Simulation of Memory Management in Virtual Machines

Hayley Wu, Shuye Wu and Wei Zhang
COEN 283: Operating Systems, Spring 2011

Table of Contents

Abstract

Virtualization has gained widespread usage over the past decade due to the need for better utilization of idle hardware resources. In virtualized systems, hardware resources are multiplexed and shared across one or more virtual machines (VMs) containing its own operating system. Idleness in hardware such as CPUs and I/O devices have gotten more utilization out of virtualization, however, this has not been the norm for main memory. In this project, we explored the benefits of statically vs. dynamically allocating memory in concurrent VMs by simulation through multi-threaded code. A main thread was used as the virtual machine monitor (VMM) and multiple threads were used as the VMs. Page sequences implemented as linked lists and subjected to the least recently used (LRU) page replacement algorithms were fed to the VMs. Analysis of the outputs showed that dynamically allocating memory to the VMs produced less page faults for the two test cases we used, however the static memory implementation was far simpler, which is a desirable characteristic when interfacing with hardware. This exercise, thus, shows the promise and future trend of dynamically allocating memory across virtualized systems.

Introduction

Virtual machines were first introduced in the 1960's, the motivation then was to multiplex scarce resources of main frame computers. In the past decade or two, as hardware costs have decreased dramatically while computing performance has improved dramatically, interest in taking advantage of underutilized computing resources have been rekindled. In virtualization, the illusion of several smaller machines is provided. This is achieved via a hypervisor (or VMM), the hypervisor provides multiplexing of hardware resources and manages guest OSes. VMMs are generally seperated into type I and type II. In type I, the hypervisor sits directly on top of the hardware and controls guest OSes, in type II, the hypervisor sits on top of a host operating system and does not have direct access to the hardware. Figure 1 illustrates the difference between type I and type II hypervisors.

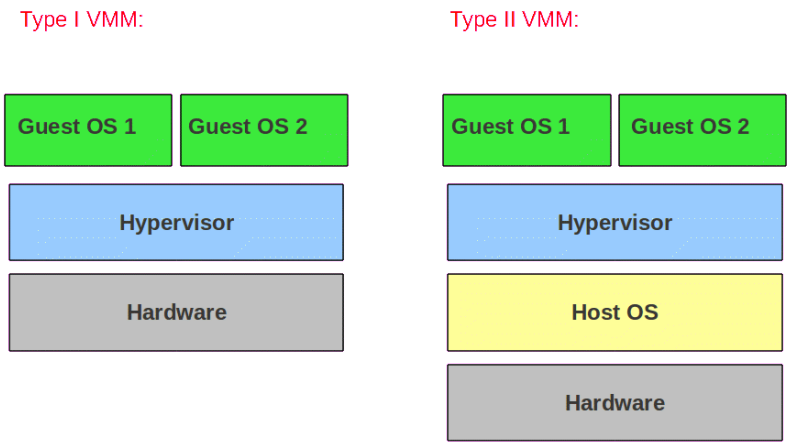

Figure 1: Type I vs. Type II hypervisor

In most commodity virtualized systems, CPUs are assigned to guest OSes on demand while memory is assigned statically (each VM is alloted a certain amount of main memory during configuration/setup). For example, given a system with 1 GB of memory and 10 VMs, each VM will be allocated 100 MB. It is clear that such a scheme will incur memory idleness for a VM that is mostly idle. Recently, a few virtualization products such as Xen and VMware ESX have introduced dynamic memory allocation to reclaim idle memory from one idle VM to other memory-needy VM(s). In dynamically memory managed VMMs, memory balancing is required.

We seek to investigate and understand the advantages and disadvantages of static vs. dynamic memory allocation in virtualized systems. We do so by simulating memory management of a hypervisor. This simulated hypervisor will hold more than one guest OS(es) and will allocate memory to these VMs either dynamically or statically. Each VM will be implemented as a process; we simulate a sequence of memory demands (trace) and pass them to each VM and monitor the page miss ratios using LRU page replacement algorithm. A checkpoint happens at the hypervisor for every page that gets sent to the VMs. Figure 2 shows the LRU replacement algorithm implemented as a linked list and Figure 3 illustrates our simulation strategy. For static memory allocation, the number of pages for each VM is fixed, and for dynamic memory allocation, two different strategies are employed. On the first one, page “stealing” occurs between pairs of VMs that page fault and don’t page fault at the latest checkpoint. In the second strategy, a running sum of page faults is computed at every checkpoint, and pages are reallocated based on the proportion of page faults to total

available “bonus pages.” Such balancing is an adaptation similar to Zhao & Wang’s Memory Balancer (MEB)[2] and will be explained further in the Theoretical Basis and Literature section below.

