challenging. Different applications have different memory requirements. Even a single application shows varied working set sizes during its execution. An optimal memory management strategy under a virtualized environment thus needs to dynamically adjust memory allocation for each virtual machine, which further requires a prediction model that forecasts its host physical memory needs on the fly. This paper introduces *MEmory Balancer* (MEB) which dynamically monitors the memory usage of each virtual machine, accurately predicts its memory needs, and periodically reallocates host memory. MEB uses two effective memory predictors which, respectively, estimate the amount of memory available for reclaiming without a notable performance drop, and additional memory required for reducing the virtual machine paging penalty. Our experimental results show that our prediction schemes yield high accuracy and low overhead. Furthermore, the overall system throughput can be significantly improved with MEB. [1]

Categories and Subject Descriptors D.4.8 [*OPERATING SYSTEMS*]: Performance; D.4.8 [*OPERATING SYSTEMS*]: Storage Management

General Terms Design, Experimentation, Management, Measurement, Performance

Keywords Virtual Machine, Memory Balancing, LRU Histogram

1. Introduction

Recently, virtualization technologies, whose roots can be traced back to the mainframe days, are drawing renewed attention from a variety of application domains such as data centers, web hosting, or even desktop computing, by offering the benefits on security, power-saving, and resource-efficiency. Virtual machines (VMs) run on top of the hypervisor or the virtual machine monitor (VMM)[2], which multiplexes the hardware resources. The VMM has the ultimate control on all hardware resources while offering each guest OS an illusion of a raw machine by virtualizing the hardware. No matter if we use a virtualized system for security or hardware multiplexing, we usually boot several virtual machines on a single computer. Those machines eventually share or compete for the hardware resources.

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. The host memory allocation is mostly static–each virtual machine is assigned a fixed amount of host memory in the beginning. Although Xen and VMware provide a ballooning driver to dynamically adjust host memory allocation, existing studies are insufficient to tell when to reallocate and how much memory a virtual machine needs or is willing to give up to maintain the performance of the applications running on it (Barham et al. 2003; Waldspurger 2002). This paper proposes *MEmory Balancer* (MEB) which dynamically monitors the memory usage of each virtual machine, accurately predicts its memory needs, and periodically reallocates host memory to achieve high performance.

The VMware ESX server (Waldspurger 2002) uses a *share*-based allocation scheme. The share of a virtual machine is determined by its memory utilization and a maximum and minimum memory quota predefined by the system administrator. VMware uses a sampling method to determine memory utilization. The system periodically samples a random set of pages and the utilization is the ratio of the pages actively in use. A virtual machine with low memory utilization has a lower share and thus is more likely to get its memory reclaimed. One drawback of the sampling approach is that it can only predict the memory requirement of a virtual machine when there is free memory. Furthermore, it cannot tell the potential performance loss if more memory than what is currently idle is reclaimed.

Recent studies have used Mattson et al.'s LRU stack distance-based profiling (Mattson et al. 1970) to calculate page miss ratio of a process with respect to memory allocation (Zhou et al. 2004; Yang et al. 2006). These studies either require tracking virtual addresses or need interaction between processes and the OS, and thus cannot be directly applied to virtual machines where we need to estimate memory needs of each virtual machine. In a virtualized environment, execution of a memory access instruction is usually handled by the hardware directly, bypassing the hypervisor unless it results in a page fault. On the other hand, page tables are considered as privileged data and all updates to page tables will be captured by the VMM, which provides us a chance to manipulate the page table entry for each page table update request. Inspired by previous process-level memory predictor, our predictor, which

[1] This work is based on an earlier work: Dynamic Memory Balancing for Virtual Machines, in proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments, March 2009 © ACM, 2009. http://doi.acm.org/10.1145/1508293.1508297 This version presents some new experimental results with revised SPEC WEB 2005 benchmark for better performance comparison. We also added a more detailed overhead analysis.

[2] The terms *hypervisor* and *VMM* are used interchangeably in our paper

is located in VMM, tracks normal memory accesses by revoking user access permission and trapping them into VMM as minor page faults (Zhou et al. 2004). Note that it is prohibitive to track virtual addresses for each process in a VM. We instead build an LRU histogram based on host physical addresses (or machine addresses). As a consequence, this estimator cannot provide the relationship between page miss and memory size beyond the size of machine memory allocation. When a VM's memory is under-allocated, the miss ratio curve is unable to predict how much extra memory is needed. MEB uses a simple predictor to monitor the swap space usage and uses it as a guide to increase the memory allocation for a virtual machine.

We have implemented MEB in Xen (Barham et al. 2003). Experimental results show that it is able to automatically balance the memory load and significantly increase the performance of the VMs which suffers from insufficient memory allocation with a slight overhead to the initially over-allocated VMs.

The remainder of this paper is organized as follows. Section 2 presents the background and related work. Section 3 describes the dynamic balancing system in detail. Section 4 discusses experimental results, comparing with previous approaches, and Section 5 concludes.

2. Background and Related Work

2.1 LRU Histogram

The LRU histogram has been widely used to estimate cache and page miss rate with respect to cache size or memory size (Mattson et al. 1970; Chandra et al. 2005; Zhou et al. 2004; Yang et al. 2006; Jones et al. 2006; Sugumar and Abraham 1993). As the LRU (or its approximation) replacement policy is mostly used for page or cache replacement, one can generate an LRU histogram to track the hit/miss frequency of each LRU location. Assuming a system with only four pages of physical memory, Figure 1 shows an example LRU histogram in the top half of the figure for an application with a total of 200 memory accesses. The histogram indicates that 100 accesses hit the MRU page, 50 accesses hit the second MRU slot, and so on. Apparently the page hit rate is $(100 + 50 + 20 + 10)/200 = 90\%$. We can tell that if we reduce the system memory size by a half, the hits to the first two LRU slots are still there while the hits to the next two LRU slots now become misses. The hit rate becomes $(100 + 50)/200 = 75\%$. The LRU histogram thus can accurately predict miss rate with respect to the LRU list size. The bottom part of Figure 1 is the miss ratio curve corresponding to the histogram on the top. Zhou et al. (Zhou et al. 2004) use virtual page addresses to organize the LRU list and thus can predict the miss rate with respect to any physical memory size for a process. We instead implement an LRU histogram in the hypervisor and track host physical (machine) addresses. Therefore the scale of the memory size cannot be greater than the host memory size allocated to a virtual machine.

2.2 Memory Management

Typically, when a VM starts, the hypervisor assigns a fixed amount of memory to the VM and passes a nominal value of memory size to the guest OS. This value serves as the maximum memory size that the guest OS can manage. When a VM is scheduled to run, most memory accesses are directly handled by the hardware without involving of the hypervisor. It is the guest OS's responsibility to effectively utilize the allocated memory. However, page tables are usually considered as privilege data. The related operations, for example, installing a new page table or updating a page table entry, should be validated by the hypervisor first. Typically, the page table area will be protected by setting higher access permission. Page table updates should be requested via an explicit service call as in paravirtualization or handled implicitly as page faults as in full virtualization.

