Michigan Technological University
Digital Commons @ Michigan Tech

Dissertations, Master's Theses and Master's Reports - Open

Dissertations, Master's Theses and Master's Reports

2011

Memory resource balancing for virtualized computing

Weiming Zhao
Michigan Technological University

Follow this and additional works at: https://digitalcommons.mtu.edu/etds

Recommended Citation

Follow this and additional works at: https://digitalcommons.mtu.edu/etds

MEMORY RESOURCE BALANCING FOR VIRTUALIZED COMPUTING

By

WEIMING ZHAO

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Science)

MICHIGAN TECHNOLOGICAL UNIVERSITY

2011

This dissertation, "Memory Resource Balancing for Virtualized Computing," is hereby approved in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE.

Department of Computer Science

Signatures:

Dissertation Advisor ————————————————————
Dr. Zhenlin Wang

Committee Member ————————————————————
Dr. Steven Carr

Committee Member ————————————————————
Dr. Steven Seidel

Committee Member ————————————————————
Dr. Jindong Tan

Department Chair ————————————————————
Dr. Steven Carr

Date ————————————————————

To my parents and my wife

Contents

List of Figures

List of Tables

Acknowledgments

Foremost, I owe my deepest gratitude to my advisor, Dr. Zhenlin Wang for the continuous research guidance, for his patience, motivation, enthusiasm, immense knowledge in my Ph.D. study and research. His guidance helped me in all the time of research and writing of this dissertation. I appreciate all his contributions of time, ideas and funding to make my Ph.D. experience productive and stimulating. I could not have imaged having a better advisor and mentor for my Ph.D. study.

My sincere thanks also go to Dr. David Poplawski, for his invaluable guidance and encouragement on my two-year's teaching experience. Before I came to Michigan Tech, I had no idea at all about how to lecture a course. I still remember how nervous I was when I first stood at the classroom. But in the next year, I built the confidence and even started to enjoy the lecturing, which would not happen without Dr. Poplawski's support and mentoring.

I also would like to thank the rest of my thesis committee: Dr. Steven Carr, Dr. Steven Seidel and Dr. Jingdong Tan, for their kind help and valuable comments on this work.

Last but not the least, I would like to thank my parents for giving birth to me at the first place and supporting me spiritually throughout my life.

Abstract

Virtualization has become a common abstraction layer in modern data centers. By multiplexing hardware resources into multiple virtual machines (VMs) and thus enabling several operating systems to run on the same physical platform simultaneously, it can effectively reduce power consumption and building size or improve security by isolating VMs.

In a virtualized system, memory resource management plays a critical role in achieving high resource utilization and performance. Insufficient memory allocation to a VM will degrade its performance dramatically. On the contrary, over-allocation causes waste of memory resources. Meanwhile, a VM's memory demand may vary significantly. As a result, effective memory resource management calls for a dynamic memory balancer, which, ideally, can adjust memory allocation in a timely manner for each VM based on their current memory demand and thus achieve the best memory utilization and the optimal overall performance.

In order to estimate the memory demand of each VM and to arbitrate possible memory resource contention, a widely proposed approach is to construct an LRU-based miss ratio curve (MRC), which provides not only the current working set size (WSS) but also the correlation between performance and the target memory allocation size. Unfortunately, the cost of constructing an MRC is nontrivial. In this dissertation, we first present a low overhead LRU-based memory demand tracking scheme, which includes three orthogonal optimizations: AVL-based LRU organization, dynamic hot set sizing and intermittent memory tracking. Our evaluation results show that, for the whole SPEC CPU 2006 benchmark suite, after applying the three optimizing techniques, the mean overhead of MRC construction is lowered from 173% to only 2%.

Based on current WSS, we then predict its trend in the near future and take different strategies for different prediction results. When there is a sufficient amount of physical memory on the host, it locally balances its memory resource for the VMs. Once the local memory resource is insufficient and the memory pressure is predicted to sustain for a sufficiently long time, a relatively expensive solution, VM live migration, is used to move one or more VMs from the hot host to other host(s). Finally, for transient memory pressure, a remote cache is used to alleviate the temporary performance penalty. Our experimental results show that this design achieves 49% center-wide speedup.

Chapter 1

Introduction

Virtualization is becoming pervasive in massive data centers, cloud computing, and enterprise infrastructure, driven by a number of important benefits, such as dramatic cost reduction, increased application availability and more efficient IT administration. According to Gartner (16), today, 25% of installed server workloads are virtualized. IDC even forecasts that, by 2014, more than 70% of applications on newly shipped servers will run in virtual machines (29). However, in a virtualized environment, efficient and effective memory resource management is still a challenging problem. In this dissertation we propose a memory resource balancing scheme to improve performance and memory resource utilization for center-wide virtualized computing. We demonstrate that our solution can accurately monitor memory demand of each virtual machine with very low overhead and can effectively improve overall system performance.

Virtualization technologies like Xen(9), VMware(54), and Denali(56) have become a common abstraction layer in modern data centers. They enable multiple operating systems to run on their own virtual machines independently. Figure 1.1 shows an example, where the hypervisor multiplexes the hardware of a single physical machine with several virtual machines and a guest OS runs inside each virtual machine independently. One of the main advantages of using virtualization is server consolidation. It is not uncommon to achieve a 15-to-1 or even higher *consolidation ratio* (11), which is the ratio of virtual to physical machine without disruptive performance impact. For a data center that hosts a large number of servers, this can effectively save power consumption, floor space occupancy and air conditioning costs. Besides, virtualization can improve availability by live migration (15). When one physical server fails or needs maintenance, the virtual machines it hosts can be transparently migrated to another physical machine with negligible application downtime.

The core of virtualization is the *virtual machine monitor* (VMM), which is also called *hypervisor*. VMM is responsible for creating and managing multiple instances of virtual hardware platforms. A lot of physical resources like CPUs or network interface cards can
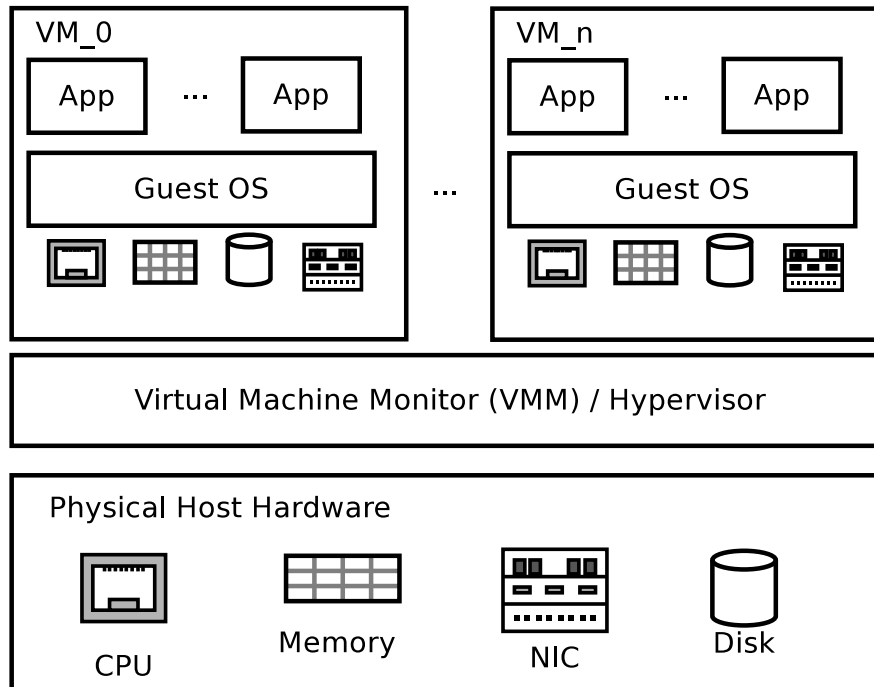
Figure 1.1: Organization of virtualization

be multiplexed in a time-sharing manner, which is similar to how multiple processes of a native OS would share them. However, the memory system is shared through address space partitioning. That is, each virtual machine is allocated with a fixed amount of address space of physical memory. However, differing from how a native OS manages virtual memory and physical memory for its processes, for the purpose of fidelity, the VMM is not actively involved in memory management of each VM. More specifically, when created, each VM is allocated with a fixed amount of physical memory. Then, it is the guest OS's responsibility to manage that amount of physical memory without the involvement of the hypervisor. As a result, the hypervisor is unaware of memory demand of VMs and thus unable to dynamically balance memory resources.

In our solution, we first design a low cost but accurate LRU-based working set size tracking scheme as the foundation of memory resource balancing. The LRU-based working set size model correlates memory allocation size and performance impact. Based on the model, we design a local memory balancing scheme, which dynamically adjusts memory allocation amount via ballooning (54, 9) on a single physical machine. Then it is extended to a global environment, where the physical memory of all interconnected machines is balanced via live migration and remote caching. To the best of our knowledge, our work uniquely coordinates the global memory balancing techniques with a local balancing scheme. Without effective local memory balancing, the effectiveness of global memory balancing will be significantly weakened. Figure 1.2 shows the overview of our solution.
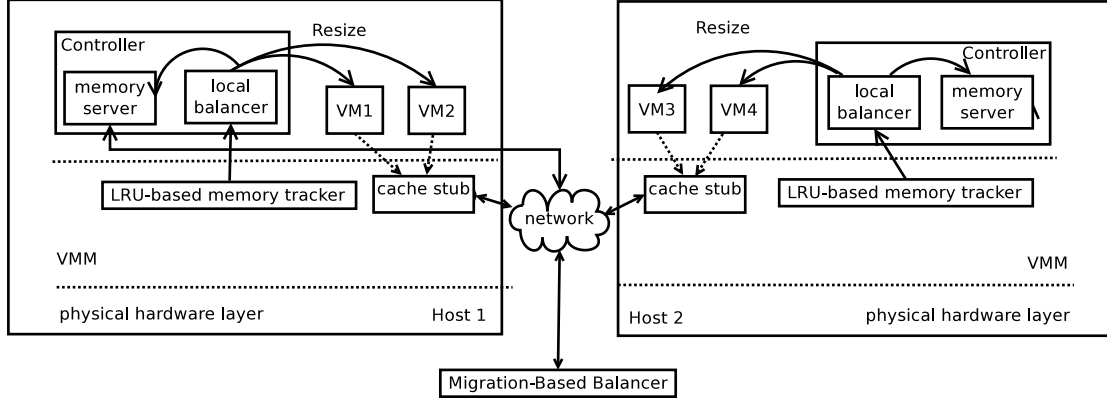
Figure 1.2: Solution overview

1.1 Our Low Cost Working Set Size Tracking Method

A widely proposed method to guide memory resource allocation is the LRU-based miss ratio curve (MRC) (38, 12, 67, 60, 30, 51) that plots memory access miss ratios versus various physical memory amount. When the memory resources of a system are well balanced, the number of its page fault occurrences is minimal and thus it achieves optimal performance. Unfortunately, the cost of maintaining an MRC is nontrivial. For example, our experiments show that, a straightforward implementation of the MRC brings a mean overhead of 173% for SPEC CPU 2006 benchmark suite.

To minimize the overhead, we introduce three optimizations (65, 66):

1. An AVL-tree-based implementation of LRU list, denoted as *ABL*. Typically, an LRU list is implemented by a linked list. However, it requires a linear search to find a node's *LRU distance*, which is the distance of the node from the head of the list to its current position. In our implementation, nodes are organized into an AVL-tree in such a way that a pre-order traversal of the tree gives the same sequence as given by traversing the linked list. With the help of an auxiliary field in each node, we can find a node's LRU distance by traversing from the current node to the root of the tree, which takes $O(\log N)$ time instead of the $O(N)$ time of using linked list, where N is the memory allocation size of the monitored VM.

2. Dynamic hot set sizing, denoted as *DHS*. *Hot set* is the set of pages that are most recently accessed and intercepted. In order to construct the MRC, a sufficient number of memory accesses need to be intercepted to acquire the addresses of their access targets. To avoid excessive interceptions that may cause significant overhead, after each interception, the accessed page is put into the hot set. Subsequent accesses to the pages in the hot set will not be intercepted until they are evicted from the hot

set. When a hot set is full, the oldest page is evicted. Hence, the larger the size of a hot set, the less the memory accesses are intercepted and vice versa. However, increasing the hot set also lowers the accuracy of the MRC. Our design is able to dynamically adjust hot set size to balance between overhead and accuracy. In some cases, if the hot set size exceeds the working set size, it substantially overestimates the memory demand. Our design can also detect such pathological cases and then adjust the estimation result.

3. Intermittent memory tracking, denoted as *IMT*. Most programs exhibit phasing behaviors in terms of IPC, branch prediction miss rates, and memory demand, etc. That is, the metrics are stable within a period while there exist disruptive transitions between the periods. Exploiting this attribute, we design an intermittent tracking scheme, which is able to turn off memory tracking when the monitored VM enters a stable phase and re-enable it when a new phase is encountered. Experimental results show that, by using this intermittent tracking design, memory tracking can be turned off for 82% of the execution time while accuracy loss is no more than 4%.

The three optimization techniques are orthogonal to each other. When combined together, the mean overhead is lowered to only 2%.

1.2   Local Memory Resource Balancing

Based on the working set size tracking scheme, we design a local memory resource balancer, which dynamically adjusts physical memory allocation of all VMs of a single physical machine to improve overall performance (64). By analyzing a VM's miss ratio curve, its performance impact with respect to various memory allocation sizes can be estimated. This provides the necessary information for arbitration when multiple VMs compete for memory resources. Though there exists an allocation plan that brings the best overall performance, finding an optimal solution requires brute force searching of nearly $O(M^V)$ time, where M is the total allocable memory size and V is the number of VMs to be balanced. We propose a quick heuristic based algorithm, which can find the near optimal solution in $O(M)$ time.

Experimental results demonstrate that, on a host with 2 VMs, when both VMs run a CPU intensive workload and a memory intensive workload in an interleaved way, our local memory balancer reduces the total number of page faults by a factor of 25. As a result, the performance of the memory intensive workload is boosted by 11 and 8 times, respectively, while the execution time of CPU intensive workload is nearly unaffected. Even in a complicated case, in which 4 VMs with mixed workloads compete for memory during the execution, it still achieves an overall speedup of 1.72.

1.3 Global Memory Resource Balancing

Although local memory balancing improves memory utilization of a single host, from the perspective of the whole data center, memory resource imbalance may still exist. It is quite often that some hosts suffer from memory overloading while others have idle memory, which calls for a global memory balancer. On the other hand, local memory balancing reclaims free memory, which is the base of global memory balancing.

We propose two mechanisms to balance center-wide global memory resources: live migration based balancing and remote cache based balancing, which is close to the idea of Williams *et al.* (57). A major distinction between them is whether free memory is dynamically reclaimed or not. In the work of Williams *et al.*, the memory used for global balancing is statically reserved, while in our work, the local memory balancer dynamically reclaims idle memory and utilizes them for global balancing. As a result, our scheme can achieve higher memory resource utilization. In addition, the decision making algorithm of migration is different too. More details will be discussed in Section 2.5.

1.3.1 Live Migration Based Global Balancing

Using live virtual machine migration (15), VMs can be moved from one host to other hosts with negligible service downtime, given that all VMs share network storage systems, which is commonly seen in a data center. This allows us to move VMs from the memory overloaded host to others with sufficient free memory and thus improve global memory utilization. Unfortunately, migration is an expensive operation. Thus, we design a heuristic based algorithm which initiates migration only if the memory pressure is predicted to sustain for enough long time.

1.3.2 Remote Cache Based Global Balancing

At the other end of the spectrum is burst memory pressure. Even if a host has enough free memory, the local balancer may not response so quicly to completely eliminate page swapping. Based on the fact that for page level data transferring, the latency of a 1 Gbps Ethernet is about 1 to 2 orders of magnitude lower than disk I/O (14), we design a remote cache to convert relatively slow disk I/Os to faster network transfer. As shown in Figure 1.2, disk I/Os are intercepted by VMM and then the requests are sent to a remote host. The remote host runs a sizable memory server, which works as a disk cache. The local memory balancer on the remote host manages the capacity of its cache to ensure that only free memory will be used to serve other hosts.

1.4   Dissertation Organization

The rest of this dissertation is organized as follows. In Chapter 2, we briefly cover relevant background knowledge and discuss related work.

In Chapter 3, our low cost LRU-based working set size tracking scheme is presented. We then describe our implementation and experimental results.

In Chapter 4, we describe how we use the tracked working set size information to balance memory resources for all VMs on a single host. We show that our local balancing can significantly boost overall performance.

In Chapter 5, we first design a migration based global balancing mechanism to solve long term memory imbalance. We then present a remote cache based global balancing scheme to mitigate the performance penalty when spikes of memory demand occur or when migration is impossible.

We conclude this dissertation in Chapter 6 by discussing the limitations and possible future work.

1.5   Summary of Contributions

We make the following contributions in this dissertation:

1. A novel low cost LRU-based WSS estimation scheme which employs a more efficient data structure and exploits hardware performance counters to construct LRU-based miss ratio curve and accurately estimate the current WSS of a VM.

2. A memory balancing mechanism to dynamically adjust the amount of each VM's memory allocation to improve performance.

3. An algorithm to predict memory trend and decide when to apply VM migration to improve global memory resource utilization.

4. A design of remote cache that uses free memory of a remote host to alleviate the impacts of a burst of memory requirement.

We implement all the design on an open source hypervisor and demonstrate that our WSS estimator can effectively and efficiently track working set size. And the local balancer is able to boost performance. Even when there exists significant memory contention, the

overall performance can be improved. And the experimental results also reveal that our global memory balancer can further improve system performance.

Chapter 2

Background And Related Work

This chapter discusses background materials and related work. It first describes the most pertinent background knowledge about how memory is managed under virtualization and discusses various approaches to estimate memory demand. It then discusses virtual machine migration and existing studies about remote memory, the two schemes that can be applied to global memory balancing, followed by extensive research on memory balancing for virtual machines. Finally, the background and related work about program phases are discussed.

2.1 Memory Management

In this section, we first briefly introduce how memory is managed in a native operating system. Then we describe how memory management is adapted for virtualization and the challenges that virtualization brings.

2.1.1 Memory Management In A Native OS

Virtual memory was first described in 1960s (21) and it has become a standard feature of modern general-purpose operating systems. It gives each process an illusion that it is running on a standalone and contiguous memory address space, called *virtual address space*. The size of a virtual address space can be larger than the amount of available real, physical memory. Typically, memory is allocated and reclaimed at a fixed granularity, called *page*, whose size is determined by processor architecture. Both virtual address space and physical address space are measured at the unit of page size. Inside the operating system, it maintains a translation table for each process, named *page table*, which maps a

9

virtual page number to a physical page frame number. Inside a processor, there is a *memory management unit* (MMU) that dynamically translates virtual addresses that an application references to the corresponding physical addresses by looking up the current page table.

When a system runs out of physical memory, and if some process requests for a free page, the OS selects some physical frame, saves its contents on a secondary storage (called *swapping-out* or *paging-out*) and allocates the page to the process. Later on, if the previously paged out data is needed, it is loaded into a free physical memory frame, which is called *paging-in* or *swapping-in*. This page swapping scheme is called *demand-paging*. The strategy that determines which pages are swapped out is called *page replacement policy*.

Since the latency of disk accesses is usually thousands of times longer than that of memory accesses, frequent page swapping will significantly damage the performance. A theoretically optimal page replacement policy should select a page whose next use will be the farthest in the future (2). However, in a general purpose OS, it is impossible to precisely predict the future memory access behaviors. In real world, a commonly used algorithm is the *least recently used (LRU)* replacement policy or its approximation. It works based on the property of program locality. That is, the pages that have been heavily and recently used are also most likely to be heavily used in near future. For an OS that uses demand-paging and the LRU page replacement policy, a process that heavily uses a set of pages will automatically secure those pages in physical memory.

2.1.2   Memory Management With Virtualization

To create a virtualized environment, the hypervisor runs at the most privileged level that a native OS runs at. For CPUs without virtualization support, guest OSes and its processes run at the non-privileged level. Those privileged operations in the guest OS, such as page table setup, I/O instructions and etc. are either statically replaced with calls to the hypervisor or dynamically trapped and emulated via binary rewriting by the hypervisor. With hardware virtualization support, an unmodified guest OS runs on the CPU directly without intervention by the VMM until it tries to execute a restricted instruction. At this point, the hypervisor takes the control to emulate the instruction.

When an operating system runs on the top of hypervisor, another level of memory address space, *guest physical address (GPA)* space, is introduced. To avoid confusion, in virtualization, the real physical address is specifically referred as *machine physical address (MPA)*. GPA is used by guest OSes in their physical address space. The purpose of using GPA is to provide the guest OS the impression that it is running on a real machine with certain amount of contiguous physical memory starting from address 0 because there is no guarantee that every guest OS will be allocated with contiguous machine memory and most OSes

Virutal Address Space
(Process 1)

Machine Address
Space

Guest Physical
Address
Space

Virutal Address Space
(Process 1)

Physical
Address
Space

Virutal Address Space
(Process 2)

Virtual Disk

Guest OS

Disk

(a) A native OS   (b) A virtualized OS
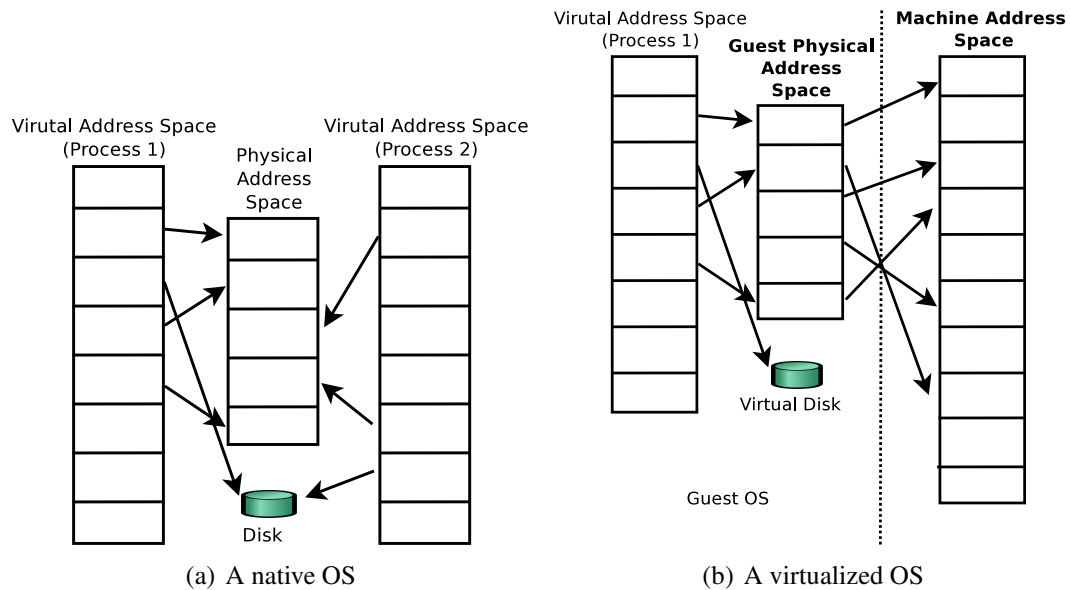
Figure 2.1: Address spaces in a native OS and a virtualized OS

do not support fragmented physical address space. The size of the GPA space of an OS is the same as the amount of machine memory that the hypervisor statically allocates to the guest OS when it is created. Meanwhile, the hypervisor maintains a mapping table to map the contiguous GPA space to the possibly scattered MPA space. Figure 2.1 contrasts the address spaces between a native OS and a virtualized OS.

To support the translation from GPA to MPA, currently there are two kinds of approaches based on whether the processor supports MMU virtualization or not (1, 53, 39): two software-based techniques and a hardware-based approach.

2.1.2.1 Software-Based Techniques

Without hardware assistance to automatically translate GPA to MPA, two software-based approaches, *paravirtualization* and *shadow page table* are used. Since the processor only supports VA to MPA translation, the hypervisor has to translate GPA to MPA and setup the page tables.