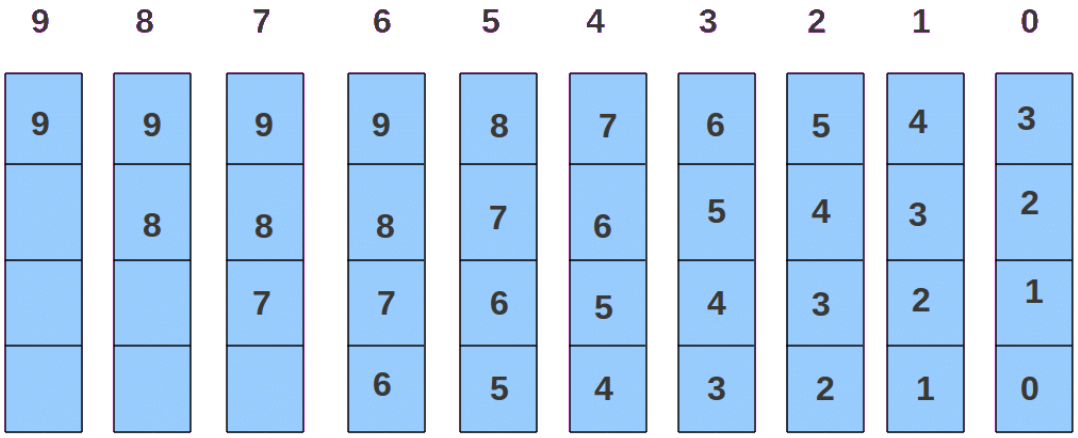

Figure 2: LRU replacement algorithm implemented as a linked list.

This project will cover topics in multiprocessing, virtualization, and memory management. Our side by side simulation, comparison, and analysis of static vs. dynamic memory allocation in VMs will serve as a broad overview/review of current memory management schemes in virtualization; it will also provide insights for future trends in memory management of virtual machines.

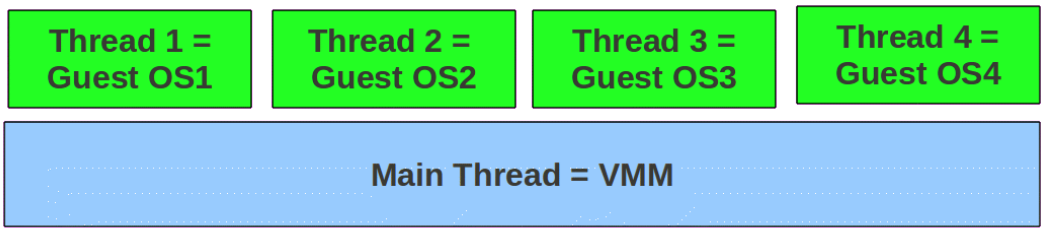

Figure 3: Main thread is used to simulate the hypervisor and child threads are used to simulate guest OSes.

Theoretical Bases and Literature Review

A virtual machine (VM) is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine. A system virtual machine provides a complete system platform which supports the execution of a complete operating system (OS). It allow the sharing of the underlying physical machine resources between different virtual machines, each running its own operating system. The advantages of VMs are visible: multiple OS environments can co-exist on the same computer, in strong isolation from each other. And it can provide an instruction set architecture(ISA) that is somewhat different from that of the real machine. However, a virtual machine is less efficient than a real machine when it accesses the hardware indirectly (as in type II hypervisor). And when multiple VMs are concurrently running on the same physical host, each VM may exhibit a varying and unstable performance (speed of execution), which highly depends on the workload imposed on the system by other VMs. The memory management is intimately tied to the instruction set of the VM. A better design of memory management of the virtual machine makes both the host system and the guest system performance better.

P. Zhou, V. Pandey have already studied the dynamic tracking of page miss ratio curve (MRC) for memory management in a single operating system with multiprogramming[1]. They designed a hardware MRC monitor to dynamically track MRC for applications at run time. The MRC monitor applies well to direct memory allocation to maximize memory utilization in multiprogramming systems.

We will use LRU (Least Recently Used) page algorithm. Though it is similar in name to NRU (Not Recently Used), differs in the fact that LRU keeps track of page usage over a short period of time, while NRU just looks at the usage in the last clock interval. LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory (almost as good as Adaptive Replacement Cache), it is rather expensive to implement in practice.

Wang and Zhao present Memory Balancer (MEB) in Xen, which dynamically monitors memory usage across VMs, predicts memory usage, and reallocates them[2]. Their memory allocation method is based on the working size set and a minimum amount of memory allocation to each VM. A bonus pool of memory can be redistributed to the memory needy VMs.

Barham et al. present Xen[3,7], a high performance resource-managed virtual machine monitor(VMM) which enables applications such as server consolidation, co-located hosting facilities, distributed web services, secure computing platforms and application mobility.