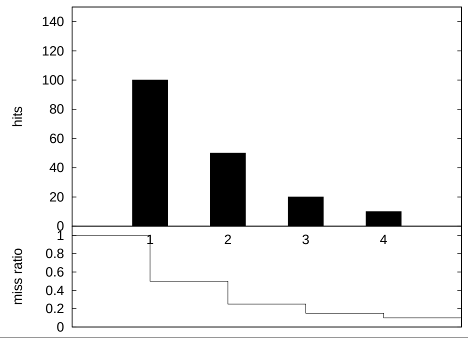


Figure 1. LRU Histogram Example

To dynamically change the memory allocation of a VM, a ballooning (Waldspurger 2002) mechanism is proposed by Waldspurger. Conceptually, as a kernel space driver, the *balloon driver* gives the OS an illusion that this driver needs more memory to run while actually this memory is returned to the hypervisor, which essentially reduces memory allocation of the VM. Similarly, releasing memory from the ballooning driver increases the memory allocation for a VM. The ballooning mechanism takes advantage of a guest OS's knowledge about which pages to reclaim.

2.3 Related Work

As discussed in Section 1, in (Waldspurger 2002), a page sampling strategy is presented to infer the memory utilization and thus the memory needs of a VM. During a sampling interval, accesses to a set of random pages are monitored. By the end of the sampling period, the page utilization of the set is used as an approximation of global memory utilization. Reclaimed memory is reallocated to the VMs based on their utilization. Our experiments in Section 4 show that the accuracy of this sampling-based scheme is lower than our LRU histogram based estimation. And since the performance of a virtual machine is not usually proportional to the size of allocated memory, it is hard to evaluate the performance impact of various amount of memory allocation to a VM. In particular, it cannot give the performance impact if some actively used memory is reclaimed for usage in another virtual machine for a more important application. Therefore it may miss an opportunity to minimize system-wide page misses. Magenheimer (Magenheimer 2008) uses the operating system's own performance statistics to guide the memory resource management. However, the memory usage reported by most modern operating systems includes the infrequently used areas, which can be reclaimed without a notable performance penalty.

Zhou et. al (Zhou et al. 2004) propose the page miss ratio curve to dynamically track working set size. They suggest two approaches: hardware based and OS based. The former one tracks the miss ratio curve for the entire system by monitoring physical memory accesses. The latter one maintains a per-application miss ratio curve based on applications' virtual addresses. Unfortunately, neither of them fits in the domain of virtualization well:

1. The hardware solution requires an extra circuit, which is unavailable on commodity processors.
2. The hardware solution monitors physical memory, which cannot estimate the effect of a memory increment beyond the allocated size.
3. The virtual address based LRU does solve the problem in (2), but it works on the per-process level. To track the memory usage at the system scope, it has to maintain an LRU list and a

histogram for every process, resulting in prohibitive space and management cost.

Nonetheless, the page miss ratio curve and memory access intercepting are extensively used in recent studies with extensions. CRAMM (Yang et al. 2006) is an LRU histogram based virtual memory manager, which maintains an LRU histogram for each process. A modified JVM communicates with CRAMM to acquire intelligence about its own working set size (WSS) and the OS's available memory, and adjusts the heap size to minimize the occurrence of garbage collection without incurring an unacceptable number of major page faults. CRAMM builds a model that correlates the WSS of a Java application and its heap size. Hence, by detecting WSS changes, the heap size can be adjusted accordingly. However, there is no such correlation between a virtual machine's WSS and its allocated memory size. In addition, CRAMM requires modifications to the OS, which we manage to avoid.

Geiger (Jones et al. 2006), on the other hand, detects memory pressure and calculates the amount of extra memory needed by monitoring disk I/O and inferring major page faults. However, when a guest is over-allocated, it is unable to tell how to shrink the memory allocation, which prevents the resource manager from reclaiming the idle memory for use in other guests. The Black-box and Gray-box strategies (Wood et al.) migrate hot-spot guests to the physical machines with enough resources. Its memory monitoring function is a simplified version of Geiger and therefore inherits its drawbacks.

Hypervisor exclusive cache (Lu and Shen 2007) is most relevant to our research. The VMM uses an LRU based miss ratio curve to estimate WSS for each virtual machine. In this design, each VM gets a small amount of machine memory, called *direct memory*, and the rest of the memory is managed by the VMM in the form of exclusive cache. By allocating a different amount of cache to a VM, it achieves the same effect as changing its memory allocation. Since this approach can track all memory accesses above the minimum memory allocation, both WSS increases and decreases can be deduced, although the maximum increment is still bounded by the cache size. This design introduces an additional layer of memory management. Moreover, since the cache is exclusive, system states spread across a VM's direct memory and the VMM, which breaks the semantics of the VMM. It is therefore hard to migrate VMs. Our solution uses ballooning which keeps the VMs migrate-able. The hypervisor exclusive cache requires modification to the guest OS to notify the VMM of page eviction or release. Our approach only needs to trace page table changes and the whole balancing is transparent to guest OSes.

3. Memory Monitoring and Automatic Resizing

3.1 System Overview

MEB consists of two parts: an estimator and a balancer. The estimator 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 estimator. Figure 2 illustrates the system architecture of MEB.

1. Memory accesses from the VMs, denoted by dashed arrows, are intercepted by the VMM and then used to update the histogram for each virtual machine.
2. Optionally, each VM may have a background process, *Mon*, which gets the swap usage from the OS performance statistics and posts the swap usage statistics to the central data storage.

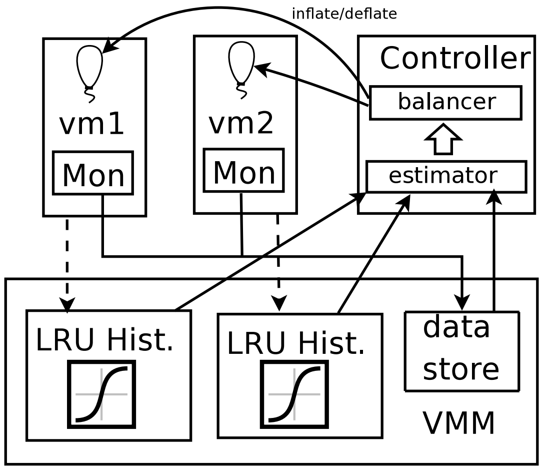


Figure 2. System Overview

3. The estimator reads the histograms and collects information from the central store periodically. It computes the expected memory size for each VM.
4. The balancer arbitrates memory contention, if it exists, and sets the target memory size for each VM via ballooning.

3.2 The LRU-based Predictor

Although operating systems keep track of memory usage, most of them reclaim inactive memory lazily. For example, in the Linux kernel, when there is no memory pressure, inactive pages may not be reclaimed. As discussed in Section 2.1, if we can build an LRU histogram that tracks memory accesses for each guest OS, the histogram can indicate how inactive those frames near LRU locations are. A page miss ratio curve, which derives from the LRU histogram, indicates the relationship between memory size and page miss rate. We define the *Working Set Size* (WSS) of a guest OS as the amount of machine memory needed without causing significant page swapping. More specifically, let δ be a small page miss rate we can tolerate, the WSS can be selected as the minimum memory size that yields page miss rate no larger than δ, which can be derived from the miss rate curve. By dynamically monitoring memory accesses and constructing the LRU histogram, we thus can predict the WSS of a guest OS. If the WSS is smaller than the actual host memory size, the guest OS can give up some host memory accordingly.