Using paravirtualization, the guest OS is modified to make explicit requests to its hypervisor for page table updating. For example, when the guest OS attempts to map a virtual address v to its guest physical address g, instead of directly updating its page table, it delegates the operation to its hypervisor. When the hypervisor receives the request, it first finds the corresponding machine address m for g by looking up its GPA-to-MPA table, and then it fills up page table entry of v with m. As a result, when the guest OS accesses v, the
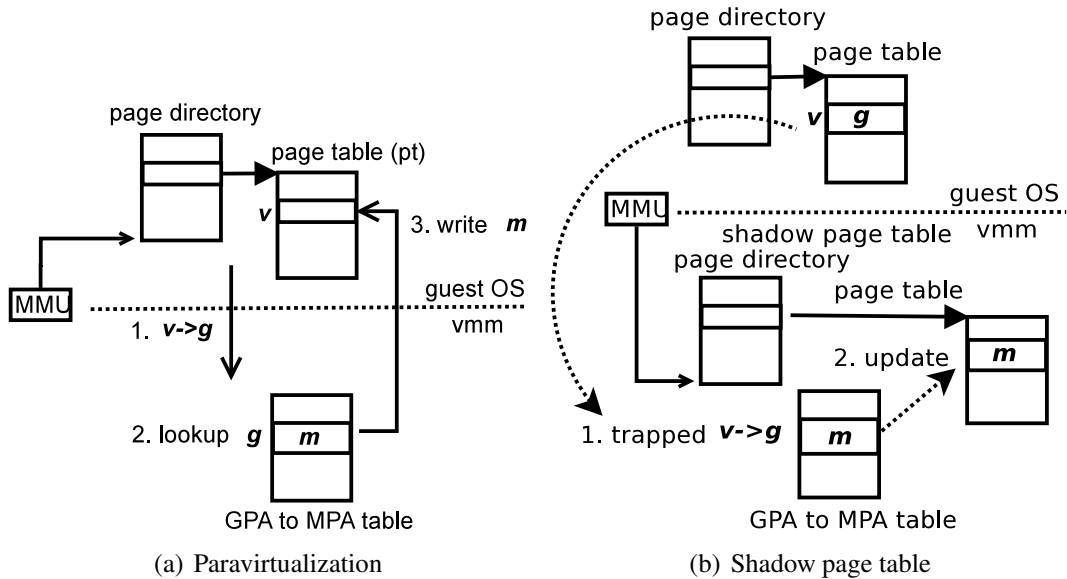
Figure 2.2: Examples of address translation

The two figures show the operations when the guest OS attempts map a virtual address *v* to a guest physical address *p* that corresponds to a machine address *m*, using paravirtualization and shadow page table, respectively

MMU will map it to *m*. Figure 2.2(a) illustrates the operations. This approach is adopted by Xen (9). It is highly efficient but requires some modifications to the guest OS kernel.

An alternative technique is the shadow page table, which is adopted by VMware ESX(54). In this scheme, there exist two sets of page tables, one is used by the guest OS for VA to GPA mapping, and the other one is maintained by hypervisor and used by MMU, named *shadow page tables*, that maps VA to MPA directly. The shadow page tables are invisible to guest OSes while the page tables used by guest OSes are not used by the processor. With hardware assistance or binary code rewriting, all page table creations or updates from guest OSes will be trapped by the hypervisor and it will then update its shadow page tables accordingly. For instance, as Figure 2.2(b) shows, when a guest OS maps a virtual address *v* to a guest physical address *g* by updating its page table, this operation is intercepted by the hypervisor. The hypervisor then finds the machine address *m* for *g* by looking up its GPA-to-MPA table and sets up a mapping of *v* to *m* in the corresponding shadow page table. Since only the shadow page table is exposed to the MMU, an access to *v* will be mapped to the machine address *m*. On x86/x86-64 platform, both the first generation virtualization technologies from Intel and AMD, named VT-x(53) and AMD-V(6), respectively, support the interception of page table updating but are lack of hardware assisted MMU virtualization.

12

page directory

page table (pt)

v | g

MMU .... guest OS

vmm

g | m

GPA to MPA table
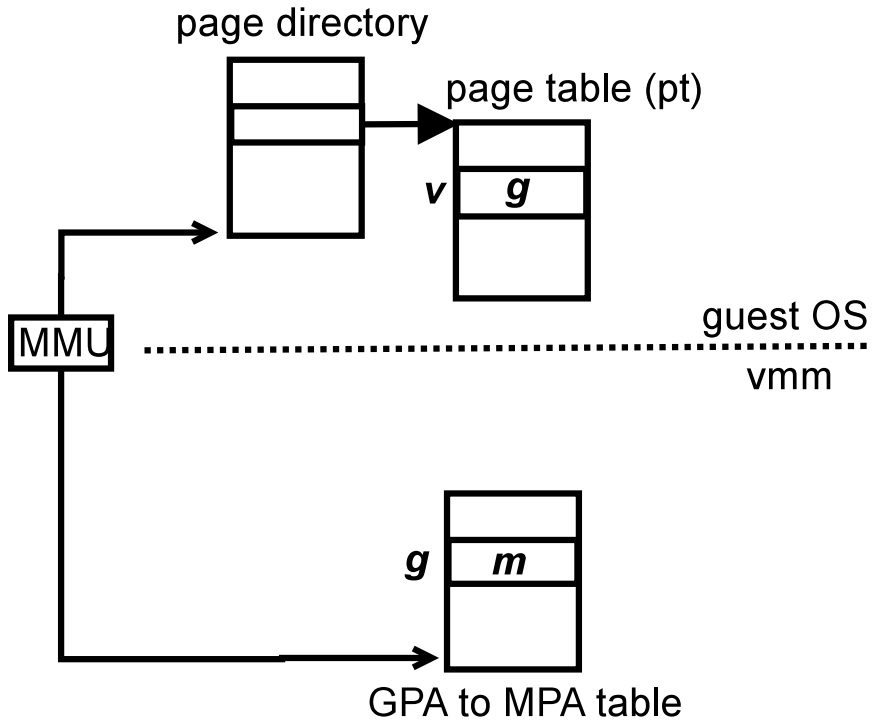
Figure 2.3: Address mapping with MMU virtualization

2.1.2.2 Hardware-Based Technique

In the second generation virtualization technology, Intel and AMD introduce *Extended Page Table* and *Rapid Virtualization Indexing*, respectively. Both of them support MMU virtualization, with which, both the page tables in the guest OS and the GPA-to-MPA mapping table are exposed to MMU, as illustrated by Figure 2.3. Given a VA, the hardware will first index the page tables used by the guest OS to get the GPA, and then index the set of page tables provided by the hypervisor to map the GPA to MPA.

Note that, no matter which technique is used, the hypervisor is only responsible for assisting the mapping from GPA to MPA. It does not involve in memory management, such as page allocation, reclaiming or swapping. Therefore, the hypervisor has little knowledge about its VMs' memory utilization.

2.1.3 Dynamic VM Memory Allocation Resizing

The memory allocation size of a VM is determined when it is created. Resizing physical memory allocation of a guest OS in runtime is similar to changing physical memory

13

amount to a native OS without rebooting, which is nontrivial and not well supported by most OSes. Though memory hot plugging, a mechanism that allows plugging/unplugging DIMMs physically, has been proposed (24, 42), it is not widely adopted by most OSes. Even it is supported by some OSes, it is not suitable for VM memory resizing because of its high cost and coarse granularity. To perform a DIMM unplugging, the OS has to migrate pages to gather those scattered free pages into a contiguous, DIMM-aligned address space, which is an expensive operation. Besides, hot plugging/unplugging usually works on a large granularity.

Waldspurger proposes the *ballooning* (54) mechanism, which has been widely adopted for VM memory resizing. The "balloon" is installed into the guest OS as a kernel space driver. To decrease an OS's memory allocation by n pages, the hypervisor instructs the balloon to "inflate" by n pages. The driver then applies for n pages from the guest OS. From the guest OS's perspective, those pages are owned by the driver. But actually, those pages are reclaimed by the hypervisor and can be allocated to other VMs. Similarly, releasing memory from the ballooning driver restores the memory allocation for a VM. As the balloon is completed deflated, the allocation amount of the guest OS is just the same as it sees during the boot time, which is the upper bound of its resizing range. And the lower bound of the resizing range is determined by how many pages the balloon driver can apply for. Theoretically, all user-owned pages can be reclaimed by swapping them out but it may result in system unstability.

One advantage of the ballooning mechanism is that it takes advantage of a guest OS's knowledge about which pages are the most suitable for reclaiming.

2.2 Working Set Size Estimation

In this section, we first introduce the concept of working set. We then discuss various techniques to estimate the size of a working set.

2.2.1 Working Set

Denning (17) first defined the working set as the set of memory pages referenced by a process during a time interval. The size of the working set (WSS) is the amount of memory that a process needs without paging. Even if the process has very large memory footprint, those pages not in its working set can be reclaimed without performance penalty. The same idea can be extended to VMs. Assuming a guest OSes can well utilize their allocated physical memory, and if the memory allocation amount of each VM is exactly its WSS, then the physical memory allocation on the host is optimal. Therefore, working set size estimation

provides a necessary metric that ensures the system to reach the maximum performance as expected.

2.2.2 Miss Ratio Curve Based Working Set Size Estimation

The miss ratio curved (MRC) based WSS estimation is a widely proposed technique (38, 13, 67, 60, 30, 51). A page miss ratio curve plots the page miss ratios against various amount of memory allocation. When the allocation size is no less than a system's WSS, the miss ratio is 0, which means all accesses will hit in main memory. When the allocation size is less than its WSS, the MRC tells the ratio of many accesses that will cause page swapping. 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. For example, when two applications compete for memory resources, in order to achieve optimal overall performance, the arbitration scheme needs to evaluate how the performance would be impacted by varying their allocation sizes

A commonly used method to calculate MRC is Mattson's stack algorithm (38). It was initially proposed to reduce the time of trace-driven cache simulation. The algorithm uses an LRU stack to store the page numbers of accessed page frames. For each entry of the LRU stack, its distance to the top of the stack is called *stack distance* or *LRU distance*. Each stack entry i is associated with a counter, denoted as $Hist(i)$. When a page is referenced, the algorithm first searches the page number in the stack and computes it stack distance, $dist$. It then increments the hit counter $Hist(dist)$ by one. Finally, it updates the stack by moving the page number to the top of the stack. One can plot an *LRU histogram* by relating each counter value to its corresponding LRU distance.

If there is a stack with depth D and we reduce it to depth d, then the expected miss ratio can be calculated as follows:

$$Miss\_ratio(d) = \frac{\sum_{i>d}^{D} Hist(i)}{\sum_{i=0}^{D} Hist(i)}$$

For example, given a system with only four pages of physical memory, the top half of Figure 2.4 shows the hit counts for some application that makes a total of 200 memory accesses. The histogram indicates that 100 accesses hit the most recently used (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
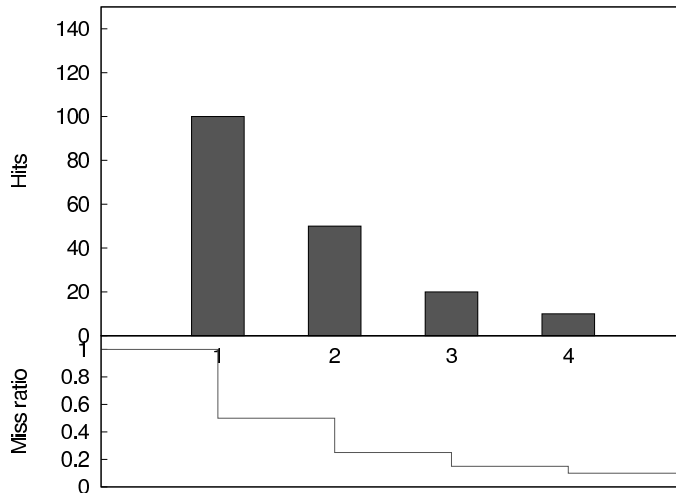
Figure 2.4: LRU histogram example

by a half, the hits to the first two MRU slots are still there while the hits to the next two MRU 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 2.4 is the miss ratio curve corresponding to the histogram on the top.

2.2.2.1 MRC Tracking

Since the Matterson's algorithm was originally used for off-line analysis, to track the MRC at runtime, it needs to collect sufficient amount of memory access addresses.

Zhou *et al.* (67) propose two approaches to capture memory accesses and construct page MRC for an application. The first is a hardware approach, where an MRC monitor is connected to the memory bus to track all memory accesses. For efficiency, pages are grouped into multiple page groups and pages in the same group are assumed to have the same stack distance. Thus, MRC is constructed at the granularity of page group. The MRC monitor consists of three array-like components: an array that servers as the LRU stack where each entry is doubly linked , a hit counter array, and a group header array that keeps the heads of page groups. Since the MRC monitor snoops physical memory accesses without capturing context switching, it tracks the MRC of the entire system. Though this approach has little runtime overhead, it is unavailable on current commodity processors. The second approach is called *OS approach*, which is implemented in an OS and requires no extra hardware support. Though the components of the MRC monitor can be easily implemented in software, it is challenging to acquire enough memory accesses. To solve this problem, it partitions the physical pages into two groups: frequently accessed pages and infrequently accessed pages. For the first group, it scans the "accessed" bit of page tables. For the second group, it uses page protection techniques to detect references to those pages. That is, the OS re-

vokes access permission for them. As a result, an access to such a page will be trapped as a protection fault. One difference between the OS approach and the hardware approach is that, in the former one, it constructs MRC based on virtual page numbers, while in the later one, MRC is constructed based on physical page numbers.

Based on the OS approach of Zhou *et al.*, 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(60, 59). In their work, the MRC implementation is improved by attaching an AVL tree to the page list to accelerate LRU distance computing. Specifically, each leaf node of the tree points to up to k consecutive pages of the linked list of size N and each non-leaf node of the tree records the total number of pages of its subtree. Therefore, there are $\lceil \frac{N}{k} \rceil$ number of leaf nodes. To find a page's LRU distance, it first searches linearly to find the page's position within the group, which takes $O(k)$ time. It then walks up the AVL tree and counts the number of pages that are positioned before the accessed page, which takes $O(\log \lceil \frac{N}{k} \rceil)$ time. When a page of LRU distance i is accessed, it needs to be moved to the head of the linked list. However, if the leaf nodes that correspond to the first $i-1$ pages of the linked list are already full, after the moving, those leaf nodes need to be updated to guarantee that each of them points to no more than k pages. Otherwise, the leaf node that points to the first page group will contain $k+1$ pages. The number of leaf nodes to be updated is $O(\lceil \frac{N}{k} \rceil)$. As a result, the total time of each MRC update is $(O(k) + O(\log \lceil \frac{N}{k} \rceil) + O(\lceil \frac{N}{k} \rceil))$. Therefore, given a small k, the total time cost for MRC updating is bounded by $O(\lceil \frac{N}{k} \rceil)$, though when locality is good the overhead is close to $O(\log N)$. In this dissertation, we further improve the LRU list by completely replacing the linked list with AVL tree and achieves an $O(\log N)$ time complexity.

Instead of using the page protection based technique to intercept memory accesses, the hypervisor exclusive cache (34) intercepts memory accesses from a VM by capturing its disk I/Os. It is designed as a memory balancing mechanism for virtual machines. In this scheme, each VM gets a small amount of machine memory, called *direct memory*, and the rest of the memory is managed by the hypervisor in the form of exclusive cache. Because of the small amount of memory allocation, the guest OS will use page swapping heavily. Those disk I/Os are intercepted by the hypervisor and if they hit the exclusive cache, they are actually redirected to native memory operations in the form of cache admissions or cache hits. With the information of disk I/O requests, it can infer the memory accesses, construct an MRC for each VM and use it to estimate WSS for each VM. By adjusting the cache size of a VM, it achieves the effect of physical memory balancing. However, this design 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 a hypervisor and complicates VM migration. In addition, it requires modification to the guest OS to notify the hypervisor of page release.

Besides the memory access interception techniques, another approach to monitor page accesses is by periodically checking if a page is accessed or not. On most processors, when a

page is accessed, the hardware will automatically set the "access bit" in the corresponding page table entry (28). By periodically checking and scanning the access bit of a process's page table, the access frequency of each page of the process can be estimated and used to construct MRC. However, sequentially scanning the whole page table is expensive. To accelerate this process, Zhang *et al.* (63) present a "locality jumping" scheme that utilizes the spatial locality of a program to skip checking many non-accessed pages. This scheme is proposed for an OS to improve its cache utilization through tracking the MRC of each process and applying page coloring. However, in a virtualized environment, setting and clearing the access bit in hypervisor may disturb a guest OS's memory management because its page replacement policy usually relies on the access bits to infer page usage.

RapidMRC (52) collects memory accesses by using the sampling functions available in the performance monitoring units (PMU) of PowerPC processor and builds an MRC for an L2 data cache. The PMU can be configured to record the address of a memory access that matches predefined criterion (e.g. L2 cache miss) into a *sampled data address register*. By periodically sampling the register, it can collect enough information of memory accesses to build an MRC. However, on x86/x86-64 platform, not all processors support recording the addresses of memory accesses. For example, the PMUs on processors prior to the latest Nehalem microarchitecture only record the architectural states of the processor (i.e. state of the general purpose registers, the flag register and the instruction pointer register). Only on Nehalem microarchitecture, the PMU is enhanced with load latency information that contains the memory address for delayed load operations that are randomly selected by hardware (28).

2.2.3   Other Techniques For Working Set Size Estimation

In addition to the MRC-based WSS estimation techniques, many other approaches have been proposed.

VMware ESX server adopts a sampling strategy (54). During a sampling interval, accesses to a set of random pages are monitored. By the end of sampling period, the page utilization of the set is used as an approximation of global memory utilization. The merit of this technique is the very low cost. However, it lacks the model between allocation size and performance impact.

Magenheimer (36) 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 working set size because it includes the infrequently used areas that can be reclaimed without a notable performance penalty.

Geiger (30), proposed by Jones *et al.*, 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. For example, when a page is read from disk, if the memory page is associated with a different disk location, then the previous content of page is assumed to be evicted. It uses this information to simulate an exclusive cache and to construct an MRC, which is similar to Lu *et al.*'s exclusive cache (34) except that it does not actually provide caching. When a guest OS's memory is overloaded, the MRC tells how much extra memory is needed. However, it is unable to detect over-allocated memory because, in this case, there is no page eviction.

2.3   Virtual Machine Migration

In virtualization, transferring a VM from its current host to another is called *VM migration*. When the source and target hosts have different available memory resources, VM migration can be used for memory load balancing though it is a relatively heavyweight solution.

Typically, migration requires both the source and destination hosts reside in the same network and share centralized storage (like NAS or SAN), thus only memory states need to be synchronized. Two metrics are commonly used to evaluate the time cost of migration: *downtime* and *total migration time*. The former one is the period during which the VM is frozen and its services are temporarily unavailable in order to complete state transfer. The latter one is the period from the initiation to the completion of the whole migration process, during which performance degradation may occur due to state synchronization. If the downtime is sufficiently low (e.g. lower than the timeout thresholds of network protocols), it can be classified as *live migration*. That is, from user's perspective, service unavailability is unnoticeable during the migration.

A straightforward *stop-and-copy* (44) migration scheme, which first halts the source VM, then copies all pages to the destination host and finally resumes service on the new VM, has the minimum total migration time, but its downtime is equal to its total migration time, which is usually unacceptable for live services. At the other end of the spectrum is the demand-driven *copy-on-reference* migration scheme(61). It first uses a very short *stop-and-copy* phase to send essential data, then the new VM is started. The rest of the pages are transferred when they are first used. Though it has very short downtime, its total migration time is very long.

Clark *et al.* propose a balanced approach (15) for live migration, called *pre-copy migration*, which copies memory pages in multiple rounds from the source host to the destination host. When migration starts, it iteratively copies the pages that are dirtied during the previous iteration. Meanwhile, the VM is still running on the source host. During the migration, all pages that have been copied are write-protected by setting the corresponding page table

field, thus any polluted pages will be logged by the hypervisor. Eventually, after certain rounds of copying, if the number of all dirty pages is small enough or the number of pre-copying iterations reaches a preset threshold, a stop-and-copy phase finishes the migration. It has low downtime and its total migration time depends on the page dirty rate of the guest OS. For example, in our evaluation, for a VM with 1200 MB memory, when it runs `186.crafty`, a program with low memory activity, the migration time is only 13 seconds. However, for the same VM, when it runs `SPEC JBB`, a program with much more memory writes, the migration time is 64 seconds.

2.4   Remote Memory

Due to the mechanic nature of hard disk drives (hdd), the performance of hdd is mainly limited by the seek time and rotational delay, which are both on the order of milliseconds. For example, for a high-end 15000 rpm HP SAS Enterprise drive, the average latency is 2.58 ms (27). Compared with disk I/Os, the high speed networking has much lower latency. For example, Chen *et al.* demonstrate that, when continuously requesting a 4KB data block, the latency of a Gigabit Ethernet is about 100 microseconds, which is more than 100 times longer than that of native memory copying but it is still 2 orders of magnitude lower than the 10,000-microsecond latency of a disk (14). The gap between native memory copying and network transferring is even smaller with faster networking technologies such as InfiniBand, Myrinet and Quadrics that provides up to 10 Gbps throughput.

As motivated by the outstanding performance of networking against hard drives, much research proposes to utilize remote memory to improve performance for a cluster environment.

Markatos *et al.* implement a *reliable remote memory pager* (37) that uses remote main memory for paging. In this design, the client forwards the paging requests to a remote server over Ethernet network. Servers are user level programs that listen to a socket and accept connections. Clients are implemented as a block device that redirects swapping operations to the servers. A client can distribute its swap space allocation requests to multiple servers. Meanwhile, a machine that acts as memory server can service multiple clients by instantiating a separate server instance for each client. In order to prevent data loss due to server crash, it introduces two RAID-like schemes: mirroring and parity-based redundancy. Using mirroring, the client sends a write request to two different servers. With parity-based scheme, pages to be swapped out are XORed and the parities are also transferred to servers.

Flouris *et al.* design the *Network RamDisk* (20), which is similar to the remote memory pager. It improves the performance of parity checking by using a parity cache. The experimental results show that the Network Ramdisk brings a four to eight fold speedup against the hard drive.

20

Newhall *et al.* design the *Nswap* (41), which supports limited dynamic growing and shrinking of the remote memory pool. The shrinking is performed by migrating some pages from the current memory server to other servers. Therefore, the total cache size in the whole network is unchanged.

Liang *et al.* present a *high performance network block device* (HPBD) (33), which runs over InfiniBand networking and utilizes its Remote Direct Memory Access (RDMA) operations for one sided communication. Its server design is a simple memory pool, without extra reliability policies. The evaluation results show that, the latency of transferring 4KB data is only a few microseconds, which is at the same magnitude as native memory transferring. As a result, HPBD is only 1.45 times slower than local memory operations, but is up to 21 times faster than the local disk.

To improve the communication efficiency, Werstein *et al.* propose a kernel-to-kernel network memory for page swapping (55). It uses a lightweight UDP-like communication channel over Ethernet and works within the kernel space.

In a virtualized environment, block devices like disks are virtualized to guest OSes. A special driver domain then redirects the disk I/Os from guest OSes to physical devices. Hence, there are two places for the client module. One is within the guest OS and acts as a kernel block device driver, the same way as the client module in a native OS. The other way is to postpone the redirection to the driver domain, just before it is actually sent to the physical disks. Hines *et al.* compare the two options in their MemX (26), a remote memory implementation for Xen virtual machines. It shows that the former option incurs slightly more overhead because the network traffic needs to traverse the extra virtual network of the guest OS.

Instead of using remote memory to replace a whole disk device, another approach is to use it as a cache between memory and disk. An important advantage of cache design is that it supports variable size. If the remote machine has less idle memory, it can shrink the capacity of its memory server. Using a write through cache design, it can simply discard some pages. For caches with write back policy, if the disks are network shared, the server can write the dirty pages back to the disk before discarding them. One example is REMOCA (14) that uses remote memory as an exclusive cache above disks for Xen virtual machines. For read requests, the client will look up its directory to check the availability of the pages. If the requested page is not in remote memory, it will read from local disk. For write requests, it adopts write through policy for better reliability. In order to achieve better cache utilization, it employs exclusive cache design. That is, after the client retrieves a page from the cache, the requested page is evicted by the cache. In order to cache a clean page evicted by the guest OS, it modifies the guest OS kernel to explicitly notify the eviction.