Hwang et al. provide a full-fledged non-obstructive memory transfer scheme enhanced with the reference pattern-based victim selection and hypervisor-level reclamation[5]. His group also proposes HyperDealer, which makes balancing operation free from the involvement of a victim VM and extends the dwell time of reclaimed pages in the reclaimed state. By evaluation, HyperDealer significantly accelerates the balancing operation with a low overhead, thereby increasing the effect of additional memory on the beneficiary VM.

Some high-level resource management policies compute a target memory allocation for each VM based on specified parameters and system load. These allocations are achieved by invoking lower-level mechanisms to reclaim memory from virtual machines. In addition, a background activity exploits opportunities to share identical pages between VMs, reducing overall memory pressure on the system[4].

New Virtual Machine(VM) technology allows computing processes within the same computing node but different VMs to communicate through shared memory pages. In Huang's paper[6], they find a One-copy protocol for Inter-VM shared memory

communication. By using the new protocol, they can save quite a lot data copying time, and raise the VMs performance.

Hypothesis

We hypothesize that the page miss ratio of VMs will be decreased by dynamic memory balancing. On the other hand, if the frequency and page miss ratio of VM's page access are almost the same (equally distributed amongst the VMs), dynamic memory balancing will not decrease the page miss ratio but only increase the overhead.

Methodology

The main program maintains all the memory allocated to VMs as a hypervisor. VMs are simulated by threads or processes. For static page management, the number of pages allocated to each VM is fixed and page replacement follows an LRU algorithm. For dynamic page allocation, at each checkpoint, the hypevisor examines page usage of each VM, then deallocates the unused pages of certain VMs and allocates them to the ones that needs more page. The hypervisor has a table of all the pages and knows the allocation to each VM. Each VM has its own page table and uses LRU algorithm to swap pages. The hypervisor and VMs know the page miss ratio at any time.

The simulation will be implementd in the C programming language and compiled with gcc under the Linux environment.
Under debug mode, the sequences and page misses will be output to the terminal. From these outputs, tables and charts will be generated and used to analzye the advantages and disadvantages of statically vs. dynamically allocating memory to virtual machines.

The readme.txt file contains instructions on how to compile and run the code. To obtain all outputs, run "make debug", then call the program by ./main test<n>.dat, where n is 1,2,or 3. Refer tot he readme.txt file for more information.

Implementation

Methods and Headers

main.c, page.h: Initializes the program. Contains the main thread (the hypervisor), creates threads based on the number of VMs. Semaphores are used to protect shared data. Gets the strategy as input from the user. page.h contains the linked list structure of the pages.

helper.c, helper.h: Reads the page sequence from test<n>.dat
VM.c, VM.h: Accesses page tables.
strategies.c, strategies.h: Contains the dynamically balancing page management schemes.

Test Data: These are text files containing page sequences and number of VMs.
test1.dat
4 VMs with the following trace:
10 9 8 7 6 5 4 3 2 1 0
10 9 8 7 6 9 8 7 6 9 8
10 9 8 7 6 9 8 7 6 5 4
10 9 8 7 6 5 4 3 2 1 0

test2.dat
2 VMs with the following trace:
15 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

test3.dat
3 VMs with the following trace:
12 0 1 2 3 0 3 6 9 4 8 6 4
12 2 4 2 9 8 7 9 8 0 3 1 5
12 3 5 6 7 8 9 3 4 5 6 0 1

Static memory allocation
Each VM is distributed with an equal amount of pages. And page sequences are sent to the VMs. Page faults will occur based on the fixed amount of pages allocated following the LRU page replacement algorithm.

For dynamic balancing, we use two strategies to manage pages between virtual machines, these are described below.

Dynamic balancing based on last n page access
The hypervisor calculates the page miss ratios of each VM for every n page accessed. At the checkpoint, the hypervisor reads the last n page access records of one VM to find how many page misses exit, and then get the page miss ratio of those n accesses of this VM. If VM-A has more page misses than VM-B in last n accesses, whicfh means VM-A's page miss ratio is bigger than VM-B's, the hypervisor reclaims one page from VM-B and assign it to VM-A, vice versa.

Dynamic balancing based on proportion of total missed pages

The hypervisor can compute the historical total number of page misses at each checkpoint. Each VM is allowed to have a minimum amount of pages it can operate on, so that there is a total amount of bonus points that can be used by all other VMs. The total bonus pages are divided based on the proportion of page misses of the VMs with the ones getting the most misses getting a bigger share of the bonus pages, the minimum amount of pages each VM can have is fixed.

Data analysis and Discussion

IIn order to test our new Dynamic balancing algorithm, we first create a simple test file as the input data for the pages. Then we read in the data and store it into the pages.