3.2.1 Intercepting Memory Accesses

Constructing the LRU histogram requires the knowledge of target addresses of memory accesses which are usually transparent to the VMM. We follow a widely used technique to trap memory accesses as minor page faults by setting a higher access privilege to the pages of concern. For example, on x86 processors, bit 2 of page table entry (PTE) toggles the *user* or *supervisor* privilege level (Application 2006). Changing the bit from *user* to *supervisor* removes guest's permission and makes normal accesses trap into the VMM as minor page faults.

Trapping all memory accesses would incur prohibitive cost. Instead, we logically divide host pages into two sets, a hot page set H and a cold page set C. Only accesses to cold pages are trapped. Initially, all pages are marked as *cold* when a new page table is populated (e.g. when creating a new process) and the VMM removes permissions from all entries. Later, when an access to a cold page causes a minor page fault, it traps into the VMM. In the

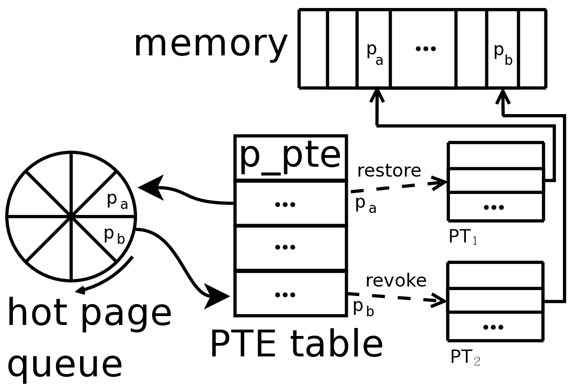


Figure 3. Hot Set Management
(Suppose an access to machine page p_a is trapped. Then the permission in the corresponding PTE in page table PT_1 is restored and the address of the PTE is added to the PTE table. Next, p_a is enqueued and p_b is dequeued. Use p_b to index the PTE table and locate the PTE of page p_b. The permission in the PTE of page p_b is revoked and page p_b becomes *cold*.)

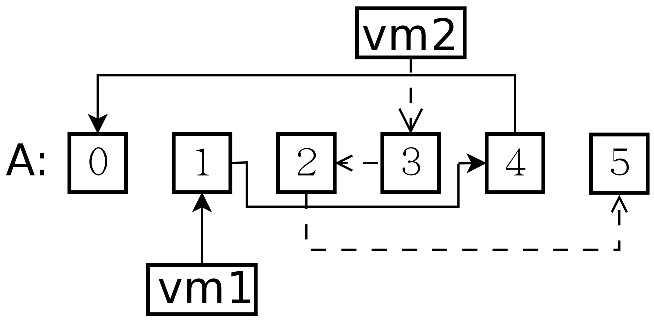


Figure 4. The LRU Lists in VMM
(All nodes are organized as an array, A. The solid line links the LRU list of vm1: $node_1$, $node_4$ and $node_0$; the dotted line represents the list of vm2: $node_3$, $node_2$ and $node_5$. For simplicity, a singly linked list is drawn.)

fault handling routine, the context information provides the virtual address and the location of the associated page table, from which the PTE and the machine address can be located in $O(1)$ time. The routine further restores the permission for future non-trapping guest accesses. As a result, the recently accessed page turns *hot*. Since any subsequent accesses to the hot page will not be trapped, those accesses will not incur any overhead. To maintain reasonable accuracy, the number of hot pages should be limited. Though the concept of *hot/cold* pages is similar to the active/inactive lists in (Yang et al. 2006), our design decouples the page-level hot/cold control from the LRU list management. The LRU list can work on a coarser tracking granularity to save space and time, which will be elaborated in Section 3.2.2.

We implement the hot page set, H, as an FIFO queue. Once a page is marked as hot, its page frame number is appended to the tail of the queue. When the queue is full, the page referred to by the head entry is degenerated to the cold set and protected by turning off *user* permission. A subsequent reference to this page will trigger a minor fault. We only store the machine page frame number in H. To quickly change the permission in the corresponding PTE of the page, we need a map from machine frame numbers to PTEs. We introduce a *PTE table*, which stores the PTE address for every machine page during the page fault handling. When a page is dequeued from H and transited from *hot* to *cold*, we look up its PTE address in the PTE table which is indexed by frame number. Occasionally, the PTE table may contain a stale entry (e.g. the PTE is updated), which can be easily detected by comparing the requested page frame number and the actual page frame number in the PTE. Figure 3 explains the whole process with an example.

The intuition behind our design stems from the program locality. Typically, accesses to a small set of pages account for a large portion of overall accesses. These pages are hot pages and the working set size can never be smaller than the total size of the hot pages. We thus only need to track page miss ratio with respect to the cold set. Furthermore, due to locality, the number of accesses to the cold set is small even with a small hot set. The overhead caused by the minor faults is thus significantly reduced.

3.2.2 The LRU List and Histogram

The LRU histogram is built with respect to the cold pages. Maintaining the LRU list at the page level requires a large amount of memory as each page needs a node in the list and each node requires a counter in the histogram. To reduce the space cost, we instead group every G consecutive pages as a unit. Every node in the LRU list corresponds to G pages. Let M be the number of machine memory pages and G be the granularity of tracking. $N = \frac{M}{G}$ is the maximum number of nodes in the LRU list. A smaller G means a larger memory requirement and higher prediction accuracy. The choice of G is determined experimentally (see Section 4.1.1) .

Whenever a memory reference is trapped, three operations will be performed on the list: locating the corresponding node in the LRU list, deleting it from the list, and inserting the node to the list head. The latter two are apparently constant time operations, given that we organize the LRU list as a doubly linked list where each node has two pointer fields – $prev$ and $next$. To expedite the first operation, all LRU nodes of all VMs are stored in a single global array A indexed by page frame number divided by G. Given a machine page frame number PFN, the corresponding node can be located in $O(1)$ time: $A[\frac{PFN}{G}]$. For each guest OS, there is a head pointer that points to the first node in its own LRU list. With this data structure, the costs of all three operations are all constant. Figure 4 gives an example when two VMs are monitored.

Updating the LRU histogram takes more time. The key point is to locate the LRU position of a node in the linked list. When a node at position i is accessed, the counter of the bar i in the LRU histogram is incremented by one. A simple solution to locate the position is to visit the linked list sequentially from the head. Although it takes worst-case linear time, for a program with good locality, most searches will hit within the first several nodes. Moreover, a large tracking granularity will reduce the overall size of the LRU list and thus improve average search time. Our experimental results show that when G is 32, the overhead is negligible and the precision is enough for automatic memory resizing (see Section 4.1.1).

A program with poor locality can exhibit large overhead caused by histogram updating as many LRU position searches would be close to the worst case. Yang et al. adaptively change the size of hot pages by monitoring the overhead (Yang et al. 2006). The size of hot pages is increased when the overhead is large. We instead directly predict the overhead using the miss rate curve generated based on the LRU histogram. We increase the hot set size when we find the locality is poor and thus the overhead is high. Let $m(x)$ be the function of the miss ratio curve, where x is the miss ratio and $m(x)$ is the corresponding memory size. A case of bad locality occurs when all accesses are uniformly distributed, resulting in $m(50\%) = \frac{WSS}{2}$. We use $\alpha = m(50\%)/\frac{WSS}{2}$ to quantify the locality. The smaller the value of α is, the better the locality. In our

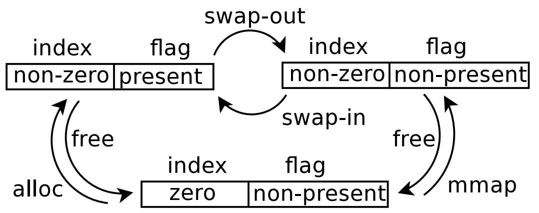


Figure 5. Transitions of PTE States

system, we use $\alpha \geq 0.5$ as an indicator of poor locality. When such a situation is detected, we incrementally increase the size of hot set until α is lower than 0.5 or the hot set reaches a preset upper bound.