2.5 Other Techniques Of Memory Resource Balancing For Virtual Machines

Two migration-based VM load balancing schemes, Black-box and Gray-box strategies (58) are proposed by Wood *et al.*. Both of the strategies migrate hot-spot VMs to other hosts with enough resources, but differ in the way of collecting VM load information. The black-box approach is fully OS-agnostic, which inherits Geiger's disk I/O monitoring scheme to infer memory load and hence inherits its drawbacks too. The gray-box approach queries the performance statistics provided by each guest OS to estimate memory load, which is insufficient to support memory load balancing on a single host.

Williams *et al.* present overdriver(57), a mechanism that employs VM migration and network memory to balance memory resources for a data center. When a VM's memory overload is predicted to sustain for sufficiently long time, a migration-based solution will be used to alleviate memory pressure on that host. Otherwise, it will use physical memory on a remote host to mitigate the penalty of page swapping. In order to decide if the duration of memory overload is transient or not, it first constructs a workload profile for the VM that records the ratio of occurrences for each duration length of overloading. When memory overloading is detected, it uses the profile to get the ratio of overload of its current duration and compare the ratio against a threshold to decide if an overloading is transient or not. However, without local memory tracking and balancing, it is unable to effectively reclaim sufficient free memory to allow for migration, nor can it decide the appropriate size of the remote cache at runtime. Thus, waste of memory resources may still occur. Besides, its workload profile is only meaningful for a fixed memory allocation, which makes this scheme unsuitable for VMs with dynamically changed memory allocation. And in order to construct the profile, it needs to collect enough historic overloading events, which lowers its effectiveness during that time window.

2.6 Program Phases

Prior studies (18, 46) have shown that, an executing program usually exhibits phase behaviors. That is, for some metrics such as the number of instructions per cycle (IPC), branch prediction miss rates and memory access patterns, etc. are relatively stable within some temporal intervals while there exist abrupt changes in between. For example, Figure 2.5 shows the phases in terms of the WSS, the numbers of L1 cache references, L2 cache misses, data TLB misses and number of instructions executed of a benchmark program `429.mcf`.

A lot of approaches have been proposed to capture, characterize or even predict phase
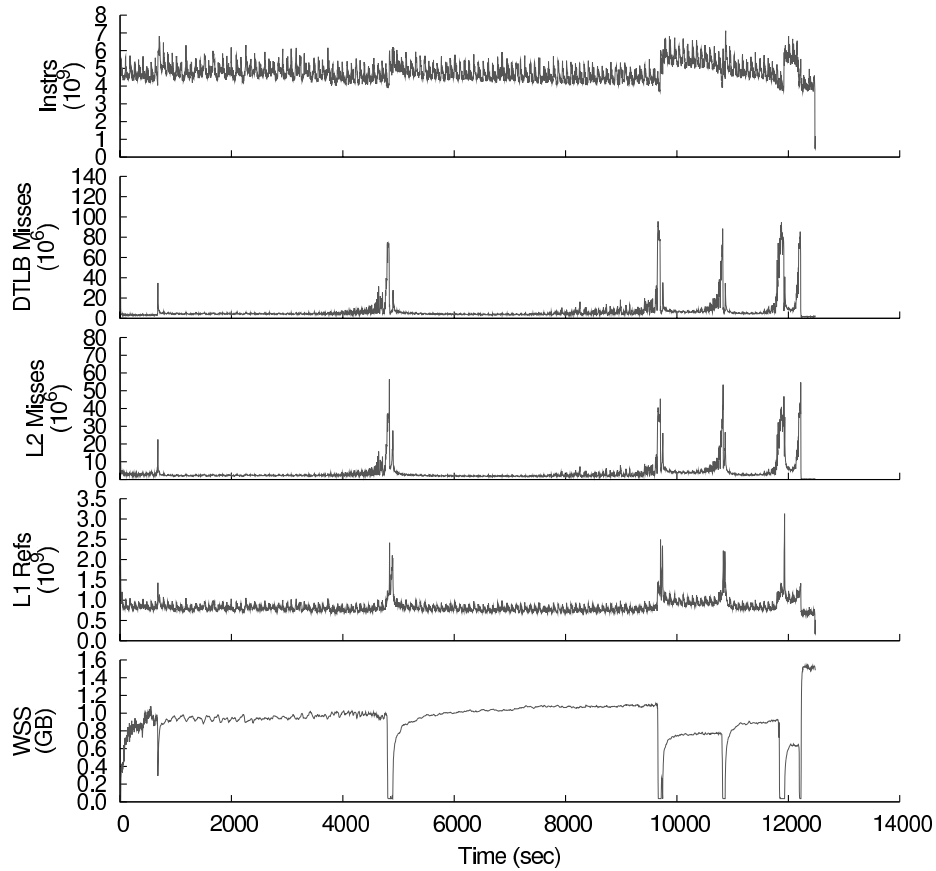
Figure 2.5: Phases of 429.mcf

behaviors. For example, Sherwood *et al.* (48) first use Fourier analysis upon 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 (49). Shen *et al.* (46) predict locality phases at runtime by first identifying the locality phases of a training input through offline analysis and then inserting phase marks into the binary code.In the offline analysis, it first samples representative reuse distances that are sufficiently long for a representative set of memory locations. It then uses discrete wavelet transformation as a filter to find abrupt changes in reuse distance for each tracked memory location. Finally, it marks phases by looking for basic blocks that only occur near the beginning of a phase and inserts a predictor that predicts phases based on the online phase history for other inputs.

2.6.1 Online Phase Detection

For accurate phase detection, prior approaches use offline profiling because it provides a whole view of program characters and allows for expensive analysis such as the wavelet analysis used in (46). However, in some cases, for example, to capture the phase behaviors of a virtual machine that may run any program instead of a particular one, online phase detection is desirable.

Dhodapkar *et al.* (19) show an online phase detection scheme that is used to guide the dynamic tuning for multiple configurable hardware units such as caches, TLBs, branch predictors, instruction widows, etc. for performance and/or energy. It first collects instruction working set, which is the set of instructions executed during a given interval. It uses the *relative working set distance* to compare the similarity between two sets of collected instructions. The relative working set distance is defined as the percentage of the number of non-common instructions between the two sets to the size of the union of the two. Then, phase changes are identified by comparing the relative working set distance against a fixed threshold. It requires hardware support to collect every committed instruction and computes the distance.

Nagpurkar *et al.* (40) summarize the framework for online phase detection. It includes two data windows: a *current window* for the most recently consumed data and a *trailing window* for the next most recently consumed data. Phase transitions are identified by comparing the similarity of the elements in the two windows. The detection algorithm consists of three policies:

1. A *window policy* specifies the size of the data window, the number of elements consumed at a time and whether the size of trailing window is variable.

2. A *model policy* defines how the similarity between the two data windows is computed. For example, it could be the ratio of elements in the current window that are also in the trailing window.

3. An *analyzer policy* eventually decides whether the similarity value is sufficiently large to indicate a stable phase or not. It proposes to compare the current similarity with a historic average value. If the current similarity is less than the average value by a given threshold, it assumes a phase transition.

In our design, we utilize phasing behavior to lower the overhead of WSS tracking. When the target system is in a stable phase in term of WSS, we disable the WSS tracking. We then predict phase changes by monitoring memory related hardware events. Once a phase transition is predicted to occur, the WSS tracking is re-enabled. For efficient online phase detection, we use a moving window filter for de-nosing and a threshold based mechanism to identify phase changes.

24

Chapter 3

Low Cost Working Set Size Tracking

This chapter 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 technique to intercept memory accesses and construct LRU miss ratio curves proposed by Zhou *et al.* and Yang *et al.* (60, 67). However, the original idea targets the memory demand tracking for individual applications. In addition, a straightforward implementation of that idea incurs high overhead, especially for programs with large memory demand or poor locality. We first adapt the idea to VM-level WSS tracking and then solve the overhead problem by adopting three innovative optimizing techniques. Experimental results show that, the mean overhead for an extensive set of benchmark programs is lowered from 173% to only 2%.

In the following sections, we first present our basic design of WSS tracking in a hypervisor and analyze its overhead. We then introduce three optimizations: (1) AVL-tree based LRU list design, (2) dynamic hot set sizing and (3) intermittent memory tracking, to lower the overhead. Finally we discuss our experimental method and evaluation results.

3.1   LRU-Based Working Set Size Estimation

In order to estimate the working set size of a VM, it is necessary to infer its memory access behaviors. Although most memory accesses from guest OSes are transparent to the hypervisor, page table operations such as creation, modification, etc. are still visible to the hypervisor because it requires the hypervisor to map pseudo physical pages that guest OSes use to real machine pages. For a fully virtualized system with hardware assistance, page table updating will be trapped by the hypervisor via hardware exceptions. For a para-virtualized system, a guest OS has to make explicit calls, called *hypercalls*, to notify the hypervisor about the operations.

We modify the hypervisor such that once those requests are received, 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.

For process level WSS estimation, using tracked virtual addresses to construct LRU miss ratio curve 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 ambiguity problem because each process of the guest OS uses the same virtual address space. On the other hand, using tracked physical addresses limits the ability of estimation. 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 to guide 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. The swapping activities can be inferred by monitoring page table changes. Figure 3.1 illustrates the changes in a page table entry under different scenarios. When a guest OS decides to swap out a page to disk, the "presence" bit of its corresponding page table entry (PTE) is marked as "non-present" and its index field is changed from its physical frame number to the disk location. Later on, when the page is needed, a page fault occurs and the reverse operations are applied by the guest OS to load the disk content into the physical memory. Thus, by monitoring page table updates, the swapping behavior can be inferred. Using a disk location based MRC, the amount of memory growth can be estimated. Geiger (30) 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 obsoleted and thus overestimates the memory demand.

Alternatively, we can just take the swap space usage as the growth amount. On most modern OSes, 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, intuitively, compared with 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 it footprint on the swap space. Moreover, compared with constructing an MRC, the overhead of this OS statistics-based
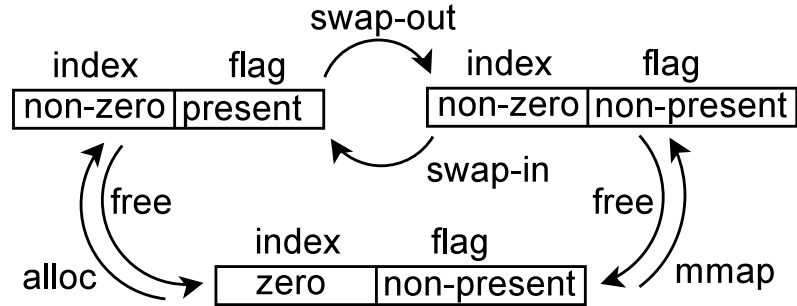
Figure 3.1: Transitions of PTE states

estimation is almost negligible and it reflects the swap usage decrease more rapidly than the LRU-based solution. In Section 3.5.2, we compare the two design alternatives.

3.1.1 Basic Design of LRU List

To track the LRU order of each physical memory frame, we maintain an ordered sequence of tags to reflect the recency of page accesses. It is quite natural to organize the tags into a linked list, where the tag at the head of the list corresponds to the most recently accessed page, while the tag at the tail of the list corresponds to the least recently accessed one.

Whenever a memory reference is trapped, three operations will be performed on the list: locating the corresponding tag in the LRU list, finding the LRU distance of the tag and moving the tag to the head of the list. Since for a given machine, its physical memory size is fixed, we can expedite the tag locating by pre-allocating all tags in a single array. Given a physical frame number i, its corresponding tag can found in $O(1)$ time: $A[i]$, where A denotes the array. For each guest OS, there is a head pointer that pointers to the first node of its own LRU list. Figure 3.2 gives an example when two VMs are monitored. To facilitate tag moving in LRU list, the list is doubly linked. In order to find the LRU distance of some tag, it requires a linear search, which is an $O(N)$ operation, where N is the size of the linked list. Hence, the cost of finding the LRU distance dominates the overhead of each update of LRU histogram.

Maintaining the LRU list 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. 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 $A[\frac{p}{G}]$. As a result, it not only reduces the length of the linked list, but also shortens the average search time in finding LRU distance.

Let M be the number of physical pages, the length of the LRU list is $N = \frac{M}{G}$. We call G the granularity of tracking.

27
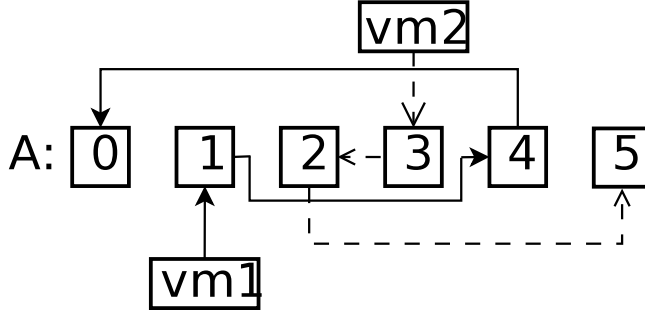
Figure 3.2: An example of LRU lists

(All tags are organized as an array, *A*. The solid lines link the LRU list of VM_1: $node_1$, $node_4$ and $node_0$; the dotted lines represent the list of VM_2: $node_3$, $node_2$ and $node_5$. For simplicity, a singly linked list is drawn.)

Though a coarser tracking granularity may lead to less accurate estimation, in Section 3.5.1, we show that when *G* is 32, the accuracy loss is negligible for the purpose of memory balancing.

3.1.2   Overhead Analysis

The overall time cost of the LRU-based memory demand tracking scheme can be represented as

$$interception\ number \times (page\ fault\ handling\ time + LRU\ updating\ time)$$

For programs with very good locality or very small WSS, most linear searches for LRU distance computing will hit near the head of the list. So the time spent on LRU updating is relatively small. For example, for `401.bzip2` and `416.gamess`, whose WSS is only 24 MB and 45 MB, respectively, their tracking overhead is only 3% and 1%, respectively. However, for programs with average or poor locality, the overhead is significant. For instance, for `429.mcf`, whose average WSS is 859 MB, it execution time is slowed down by 58 times when WSS tracking is applied. For the whole SPEC CPU 2006 benchmark suite, the mean overhead is 173%.

To lower the overhead, we can either decrease the number of memory interceptions or reduce the LRU updating time, or both. First, we design an AVL-tree-based LRU list, which lowers the cost for each LRU updating from $O(N)$ to $O(\log N)$, where *N* is the number of tags in an LRU list. Secondly, we use a dynamic hot set sizing design to decrease the number of interceptions. Thirdly, we present an intermittent memory tracking scheme, which can temporarily turn off the whole memory tracking system. In Section 3.2, 3.3 and 3.4, we elaborate these three optimizations, respectively. Eventually, we lower the

28

(a) Before   (b) After
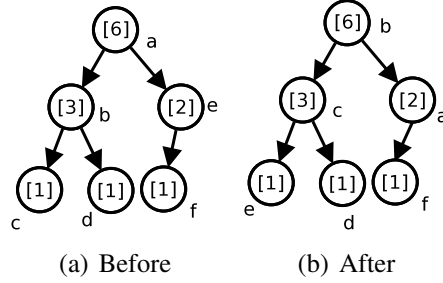
Figure 3.3: An example of AVL-based LRU list

(Before page e is visited, the LRU sequence is c,b,d,a,f,e. Then, e is moved to the left-most position, and then the tree is re-balanced. In-order traversal gives e,c,b,d,a,f.)

average overhead to 2% by combining the three optimizations.

3.2 AVL-Tree Based LRU Design

Though linked list is a common design to maintain the recency order of page accesses, the LRU sequence can also be logically organized as a binary tree. For any tag, if its left child has shorter LRU distance and its right child has longer LRU distance, then an in-order traversal of the tree gives the same sequence as given by a linked list.

To facilitate finding out the LRU distance of a tag, each tag has a field, *size*, which counts the size of the sub-tree rooted from itself. For example, in Figure 3.3, 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:

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

in which functions $size(x)$ and $LC(x)$ denote the size of the sub-tree rooted as x and x's left child, respectively. Function $ANC(x)$ returns either *nil* or the nearest ancestor of x such that x is in its right sub-tree. For example, in Figure 3.3(a), $ANC(c)$ is *nil* and $ANC(f)$ is a. Since function ANC walks up along the tree towards the root, for a balanced tree with N nodes, 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 left most leaf. During the insertion or removing, at most all the tag's ancestors' *size* fields need to be updated, which takes $O(\log N)$ time, the same cost upper bound for tree re-balancing. As a result, the overall time cost is lowered to $O(\log N)$), while the space cost is still $O(N)$.
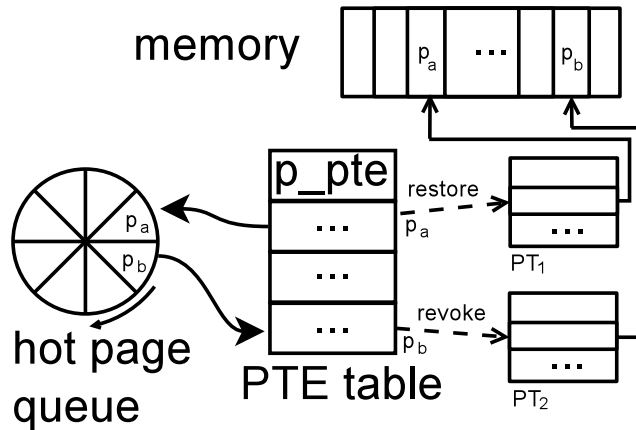
Figure 3.4: Hot set management. Suppose an access to physical page p_a is trapped. Then the permission in the corresponding PTE in page table PT_1 is restored and the address of the PTE is added to the PTE table. Next, p_a is enqueued and p_b is dequeued. Use p_b to index the PTE table and locate the PTE of page p_b. The permission in the PTE of page p_b is revoked and page p_b becomes *cold*.

3.3 Dynamic Hot Set Sizing

Trapping all memory accesses would incur prohibitive cost. Instead, 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 installed (e.g. when creating a new process), all pages are *cold* . Later on, 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.

A hot set is implemented as an FIFO queue with limited size. It stores physical frame numbers (PFNs). Once the queue is full, the page referred to by the head entry is dequeued and made cold again by revoking its access permission in the corresponding 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 3.4 explains the whole process with an example.

The hot set design can significantly reduce the number of page interceptions because of program locality. Typically, a large portion of recent accesses go to a small set of pages. However, for programs with poor locality, in order to have the same interception number, it requires a larger hot set. Yang *et al.* adaptively change the size of hot set by monitoring the overhead (60). The size of the hot set is increased until the overhead is decreased below a preset threshold. We instead directly predict the locality from the shape of its miss ratio

curve and then adjust the hot set size accordingly. 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, the better the locality. In our 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 maximum size.

Usually, since the hot set size is far less than the WSS and minimum memory allocation size of a VM, the accuracy loss of WSS estimation is allowable. However, we observe that there exist some unusual cases where the working set size is less than the hot set size. In these cases, as most memory accesses fall into the hot set, the LRU histogram appear more flat and cause an overestimate of memory requirement.

One example is 171.*swim*. We use a precise instrumentation tool that is based on PIN (35) to record every memory access of this program and then simulate the process of our LRU monitoring with different hot set sizes. Figure 3.5 illustrates the miss rates (in solid lines with respect to the left axes) and accumulated hits (in dashed lines with respect to the right axes) for two hot sizes respectively. As Figure 3.5(a) shows, the predicted miss rates are above 5% until memory size reaches around 71000 pages, which suggests a memory requirement of 277 MB. However, when the hot set size is 0, from Figure 3.5(b), we can observe that 12 pages of memory cover more than 95% memory accesses. And by comparing the accumulated hits, we can see that when the hot set size is 12, the VMM only intercepts about 0.1% memory accesses. To avoid this problem, our solution is to use the number of data TLB misses as an approximation of the number of memory page accesses. We observe that, normally, the number of the TLB misses, T_m, is no more than 2 orders of magnitude larger than the number of intercepted memory accesses, I. If $T_m >> I$, it implies that there are a great number of memory accesses that are not intercepted by the system and thus fall into the hot set. As long as such situation occurs, we can either reduce the hot set size or correct the estimation result.

3.4   Intermittent Memory Tracking

Most programs show typical phasing behavior in terms of memory demands. Within a phase, the working set size remains nearly constant. This inspired us to temporarily disable memory tracking when the monitored program enters a stable phase and re-enable it when a new phase is encountered. Through this approach, the overhead can be substantially lowered. However, when memory tracking is off, the memory tracking mechanism itself is unable to detect phase transitions anymore. Hence, an alternative phase detection method is required to wake up memory tracking when it predicts a phase change.
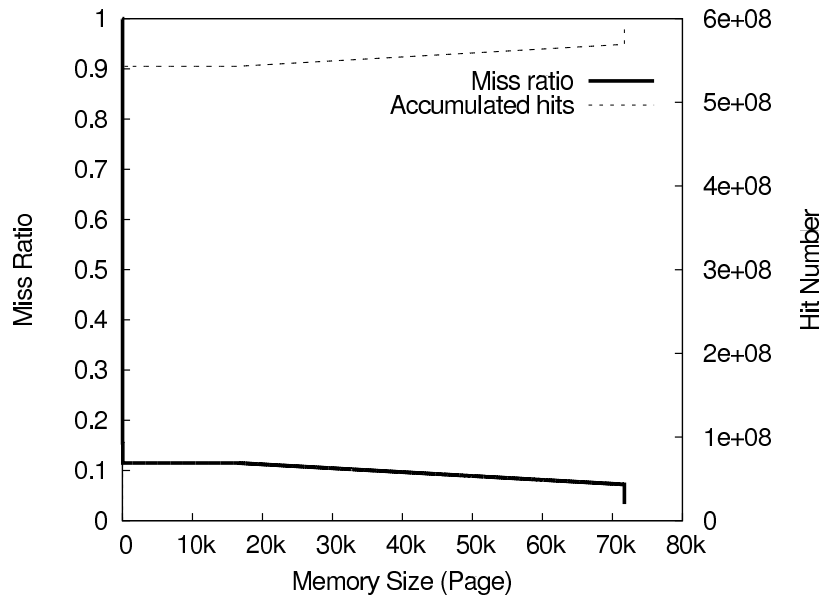
(a) Hot set size is 12
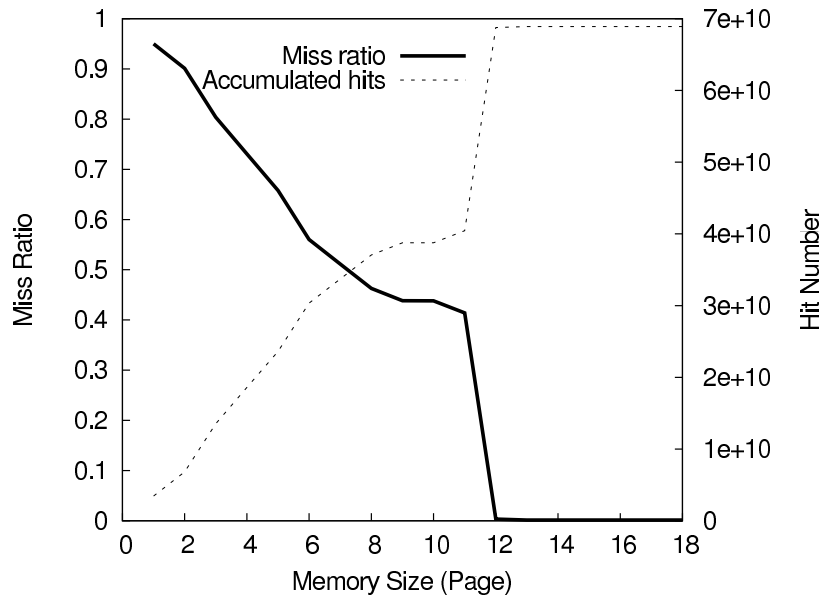
(b) Hot set size is 0

Figure 3.5: Miss ratio curve of *171.swim*

We find that a sudden change of working set size tends be accompanied by sudden changes of the occurrences of memory-related hardware events like TLB misses, L2 misses, and L1 accesses, etc. And when the working set size remains stable, the number of occurrences of those events is relatively stable as well. Occurrences of these events can be monitored through special registers built into most modern processors and be accessed with negligible overhead. However, a key challenge is to effectively differentiate phase changes from

random fluctuations. The remainder of this section will focus on our solutions to this problem.