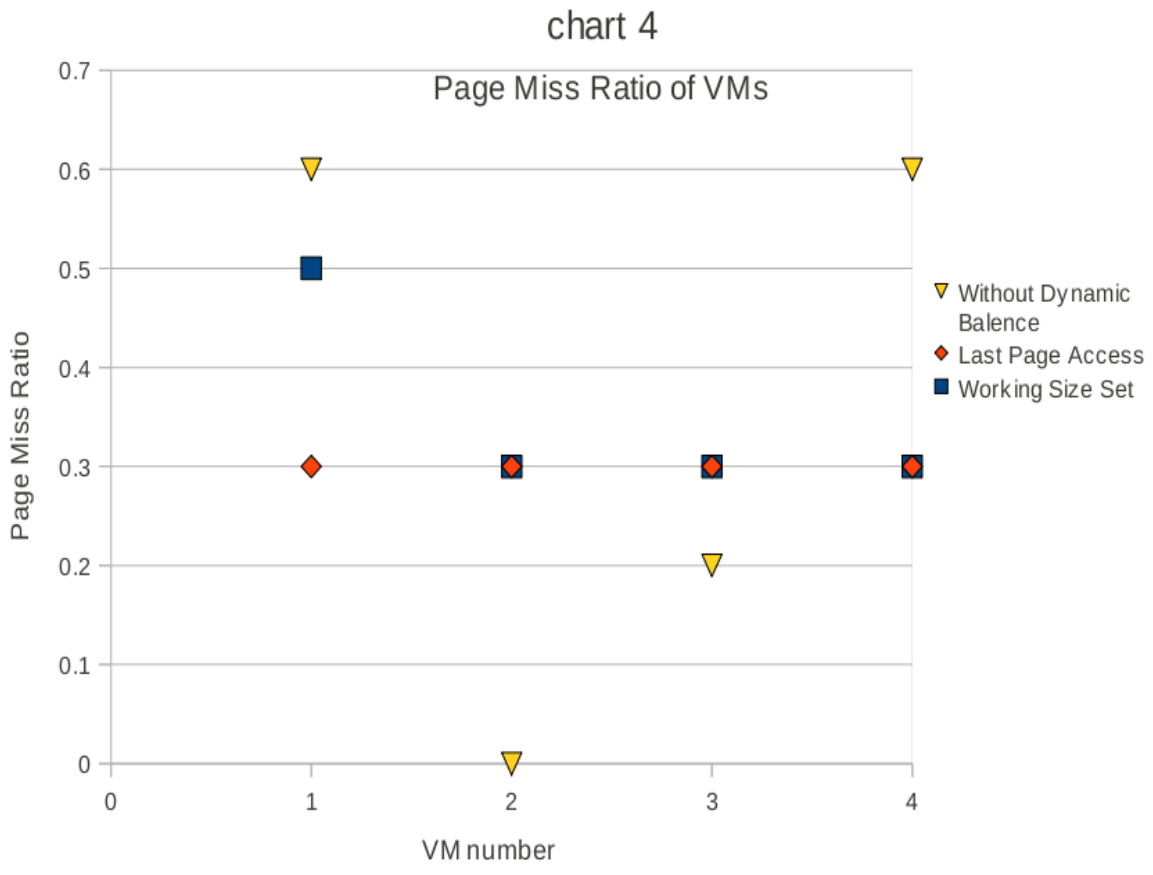

Figure 4: page miss ratios for three different memory allocation strategies across four VMs using test file *test1.dat*

Whenever there is a page miss, a “1” will be generated for true. Figure 4 shows the page miss ratio for “test1.dat” in our project. Three different color represents the three different strategies in our design. Yellow is for the static strategy, red is for the dynamic balancing based on last n page access, and blue is for the dynamic balancing based on working size set or total number of missed pages.