The LRU histogram is sensitive to WSS increase but responds slowly to the decrease of memory usage in some cases. For instance, when a memory intensive application exits, the LRU histogram may still show high memory usage because no updates are made to it. Though the decay (Yang et al. 2004) algorithm was proposed to alleviate this problem, responses to memory decreases are still not as rapid as to memory increases. However, when there is no severe memory resource contention, slow responses of the LRU histogram will not affect the performance much.

3.3 Dynamic Memory Growth

The LRU-based WSS prediction in Section 3.2 can estimate WSS no larger than the current memory reservation. To detect memory insufficiency, this section describes two approaches. One approach resembles the eviction-based LRU histogram as proposed in (Jones et al. 2006) and the other is simply based on OS statistics. We turn on memory insufficiency prediction only when the number of major page faults is above a threshold during an interval.

To figure out how much extra memory is needed, a natural solution is to treat the disk as memory and construct an LRU histogram to estimate the WSS on disk. By monitoring page table changes, the page swapping activities can be inferred. Figure 5 illustrates the changes in a page table entry under different scenarios. When an OS dumps the content of a page to disk (swapping-out), the *Presence* bit of the PTE will be flipped and the index field will be changed from the machine frame number to the location on disk. Later, when the page is needed, a major page fault happens and the reverse operations will be applied on the PTE (swapping-in). As all page table updates are visible to VMM, the pattern of swap space usage can be inferred. By tracking the page swapping-in, an LRU histogram based on disk locations can be established and its WSS can be used to guide memory growth.

A more straightforward approach is to use swap usage as the growth amount. On most modern OSes, swap usage is a common performance statistic and is available for user-level access. A simple background process which runs on each VM can be designed to retrieve the value periodically and pass it to the estimator. Though, intuitively, it may overestimate the WSS, it is actually close to the histogram based estimation because the accesses to disk are filtered by in-memory references and therefore show weak locality. Moreover, it causes less overhead and reflects the swap usage decrease more rapidly than the LRU based solution. Both approaches are implemented in MEB and the results are compared in Section 4.1.2.

3.4 Automatic Memory Resizing

Once the estimator computes the work set sizes for all VMs, the balancer determines the target allocation sizes for them. Assume P is the size of all available host machine memory when no guest is running, V is the set of all VMs. For QoS purposes, each VM$_i \in V$ is given a lower bound of memory size L_i. Let $E_i = max(L_i, WSS_i)$ be the *expected memory size* of VM$_i$. When $P \geq \sum E_i$, all VMs can be satisfied. We call the residue of allocation $(P - \sum E_i)$ as *bonus*. The *bonus* can be spent flexibly. In our implementation, we aggressively allocate the bonus to each VM proportionally according to E_i. That is $T_i = E_i + bonus \times \frac{E_i}{\sum E_i}$, where T_i is the eventual target memory allocation size.

When $P < \sum E_i$, 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. Let $mrc_i(x)$ be the miss ratio curve and nr_i be number of memory accesses in a recent epoch of VM$_i$. Given a memory size m, the number of page misses is $miss_i(m) = mcr_i(m) \times nr_i$. The balancer tries to find an allocation $\{T_i\}$ such that $\sum_{i \in V} miss_i(T_i)$, the total penalty, is a minimum.

Since ballooning adjusts memory size on a page unit, a brute force search takes $O(M^{|V|})$ time, where M is the maximum number of pages a VM can get. We propose a quick approximation algorithm. For simplicity of discussion, we assume that there are two VMs, VM$_1$ and VM$_2$, to balance. Choosing an increment/decrement unit size S ($S \geq G$), the algorithm tentatively reduces the memory allocation of VM$_1$ by S, increases the allocation of VM$_2$ by S, and calculates the total page misses of the two VMs based on the miss rate curves. We continue this step with increment/decrement strides of $2S$, $3S$, and so on, until the total page misses reach a local minimum. The algorithm repeats the above process but now reducing allocation of VM$_1$ while increasing allocation for VM$_2$. It stops when it detects the other local minimum. The algorithm returns the allocation plan based on the lower of the two minima. This algorithm can run recursively when there are more than two VMs.

Sometimes, the two minima are close to each other in terms of page misses but the allocation plans can be quite different. For example, when two VMs are both eager for memory, one minimum suggests $\langle VM_1 = 50MB, VM_2 = 100MB\rangle$ with total page misses of 1000, while the other one returns $\langle VM_1 = 100MB, VM_2 = 50MB\rangle$ with total page misses of 1001. The first solution wins slightly, but the next time, the second one wins with a slightly lower number of page misses and this phenomenon repeats. The memory adjustment will cause the system to thrash and degrade the performance substantially. To prevent this, when the total page misses of both minima are close (e.g. the difference is less than 10%), the allocation plan closer to the current allocation is adopted.

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. So during each balancing, we limit the maximum reduction to 20% of its current memory size to let it shrink gradually.

4. Implementation and Evaluation

MEB is implemented on Xen 3.2 on an Intel server. The balancer is written in Python and runs on Domain-0. Paravirtualized 64-bit Linux 2.6.18 runs on VMs as the guest OS. The server has 8 cores (two Intel Xeon 5345 2.33 GHz Quad-Core processors) and 4 GB 667 MHZ Dual Ranked memory. To exclude the impact of CPU resource contention when multiple VMs are running, each VM is assigned a dedicated CPU core.

To evaluate the effect of MEB, various benchmarks are measured. We run two micro kernel benchmarks, DaCapo (Blackburn et al. 2006), SPEC CPU 2000 (spe a) and SPEC Web 2005 (spe b). Originally, SPEC Web 2005 was designed to measure the maximum number of simultaneous user sessions during a long time period with respect to a QoS threshold. To match up with the short completion time of other benchmarks that run side by side on other

VMs, the SPEC Web harness program is modified to measure completion time for a specified number of total requests.

Two micro kernel benchmarks, `random` and `mono`, are designed to test the accuracy of WSS estimation. Given a range $[low, high]$, during each phase, `random` first allocates a random amount of memory of size $r \in [low, high]$ and then randomly visits the allocated space for a fixed number of iterations. When the iterations are finished, the memory is freed and a new phase starts. `Mono` is similar to `random` except that the memory amount allocated in each phase first increases monotonically from low to $high$ and then decreases monotonically from $high$ to low.

DaCapo is a Java benchmark suite, which includes 10 real world applications with non-trivial memory requirements for some of them. We use JikesRVM 2.9.2 with the production configuration as the Java virtual machine (Alpern et al. 1999, 2000; Arnold et al. 2000; jik). By default, the initial heap size is 50 MB and the maximum is 100 MB.