3.4.1 Selection of Events

There exist numerous memory-related hardware events in modern processors, such as L1/L2 accesses/misses, TLB accesses/misses, and so on. Since a change of working set size implies a change in memory access pattern at page level, it suggests that data TLB (DTLB) misses would be a good candidate. L1 accesses and L2 misses may also be candidates for detecting phase changes of memory demands. For example, in Figure 3.6, it shows the correlation between WSS and the occurrences of three events, L1 accesses, L2 misses and data TLB misses for four SPEC CPU 2006 programs. In these examples, the three events are all closely correlated with WSS.

In addition, since some processors support counting multiple events simultaneously, a phase change could be determined by multiples events and different policies. An *aggressive policy* would determine a phase change only if all the events encounter phase changes. This policy minimizes the time of memory tracking. On the contrary, a *conservative policy* is sensitive to any possible phase changes. That is, if any of the events show phase changes, it will turn on memory tracking. This policy maximizes the tracking accuracy. Beside, a moderate *voting policy* decides a phase change only when the majority of the events show phase changes. Without extensive experiments, it is difficult to conclude which events and policy are the most appropriate ones. In Section 3.5.5.2, we compare the results of using different events and policies.

3.4.2 Phase Detection

Previous studies rely on sophisticated signal processing techniques such as Fourier transformation or wavelet analysis (46, 47) to detect cache-level phases. Though these techniques are able to effectively filter out noises and identify phase changes in off-line analysis, their prohibitive 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 for signal denoising. Let v_i denote the sampled value (memory demand or the number of occurrences of some hardware event) during ith time interval. We pick $f(i) = (v_i + v_{i-1} + \ldots + v_{i-k+1})/k$ 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, it means there is not enough information to make any decisions. So memory tracking is always
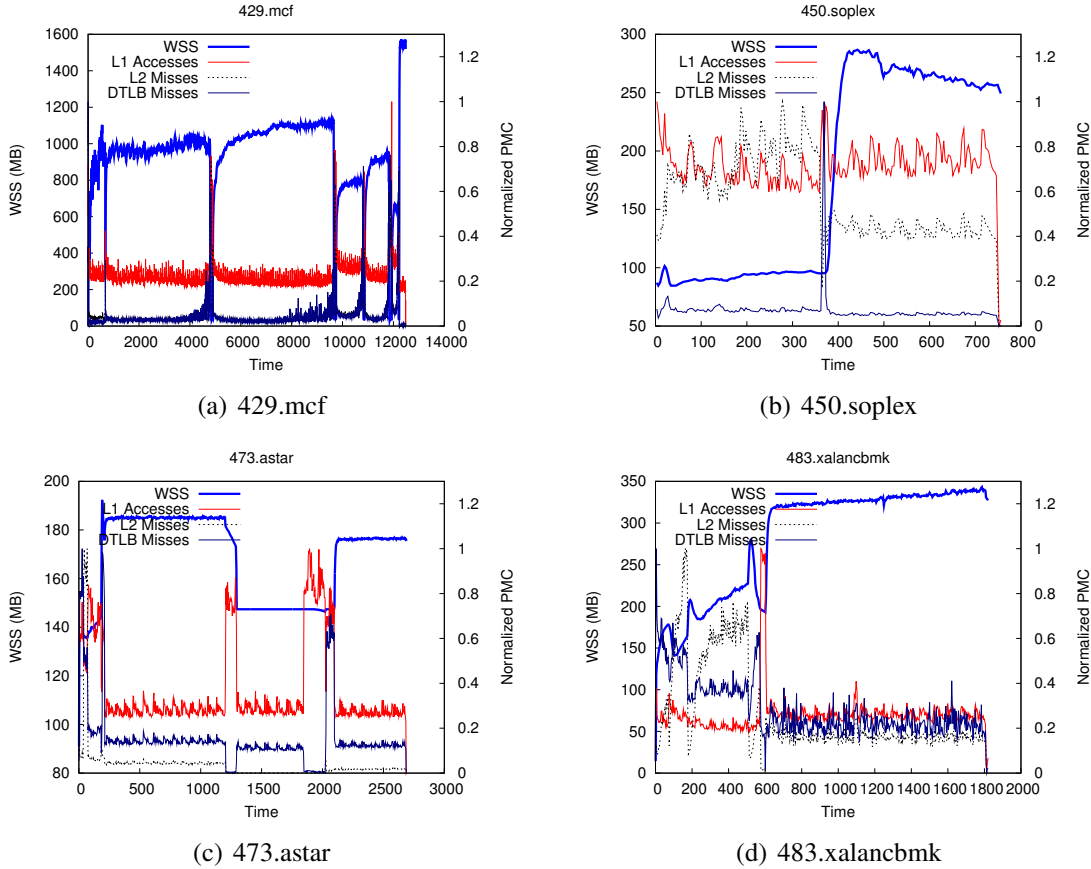
Figure 3.6: Examples of WSS and performance events

The left Y-axes show the working set size (MB) and the right Y-axes show normalized occurrences of the three events. Sampling interval is 3 seconds.

In Figure 3.6(a), the lines for L2 and DTLB misses overlap most of the time and the spikes of L1 accesses overlap with sudden drops of WSS as well.

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 $err_r \in [1 - \mathbf{T}, 1 + \mathbf{T}]$, where $\mathbf{T}$ 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.
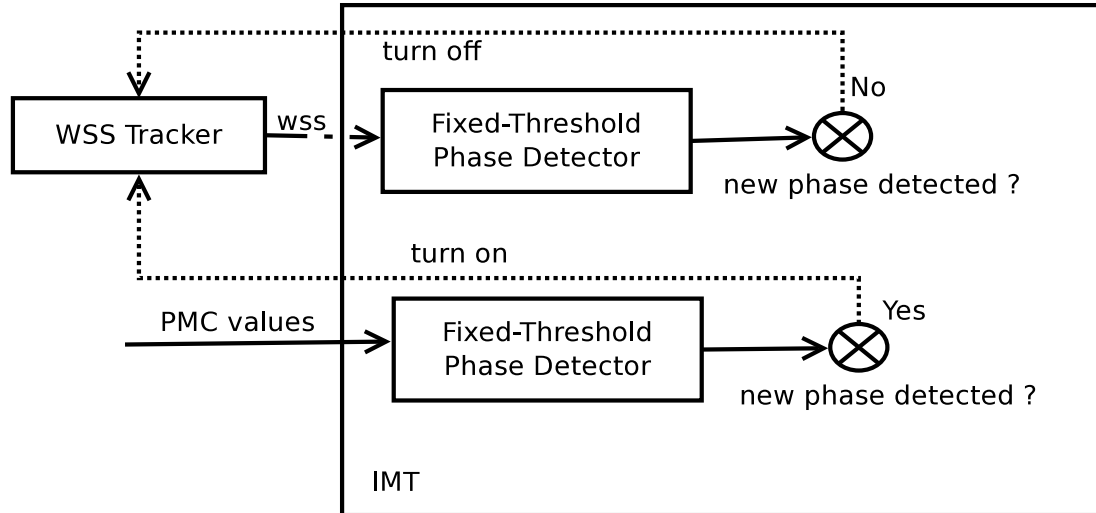
Figure 3.7: Fixed-threshold IMT

3.4.2.1 Fixed-Threshold Phase Detection

T is the key parameter in IMT because it determines the accuracy and effectiveness of phase detection. We first propose a scheme that uses a fixed value of **T**.

Figure 3.7 illustrates its organization. The phase detector on the top is based on the past WSSs. It checks if the memory demands reach a stable state or not. If so, the WSS tracking will be turned off. The phase detector at the bottom uses the occurrences of hardware events that are collected by performance monitoring counters (PMCs) to check if a phase change is seen or not. If yes, the WSS tracking will be woken up.

For stability test of memory demands, it can use a small **T** (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 a MB granularity, then as long as $err_a < 1$MB, WSS can still be assumed in a stable state even if $err_r > $ **T**.

For phase detection of hardware performance events, an over-strict threshold may cause memory tracking to be turned on unnecessarily and thus undermine its effectiveness. On the other hand, if the threshold were too large, WSS changes would not be detected, which results in inaccurate tracking results. Unfortunately, our experimental results 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. However, for each individual program, this **T** is not the optimal one.

3.4.2.2 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 in Figure 3.8. 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 the results are consistent, nothing will be changed. If the current memory demand is stable, while the PMC-based detector makes the opposite decision ($err_r > $ **T**), it implies that the current threshold for PMC phase detection 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 demand is unstable, while the PMC-based phase detector assumes a stable state, i.e. $err_r < $ **T**, it implies an over-relaxed 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 overly sensitive. 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 checkpoint design. When memory tracking has been turned off for *ckpt* consecutive sampling intervals, it is woken up to check if **T** should be adjusted or not. If no adjustment is needed, it will be turned off again until it reaches the next checkpoint or meets a new phase. In the ideal case, memory tracking will be deactivated except for checkpointing. The value of *ckpt* is adaptive. Initially, it is set to some pre-defined value $ckpt_{init}$. Afterward, if no adjustment is made in the previous checkpoint, it can be increased by some amount ($ckpt_{step}$) until it reaches a maximum value $ckpt_{max}$. Whenever an adjustment is made, *ckpt* is restored to its default value, $ckpt_{init}$. In the ideal case, the ratio of the time that memory tracking is on to the whole execution time, called *up ratio*, is nearly $1/ckpt_{max}$.

3.5 Experimental Evaluation

We use Xen 3.4 (9), an open source virtual machine monitor, as the base of our implementation. Xen's code is modified to support our page access interception, working set size estimation and performance monitor counter access. Each guest machine runs para-
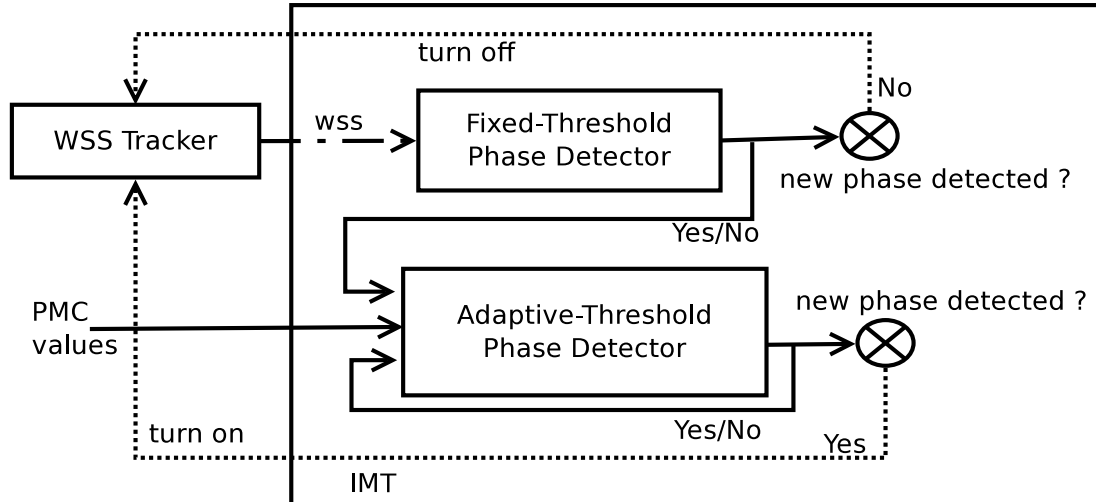
Figure 3.8: Adaptive-threshold IMT

virtualized 64-bit Linux 2.6.18, configured with 1 virtual CPU and assigned a dedicated physical CPU core.

All experiments are performed on a server equipped with one 2.8 GHZ Intel Core i5 processor (4 cores with HT enabled) and 8 GB of 800 MHz DDR2 memory.

To evaluate the accuracy of memory demand estimation, we design two micro kernel benchmarks, `random` and `mono`. Both of the two benchmarks randomly visit a specified memory space of size S for a fixed number of iterations. The value S can be seen as its memory demand during those iterations. In our experiments, we compare S against the estimated memory demand to evaluate it accuracy. The behaviors of `random` and `mono` are similar except for the way of how S is varied. In `random`, S is changed randomly among a pre-specified range $[low, high]$, while in `mono`, the size of S first increases monotonically from *low* to *high* and then decreases monotonically from *high* to *low*.

We also use DaCapo (10) and SPEC CPU 2006 to measure overhead. DaCapo is a Java benchmark suite that includes 10 real world applications with non-trivial memory requirement for some of them. We use JikesRVM 2.9.2 with the production configuration as the Java virtual machine (4, 5, 7, 3). By default, the initial heap size is 50 MB and the maximum is 100 MB.

3.5.1 Tracking Granularity

In Section 3.1.1, we use a tracking granularity of G to construct the LRU histogram. The choice of tracking unit size is a trade-off between estimation accuracy and monitoring

37
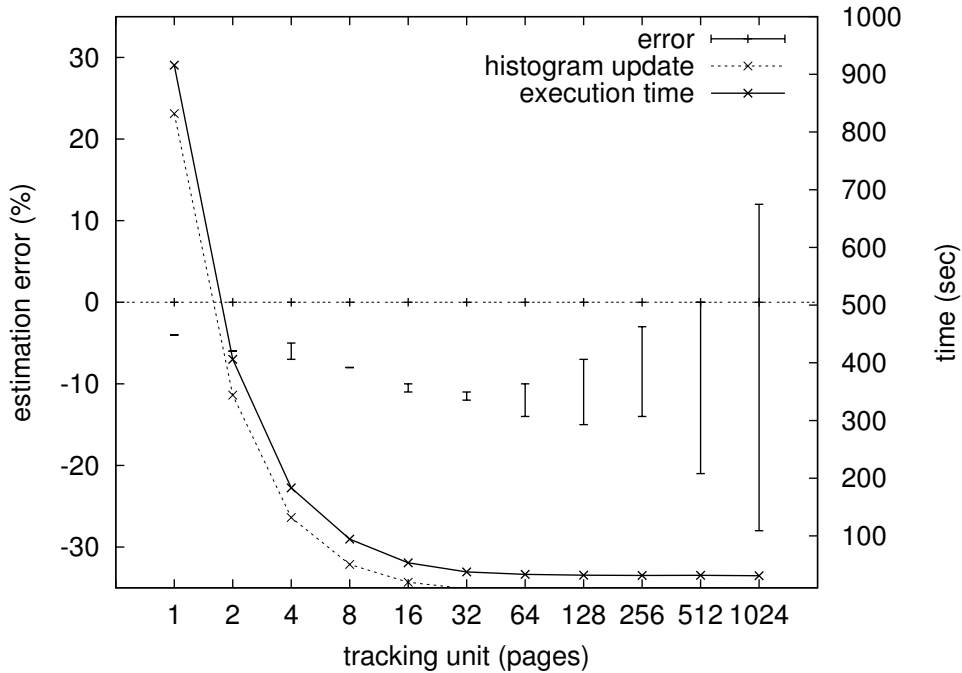
Figure 3.9: Relationship between tracking unit, accuracy and overhead

overhead. To find out the appropriate tracking unit size, we measure the execution time and estimation accuracy for various tracking granularities from 1 page to 1024 pages. We first run a program which constantly visits 100 MB of memory space. Therefore, its WSS is known as 100 MB. During the program execution, estimations are reported periodically and the highest and lowest values are recorded.

As illustrated by Figure 3.9, 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 3.10 shows the total execution time for the selected DaCapo and SPEC INT benchmarks with various tracking unit sizes[†]. As the figure shows, 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 granularity for the remaining evaluation.

[†]For some DaCapo programs, when tracking granularity is smaller than 4 pages, the excessive overhead causes JVM unstable.
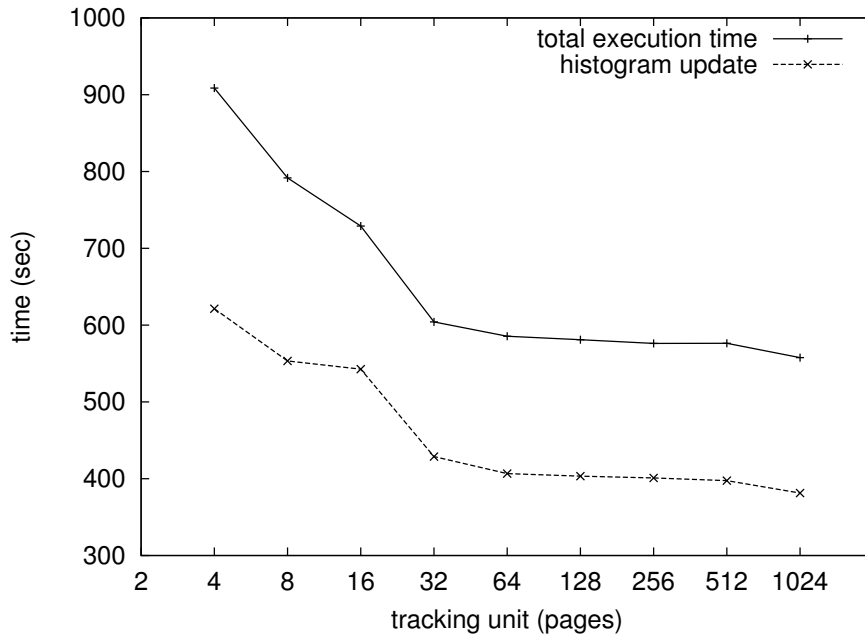
(a) DaCapo
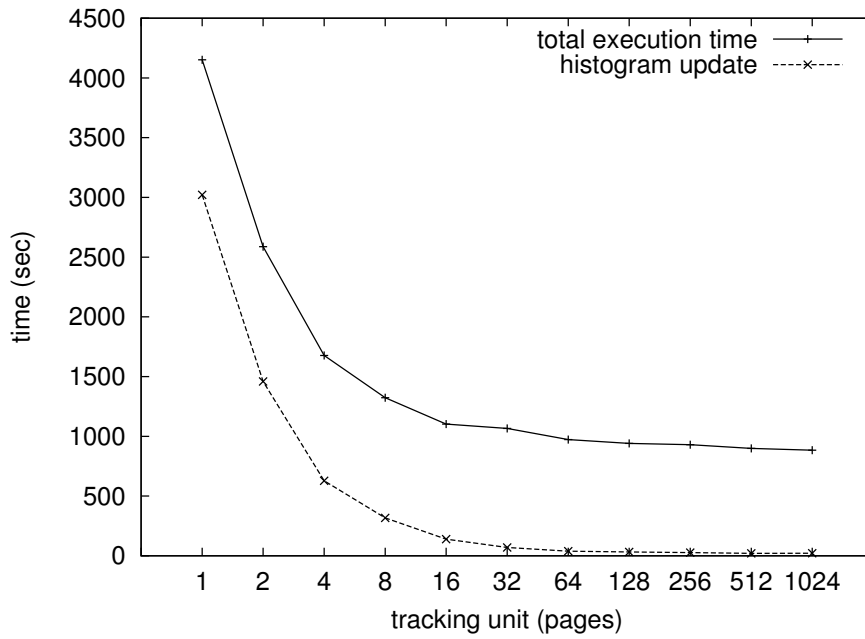
(b) SPEC INT 2000

Figure 3.10: Relationship between tracking unit and performance

3.5.2 OS-based Vs. LRU-based Memory Growth Estimation

We have implemented both the memory growth estimators discussed in Section 3.1: the OS statistics-based and the LRU-based. We run `random` and `mono` with a range of $[40, 350]$ MB on a VM with 214 MB memory allocation. Figure 3.11 shows the amount of memory growth estimated by the two predictors. The OS-statistics based estimator follows the memory usage change well. In both benchmarks, it tracks the memory allocation and release 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 memory growth estimation.

3.5.3 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 the sampling based estimation as used in the VMware ESX server (sample size is 100 pages, interval is 30 seconds[†]). As Figures 3.12(a) and 3.12(b) show, 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 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 3.12(c) and 3.12(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 214 MB 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.

3.5.4 Effectiveness of Dynamic Hot Set Sizing and AVL-Tree Based LRU List

To measure the effects of various techniques on lowering overhead, we first measure the running time of SPEC 2006 and DaCapo benchmark suite without memory tracking as the baseline performance. For SPEC CPU 2006, each program is measured individually. While for DaCapo, each of them is run in alphabetic order and the total execution time is

---

[†]the same parameters as used in (54)
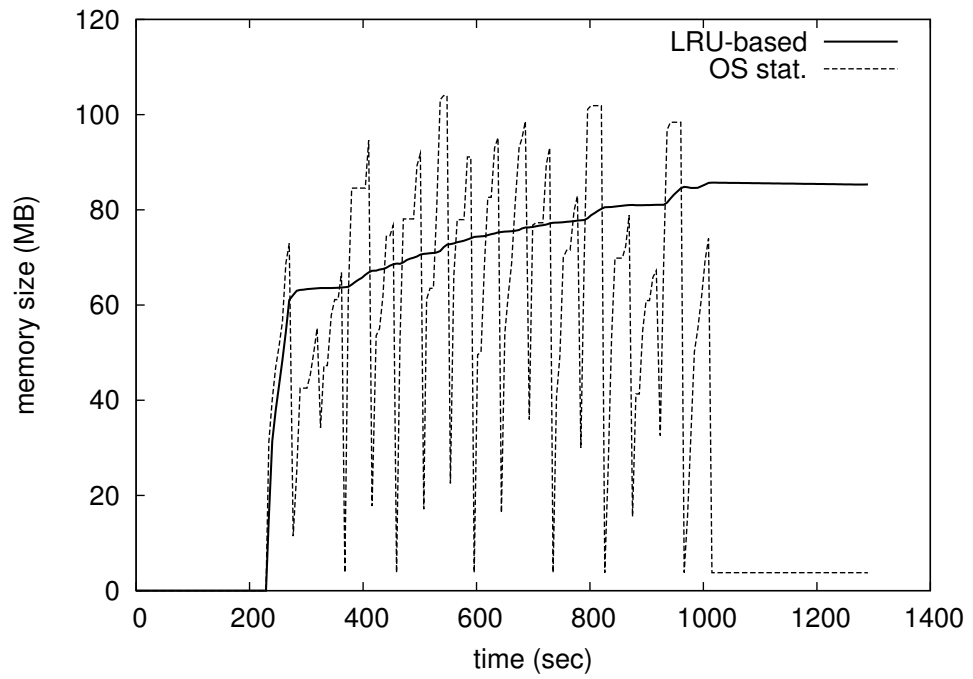
(a) `mono`
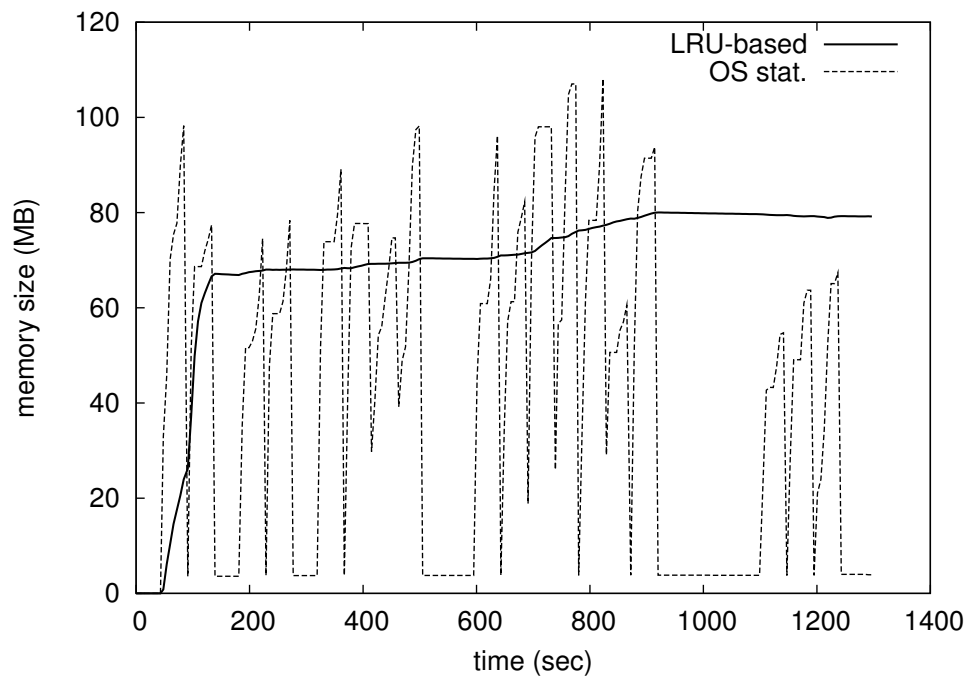
(b) `random`