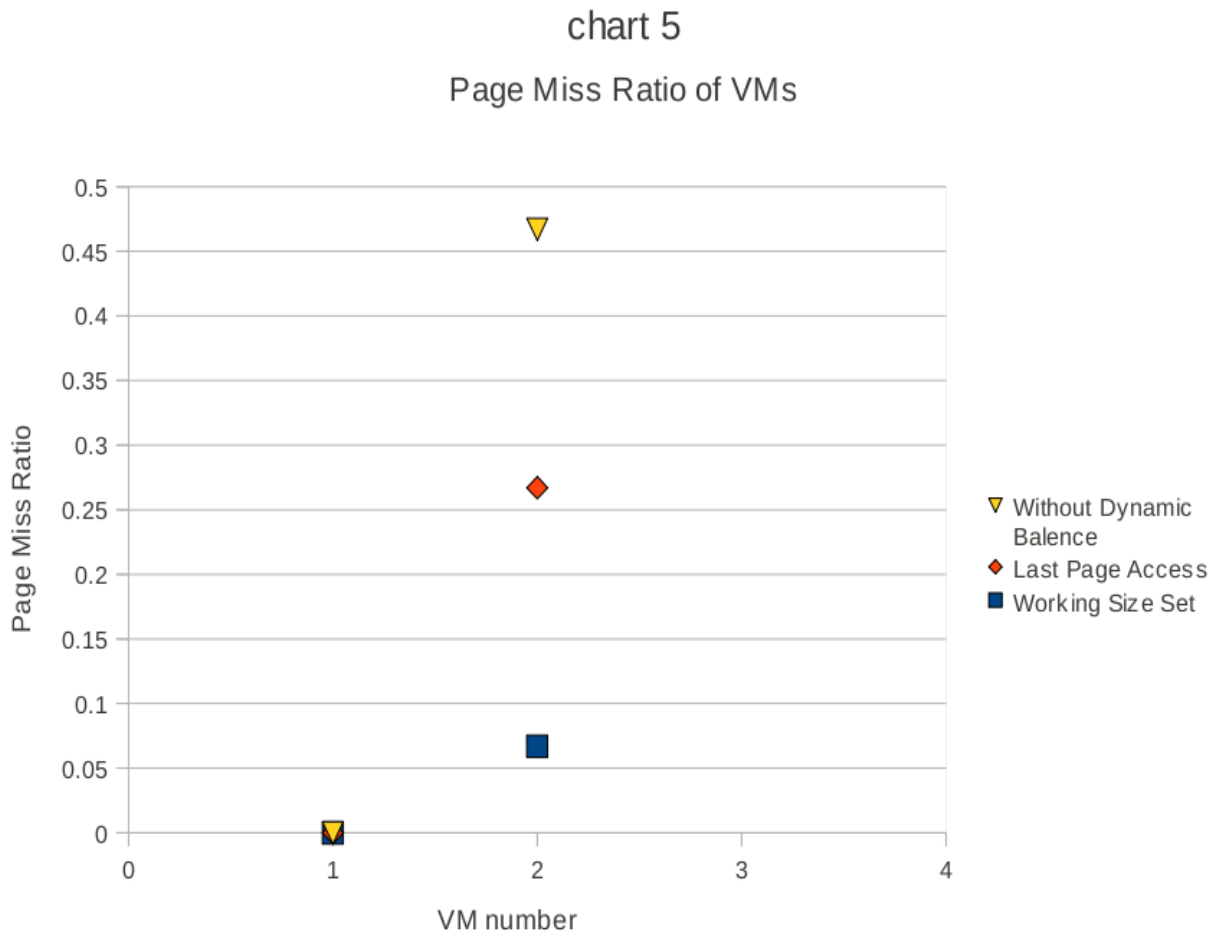

Figure 5: page miss ratios for three different memory allocation strategies across 2 VMs using test file *test2.dat*