In our experiments, MEB monitors memory accessing behavior for each VM and balances memory allocation 5 times per minute. For WSS estimation, we select the cutoff point at 5% (i.e. 5% page miss ratio is acceptable). By default, all VMs are allocated with 214 MB of memory initially. The baseline statistics are measured on VMs without monitoring and balancing turned on.

4.1 Key Parameters and Options

4.1.1 LRU Tracking Unit

In Section 3.2.2, we use a tracking unit consisting of G pages to construct the LRU histogram. The choice of tracking unit size is a trade-off between estimation accuracy and monitoring overhead. To find out the appropriate tracking unit size, we measure the execution time and estimation accuracy for various tracking granularity from 1 page to 1024 pages. To evaluate the overhead introduced solely by different tracking granularity, MEB is turned on but not adjusting memory. Thus MEB only monitors the WSS. We first run a program which constantly visits 100 MB of memory space (no swap space is used). During the program execution, estimations are reported periodically and the highest and lowest values are recorded. As illustrated by Figure 6, when G increases from 1 to 16, 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. Figure 7 shows the total execution time for the selected DaCapo and SPEC INT benchmarks with various tracking unit sizes. [3] As indicated by Figure 7, when G grows from 1 to 32, both the execution time and the histogram updating overhead drop significantly , while the curves become flat when $G \geq 32$. Therefore, we use 32 pages as the LRU tracking unit for the remaining evaluation.

4.1.2 OS-based vs. LRU-based Memory Growth Estimation

We have implemented both of the memory growth predictors discussed in Section 3.3: the OS statistics based and the LRU-based. We run `random` and `mono` with a range of $[40, 350]$ MB. Figure 8 shows the extra memory estimated by the two predictors. The OS statistics based predictor follows the memory usage change well. In both benchmarks, it tracks the memory allocation and free in each phase as the curves go up and down in both benchmarks. The LRU-based estimation changes slowly especially when swap usage drops. In the environments with rapid memory usage changes like our benchmarks, the former one is more suitable. The following experiments all use the OS statistics based predictor.

[3] For some DaCapo programs, when tracking unit is less than 4 pages, the excessive overhead causes JVM unstable.

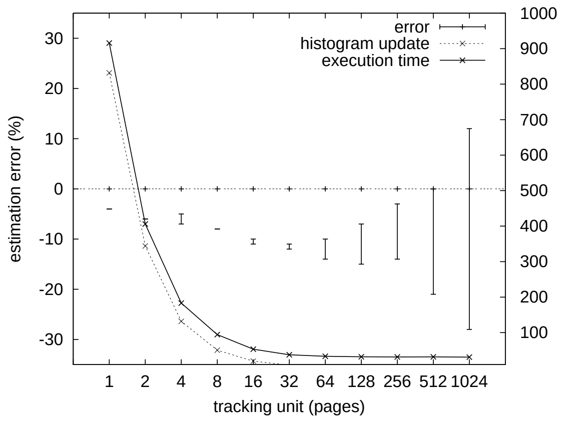


Figure 6. Relationship Between Tracking Unit, Accuracy and Overhead

4.2 Working Set Size Estimation

First, we run `mono` and `random` with a range of $[40, 170]$ MB. 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 sampling based estimation as used in the VMware ESX server (sample size is 100 pages, interval is 30 seconds[4]). As Figures 9(a) and 9(b) show, when memory usage increases, our predictor follows closely. Due to the nature of the LRU histogram, it responds slowly to memory usage decrease as discussed before. The average error of the LRU-based and sampling-based estimations is 13.46% and 74.36%, respectively, in `random`, and 5.78% and 99.16%, respectively, in `mono`. The LRU-based prediction is a clear winner.

Figure 9(c) and 9(d) show the results when the WSS of both benchmarks varies from 40 MB to 350 MB. Now the WSS can be larger than the actual memory. In this case, swap usage is involved in calculating the WSS. The sampling scheme cannot predict WSS beyond the current host memory allocation, while combining LRU-histogram and OS swap usage tracks the WSS well.

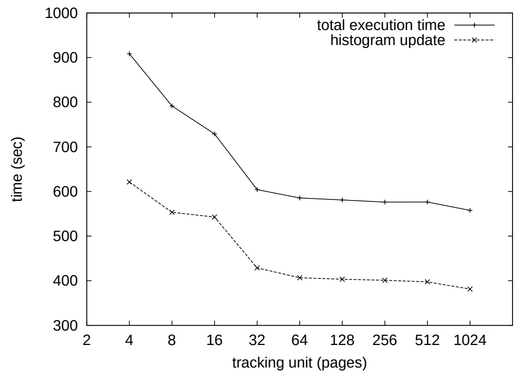
4.3 System Overhead

The memory overhead of MEB mainly comes from four data structures: the hot sets, the PTE table, the LRU list array, and the LRU histograms. The PTE table and the LRU list array are global, while one hot set and one histogram are local to each VM. The total size of the PTE table is $M \times sizeof(void*)$, where M is the number of physical memory frames on the host machine. The number of nodes in the LRU list array and the number of entries in each histogram are both equal to $\frac{M}{G}$, where G is tracking unit. Each LRU list node has two pointers, $prev$ and $next$. Each slot of the histogram is a long integer. The number of entries in the hot set is configurable and each entry uses four bytes to store the page number.

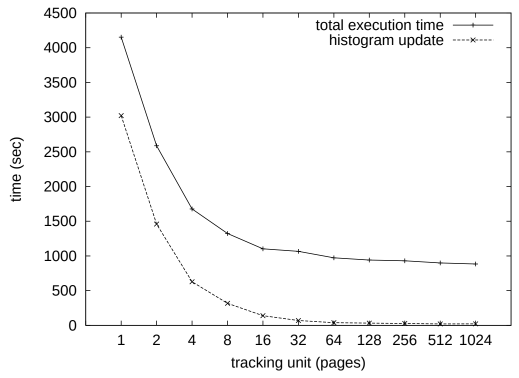
On our experimental platform, $M = 2^{20}$, given the total machine memory of 4 GB and the page size of 4 KB. A pointer and a long integer are 8 bytes each. The number of hot set entries is configured as $8K$ by default with a maximum of $16K$. Therefore, when $G = 32$, the total memory needed is: 8 MB for the PTE table, 0.5 MB for the LRU list array, 0.25 MB for each histogram and 64 KB maximum for each hot set, which is quite scalable to the physical memory size and the number of virtual machines.