Figure 3.11: Swap LRU vs. swap usage

(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)
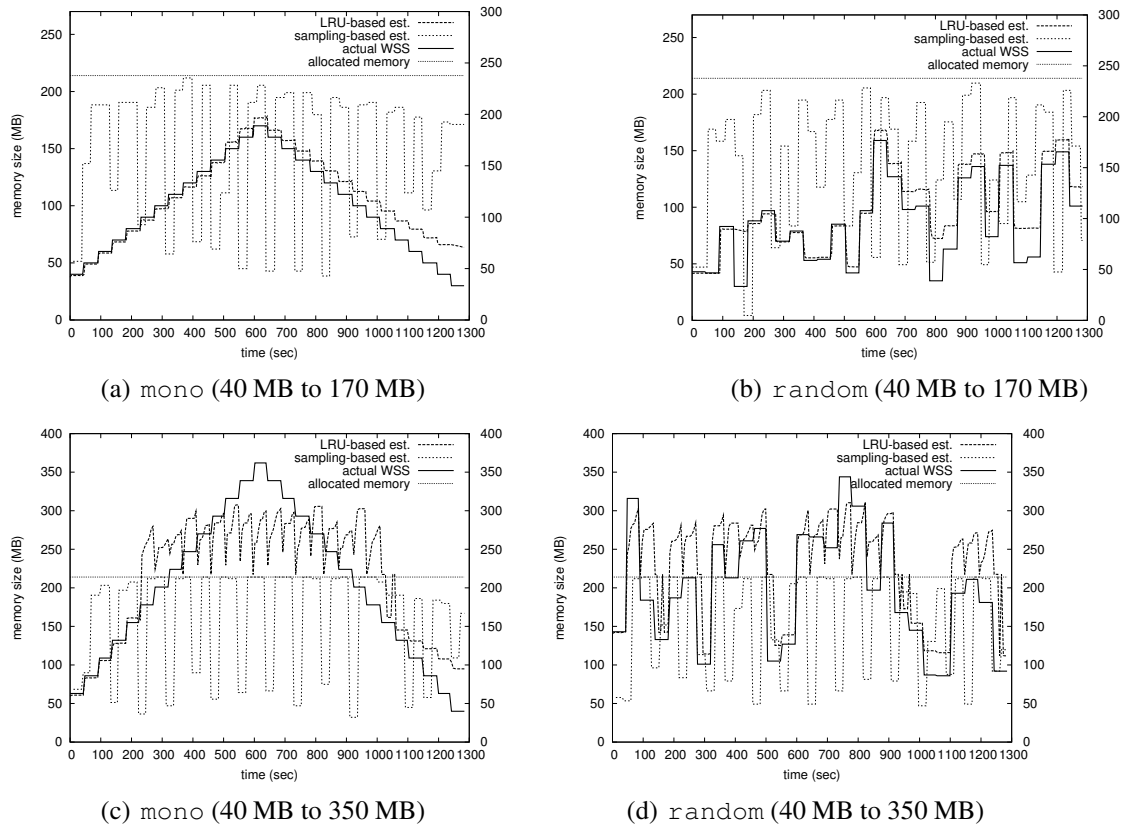
Figure 3.12: WSS estimation

measured because some programs finish in several seconds, which is too short for IMT to start working. Then we measure the running time of with memory tracking enabled, using plain linked list LRU implementation, dynamic hot set sizing (DHS), AVL-tree based LRU (ABL) implementation and the combination of the latter two, respectively.

Columns two to five of Table 3.1 list the normalized execution time against the baseline setting. For the whole SPEC 2006 benchmark suite, the plain linked-list design incurs a mean overhead of 173%. Using DHS and ABL separately, the mean overheads are lowered to 39% and 43%, respectively. Applying DHS and ABL together, the mean overhead is further reduced to 16%. When the memory working set size 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 performance gain of them is prominent. For example, in SPEC CPU 2006 suite, the top three programs with the largest WSSs are `459.GemsFDTD`, `429.mcf` and `410.bwaves`, whose WSSs are 800 MB, 680 MB and 474 MB, respectively (22).

Using DHS and ABL together, the overhead against the linked-list setting is reduced by 69.8%, 98.7%, and 85.7%, respectively. For `483.xalancbmk`, although its WSS is

Table 3.1
Overhead reduction with DHS and ABL

Program	Normalized Execution Time			
	Linked-List	DHS	ABL	DHS+ABL
400.perlbench	1.54	1.28	1.07	1.05
401.bzip2	1.03	1.04	1.01	1.02
403.gcc	1.96	1.17	1.37	1.08
410.bwaves	3.30	2.81	1.30	1.33
416.gamess	1.01	1.01	1.01	1.01
429.mcf	59.16	9.07	3.21	1.75
433.milc	13.08	8.45	3.35	2.74
434.zeusmp	2.49	1.33	1.22	1.14
435.gromacs	1.01	1.01	1.00	1.00
436.cactusADM	1.20	1.12	1.08	1.09
437.leslie3d	2.68	1.01	1.37	1.00
444.namd	1.02	1.01	1.01	1.01
445.gobmk	1.07	1.02	1.02	1.02
447.dealII	1.34	1.09	1.17	1.03
450.soplex	3.16	1.27	1.43	1.13
453.povray	1.01	1.01	1.01	1.01
454.calculix	1.03	1.01	1.01	1.01
456.hmmer	1.01	1.01	1.01	1.01
458.sjeng	9.74	1.13	1.79	1.06
459.GemsFDTD	6.79	4.17	3.28	2.75
462.libquantum	7.89	1.02	1.29	1.02
464.h264ref	1.04	1.02	1.02	1.01
465.tonto	1.01	1.00	1.01	1.00
470.lbm	4.31	2.34	1.71	1.65
471.omnetpp	41.13	1.07	4.60	1.05
473.astar	15.77	1.02	2.92	1.01
481.wrf	1.18	1.12	1.12	1.13
482.sphinx3	1.03	1.01	1.02	1.02
483.xalancbmk	7.81	1.33	2.28	1.05
Mean (SPEC CPU 2006)	2.73	1.39	1.43	1.16
DaCapo	1.28	1.07	1.14	1.07

merely 28 MB, its poor locality leads to a 681% overhead under the linked-list design. Replacing the linked-list LRU with the AVL-tree based LRU and applying DHS, its overhead is cut to only 5%. However, even both ABL and DHS are enabled, for the whole SPEC CPU 2006 benchmark suit, the mean overhead of 16% is still non-trivial.

3.5.5 Evaluation of Intermittent Memory Tracking

The performance of intermittent memory tracking is evaluated by two metrics: (1) the time it saves by turning off memory tracking, reflected by *up ratio*, and (2) the accuracy loss due to temporary deactivation of memory tracking, indicated by *mean relative error*. We first run each of the SPEC CPU 2006 benchmarks and sample the memory demands and hardware performance counters every 3 seconds without intermittent memory tracking. Then, we feed the trace results 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.

3.5.5.1 Fixed-Threshold Vs. Adaptive-Threshold

To evaluate the performance of fixed and adaptive thresholds for IMT, we use a DTLB miss as the hardware performance event for phase detection. For fixed thresholds, **T** varies from 0.05 to 0.3, two extreme ends of the spectrum. Table 3.2 shows the details.

Using fixed thresholds, when **T** = 0.05, memory tracking is off nearly three fourths of the time with an MRE of about 6%. When **T** is increased to 0.3, memory tracking is activated for only about one tenth of the time, while the MRE increases to 13%. With adaptive thresholds, its up ratio is nearly the same as that of **T** = 0.3, while its mean relative error is even smaller than that of **T** = 0.05. Clearly, the adaptive-threshold algorithm outperforms the fixed-threshold algorithm.

Figures 3.13, 3.14 and 3.15 show the results of several 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. Figures 3.13(a) and 3.13(b) show two simple cases. 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 checkpointing. Figures 3.14 shows the typical cases where there are multiple phases

Table 3.2

Up ratios and MREs of various fixed-thresholds and adaptive-threshold

	Thresholds													
	T = .05		T = .10		T = .15		T = .20		T = .25		T = .30		Adaptive	
Program	U	M	U	M	U	M	U	M	U	M	U	M	U	M
400.perlbench	47	7.5	33	11.9	20	16.2	19	16.7	19	16.7	19	17.1	29	3.1
401.bzip2	83	0.3	79	0.7	67	1.3	61	1.6	58	2.3	50	3.8	22	3.5
403.gcc	84	0.2	79	0.8	68	1.7	64	1.7	60	1.7	50	2.0	23	3.6
410.bwaves	46	2.0	44	2.2	44	2.2	42	1.2	40	1.4	34	1.9	13	4.5
416.gamess	26	1.7	2	4.5	2	4.5	2	4.5	2	4.5	2	4.5	6	0.9
429.mcf	52	2.5	39	4.2	35	10.3	32	14.6	29	35.2	24	62.1	23	38.7
433.milc	21	5.4	6	10.8	5	13.2	5	13.2	1	16.7	1	16.7	8	7.7
434.zeusmp	79	0.4	54	0.9	4	1.8	3	1.8	3	1.8	3	1.8	12	1.7
435.gromacs	13	0.5	4	1.1	4	1.1	4	1.1	4	1.1	4	1.1	8	0.2
436.cactusADM	7	11.4	5	9.5	4	8.6	3	8.3	3	8.3	3	8.3	8	1.5
437.leslie3d	5	0.9	2	1.6	2	1.6	2	1.6	2	1.6	2	1.6	6	0.3
444.namd	10	0.2	5	0.3	4	0.3	4	0.3	4	0.3	4	0.3	9	0.1
445.gobmk	59	0.4	14	1.2	5	2.1	5	2.1	4	2.1	4	2.1	9	0.8
447.dealII	98	0.0	97	0.0	96	0.0	98	0.0	95	0.1	87	1.0	50	3.9
450.soplex	18	4.1	18	4.1	13	8.3	13	8.4	13	8.4	13	8.4	16	3.7
453.povray	16	6.2	16	6.2	16	6.2	15	6.4	15	6.4	14	6.5	16	1.1
454.calculix	13	1.5	2	6.3	1	6.3	1	6.3	1	6.3	1	6.3	5	0.5
456.hmmer	22	5.4	2	74.0	2	74.0	2	74.0	2	74.0	2	74.0	13	3.5
458.sjeng	9	2.9	6	3.1	4	5.9	2	6.1	2	6.1	2	6.1	9	2.6
459.GemsFDTD	3	0.4	1	0.5	1	0.5	1	0.5	1	0.5	1	0.5	5	0.6
462.libquantum	4	0.0	3	0.2	3	0.2	3	0.2	3	0.2	3	0.3	7	0.2
464.h264ref	59	0.2	56	0.6	13	1.4	6	2.2	6	2.2	6	2.2	7	0.5
465.tonto	79	0.1	28	1.7	14	10.9	14	10.9	14	10.9	14	10.9	30	3.5
470.lbm	5	2.5	5	2.5	4	2.5	4	2.5	4	2.5	4	2.5	9	0.6
471.omnetpp	2	42.6	2	42.6	2	42.6	2	42.6	2	42.6	2	42.6	7	3.6
473.astar	15	0.4	12	0.4	13	0.6	11	0.9	10	1.7	10	1.7	8	1.0
481.wrf	24	1.7	17	1.9	5	6.1	5	2.8	5	2.8	5	1.1	7	0.5
482.sphinx3	29	0.4	4	5.1	4	5.1	4	5.1	3	5.2	3	5.2	8	0.5
483.xalancbmk	18	7.0	13	10.6	10	11.2	9	10.6	9	10.6	9	10.7	13	3.2
Mean	27	5.7	19	8.9	14	9.9	13	10.0	12	11.3	11	12.6	11	3.9

in terms of WSS and DTLB misses. `416.gamess` (see Figure 3.15(a)) is an exception. When examined from an overall scope, its WSS varies gradually. However, the WSS looks more stable when examined from each small time window. This makes the program assume that the WSS is in the stable mode and thus turns off memory tracking. Nonetheless, with the checkpointing mechanism, the WSS variances are still captured. Figure 3.15(b) shows that, though the WSS is stable most of the time, the DTLB miss fluctuates randomly. With the adaptive algorithm, the noise is filtered by increased thresholds.

Overall, adaptive-threshold based IMT achieves an up ratio of 11.5% with MRE of 3.9%.

(a) 450.soplex　　　　　　　　　(b) 400.perlbench
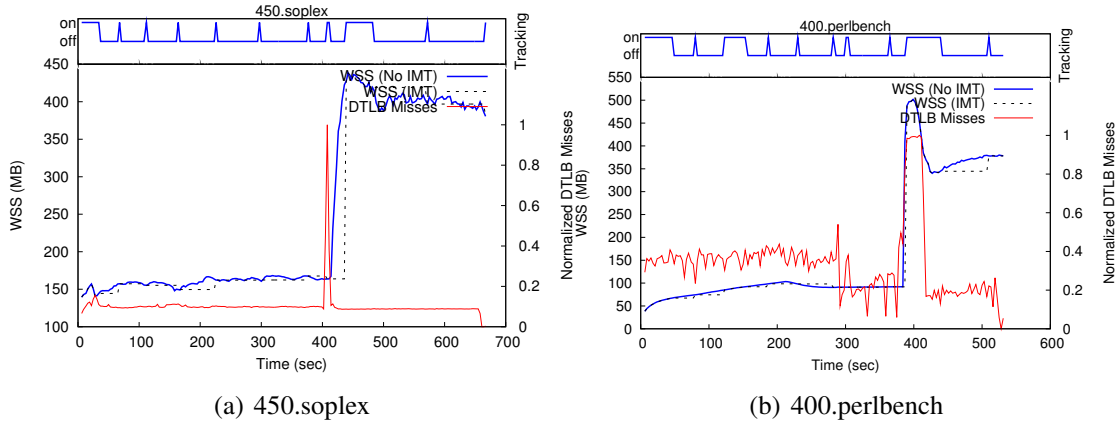
Figure 3.13: Examples of using adaptive-thresholds IMT (simple cases)

With adaptive-threshold IMT, the mean MRE of all other programs except for `429.mcf` is merely 2%. For `429.mcf`, as Figure 3.14(d) shows, most of the time, the WSS estimation using IMT follows the one without using IMT. The high relative error is because its WSS changes dramatically up to 9 times at the borders of phase transitions. Though after a short delay, IMT detects the phase change and wakes up memory tracking, those exceptionally high relative errors lead to a large MRE. More specifically, during 67% of its execution time, the relative errors are below 4%, and during 84% of the time, the relative errors remain within 10%.

3.5.5.2　Selection of Hardware Performance Events

In addition to the fixed/adaptive threshold algorithms, the other dimension of the design space of IMT is the selection of hardware performance events. For comparison purpose, three memory related events are traced simultaneously: DTLB misses, L1 references and L2 misses. First, each of the events is used separately. Second, all combinations of these events are tested with different phase identification policies. Table 3.3 lists the test results. The first column lists the name of the monitored events and the letter in parentheses denotes the adopted policy: "a", "c" and "v" stand for the aggressive policy, the conservative policy and the voting-based policy, respectively. (The details of each policy are discussed in Section 3.4.1).

Interestingly, each of the single events and any combinations of them with the aggressive policy achieve similar performance. The conservative policy gives the best accuracy. However, the up ratios are the highest too. When all three events are used, the voting policy shows a moderate result. Since using more than one event does not boost performance significantly, in the following experiments, we use only DTLB misses.

(a) 403.gcc

(b) 410.bwaves

(c) 447.dealII

(d) 429.mcf

(e) 433.milc

(f) 473.astar
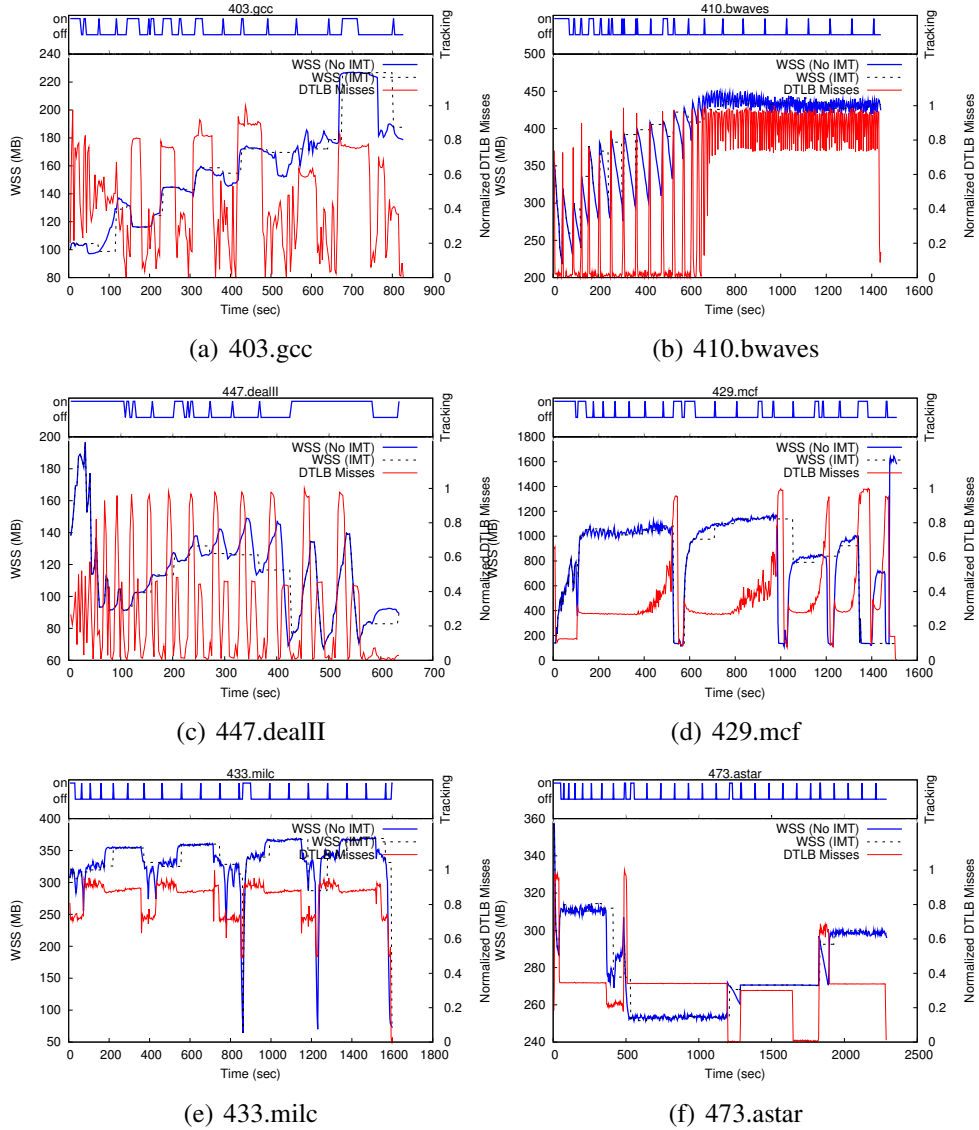
Figure 3.14: Examples of using adaptive-thresholds IMT (common cases)

3.5.6 Overhead Revisited

As discussed in Section 3.5.4, using dynamic hot set sizing and AVL-based LRU to-gether, the overhead for most programs is successfully lowered except for some programs with large WSSs and/or bad locality. For example, for high-overhead programs, such as `429.mcf` and `433.milc`, the average WSSs are 859 MB and 334 MB, respectively.

In Table 3.1, we compare the normalized execution time of using AVL-based LRU plus dynamic hot set sizing against the baseline setting. Now, in Table 3.4, we present the nor-malized execution time when IMT is augmented, as well as the up ratios of IMT. For easy

(a) 416.gamess   (b) 444.namd
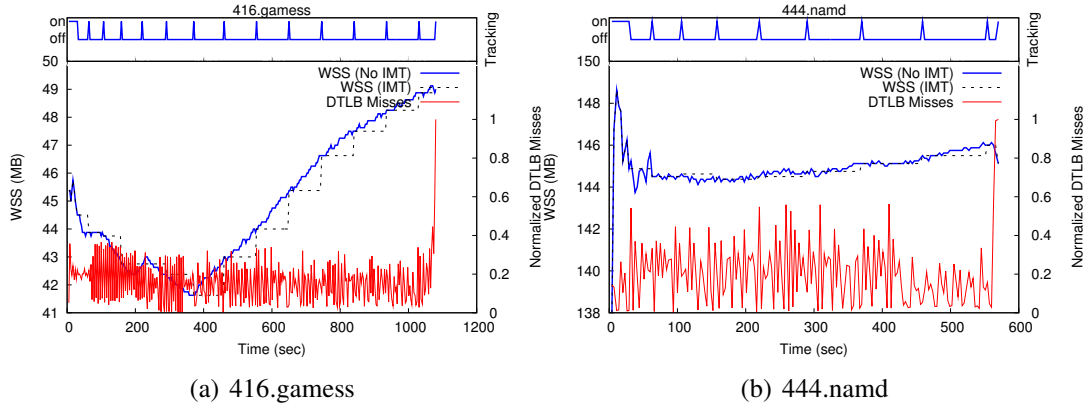
Figure 3.15: Examples of using adaptive-thresholds IMT (special cases)

Table 3.3
Effects of different hardware performance events and policies

Events and policies	UR	MRE
DTLB	0.115	0.039
L2 Miss	0.103	0.046
L1 Ref	0.118	0.041
DTLB+L1 (a)	0.114	0.031
DTLB+L1 (c)	0.185	0.016
DTLB+L2 (a)	0.102	0.044
DTLB+L2 (c)	0.179	0.017
L1+L2 (a)	0.106	0.041
L1+L2 (c)	0.177	0.020
DTLB+L1+L2 (a)	0.105	0.041
DTLB+L1+L2 (v)	0.137	0.025
DTLB+L1+L2 (c)	0.190	0.018

comparison, the results of ABL plus DHS are repeated in the second column of Table 3.4. With the same setting, enhanced with IMT of fixed-threshold ($\mathbf{T} = 0.2$), the mean overhead of SPEC 2006 is lowered to merely 6%. Using adaptive-threshold IMT, the mean overhead is further reduced to 2% by further cutting off half of the up time of memory tracking.

3.6   Chapter Summary

In conclusion, combining the three optimizations: AVL-tree based LRU list, dynamic hot set sizing and intermittent memory tracking, our LRU-based WSS estimation mechanism achieves a mean overhead of 2% with only 4% accuracy loss, which means it can be practically used to guide memory resource balancing. Based on this low overhead, accurate WSS estimator, we design a local and two global balancing schemes, which will be discussed in the next two chapters, respectively.