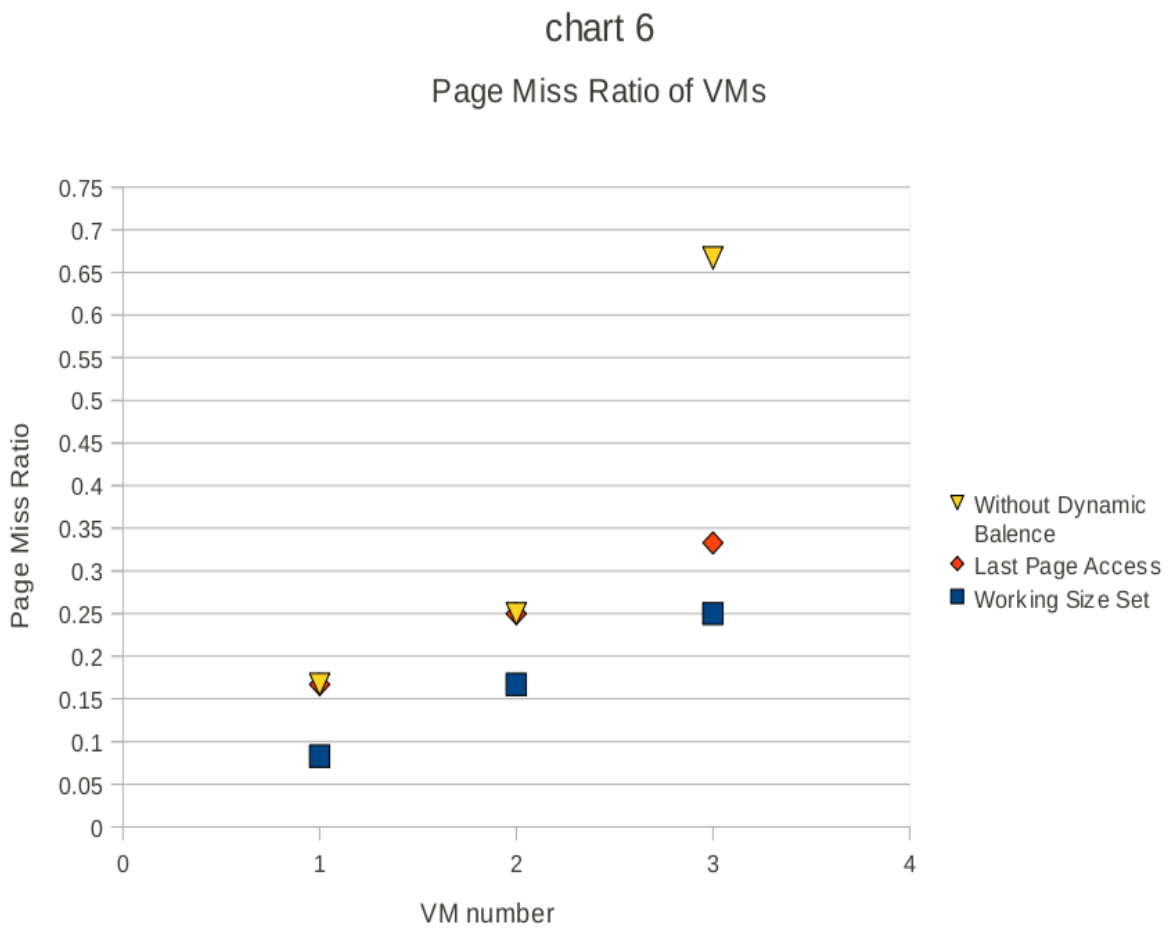

Figure 6: page miss ratios for three different memory allocation strategies across 3 VMs using test file *test3.dat*

Total Page Miss for the Three Strategies

static strategy	0.321
dynamic balance based on last page access	0.236
dynamic balance based on working size set	0.198

Table 1: the total page miss ratio for the three different strategies.

In our test the total page miss is decreased by 26.48% when using the dynamic balance based on the last page access and decreased by 38.32% when using the dynamic balance based on working size set.

Page Miss for Second Strategy With Different Frequency

frequency	VM0	VM1	VM2	VM3
1	3	3	3	3
2	5	2	3	5
3	5	3	3	5
4	5	2	2	5
5	5	5	5	5
6	5	4	4	5
7	5	3	3	5
8	5	2	2	5
9	5	1	2	5
10	6	0	2	6

Table 2: For strategy 2, page misses based on different n values, where n is a check to every n pages.

Conclusions and Recommendations

Dynamic memory allocation across VMs has been simulated in this project. It is clear through our simulation that dynamically allocating pages in virtualized systems is superior to static page allocation in that better memory usage is obtained. However, the implementation of dynamic memory allocation was more complex and difficult than static memory allocation.

Type I hypervisors, which are typically in use on server systems already implement dynamic memory allocation, and improvements in this area continues. It is easy to see why dynamic memory allocation is more easily implemented in type I than than type II: in type II hypervisors, memory will have to be shared between the host OS and VMs adding to more complexity. However, dynamic memory allocation in commodity type II VMs are gaining interests, and hosted Xen is a promising product/project that does this. We foresee that future commodity VMs that run on desktops will emulate hosted Xen;

based on our simulation, dynamic memory allocation across VMs provide more efficient memory utilization.

Bibliography

[1] P. Zhou, V. Pandey, S. Jagadeesan, A.Raghuraman, Y.Zhou, S. Kumar. Dynamic tracking of page miss ratio cuve for memory management. In ASPLOS-XI: Proceedings of the 11th international conference on architectural support for programming lanuages and operating systems, pages 177-188, New York, NY, USA 2004. ACM.
ISBN-1-58113-8-4-0. doi:https://doi.acm.org/101145/1024393.1024415.

[2] W. Zhao, Z. Wang. Dynamic memory balancing for virtual machines. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on virtual execution and environments, pages 21-30, Washington, DC, USA 2009. ACM.
ISBN-978-1-60558-375-4. doi: https://doi.acm.org/10.1145/1508293.1508297.

[3] P. Barham, B.Dragovic, K.Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the art of virtualization. SIGOPS Operating Systems Review 37(5):164-177, 2003. ISSN-0163-5980.
doi:https://doi.acm.org/10.1145/1165389.945462.