To evaluate the runtime overhead of MEB, we measure the execution time of the selected benchmarks with only the WSS estimator enabled. As shown in Figure 10, the mean overhead is

[4] the same parameters as used in (Waldspurger 2002)

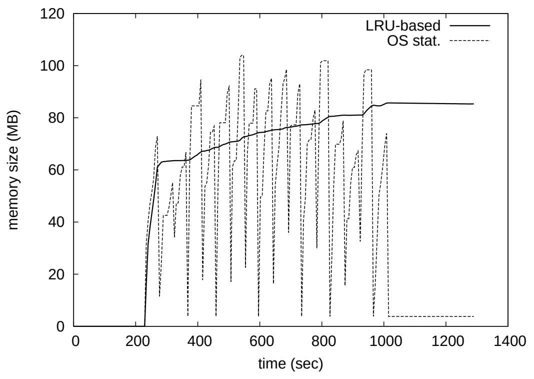


(a) DaCapo

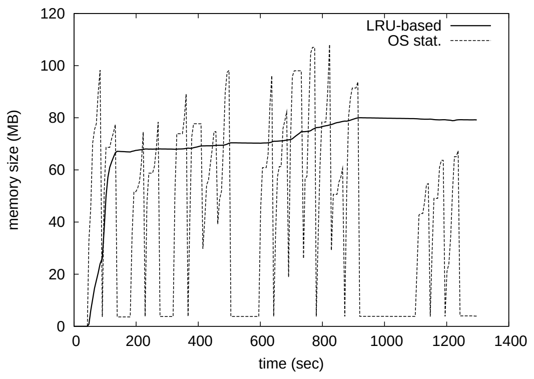


(b) SPEC INT 2000

Figure 7. Relationship Between Tracking Unit and Performance



(a) mono



(b) random

Figure 8. Swap LRU vs. Swap Size

small: 2.63% for the DaCapo suite, 2.13% for SPEC INT, and 0.1% for SPEC Web. Among the benchmark programs, `181.mcf` is the most sensitive to the hot set size. The adaptive hot set growth reduces its overhead by 70%, but there is a still 24% performance loss.

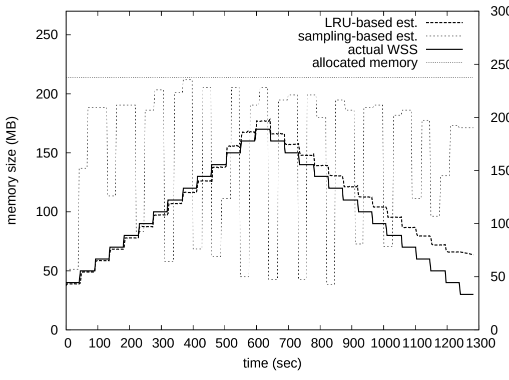
We further break down the runtime overhead into two sources: histogram updating and page fault handling. The code path of page fault handling is short, however, the latency and side effects of context switches, which include TLB flushing, saving/restoring registers, etc., may hurt performance substantially. The time spent on histogram updating can be directly measured by a kernel timer. The rest of the overhead is dominated by context switches. Figure 11 reveals the composition of the two overhead sources for each benchmark program. For `186.crafty`, `Banking`, and `Ecommerce`, the total overhead is merely 0.2, 0.026, and 0.005 second, respectively, which is comparable to the resolution of the timer we use, so the result is imprecise. The overhead of the other benchmarks is at least 100 times of the precision of the timer. Interestingly, for DaCapo and `Support`, the overhead is dominated by context switches; for SPEC INT, histogram updating causes 10% to 40% of total overhead depending on the locality of the programs. We use PIN instrumentation tool (Luk et al. 2005) to record all memory accesses and plot the actual miss ratio curves for the SPEC INT programs in Figure 12. As it shows, the miss ratio curves of `181.mcf`, `175.vpr`, and `197.parser` descend slower than the others. MEB, therefore, spends more time on locating the nodes in the LRU list for these three benchmarks.

4.4 Performance

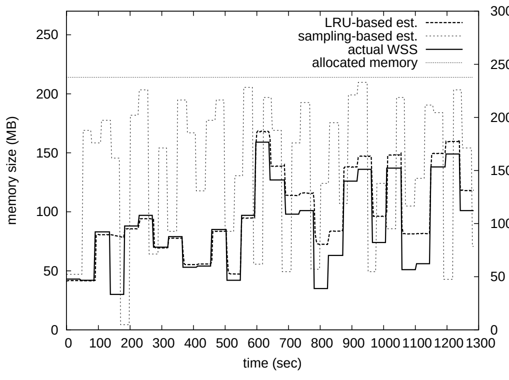
To evaluate the effect of automatic memory balancing, we first assign different tasks on two VMs, then experiments are performed on four VMs. Each VM is started with 214 MB initial memory and configured with 80 MB as its lowest possible memory allocation. Hence, a VM's memory may vary from 80 MB to $214 + (N - 1) * (214 - 80)$ MB where N is the number of VMs.

4.4.1 CPU Intensive + Memory Intensive Workloads

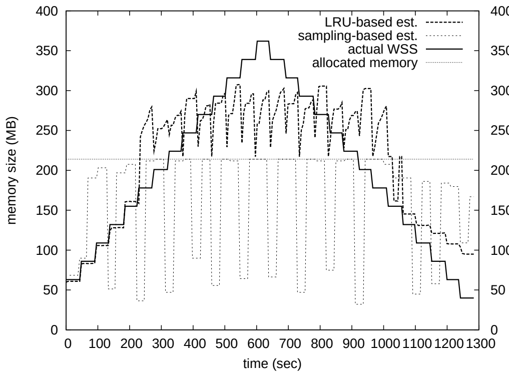
Our evaluation starts with a simple scenario where memory resource contention is rare. The workloads include the DaCapo benchmark suite and `186.crafty`, a program with intensive CPU usage but low memory load. On VM$_1$, `186.crafty` runs 12 iterations followed by the DaCapo benchmark suite. Meanwhile,



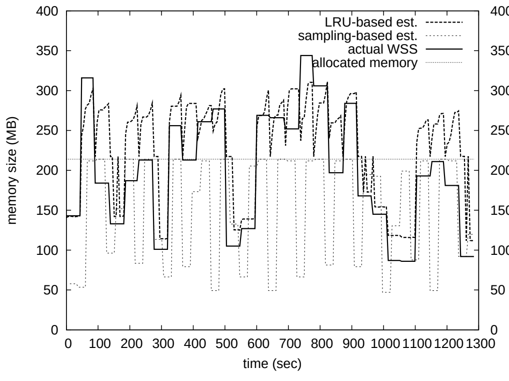
(a) `mono` (40 MB to 170 MB)



(b) `random` (40 MB to 170 MB)

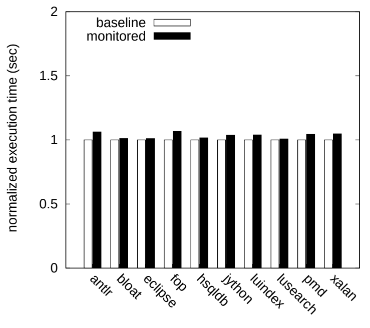


(c) `mono` (40 MB to 350 MB)

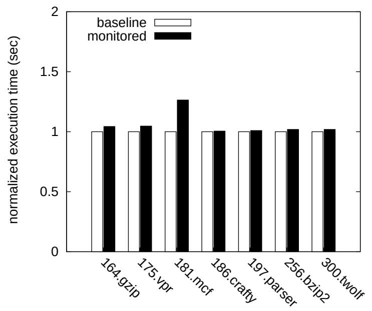


(d) `random` (40 MB to 350 MB)