Table 3.4

Overhead reduction and up ratios of IMT

Program	Normalized Execution Time			Up Ratios	
	D+A	D+A+IMT(f)	D+A+IMT(a)	IMT (f)	IMT (a)
400.perlbench	1.05	1.00	1.02	0.22	0.11
401.bzip2	1.02	1.01	1.01	0.76	0.14
403.gcc	1.08	1.02	1.03	0.87	0.11
410.bwaves	1.33	1.19	1.02	0.56	0.15
416.gamess	1.01	1.00	1.00	0.18	0.09
429.mcf	1.75	1.41	1.04	0.72	0.37
433.milc	2.74	2.46	1.05	0.52	0.11
434.zeusmp	1.14	1.06	1.06	0.13	0.21
435.gromacs	1.00	1.00	0.99	0.13	0.10
436.cactusADM	1.09	1.00	1.02	0.14	0.15
437.leslie3d	1.00	1.00	1.00	0.13	0.10
444.namd	1.01	1.00	1.00	0.15	0.11
445.gobmk	1.02	1.01	1.01	0.38	0.10
447.dealII	1.03	1.02	1.01	0.87	0.11
450.soplex	1.13	1.01	1.10	0.49	0.21
453.povray	1.01	1.00	1.00	0.26	0.09
454.calculix	1.01	1.00	1.00	0.10	0.12
456.hmmer	1.01	1.00	1.01	0.25	0.09
458.sjeng	1.06	1.01	1.01	0.31	0.10
459.GemsFDTD	2.75	0.99	0.99	0.15	0.10
462.libquantum	1.02	1.00	1.01	0.15	0.10
464.h264ref	1.01	1.01	1.00	0.15	0.14
465.tonto	1.00	1.00	1.01	0.13	0.09
470.lbm	1.65	1.01	1.00	0.17	0.10
471.omnetpp	1.05	1.00	1.04	0.26	0.23
473.astar	1.01	1.01	1.00	0.51	0.13
481.wrf	1.13	1.00	1.00	0.13	0.09
482.sphinx3	1.02	1.00	1.00	0.14	0.09
483.xalancbmk	1.05	1.02	1.00	0.57	0.09
Mean of SPEC 2006	1.16	1.06	1.02	0.26	0.12
DaCapo	1.07	1.04	1.01	0.82	0.20

D: DHS, A: ABL, IMT(f): IMT with fixed-threshold of 0.2, IMT(a): IMT with adaptive-threshold

Chapter 4

Local Memory Resource Balancing

With the LRU miss ratio curves of all VMs on a physical machine, we can dynamically adjust each VM's memory allocation size. We call this scheme *local memory resource balancing*. In this chapter, we first present our memory resource balancing and arbitration scheme in Section 4.1. When there is no sufficient physical memory to meet all VMs' memory requirement, our arbitration algorithm is able to quickly find an allocation plan that aims for the overall performance. Then, in Section 4.2, our experimental results show that using local memory resource balancing, the overall performance is significantly improved by up to 8.05 times.

4.1 Local Memory Resource Balancing And Arbitration

Once the working set sizes of all VM are estimated, the balancer needs to determine the target allocation sizes for them. Assume that P is the size of all available host machine memory when no guest is running, and 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 final 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 (see Section 2.1.3) adjusts memory size on a page unit, a brute force search takes nearly $O(M^{|V|})$ 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 ratio 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 then 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.

It is possible that 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 that is 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.

4.2   Implementations And Experimental Results

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

To evaluate the effect of automatic memory resource balancing, we first start from balancing for two VMs, each runs different workloads. We evaluate various workload combinations, including a simple scenario of CPU intensive + memory intensive workloads where there is no memory contention, a workload combination of DaCapo and SPEC WEB that has occasional memory competition, a combination of two memory intensive workloads

and a case of workloads that have large WSSs. We then extend the number of participating VMs from two to four with four different workloads, which represents a typical setting of mixed workload.

In the following experiments, WSS tracking is optimized with adaptive IMT, AVL-tree based LRU list and dynamic hot set sizing, except as described explicitly. We set the lowest possible memory allocation of each VM to 120 MB. Without explicit specification, in the following experiments, each VM is preallocated with 250 MB memory. Hence, with this default memory allocation, a VM's memory may vary from 120 MB to $250 + (N-1) * (250 - 120)$ MB where N is the number of total VMs.

4.2.1 Balancing For Two VMs

4.2.1.1 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, on VM_2, the DaCapo suite runs first, followed by the same number of iterations of `186.crafty`.

Figure 4.1(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 gradually allocated to the two VMs.

To show the performance that an ideal memory balancer could deliver, we measure the *best case* performance on two VMs, each with 380 MB fixed memory, the peak memory allocation that a VM could own during balancing.

Table 4.1 lists the number of major page faults and execution time for both VMs. With memory balancing, the number of total major faults is reduced by a factor of 25.

Figures 4.1(b) and 4.1(c) show the execution time of each benchmark in the three settings respectively: baseline, best case, and balancing. With memory balancing, the performance of `186.crafty` is nearly the same, but DaCapo gains a speedup of 11 and 8.3 on the two hosts, respectively. Most notable improvements are from `Eclipse` and `Xalan`, whose average execution time on the two VMs is cut into $1/18$ and $1/32$, respectively. These two benchmarks require around 350 MB memory, resulting in a large number of page faults without memory balancing. Eventually, using memory balancing, it achieves an overall speedup of the two VMs of 8.05.

Table 4.1
Major page faults and execution time

	Baseline		Balancing		Best	
	VM_1	VM_2	VM_1	VM_2	VM_1	VM_2
Major page faults	158,810	110,965	4,102	6,499	1,428	538
Execution time (DaCapo)	1,619	1,267	147	153	127	110
Execution time (186.crafty)	32.5	32.5	32.7	32.9	32.5	32.7

The execution time is measured in seconds. For 186.crafty, the mean execution time of 12 iterations is listed.

4.2.1.2 DaCapo + SPEC Web

We then evaluate its performance for a more balanced workload combination: running Da-Capo on VM_1 and running SPEC Web 2005 on VM_2. Compared with `186.crafty`, the SPEC Web 2005 benchmark suite requires more memory resources, which results in moderate memory competition with DaCapo. It includes three web applications: `Banking`, `Ecommerce` and `Support`.

Figure 4.2 shows the allocated memory and expected memory for both VMs and the execution of DaCapo and SPEC Web. With balancing, though the performance of `Banking` degrades by 9.1%, DaCapo on VM_1 gains a significant speedup of 6.39 and `Support` on VM_2 shows an improvement of 9%, and the performance of `Ecommerce` is almost unaffected.

4.2.1.3 Memory Intensive + Memory Intensive Workloads

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 each program in alphabetical order while VM_2 runs them in the reversed 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.

Figures4.3(a) and 4.3(b) compare the execution time with and without memory balancing, and Figure 4.3(c) shows the memory allocation with balancing during the execution. Without balancing, the number of major page faults of the two VMs is $175,797$ and $835,667$, respectively. After applying balancing, it is lowered to $51,348$ and $260,205$, respectively, which leads to a speed up of 3.31 and 3.79, respectively.
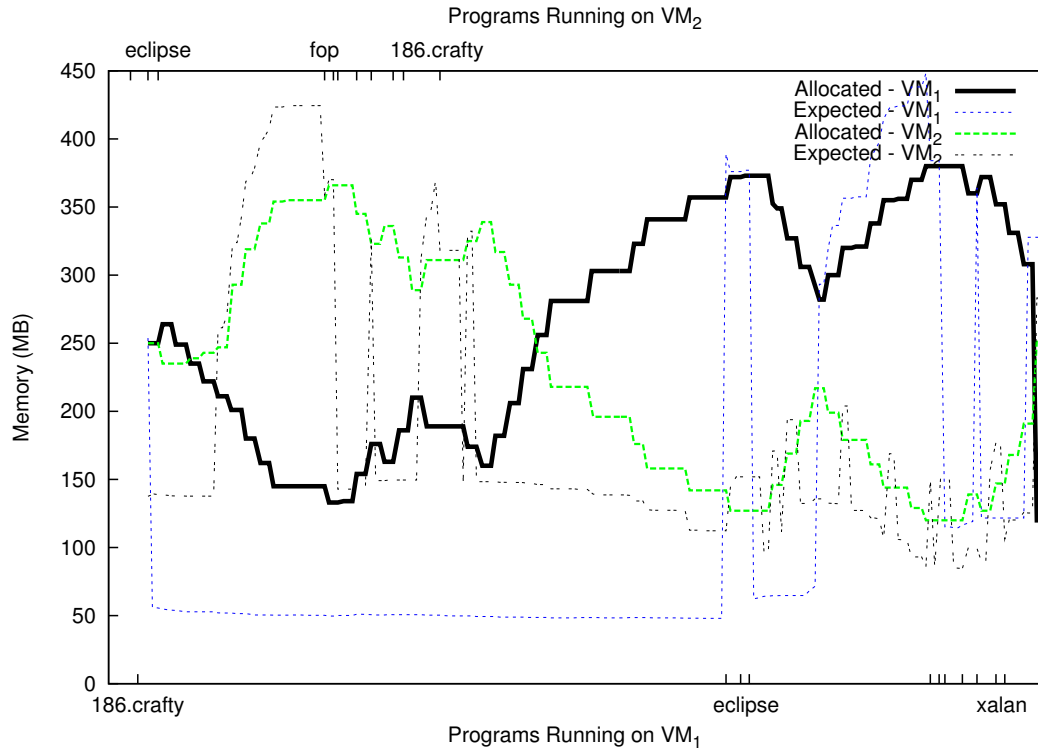
(a) Memory Allocation
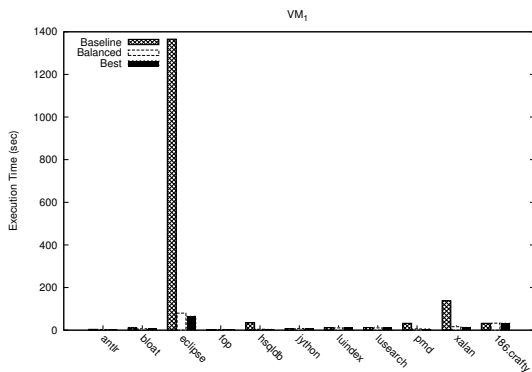
(b) Performance Comparison of VM$_1$
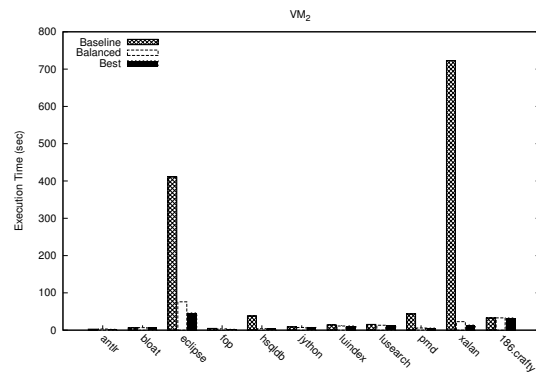
(c) Performance Comparison of VM$_2$

Figure 4.1: Local memory resource balancing: DaCapo + `186.crafty`. For readability, in Fig. 4.1(a), only a few program names of DaCapo are labeled. Fig. 4.1(b) and 4.1(c) show the complete program names in the order of execution.

4.2.1.4 Workloads With Large WSSes

To evaluate the performance impacts of the overhead of WSS tracking on memory balancing, we select two workloads with large WSSes because the overhead of WSS tracking is

(a) Memory loads on both VMs

(b) Performance comparison
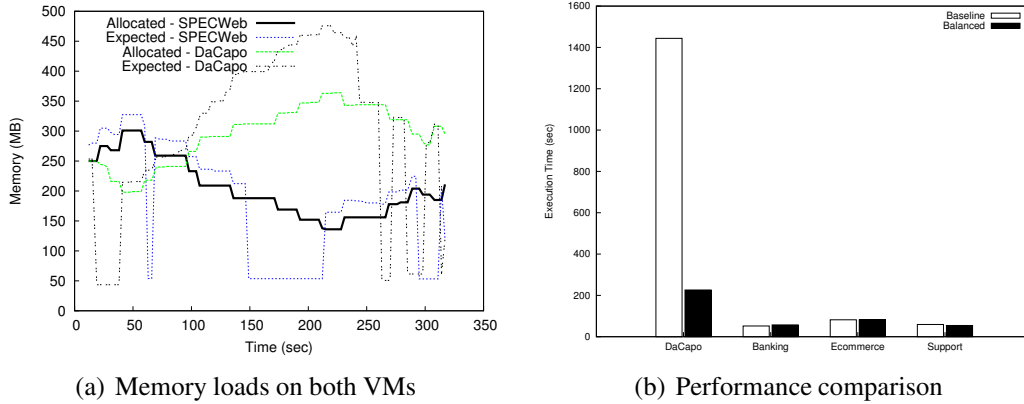
Figure 4.2: DaCapo + SPEC Web

directly related with the WSS of the monitored program. We use two benchmarks of SPEC CPU 2006: `470.lbm` and `433.milc`. The average WSS of them is 680 MB and 402 MB, respectively (22).

Initially, each VM is allocated with 700 MB of memory. VM_1 runs `470.lbm`, and VM_2 runs `433.milc` at the same time. We also compare the balancing results without using IMT, using IMT of fixed threshold of 0.2 and using IMT with adaptive threshold. Figure 4.4 shows the normalized speedups with memory balancing against the baseline setting, memory allocations and WSS tracking status.

When balanced without using IMT, the performance of `470.lbm` degrades by 10% due to the overhead of memory tracking, while the performance of `433.milc` is boosted by 2 times due to the extra memory it gets from the other VM. Using IMT, the performance impact of memory tracking on `470.lbm` is lowered to 4%. For `433.milc`, with fixed-threshold IMT, its speedup is increased from 2.96 to 3.06. Using adaptive-threshold IMT, its speedup is further increased to 3.56. The overall speedups of balancing without IMT, with fixed-threshold IMT and adaptive-threshold IMT are 1.63, 1.72 and 1.85.

4.2.2   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 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 4.2.1.3), VM_3 runs `186.crafty` for 12 iterations, and VM_4 runs SPEC Web 2005. As shown in Figure 4.5, with memory balancing, the performance of DaCapo and DaCapo' are boosted by a factor of 8.17 and 10.72, respectively, with the cost of a 70% slowdown for `Banking`. The performance of `186.crafty` and `Ecommerce` are slightly impacted

56
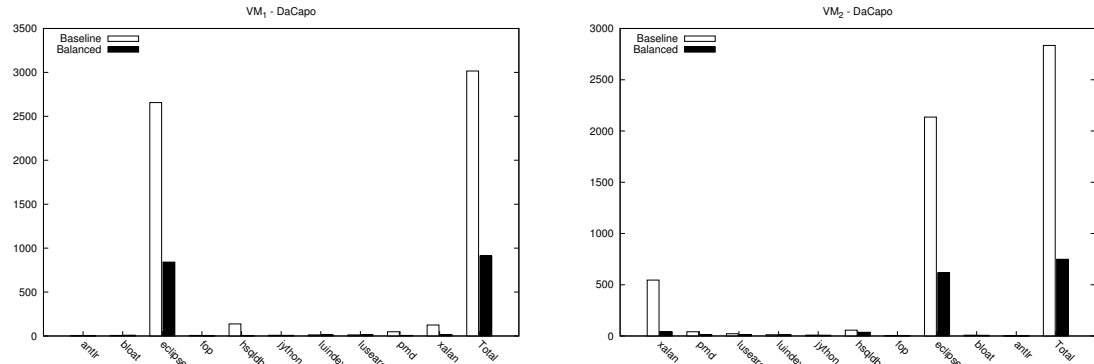
(a) Execution time (VM$_1$)

(b) Execution time (VM$_2$)
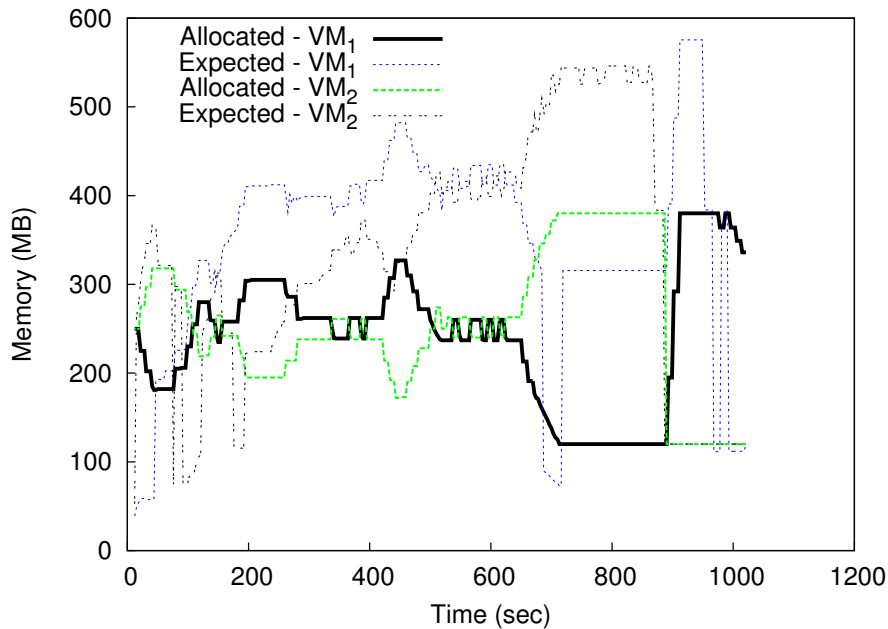
(c) Memory loads on both VMs

Figure 4.3: DaCapo + DaCapo'

by 3% and 5%. The overall mean speedup of using memory balancing is 1.72.

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 set the weight of the miss penalty of VM$_4$ as 20 times of the rest of the VMs. As displayed in Figure 4.5, after priority adjustment, `Banking` only loses 9% performance, while the significant performance improvement of the VMs of DaCapo is still maintained.
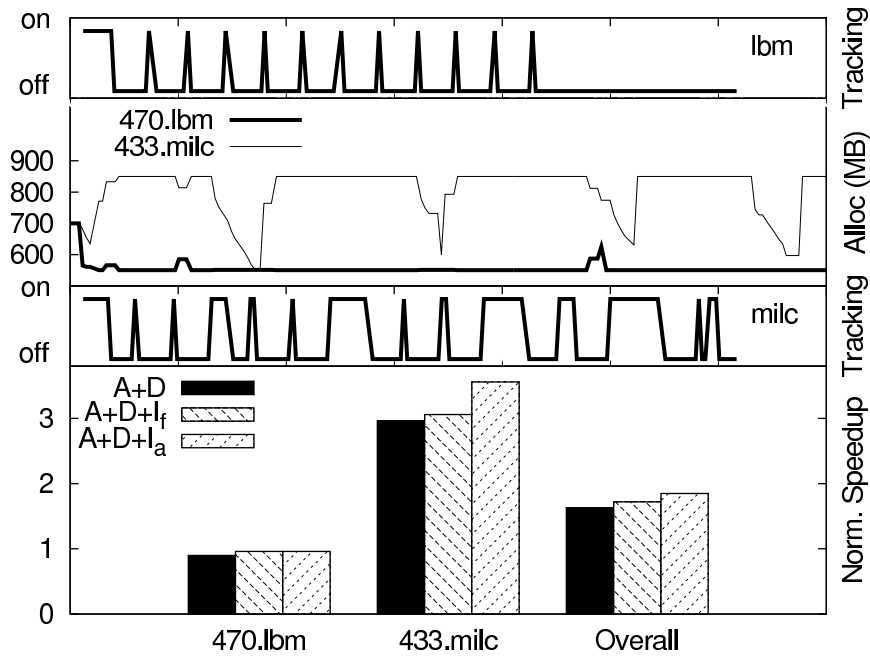
Figure 4.4: Memory balancing of `470.lbm` and `433.milc`

(A: AVL-Tree based LRU list, D: dynamic hot set sizing, I_f: fixed-threshold IMT, I_a: adaptive-threshold IMT)
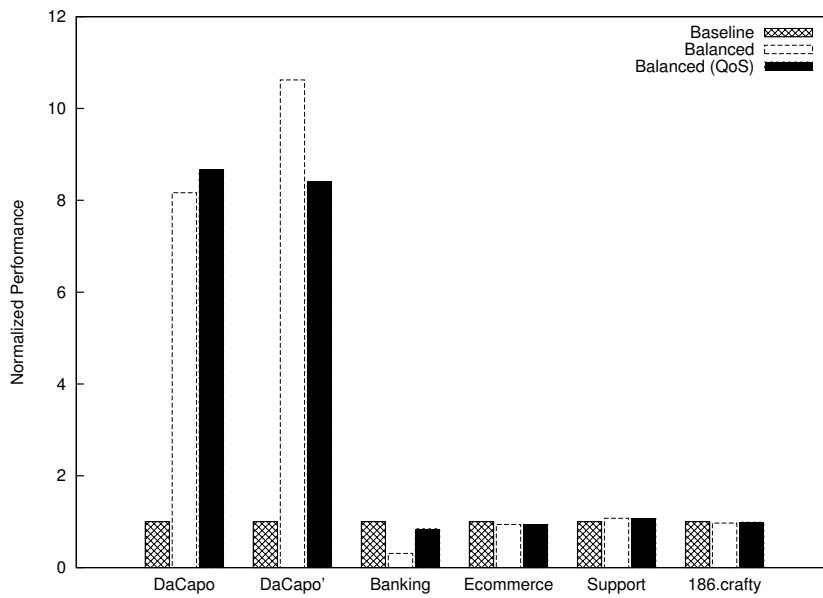
Figure 4.5: Performance comparison: DaCapo + DaCaPo' + 186.crafty + SPEC Web

4.3 Chapter Summary

As demonstrated by the experimental results, based on the WSS tracking scheme, our local memory resource balancer can effectively improve overall system performance. Even for the case with heavy memory resource competition, our arbitration algorithm still boosts the overall performance by a factor of 3. And for programs with large working set sizes, the experimental results show that the WSS tracking scheme is able to guide memory balancing with low cost and eventually boots the overall performance by 1.85 times.

The 4-VM setting shows that our balancing algorithm can balance memory resources for multiple virtual machines. To achieve better overall performance, the algorithm may sacrifice the performance of some VMs, but after applying higher priority to important VMs, the quality of service of those VMs can be guaranteed.

Though the local balancing scheme improves the memory utilization of a single host, performance penalty still exists when the total memory demand of all VMs exceeds the host's available physical memory or a spike of memory demand occurs. In the next chapter, we will present two global balancing techniques that address these problems.

Chapter 5

Global Memory Resource Balancing

In the previous chapter, we present our local memory resource balancing design for an individual host. However, for VM farms in a data center or a cloud computing infrastructure, memory resource imbalance may exist among multiple physical machines. Commonly, in a virtualized data center, all physical machines are interconnected with a fast network and all VMs run on a shared storage. Contrast to the memory resources on a single machine that are referred as local memory resources, we call the memory resources of all machines in this computing environment as global memory resources.

In this chapter, we introduce two balancing schemes, live migration based balancing and remote caching based balancing to improve global memory resource utilization.

5.1   Live Migration Based Global Balancing

Live migration enables moving one VM between two hosts by transferring its memory states from its original host to the destination host. When one host suffers from insufficient physical memory, we can migrating one or more VMs to other hosts with sufficient idle memory to relieve the memory pressure of the original host and improve the overall memory resource utilization.

Unfortunately, the cost of live migration is so high that without a wise migration policy it may cause a backfire. In addition, the decisions of target host and migration candidate play an important role as well. This decision process is similar to the decision of an OS job scheduler that moves processes among hosts (50, 23). The decision framework of the OS scheduler consists of three policies, an *information policy*, a *transfer policy*, and a *placement policy*. We apply the same framework to VM-migration based memory balancing. The information policy determines when to initiate migration, the transfer policy decides

which VM will be migrated out, and the placement policy selects the target host for migration. The following subsections address these three policies respectively.

5.1.1   Information Policy

When the total memory resources on a host become insufficient, the performance of one or more VMs will degrade significantly. Under this circumstance, there are two decision choices: migration to release the memory pressure of the whole host or balancing memory locally to minimize the overall penalty. If the memory pressure only lasts for a short period, local balancing is preferred because the time spent on migration may not pay off the benefits brought by the extra memory it receives. On the other hand, if the memory pressure sustains for a long period, the earlier to migrate, the more performance is gained. Unfortunately, without *a priori* knowledge of a system's memory demand trend, it can barely make the optimal decision.

We find this decision problem is similar to the classical *ski-rental* or *rent/buy* problem (45): a skier needs to decide to rent skis or buy a pair of new. With the perfect *a priori* knowledge that he knows he will ski n times, the optimal decision is obvious. If $n * r \leq b$, where r is the rental cost each time and b is the cost of a pair of new skis, he will rent every time. Otherwise, he will buy at the first time he skis. However, without knowing the n beforehand, it is unlikely to make an optimal solution. Suppose the skier decides to buy a pair of skis before his i-th skiing, then if $n < \lceil \frac{b}{r} \rceil$, he pays $\frac{b-r*i}{r*i}$ more than then optimal choice. That is, the earlier hey buys, the more he pays. But if $n >= \lceil \frac{b}{r} \rceil$, then the result is reversed. The known best deterministic algorithm is the *break-even* algorithm where the skier should buy immediately if his accumulated rental cost is equal to b (31). This algorithm guarantees that his spending will be bounded by two times the optimal cost. A more aggressive algorithm randomly selects the i ($i < \lceil \frac{b}{r} \rceil$) to buy the skies (32).