[4] C. Waldspurger. Memory resource management in VMware esx server. SIGOPS Operating Sustems Review, 36(SI): 181-194, 2002. ISSN 0163-5980.
doi: https://doi.acm.org/10.1145/844128.844146

[5] Woomin Hwang; Yangwoo Roh; Youngwoo Park; Ki-Woong Park; Kyu Ho Park; , "HyperDealer: Reference-Pattern-Aware Instant Memory Balancing for Consolidated Virtual Machines," *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on* , vol., no., pp.426-434, 5-10 July 2010
doi: 10.1109/CLOUD.2010.70

[6] Wei Huang; Koop, M.J.; Panda, D.K.; , "Efficient one-copy MPI shared memory communication in Virtual Machines," *Cluster Computing, 2008 IEEE International Conference on* , vol., no., pp.107-115, Sept. 29 2008-Oct. 1 2008
doi: 10.1109/CLUSTR.2008.4663761

[7] http://wiki.xen.org

Appendices

Appendix 1: main.c

```
/**
*
* main.c
* The hypervisor process.
* @author Shuye Wu
* @date 05/18/2011
*
*/

#include"helper.h"
#include"page.h"
#include"strategies.h"
#include"VM.h"

Page pages[PAGE_COUNT];

void init_pages();
void init_alloc(int, VMInfo *);
void init_page_table(PageTableItem *, int);
int choose_strategy();

int main(int argc, char *argv[])
{
   int VMCount, i, j;
   pthread_t *VMThreads;
   VMInfo *VMInfos;
   sem_t *VMSems, *hypervisorSems;
   SeqInfo *seqInfos;

   if (argc < 2)
   {
         printf("Error: the file name of VM page access sequences is missed.\n");
         exit(-1);
   }

   // Read the sequence information from the specified file.
   VMCount = read_sequences(argv[1], &seqInfos);

   if (VMCount <= 0)
   {
         return -1;
   }

#ifdef DEBUG
   for (i = 0; i < VMCount; i++)
   {
```

```
            for (j = 0; j < seqInfos[i].seqCount; j++)
            {
                    printf("%d\t", seqInfos[i].seqs[j]);
            }
            printf("\n");
    }
#endif

    // Allocate memory for handlers of threads, semaphores, VMInfos and so on.
    VMThreads = (pthread_t *)malloc(sizeof(pthread_t) * VMCount);
    VMInfos = (VMInfo *)malloc(sizeof(VMInfo) * VMCount);
    VMSems = (sem_t *)malloc(sizeof(sem_t) * VMCount);
    hypervisorSems = (sem_t *)malloc(sizeof(sem_t) * VMCount);
    init_pages();
    init_alloc(VMCount, VMInfos);

    // Choose the strategy for memory dynamic balancing.
    int strategy = choose_strategy(), frequency;
    if (strategy == 2)
    {
            printf("Input the frequency for calculating page miss ratio: ");
            scanf("%d", &frequency);
    }

    // Create the threads for each VM.
    void *exit_status;
    for (i = 0; i < VMCount; i++)
    {
            VMInfo *info = &VMInfos[i];

            info->VMNum = i;
            sem_init(&VMSems[i], 0, 1);
            info->VMSem = &VMSems[i];
            sem_init(&hypervisorSems[i], 0, 0);
            info->hypervisorSem = &hypervisorSems[i];
            info->accessCount = seqInfos[i].seqCount;
            info->accessSeq = seqInfos[i].seqs;
            info->pageTable = (PageTableItem *)malloc(sizeof(PageTableItem) *
seqInfos[i].pageCount);
            init_page_table(info->pageTable, seqInfos[i].pageCount);
            info->pageMisses = (int *)malloc(sizeof(int) * seqInfos[i].seqCount);

            pthread_create(&VMThreads[i], NULL, processVM, info);
    }

    // The process of Hypervisor.
```

```
    for (i = 0; i < seqInfos[0].seqCount; i++)
    {
        for (j = 0; j < VMCount; j++)
        {
            sem_wait(&hypervisorSems[j]);
        }

#ifdef DEBUG
        printf("This is the %d checkpoint of hypervisor.\n", i);
#endif

        dynamic_balance(strategy, VMCount, VMInfos, i, &frequency);

        for (j = 0; j < VMCount; j++)
        {
            sem_post(&VMSems[j]);
        }
    }

    for (i = 0; i < VMCount; i++)
    {
        pthread_join(VMThreads[i], &exit_status);
    }

    // Print the page miss correspond to the access sequence.
    for (i = 0; i < VMCount; i++)
    {
        printf("Page miss sequence of VM-%d: ", i);
        for (j = 0; j < seqInfos[i].seqCount; j++)
        {
            printf("%d ", VMInfos[i].pageMisses[j]);
        }
        printf("\n");
    }

    return 0;
}

void init_pages()
{
    int i = 0;
    for (; i < PAGE_COUNT; i++)
    {
        pages[i].num = i;
        pages[i].isUsed = 0;
        pages[i].pageContent = -1;
```

```
            pages[i].previous = NULL;
            pages[i].next = NULL;
    }
}

void init_alloc(int VMCount, VMInfo *VMInfos)
{
    int i, j, count = 0, average = (PAGE_COUNT + VMCount - 1) / VMCount;

    for (i = 0; i < VMCount; i++)
    {
            PageManager *manager = (PageManager *)malloc(sizeof(PageManager));

            manager->head = &pages[count];
            pages[count].previous = NULL;
            for (j = 0; j < average && count < PAGE_COUNT; j++)
            {
                    pages[count].next = &pages[count + 1];
                    pages[count + 1].previous = &pages[count];
                    count++;
            }
            pages[count - 1].next = NULL;
            manager->tail = &pages[count - 1];

#ifdef DEBUG
            printf("Allocate %d pages to VM-%d.\n", j, i);
#endif

            VMInfos[i].pageManager = manager;
    }
}

void init_page_table(PageTableItem *table, int count)
{
    int i;
    for (i = 0; i < count; i++)
    {
            table[i].localPageNum = i;
            table[i].page = NULL;
    }
}

int choose_strategy()
{
    int strategy;
```

```
    printf("Choose one of the following dynamic memory balancing strategies to run:\n");
    printf("1. No dynamic balancing.\n");
    printf("2. Dynamic balancing based on last n page access.\n");
    printf("3. Dynamic balancing based on total page misses.\n");
    scanf("%d", &strategy);
    return strategy;
}
```