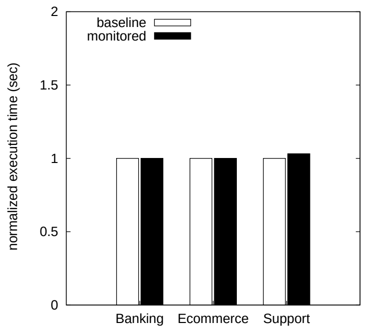
Figure 9. WSS Estimation



(a) DaCapo



(b) SPEC INT 2000

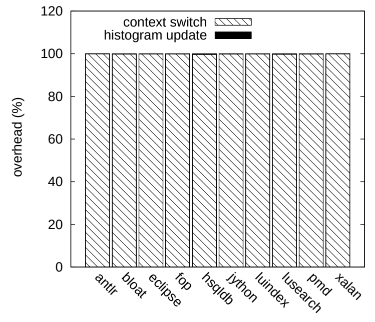


(c) SPEC WEB 2005

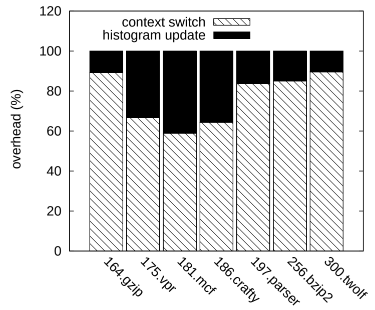
Figure 10. Overhead

on VM_2, the DaCapo suite runs first, followed by the same number of iterations of `186.crafty`. Figure 13(a) 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 generously allocated and the allocations to the two VMs meet.

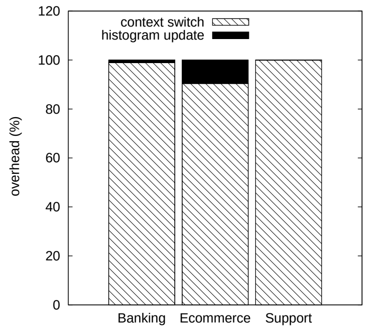
To show the performance that an ideal memory balancer could deliver, we measure the *best case* performance on two VMs, each with 428 MB fixed memory. In other words, it resembles the sce-



(a) DaCapo



(b) SPEC INT 2000



(c) SPEC WEB 2005

Figure 11. Overhead Breakdown

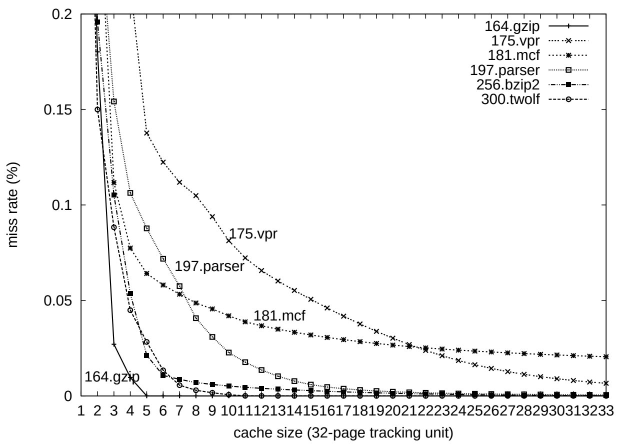


Figure 12. Miss Ratio Curve for SPEC INT

	Baseline	Balancing	Best
Major Page Faults	40,301	1,293	793
Exec. Time (DaCapo)	546.89	212.16	154.48
Exec. Time (186)	43.46	44.15	43.42

Table 1. Major Page Faults and Execution Time

nario that one VM uses up all available host memory. Figure 13(b) shows the execution time of each benchmark in the three settings respectively: baseline, best case, and balancing. As indicated by Table 1, with memory balancing, the number of total major page faults is reduced by a factor of 31.2, getting very close to the best case. Though there is 1.6% performance loss for `186.crafty` because of the monitoring overhead, DaCapo gains a 2.58 speedup. Most notable improvements are from `eclipse` and `xalan`, whose execution time is cut into $1/3$ and $1/6$ respectively. Both benchmarks require around 300 MB memory, resulting in a large number of page faults without MEB. Memory balancing leads to a 99.84% and 99.99% page fault reduction, respectively for `eclipse` and `xalan`.

Figure 14 shows the normalized performance when one VM runs DaCapo and the other one runs SPEC Web. With balancing, though the performance of `Ecommerce` degrades by 5.87%, DaCapo on VM_1 gains a 2.77 speedup, `Banking` on VM_2 shows an improvement of 7%, and the performance of `Support` is almost unaffected.

4.4.2 Memory Intensive + Memory Intensive Workloads

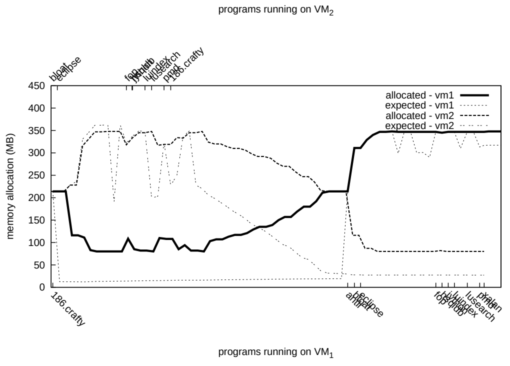
More challenging cases for memory balancing are the ones with frequent memory contention. We run the DaCapo benchmark suite on two VMs at the same time but in different orders: VM_1 runs each benchmark in alphabetical order while VM_2 runs them in the reverse order (denoted as DaCapo'). Note that `eclipse` and `xalan` require about 300 MB memory and `eclipse` takes about half of the total execution time. When the execution of two occurrences of `eclipse` overlaps, memory resource contention happens.

Figure 15(a) shows the memory allocation during the execution, and Figures15(b) and 15(c) compare the execution time with and without memory balancing. The goal of memory balancing is to improve the overall performance. Some benchmarks on one VM may sacrifice their performance to benefit the other ones. Specifically, the execution times of `eclipse` on VM_2 and `xalan` on both VMs are significantly reduced. Slowdown can happen for some benchmarks such as `eclipse` on VM_1. After applying balancing, the total number of major page faults is decreased by 45.1% and 12.9% respectively and the total execution time is shortened by 23.93% and 19.03% accordingly.

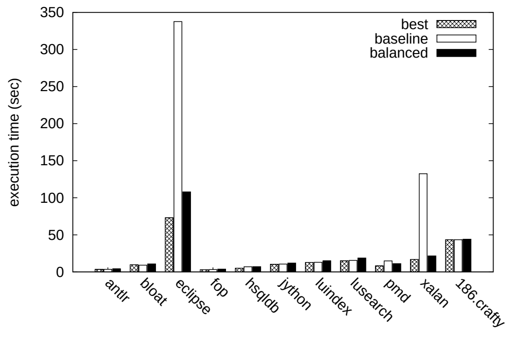
4.4.3 Mixed Workloads with Four VMs

To simulate a more realistic setting in which multiple VMs are hosted and diverse applications are deployed, four VMs are created and different workloads are assigned to each of them. VM_1 runs the DaCapo suite, VM_2 runs the DaCapo suite in reverse order (as in Section 4.4.2), VM_3 runs `186.crafty` for 12 iterations, and VM_4 runs SPEC Web. As shown in Figure 16, with memory balancing, the performance of DaCapo and DaCapo' are boosted by a factor of 4.32 and 3.25, respectively, with the cost of a 6% and 36% slowdown for `Ecommerce` and `Support`, respectively, on VM_4. Compared with the two-VM scenarios, the performance of DaCapo in the four-VM setting is significantly enhanced due to larger memory resource pool which allows the arbitrator to allocate more memory to these memory-hog programs.