Karlin *et al.* apply the break-even algorithm to snoopy cache (31), in which the problem is whether a dirty cache line should be retained or discarded for a snoopy cache. Romer *et al.* (43) use the similar idea to strike a balance between the cost of constructing a superpage and the benefits it brings to reduce TLB misses. Karlin *et al.* also present the randomized algorithm for the snoopy caching problem(32). We can inherit the ideas of the strategies to make a decision about whether to start migration or just retain the current state. Once memory pressure is detected, we first assume a victim VM would be migrated and estimate its migration time (denoted as EMT). We then predict the trend of that host's memory demand during the period of $[now, now + EMT]$. The migration decision will be made if: (1) when the host has already suffering from memory pressure for at least $EMT/2$ and (2) during the future $EMT/2$ period the memory pressure is still predicted to exist. We use the threshold, $EMT/2$, instead of the break-even value of EMT is because the prediction mechanism may increase the confidence of the decision making and thus allow a more

aggressive threshold. The prediction is performed using the memory demand in the most recent $EMT/2$ time. If a descending trend is detected, we can assume that the demand peak has passed.

5.1.1.1 Online Memory Demand Prediction

Migration time of a VM is a relative long period. The memory demand of the VM can change during this period. If the predictor finds the memory demand will go down, it can choose not to migrate, expecting that the local memory might become sufficient. We design an effective online linear regression model to predict the trend of memory demand in the next period. The memory demand of next period is $\beta \cdot time + \varepsilon$ where the coefficients β and ε are determined by linear fitting on the recent historic sample data. If the memory demand trend changes its direction, or in other words, we encounter a *break point*, we need to start a new linear function. We introduce a break point detection mechanism into our predictor. If a break point is detected in the historic data window, only the data after the break point will be used for linear fitting. Though various break point detection strategies have been suggested (8, 25, 62), we instead employ a quick and simple method to avoid high computing overhead. Let $c_1, c_2, \ldots, c_n$ be the n most recent historic data, in which c_1 is the oldest datum and c_n is the most recent datum. Starting from c_n, for each two consecutive data, we compute the slopes respectively. If the current slope, for example, the slope between c_i and c_{i-1}, differs from its previous slope by some preset threshold, it is identified as a break point and only data from c_i will be used for linear fitting.

5.1.1.2 Live Migration Time Estimation

Classical *pre-copy* live migration algorithm transfers memory pages iteratively (15). The pages that are written during the current iteration will be re-transferred in the next iteration. The whole process stops until the number of remaining pages is small enough or the number of iterations reaches some preset limit. The duration of live migration is mainly determined by network throughput, memory size and page dirty rate.

Assume that the network bandwidth does not vary much among iterations and between the time of estimation and the time of actual migration. Then, we only need to estimate the number of iterations and pages to be transferred in each of the iterations, which largely depend on the page dirty rate. Fortunately, our memory monitoring mechanism can be easily extended to collect page dirty rate by checking if the intercepted page access is a write or not. The dirty rate is derived from the number of intercepted writes within the sampling interval of the local balancer.

Given the number of pages allocated to the VM (denoted as M), available network band-

width (denoted as n) and the most recent page dirty rate (denoted as w), the following algorithm emulates the migration process and computes the estimated migration time iteratively:

Algorithm 1 Migration time estimation

$p \leftarrow M$ {p is number of pages to be sent in each iteration. Initially, all pages are sent}
$emt \leftarrow 0$ {estimated migration time}
$c \leftarrow 0$ {iteration counter}
repeat
   $t \leftarrow \frac{p}{n}$ { time needed to send pages in this iteration}
   $emt \leftarrow emt + t$
   $p \leftarrow t \times w$ {number of pages written during sending}
   $c \leftarrow c + 1$
until $p < P$ or $c > C$ {P, C are VMM specific thresholds to terminate iteratively transferring}
$emt \leftarrow emt + \frac{p}{n}$ {add the time of sending the last batch of pages}
return emt

Our experiments show that the mean estimation error is 8.34%. The details are presented in Section 5.3.1.1.

5.1.2 Transfer Policy

A prerequisite for a migration candidate is that there exists at least one target machine that has sufficient idle memory to host it. To decide the victim VM, several metrics can be applied. To minimize migration time, a *fastest-migration* criterion prefers the VM with shortest estimated migration time.

Alternatively, we can apply *fastest-increasing* criterion, which favors the VM with highest increasing rate of memory demand. This policy along with the *largest* placement policy, tries to prevent memory shortage in the future. In our implementation, we use *fastest-migration* policy.

5.1.3 Placement Policy

Once the VM to be migrated is selected, all hosts with at least the same amount of free memory as the current allocation of the victim VM are considered. Note that the idle memory in the destination host can be obtained through local balancing there. Various criteria can be designed to choose one of the candidates as the destination host. For example, a

largest criterion chooses the host with the most free memory resources. And a *fastest* selects the one with the fastest computing capability (e.g. fastest CPU). In our system, we adopt the *largest* criterion.

5.2 Remote Caching

Remote caching can be used to alleviate transient memory overloads. When a memory overload only persists for a short while, using live migration may damage the performance and relying on the local balancer may still result in page swapping due to delayed response or insufficient total physical memory. In this circumstance, directing the slow disk I/Os to network I/Os can alleviate the performance penalty. Besides, even if the memory overload sustains long enough for migration but there is no hosts that have enough free memory to hold a VM, remote caching may still be used.

We adopt a cache-style design instead of a memory server design that completely replaces the whole swap device because the amount of free memory space that is used for remote caching varies dynamically. In this section, we present our remote caching design, which is mainly based on the work of Chen *et al.* (14).

5.2.1 Overview

The remote caching system follows the typical client/server design. The host that maintains a page pool and provides cache service to multiple clients is called *cache server*. The host that uses cache service is referred as *cache client*. The client and server communicates via TCP socket. On client side, disk I/Os from the guest OS to its virtual disks are intercepted by the VMM and a cache proxy inside the VMM initiates caches requests to the server and receives data if it is a cache hit. If the request misses from the cache, it falls to the normal disk I/O operations. Figure 5.1 illustrates the basic design. Note that, the cache server can serve multiple clients and its capacity is managed by the local memory balancer on that host.

5.2.2 Cache Client Design

The cache client works as a proxy that directs read requests to the remote server, receives hit data, sends new data blocks and notifies the server to invalidate data blocks.

To guarantee system reliability, the cache adopts a write-through policy to prevent the data
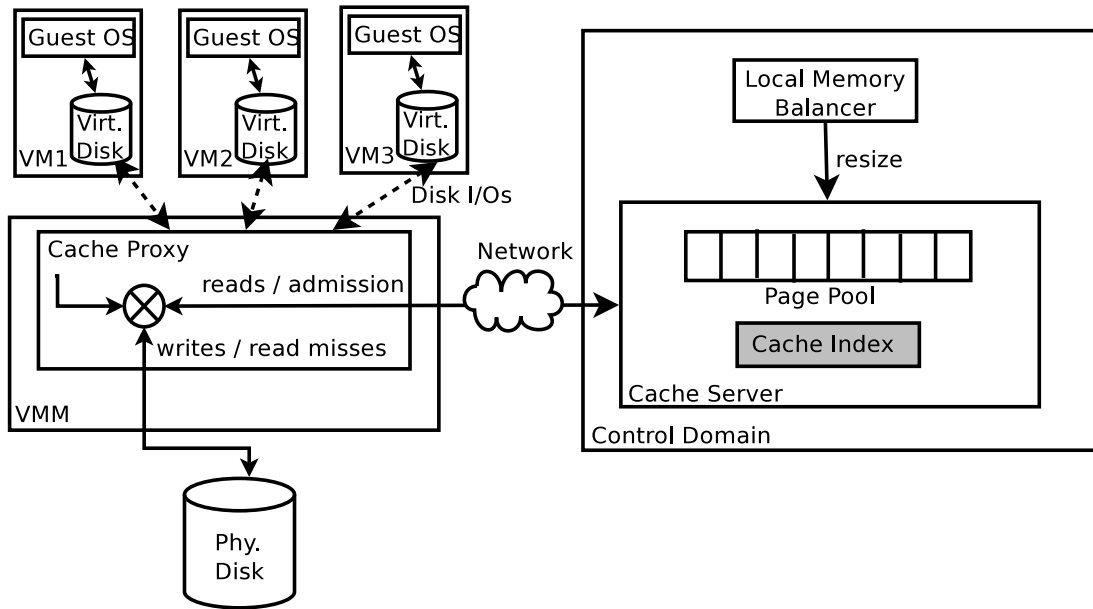
Figure 5.1: Overview of remote caching

loss in case of abnormal server down. When a write request is intercepted, the cache proxy transfers the data to disk and remote cache in parallel.

When a read request is intercepted, the disk ID and sector number are used as a key and sent to the server. If the server returns a hit alone with the corresponding data, the data is then copied to the memory location just as the disk read routine does. If the server returns a miss, the cache proxy then calls the disk read routine to complete the read request.

In our design, we do not maintain a local index inside the cache proxy as originally proposed by Chen *et al.* (14) because:

1. if the request is a hit, querying the local proxy adds overhead;

2. if the request is a miss, compared with the slow disk operations, the network overhead is negligible;

3. it eliminates the cost to update local index;

4. it is more efficient when the server employs a global cache that is shared by all its clients. For such a cache server, the cache share of each client is dynamic. Thus, synchronizing the global index and each client's local index would be costly and complicated.

Since the remote cache is designed to address short periods of memory insufficiency, page

66

swapping is supposed to be sporadic. Hence, we let a client always direct the I/Os to the server even if the capacity of the cache is very small.

5.2.3 Remote Cache Design

In the cache server, it maintains a page pool array, denoted as $pool[n]$, a hash map, denoted as H, a tag array, denoted as $tag[n]$, an LRU list and a free list, where n is the number of pages of the cache. The page pool stores the cache content and is shared by all the clients of the server. Each page in the page pool corresponds to a tag, which is linked by the LRU list. Given a read request of key k, if it is found in the hash map, the hash map returns its index $H(k)$. The corresponding content and tag can be located in $pool[H(k)]$ and $tag[H(k)]$, respectively. After transferring the page content to the client, $tag[H(k)]$ is move to the head of the LRU list and if the cache is full, the tag at the tail of the LRU list and its corresponding page are evicted.

When the server receives an allocation request, if first searches a free tag from the free list. If one is found, it is moved to the tail of the LRU list and the corresponding page is used to store the content. Otherwise, the page that corresponds to the tag at the tail of the LRU list is reclaimed. Finally, the hash map is updated.

To expand the cache size, both the page pool and tag array are enlarged. The newly added tags are appended to the free list. To shrink the cache size, it first reclaims from the free list. If there is no sufficient free space, it evicts the cache entries from the tail of the LRU list until the page pool reaches the specified size. The hash map is updated as well to reflect the eviction.

5.2.4 Manage Cache Size

When cooperating with local memory balancer, a naive approach is to allocate all its free memory to the cache server and when the host is under memory pressure, it reclaims required memory from the cache. However, allocating all the free memory may impact its local performance because when the VMs need more memory, it has to first reclaim from cache and then allocate them to the VMs, which increases the response time of the local balancer. Similarly, if a spike of memory demand on the local host occurs, reclaiming all the needed memory from the cache is unwise.

In our design, we take a simple heuristic approach. That is, in each balancing interval, we only increase the cache size by half of the available memory or decrease its size by at most half.

Table 5.1

Migration time estimation

VM Mem (MB)	Program	EMT (sec)	Actual (sec)
214	Eclipse	3.14	3.38
214	186.crafty	1.90	2.42
500	Eclipse	5.65	6.31
500	JBB (8 wh)	5.73	5.46
1200	JBB (8 wh)	64.37	68.97
1200	186.crafty	13.25	12.71

5.3 Experimental Results

Based on the experimental settings described in Section 4.2, we add a machine as the global memory balancing arbitrator. Besides, a communication component runs on each host, sending its local information to the global balancer and receiving migration commands. We first evaluate the effectiveness of migration-based global balancing, followed by cache-based balancing and their combination.

5.3.1 Evaluation of Migration Based Global Balancing

As discussed in Section 5.1.1, migration time estimation plays an important role in migration decision. In this section, we first evaluate the accuracy of migration time estimation. Then, we design various scenarios to evaluate the effect of migration based global memory balancing.

5.3.1.1 Accuracy of Migration Time Estimation

We evaluate the accuracy of migration time estimation with different memory allocation sizes and different types of benchmarks. First of all, a VM is configured with three different memory allocation sizes, 214 MB, 500 MB, and 1200 MB, that represent low, medium and high memory allocation, respectively. Then, for each memory allocation size, we run two of the three programs `186.crafty` from SPEC CPU 2000, `Eclipse` from DaCapo suite and SPEC JBB 2005. `186.crafty` has few memory writes. `Eclipse` and SPEC JBB (configured with 8 warehouses) represent the programs with moderate page dirty rate. Table 5.1 lists the testing results.

As the experimental results reveal, when the memory size is no more than 500 MB, the migration time is proportional to the VM's allocated memory size. However, when the VM's memory size increases to 1200 MB, memory write activities significantly affect migration time. For the VM that runs `186.crafty`, its migration time is still proportional to its memory size. When it runs SPEC JBB, its migration time increases by 4 times though the memory size is just roughly doubled. Our algorithm is able to catch this behavior accurately. The mean error is 8.34%, which is sufficient to support a heuristic-based migration decision.

5.3.1.2 Using Migration For Global Memory Balancing

We create five types of experiments to simulate different scenarios which cover both simple cases and complicated situations.

5.3.1.2.1 Scenario 1: Short Burst of Memory Demand In real situation, it is not uncommon to experience a short surge of memory demand. Depending on the length of the period the surge lasts, different balancing policies may respond differently and eventually bring different performance. To study the performance impacts of various spike lengths and balancing policies, we run SPEC JBB with 3, 12, and 3 warehouses alternatively. With this setting, we create a memory demand burst when the warehouse number is 12. This running pattern is denoted as $A - B - A$. Let phase A last for 120 seconds, and B last for 15, 30, 60 and 90 seconds, respectively, to simulate different lengths of spikes. We use two machines, M_1 and M_2. M_1 and M_2 are limited to use 800 MB and 1200 MB of memory for all its VMs, respectively. Initially, M_1 hosts one VM that runs the SPEC JBB and M_2 is initially idle as a potential migration destination.

We compare our migration cost based algorithm (denoted as *CS*) that is discussed in 5.1.1 with two other strategies: *immediate migration*, denoted as *IM*, which immediately initiates migration once memory is insufficient and *tolerating strategy*, denoted as *TS*, which always tolerates memory demand burst. These two strategies represent the two ends of the spectrum of migration decision, respectively.

Figure 5.2 illustrates the normalized throughputs of phase B against the tolerating strategy. When phase B lasts for 15 seconds, only with *IM*, the VM is migrated to M_2. In this case, the throughput of using migration is merely one sixth of that of non-migration.

On the contrary, when phase B lasts for 60 seconds, *IM* wins. Using *CS*, migration is initiated but after about 14 seconds, a half of the estimated migration time, since the memory pressure starts, which leads to a 25% performance loss compared to *IM*. Nonetheless, the performance loss of *CS* is roughly half of that of tolerating strategy, which is 53%. The results are similarly when phase B lasts for 90 seconds. In other words, in these cases, the
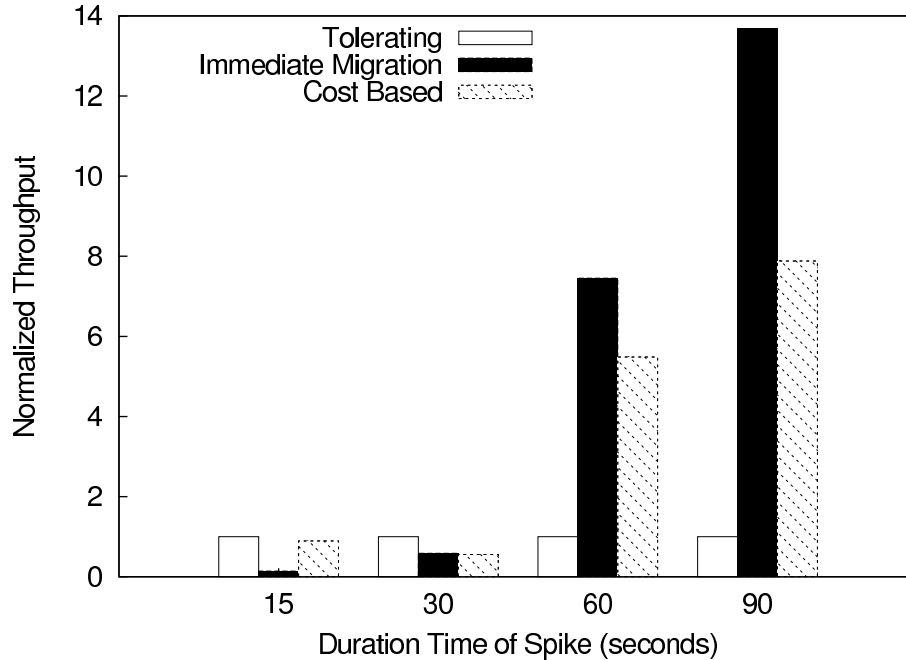
Figure 5.2: Performance of SPEC JBB with 12 warehouses using migration

performance of our cost based heuristic policy is just at the mid-point between the best and worst cases, meeting the criterion of our heuristic algorithm.

When phase B lasts for 30, migration occurs for both *IM* and *CS* and the performance of *TS* is slightly better than that of migration.

This experiment reveals that fixed policies like *IM* and *TS* can achieve optimal performance in some cases but can also lead to the worst performance in other cases, while *CS*, which uses dynamic information, can achieve a near optimal performance or a performance within a range of the optimal. In the following experiments, we always uses the cost based heuristic policy (*CS*).

5.3.1.2.2 Scenario 2: Balancing Two VMs   This is a simple case in which one machine hosts two VMs, and during execution, one VM's memory requirement increases and exceeds the total available memory on the host.

We let machine M_1 hosts two VMs, one runs `186.crafty`, the other runs DaCapo benchmark suite. Each host is configured to use at most 500 MB of memory for all guests. Initially, M_1 hosts two VMs, which evenly share the 500 MB of memory, and M_2 hosts no VM. We set the baseline case as the execution time with fixed 250 MB of memory without memory usage monitoring and memory balancing. For comparison, we also measure the performance using local-only memory balancing.

70

Figure 5.3: Performance comparison: DaCapo and 186.crafty

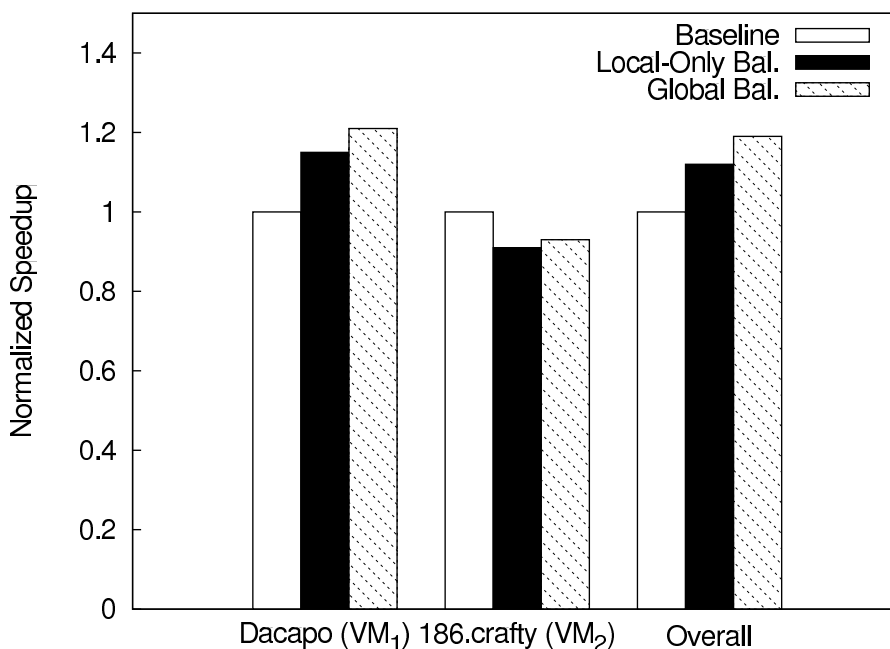Figure 5.3 shows the normalized speedups against the baseline case. Though local memory balancing improves the overall performance by 12%, using migration can further improve it to 19%. Most performance gain is from migrating `Eclipse` in DaCapo which enjoys the large available physical memory on M_2 after migration.

5.3.1.2.3 Scenario 3: Balancing Two VMs with Long Migration Time

As shown in Section 5.3.1.1, when a VM's memory allocation size is large and its memory activity is intensive, the migration time may be significant. To measure how migration affects performance, we let machine M_1 and M_2 use 1600 MB and 800 MB of maximum memory for their VMs respectively. M_1 hosts two VMs, VM_1 and VM_2, each of which is assigned with 800 MB of memory by default. VM_1 runs JBB with variable number of warehouses (1-8-1), and VM_2 runs JBB constantly with 8 warehouses.

When the VMs start to run, the local memory balancer gradually reclaims unused memory from VM_1 and assigns them to VM_2. Later on, when VM_1 reaches 8-warehouse stage, the total memory requirements on the machine exceed 1600 MB. Since VM_1's memory is previously reduced to 140 MB when its warehouse number is 1, the global balancer selects VM_1 for its shorter EMT and migrates it to M_2, which takes 20 seconds to finish. After that, both VMs can use the memory on their hosts exclusively. Table 5.2 lists the normalized speedups based on throughput against the baseline case (fixed 800 MB memory allocation for each VM, no memory monitoring). Using global balancing, the overall throughput improvement is 10%. Using local-only balancing, the overall improvement is merely 2%.

Table 5.2
Speedups of SPEC JBB

		Speedups	
VM	Warehouses	Local balance	Local balance + migration
	1	1.06	0.99
1	8	0.95	0.92
	1	1.01	1.10
2	8	1.08	1.45
	Mean	1.02	1.10

5.3.1.2.4   Scenario 4: Balancing Six VMs / Two Hosts   In this scenario, we simulate a more complex situation. We still use 2 machines: M_1 and M_2. Instead of letting one machine be idle, both machines host 3 VMs and each VM runs different workloads. On M_1, SPEC Web, 4 SPEC 2000 Integer programs (`175.gcc`, `176.vpr`, `197.parser`, `300.twolf`) and DaCapo suite are run on the three VMs respectively. On M_2, the three VMs run 6 SPEC 2000 FP programs (`172.mgrid`, `177.mesa`, `178.galgel`, `179.art`, `183.equake`, `200.sixtrack`), SPEC JBB, and `186.crafty`, respectively. The number of warehouses of SPEC JBB increases from 1 to 4 evenly. DaCapo runs for 3 iterations and 186.crafty runs for 24 iterations. Initially, all the six VMs are equally allocated with 250 MB of memory. These workloads are selected such that they have various memory usage patterns and take nearly the same time to finish without balancing. In this situation, there is no free memory without actively reclaiming unused memory through local balancing.

Table 5.3 presents the speedups for both local-only and global balancing against the baseline setting (250 MB fixed memory allocation for each VM). In this setting, since JBB requires a significant amount of memory, when `186.crafty` is migrated from M_2 to M_1, JBB receives more memory and achieves a speedup of 2.65. Compared with local balancing only, combining with migration brings 16% extra speedup.

5.3.1.2.5   Scenario 5: Balancing Six VMs / Three Hosts   To further evaluate the performance of global balancer, we introduce a third machine into our experimental system. Machine 1 and 2 are the same ones as used in previous scenarios. Machine 3 is equipped with one Intel Core i7 processor and 8 GB of 800 MHz DDR2 memory. Initially, Each machine hosts two VMs and each VM is allocated with 214 MB of memory. We limit each host to use only 428 MB of memory in total for all its VMs.