Although the results are impressive in terms of the overall performance metric, QoS might be desirable in real applications for a performance-critical VM. A naive solution to guarantee the performance is to increase its lower bound on memory size. Alternatively, by assigning more weight or higher priority to the VM during arbitration, similar effects should be acquired. In this case, to improve the performance of SPEC Web HTTP services on VM_4, we

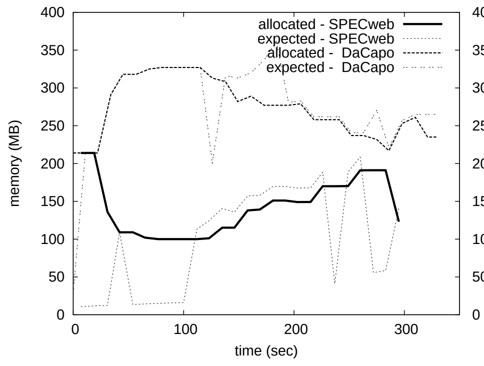


(a) Memory Allocation

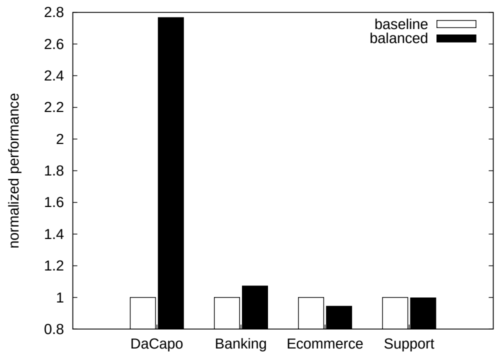


(b) Performance Comparison

Figure 13. DaCapo + `186.crafty`

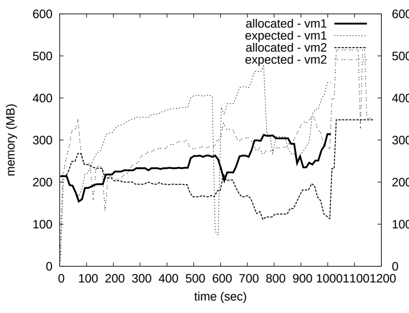


(a) Memory Loads on Both VMs

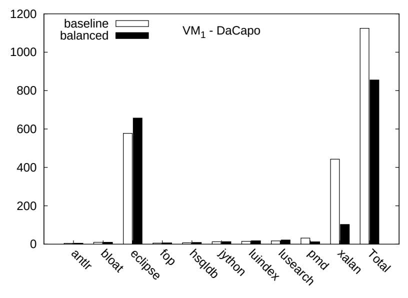


(b) Performance Comparison

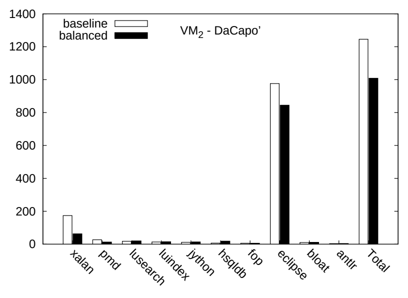
Figure 14. DaCapo + SPEC Web



(a) Memory Loads on Both VMs



(b) Execution Time (VM_1)



(c) Execution Time (VM_2)

Figure 15. DaCapo + DaCapo'

set the weight of the miss penalty of VM_4 as 20 times of the rest of the VMs. As displayed in Figure 16, after priority adjustment, `Ecommerce` and `Support` gain a 6% and 1% speedup, respectively, while the performance improvement of DaCapo is still maintained.

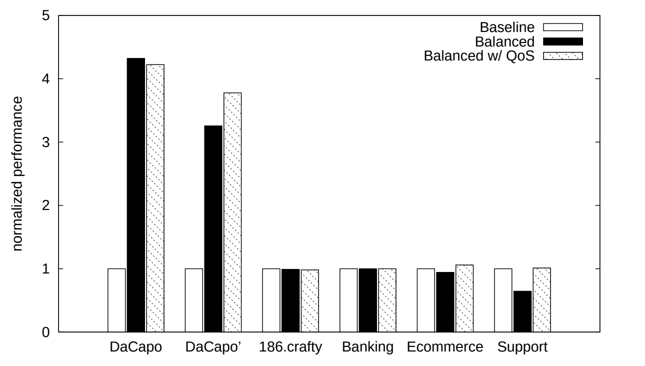


Figure 16. DaCapo + DaCaPo' + 186 + Web

5. Conclusion and Future Work

As virtualization technology gains more ground in both academics and industry, resource management in a virtualized system remains a key problem. Memory contention among virtual machines can significantly drop the overall system throughput. In this paper, we present MEB, a memory balancer that estimates the working set size of a VM and automatically adjusts the memory allocation to improve memory resource utilization. An evaluation of our Xen-based implementation shows that MEB is capable of boosting the overall performance through automatic memory balancing. In future work, we plan to extend MEB to support adaptive parameter selection to further reduce overhead while improving prediction accuracy.

Acknowledgments

We owe thanks to Steve Carr, Xiaolin Wang, and anonymous reviewers for their comments on this work. This work is supported by NSF Career CCF-0643664, NSF CCF-0811427, NSF CCF-0833082, National Grand Fundamental Research 973 Program of China under grant no. 2007CB310900, MOE-Intel Information Technology Foundation under grant no. MOE-INTEL-08-09, National High Technology Research 863 Program of China under grant no. 2008AA01Z112, NSFC-90718028, and NSFC-60873052.

References

Jikes RVM. URL `http://www.jikesrvm.org/`.

SPEC CPU2000, a. URL `http://www.spec.org/cpu2000`.

SPECweb2005, b. URL `http://www.spec.org/web2005`.

Bowen Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

Bowen Alpern et al. Implementing Jalapeño in Java. In *OOPSLA '99: Proceedings of the 14th annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 314–324, Denver, CO, November 1999.

V. Application. Intel 64 and IA-32 architecture software developer's manual, 2006. URL `citeseer.ist.psu.edu/484264.html`.

M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA '00: Proceedings of the 15th annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, October 2000.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1165389.945462.

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. doi: http://dx.doi.org/10.1109/HPCA.2005.27.

Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. *SIGOPS Oper. Syst. Rev.*, 40(5):14–24, 2006. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1168917.1168861.

Pin Lu and Kai Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–15, Berkeley, CA, USA, 2007. USENIX Association. ISBN 999-8888-77-6.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-056-6. doi: http://doi.acm.org/10.1145/1065010.1065034.

Dan Magenheimer. Memory overcommit... without the commitment, 2008.

R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 24–35, May 1993.

Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/844128.844146.

Timothy Wood, Prashant Shenoy, and Arun. Black-box and gray-box strategies for virtual machine migration. pages 229–242. URL `http://www.usenix.org/events/nsdi07/tech/wood.html`.

T. Yang, E. Berger, M. Hertz, S. Kaplan, and J. Moss. Automatic heap sizing: Taking real memory into account, 2004. URL `citeseer.ist.psu.edu/article/yang04automatic.html`.

Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. CRAMM: virtual memory support for garbage-collected applications. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 103–116, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 177–188, New York, NY, USA, 2004. ACM. ISBN 1-58113-804-0. doi: http://doi.acm.org/10.1145/1024393.1024415.