The workload on each VM is listed in Table 5.4, as well as the speedups for both local-only and global balancing against baseline setting (fixed 214 MB memory allocation for each

Table 5.3

Speedups of using migration for 6 VMs on 2 hosts

VM	Program	Local balance	Local balance + migration
1	SPEC Web	1.140	1.195
2	SPEC INT	0.817	0.881
3	DaCapo	3.506	2.900
4	SPEC FP	0.975	0.868
5	SPEC JBB	1.148	2.917
6	186.crafty	1.000	0.993
	Mean	1.241	1.402

Table 5.4

Speedups of using migration for 6 VMs on 3 hosts

VM	Program	Local balance	Local balance + migration
1	SPEC Web	0.99	0.88
2	CINT	0.90	0.83
3	DaCapo	1.38	2.07
4	CFP	0.81	0.85
5	JBB	1.42	2.05
6	186.crafty	0.99	0.89
	Mean	1.05	1.15

VM). Briefly, in this scenario, global balancing initiates 9 migrations in total, which brings an overall speedup of 15%.

Below we list the details of the migrations: (1) M_3 first experiences memory pressure due to significant demand from SPECJBB on VM_5. As a result, VM_6 is migrated to M_2, which enables VM_5 to use all the 428 MB memory of M_3. (2) Later on, M_2 suffers from memory shortage, while the local balancer on M_1 reclaims some memory, VM_6 is then migrated from M_2 to M_1. When SPEC JBB finishes, it makes M_3 to be idle. (3) After that, since VM_3 (DaCapo) on M_2 requires more memory, VM_3 is migrated to M_3. (4) Next, M_1 suffers from memory pressure, which triggers the migration of VM_6 from M_1 to M_3. (5) With the further increasing demand from VM_3 (DaCapo), it makes VM_6 migrated to M_2 because at that time M_2 has enough free space to host VM_6. After the termination of `Eclipse`, the memory demand of VM_3 drops, which makes M_3 a migration target again. More migrations happen afterward.
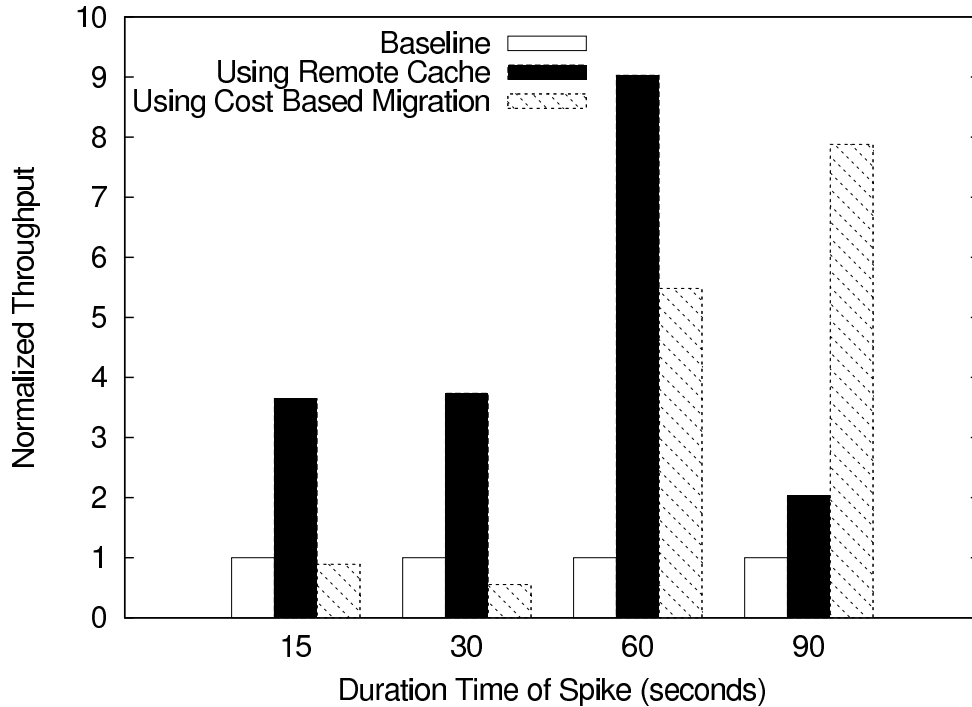
Figure 5.4: Throughputs comparison of SPEC JBB with 12 warehouses

5.3.2 Evaluation of Remote Cache Based Global Memory Balancing

5.3.2.1 Penalty Alleviation For Memory Spikes

To evaluate the effects of remote caching in alleviating performance loss of memory spikes, we again use `SPEC JBB` to create various lengths of memory demand spikes as we do in the Scenario 1 of Section 5.3.1.2. The VM that runs SPEC JBB is allocated with 800 MB memory, and the dedicated cache server is allocated with 400 MB page pool. Thus, the total memory including the cache is 1200 MB, the same size as the target VM of the migration used in Scenario 1 of Section 5.3.1.2.

Figure 5.4 shows the normalized throughputs of using remote cache against the baseline setting. For comparison, we also show the performance of using migration based balancing from Section 5.3.1.2.

Clearly, using remote cache significantly boosts performance. It increases the throughput by 8 times. Compared with using migration based balancing, when the duration of memory pressure is 60 seconds, remote caching has better performance. While when the duration of the spike is as long as 90 seconds, using migration is a better choice because the performance gap between native memory accesses and network I/Os far exceeds its migration cost.

74

Table 5.5
DaCapo + DaCapo': performance of using remote cache

Host / Program	Speed Up	Cache Hit Ratio	Speed Up of Local Bal.
VM_1 / DaCaPo	1.93	89.83%	3.31
VM_2 / DaCapo'	1.69	85.05%	3.79

5.3.2.2 Symmetrical Remote Caching

In this experiment, we use two hosts, each of them runs a VM and a cache server and uses each other's caching service. Meanwhile, the local memory balancer on each host dynamically adjusts memory allocation for its VM and cache pool. That is, each host uses its idle memory to service the other host. One VM runs `DaCapo` benchmark suite, the other one runs `DaCapo'`, the revered execution order of its programs. Each host is limited to use 250 MB memory for its guest OS and memory server. The baseline setting is using the 250 MB memory statically without any balancing technique.

Table 5.5 presents the speed ups for using remote cache and the cache hit ratios. For comparison, we also list the speedups of locally balancing the two VMs on a single host from Section 4.2.1.3.

It is not surprising that, given the same amount of total memory resources, the performance of using remote caching is inferior to that of local memory balancing. After all, the latency of a Gigabit Ethernet is more than 100 times higher than that of native memory accesses. However, it still gains a mean speedup of 1.81 against the baseline setting, which achieves its design purpose as a mitigation of page swapping.

5.3.2.3 Multiple Cache Clients

To evaluate the performance of a unified cache server that services multiple clients, we use four machines, one is used as the server and the other three host one VM each and run `DaCapo`, `DaCapo'` and `186.crafty`, respectively. Each VM is allocated with 250 MB memory. In the baseline setting, remote cache is not used. Using the unified cache server, it allocates 300 MB memory to its page pool. For comparison, we also run 3 instances of the cache service, each of which has 100 MB page pool and services one client dedicatedly.

Table 5.6 lists the speed ups against the baseline setting and cache hit rates. It shows that, using a unified cache server has better cache utilization and results in 1.13 overall speedup, which is 4% higher than that of using dedicated cache.

Table 5.6

Performance of using shared cache server and dedicated cache server

Host / Program	Shared Cache		Dedicated Cache	
	Speed Up	Hit Ratio	Speed Up	Hit Ratio
VM$_1$ / DaCaPo	1.11		1.15	83.59%
VM$_2$ / DaCapo'	1.31		1.14	76.16%
VM$_3$ / 186.crafty	1.00		1.00	0%
Overall	1.13	87.87%	1.09	

5.3.3   Putting All Together

In this experiment, we measure the performance of using the two global balancing schemes together. We use the same VM settings and benchmarks as we do in Scenario 4 in Section 5.3.1.2.4 as it represents a typical scenario in a data center. The difference is that, on each of the hosts, in addition to the three VMs, it also run a cache server , whose capacity is initially 0. And each host is configured to use the other host as a cache server.

During the running, VM$_6$, the VM that runs `186.crafty`, is migrated from M_2 to M_1 since SPEC JBB creates sustained memory pressure. The cache hit ratio of VM$_1$ and VM$_2$ is 0.904 and 0.784, respectively.

For comparison purpose, we also measure the performance of using local memory balancing augmented with remote caching but no migration based balancing. Table 5.7 lists the results, which also includes the data of Table 5.3 for reference. As the results show, combining local balancing and the two global balancing techniques, it achieves the best performance of 1.488 overall speedup. And we believe that, with faster network, our global balancing can further improve the performance.

5.4   Chapter Summary

In this chapter, we present two global memory balancing techniques: migration based and remote cache based, which are based on our local memory balancing and extend the scope of memory balancing from a single host to center wide.

Our experimental results show that, migration based global memory improves overall performance by up to 40%. When memory pressure occurs, it uses the cost (estimated migration time) based heuristic to decide if it should migrated a VM or not, which gives near optimal or intermediate performance.

Table 5.7
Speedups of global balancing for 6 VMs on 2 hosts

VM	Program	Local Bal.	Local Bal. + Mig.	Local Bal. + Remote Caching	Local Bal. + Mig. + Remote Caching
1	SPEC Web	1.140	1.195	0.978	1.501
2	SPEC INT	0.817	0.881	0.892	0.914
3	DaCapo	3.506	2.900	2.958	2.265
4	SPEC FP	0.975	0.868	0.960	0.948
5	SPEC JBB	1.148	2.917	3.450	3.711
6	186.crafty	1.000	0.993	0.991	0.992
	Mean	1.241	1.402	1.428	1.488

Remote caching effectively fills the gap where migration is not an option. By replacing disk operations with network transportation, a speed up of up to 3.79 is seen.

By combining these two techniques, we achieve an overall speed up of 1.49 for 6 VMs on 2 hosts, which is 24% more than using local balancing only.

Chapter 6

Conclusion

Imbalanced memory resource allocation undermines the effectiveness and efficiency of virtualization both for a single host and for a center-wide environment. This dissertation proposes a scheme to address this problem. We show the promise of our solution by examining each techniques in our scheme and their combination through extensive experiments. This chapter summarizes our contributions and discusses future work.

6.1   Contributions

We propose a unique and complete scheme that increases global memory resource utilization. Our memory demand tracking scheme is the cornerstone of the whole scheme. Based on an LRU miss ratio curve, it not only indicates the current memory demand of a VM, but also predicts performance impacts with respect to various memory allocation sizes, which allows optimizing memory allocation in case of memory resource competition. We successfully reduce the overhead of the online construction of an LRU miss ratio curve with a negligible accuracy loss by applying an AVL-tree based LRU list implementation, dynamic hot set sizing, and a novel intermittent memory tracking scheme. Combining the three optimizing techniques, we reduce the mean overhead of SPEC CPU 2006 from 173% to 2% while the accuracy loss is less than 4%. We believe that, in addition to memory resource balancing, this memory demanding tracking scheme can be adapted for other applications, such as power management, heap size management for JVM, etc.

Based on the memory demand tracker, we propose a local memory resource balancing scheme, which includes a quick memory arbitration algorithm to resolve memory resource competition. Using the local memory balancer, we observe a factor of 25 for page fault reduction and a speedup of up to 11 times. Even when 4 VMs frequently compete for memory, our scheme still achieves an overall speedup of 1.72.

79

We further extend the scope of balancing to a whole data center. The global memory balancing scheme utilizes the free memory that are reclaimed by our local memory balancer to serve other hosts in the same date center. We use two approaches to improve global memory resource utilization: live migration and remote caching. To utilize the heavyweight migration, we propose a heuristic based algorithm that predicts if a migration is beneficial or not. Experimental results show that our algorithm makes near optimal decision and boosts overall performance by up to 19%. Since migration is costly and unable to utilize small amount of free memory, we introduce a resizable remote cache to alleviate temporary memory pressure. When a local host has free memory, it uses them to cache disk I/Os of remote hosts and thus alleviates their performance penalty due to page faults. We observe an 81% performance gain when remote caching is used.

Combining the two global memory balancing scheme, the utilization of global memory resources is further improved. Our experiments show that, their combinations achieves 8% and 6% more performance gain than using migration based balancing alone and remote caching based balancing alone, respectively.

6.2   Future Work

Our future interests include applying the low cost page-level miss ration curve construction scheme to guide power saving for memory systems and cache-level miss ratio construction.

Memory balancing and QoS is an interesting research area too. Currently, we use preset weights to prioritize memory allocation. It is desirable to automatize this process for given QoS goals or service level agreement.

We are also interested in developing more accurate prediction model of memory demand trend and exploring more design options for remote caching such as distributed cache, reliable write back cache.

We also expect our memory balancing scheme to improve process-level memory management for a regular operating system and cluster computing.

References

[1] Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. 2006;(ASPLOS-XII):2–13.

[2] Aho AV, Denning PJ, Ullman JD. Principles of optimal page replacement. Journal of The ACM. 1971;18:80–93.

[3] Alpern B, Augart S, Blackburn SM, Butrico M, Cocchi A, Cheng P, Dolby J, Fink S, Grove D, Hind M, McKinley KS, Mergen M, Moss JEB, Ngo T, Sarkar V, Trapp M. The Jikes Research Virtual Machine project: Building an open source research community. IBM Systems Journal. May 2005;44(2).

[4] Alpern B, Attanasio CR, Cocchi A, Lieber D, Smith S, Ngo T, Barton JJ, Hummel SF, Sheperd JC, Mergen M. Implementing jalapeño in java. In Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 1999;(OOPSLA '99):314–324.

[5] Alpern B, Attanasio CR, Barton JJ, Burke MG, Cheng P, Choi J, Cocchi A, Fink SJ, Grove D, Hind M, Hummel SF, Lieber D, Litvinov V, Mergen MF, Ngo T, Russell JR, Sarkar V, Serrano MJ, Shepherd JC, Smith SE, Sreedhar VC, Srinivasan H, Whaley J. The jalapeño virtual machine. IBM Systems Journal. 2000;39:211–238.

[6] AMD . AMD64 Virtualization Codenamed "Pacifica" Technology. Advanced Micro Devices; May 2005.

[7] Arnold M, Fink SJ, Grove D, Hind M, Sweeney P. 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. October 2000:47–65.

[8] Bai J. Estimation of a change point in multiple regression models. The Review of Economics and Statistics. November 1997;79(4):551–563.

[9] Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. SIGOPS Operating Systems Review. 2003;37(5):164–177.

[10] Blackburn SM, Garner R, Hoffman C, Khan AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Phansalkar A, Stefanović D, VanDrunen T, Dincklage D, Wiedermann B. The DaCapo benchmarks: Java benchmarking development and analysis. In Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications. 2006:169–190.

[11] Chahal S, Glasgow T. Memory sizing for server virtualization. Intel Information Technology White Paper. 7 2007. Available from: `http://www.intel.com/it/pdf/memory-sizing-for-server-virtualization.pdf`.

[12] Chandra D, Guo F, Kim S, Solihin Y. Predicting inter-thread cache contention on a chip multi-processor architecture. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture. 2005:340–351.

[13] Chandra D, Guo F, Kim S, Solihin Y. 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. 2005:340–351.

[14] Chen H, Wang X, Wang Z, Wen X, Jin X, Luo Y, Li X. Remoca: Hypervisor remote disk cache. In 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications. August 2009:161 –169.

[15] Clark C, Fraser K, Hand S, Hansen JG, Jul E, Limpach C, Pratt I, Warfield A. Live migration of virtual machines. In NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation. 2005:273–286.

[16] Dawson P. Top virtualization trends you can't afford to ignore [Internet], Gartner; October 2010. Available from: `http://www.gartner.com/it/content/1434700/1434716/october_28_top_virtualization_trends_pdawson.pdf`.

[17] Denning PJ. The working set model for program behavior. Communications of The ACM. May 1968;11:323–333.

[18] Denning PJ. Working sets past and present. IEEE Transactions on Software Engineering. 1980;SE-6(1).

[19] Dhodapkar AS, Smith JE. Managing multi-configuration hardware via dynamic working set analysis. In Proceedings of the 29th annual international symposium on Computer architecture. 2002;(ISCA '02):233–244.

[20] Flouris MD, Markatos EP. The network ramdisk: Using remote memory on heterogeneous nows. Cluster Computing. October 1999;2:281–293.

[21] Fotheringham J. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. Communications of The ACM. October 1961;4:435–436.

[22] Gove D. CPU2006 working set size. SIGARCH Computer Architecture News. 2007;35(1):90–96.

[23] Gu D, Yang L, Welch L. A predictive, decentralized load balancing approach. In Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International. April 2005:131b–131b.

[24] Hansen D, Kravetz M, Christiansen B. Hotplug memory and the linux vm. In Linux Symposium. 2004:287–294.

[25] Hawkins DM. Fitting multiple change-point models to data. Computational Statistics & Data Analysis. September 2001;37(3):323–341.

[26] Hines MR, Gopalan K. Memx: supporting large memory workloads in xen virtual machines. In Proceedings of the 2nd international workshop on Virtualization technology in distributed computing. 2007;(VTDC '07):2:1–2:8.

[27] HP . HP SAS enterprise and SAS midline hard drives [Internet]; 2011. Available from: `http://h18000.www1.hp.com/products/servers/proliantstorage/serial/sas/index.html`.

[28] Intel Corporation . Intel 64 and IA-32 Architectures Software Developer's Manual. Specification, 2011.

[29] International Data Corporation . Worldwide market for enterprise server virtualization to reach $19.3 billion by 2014, according to idc [Internet]. Press Release; 12 2010. Available from: `http://www.idc.com`.

[30] Jones ST, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Geiger: monitoring the buffer cache in a virtual machine environment. SIGOPS Operating Systems Review. 2006;40 (5):14–24.

[31] Karlin A, Manasse M, Rudolph L, Sleator D. Competitive snoopy caching. Algorithmica. 1988;3:79–119.

[32] Karlin AR, Manasse MS, McGeoch LA, Owicki S. Competitive randomized algorithms for non-uniform problems. In Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms. 1990;(SODA '90):301–309.

[33] Liang S, Noronha R, Panda DK. Swapping to remote memory over infiniband: An approach using a high performance network block device. In International Conference on Cluster Computing (CLUSTER '05). 2005.

[34] Lu P, Shen K. Virtual machine memory access tracing with hypervisor exclusive cache. In Proceedings of the USENIX Annual Technical Conference. 2007:1–15.

[35] Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. 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. 2005:190–200.

[36] Magenheimer D. Memory overcommit. . . without the commitment. In Extended Abstract for Xen Summit. 2008. Available from: `http://www.xen.org/files/xensummitboston08/MemoryOvercommit-XenSummit2008.pdf`.

[37] Markatos EP, Dramitinos G. Implementation of a reliable remote memory pager. In Proceedings of the 1996 annual conference on USENIX Annual Technical Conference. 1996:15–15.

[38] Mattson RL, Gecsei J, Slutz D, Traiger IL. Evaluation techniques for storage hierarchies. IBM System Journal. 1970;9(2):78–117.

[39] Mergen MF, Uhlig V, Krieger O, Xenidis J. Virtualization for high-performance computing. SIGOPS Operating Systems Review. April 2006;40:8–11.

[40] Nagpurkar P, Krintz C, Hind M, Sweeney PF, Rajan VT. Online phase detection algorithms. In Proceedings of the International Symposium on Code Generation and Optimization. 2006;(CGO '06):111–123.

[41] Newhall T, Finney S, Ganchev K, Spiegel M. Nswap: A network swap module for linux clusters. In Proceedings of the 9th European Conference on Parallel Processing (Euro-Par). 2003:1160–1169.

[42] Pinter SS, Aridor Y, Shultz S, Guenender S. Improving machine virtualization with 'hotplug memory'. In Proc. 17th International Symposium on Computer Architecture and High Performance Computing SBAC-PAD 2005. 24–27 Oct. 2005:168–175.

[43] Romer TH, Ohlrich WH, Karlin AR, Bershad BN. Reducing tlb and memory overhead using online superpage promotion. SIGARCH Computer Architecture News. 1995;23(2):176–187.

[44] Sapuntzakis CP, Chandra R, Pfaff B, Chow J, Lam MS, Rosenblum M. Optimizing the migration of virtual computers. In In Proceedings of the 5th Symposium on Operating Systems Design and Implementation. 2002:377–390.

[45] Seiden SS. A guessing game and randomized online algorithms. In Proceedings of the thirty-second annual ACM symposium on Theory of computing. 2000;(STOC '00):592–601.

[46] Shen X, Zhong Y, Ding C. Locality phase prediction. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems. 2004.

[47] Sherwood T, Perelman E, Calder B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques. 2001;(PACT '01):3–14.

[48] Sherwood T, Perelman E, Calder B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques. 2001;(PACT '01):3–14.

[49] Sherwood T, Sair S, Calder B. Phase tracking and prediction. In Proceedings of the 30th International Symposium on Computer Architecture. 2003.

[50] Shirazi BA, Hurson AR, Kavi KM. Scheduling and Load Balancing in Parallel and Distributed Systems. IEEE Computer Society; 1995.

[51] Sugumar RA, Abraham SG. Efficient simulation of caches under optimal replacement with applications to miss characterization. In Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems. May 1993:24–35.

[52] Tam DK, Azimi R, Soares LB, Stumm M. RapidMRC: Approximating l2 miss rate curves on commodity systems for online optimizations. In Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. 2009:121–132.

[53] Uhlig R, Neiger G, Rodgers D, Santoni A, Martins F, Anderson A, Bennett S, Kagi A, Leung F, Smith L. Intel virtualization technology. Computer. may 2005;38(5):48 – 56.

[54] Waldspurger CA. Memory resource management in VMware ESX server. SIGOPS Operting Systems Review. 2002;36(SI):181–194.

[55] Werstein P, Jia X, Huang Z. A remote memory swapping system for cluster computers. In Parallel and Distributed Computing, Applications and Technologies, 2007. PDCAT '07. Eighth International Conference on. dec. 2007:75 –81.

[56] Whitaker A, Shaw M, Gribble SD. Denali: Lightweight virtual machines for distributed and networked applications. In In Proceedings of the USENIX Annual Technical Conference. 2002.

[57] Williams D, Jamjoom H, Liu YH, Weatherspoon H. Overdriver: handling memory overload in an oversubscribed cloud. In Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2011;(VEE '11):205–216.

[58] Wood T, Shenoy P, Venkataramani A, Yousif M. Black-box and gray-box strategies for virtual machine migration. In Proceedings of the 4th USENIX conference on Networked systems design & implementation. 2007;(NSDI'07):17–17.

[59] Yang T, Hertz M, Berger ED, Kaplan SF, Moss JEB. Automatic heap sizing: taking real memory into account. In Proceedings of the 4th international symposium on Memory management. 2004;(ISMM '04):61–72.

[60] Yang T, Berger ED, Kaplan SF, Moss JEB. CRAMM: virtual memory support for garbage-collected applications. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation. 2006:103–116.

[61] Zayas E. Attacking the process migration bottleneck. SIGOPS Operating Systems Review. 1987;21(5):13–24.

[62] Zeileis A, Kleiber C, Kramer W, Hornik K. Testing and dating of structural changes in practice. Computational Statistics & Data Analysis. 2003;44:109–123.

[63] Zhang X, Dwarkadas S, Shen K. Towards practical page coloring-based multi-core cache management. In Proceedings of the 4th ACM European Conference on Computer systems. 2009.

[64] Zhao W, Wang Z. Dynamic memory balancing for virtual machines. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2009:21–30.

[65] Zhao W, Jin X, Wang Z, Wang X, Luo Y, Li X. Low cost working set size tracking. In Proceedings of the 2011 annual conference on USENIX Annual Technical Conference. 2011.

[66] Zhao W, Jin X, Wang Z, Xiaolin W, Yingwei L. Efficient LRU-based working set size tracking. Technical Report CS-TR-11-01, Houghton, MI, USA, 2011.

[67] Zhou P, Pandey V, Sundaresan J, Raghuraman A, Zhou Y, Kumar S. Dynamic tracking of page miss ratio curve for memory management. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems. 2004:177–188.