Appendix B: page.h

```
/**
*
* page.h
* The structure of Page and PageManager.
* @author Shuye Wu
* @date 05/18/2011
*
*/

#include<stdio.h>
#include<stdlib.h>

#ifndef PAGE_H
#define PAGE_H

#define PAGE_COUNT 16
#define MIN_PAGE_PER_VM 2

/*Struct for global page.*/
typedef struct Page
{
    int num;      // The global page number.
    int isUsed;            // Flag indicates whether it's used.
    int pageContent;    // The number of local page for VM.
    struct Page *previous;    // The pointer to previous global page.
    struct Page *next;    // The pointer to next global page.
} Page;

/*Struct for page manager. Each VM has a page manager. It's considered as LRU list or
stack.*/
typedef struct PageManager
{
```

```
    Page *head;         // The head global page.
    Page *tail;         // The tail global page.
} PageManager;

/*Struct for page table item. Each VM has its own local page table.*/
typedef struct PageTableItem
{
    int localPageNum;    // The local page number.
    Page *page;         // The global page which allocated to the local page.
} PageTableItem;

#endif
```

Appendix C: helper.c

```
/**
*
* helper.c
* The file of helper functions.
* @author Shuye Wu, Wei Zhang
* @date 06/03/2011
*
*/

#include"helper.h"

int read_sequences(const char *fileName, SeqInfo **seqInfos)
{
    FILE *fp = fopen(fileName, "r");
    int i, j, count;

    // If fail to open the file, return -1.
    if (!fp)
    {
        printf("Error: unable to open the specified sequences file.\n");
        return -1;
    }

    // Read the count of VM and allocate memory for sequence info for VMs.
    fscanf(fp, "%d", &count);
    *seqInfos = (SeqInfo *)malloc(sizeof(SeqInfo) * count);
```

```
    // Read the sequence info of each VM.
    for (i = 0; i < count; i++)
    {
        int length, max = 0;

        // Read the length of access sequence and allocate memory for sequence.
        fscanf(fp, "%d", &length);
        (*seqInfos)[i].seqs = (int *)malloc(sizeof(int) * length);
        (*seqInfos)[i].seqCount = length;

        // Read sequence and get the page count which is the biggest access page
number.
        for (j = 0; j < length; j++)
        {
            fscanf(fp, "%d", &((*seqInfos)[i].seqs[j]));
            if (max < (*seqInfos)[i].seqs[j])
            {
                max = (*seqInfos)[i].seqs[j];
            }
        }
        (*seqInfos)[i].pageCount = max + 1;
    }

    fclose(fp);
    return count;
}
```

Appendix D: helper.h

```
/**
*
* helper.h
* @author Shuye Wu
* @date 06/03/2011
*
*/

#include<stdio.h>
#include<stdlib.h>
#include"VM.h"

#ifndef HELPER_H
#define HELPER_H
```

```
int read_sequences(const char *, SeqInfo **);

#endif
```

Appendix E: VM.c

```
/**
*
* VM.c
* The VM process.
* @author Shuye Wu
* @date 05/18/2011
*
*/

#include"VM.h"

/*Access local page.*/
void access_page(VMInfo *info, int seqNum)
{
   int pageNum = info->accessSeq[seqNum];
   PageTableItem *pageTable = info->pageTable;
   PageManager *manager = info->pageManager;

   // If the page is loaded, then access the page and move it to the head of LRU list.
   if (pageTable[pageNum].page)
   {
#ifdef DEBUG
        printf("This is No.%d page access of VM-%d. Access page %d immediately since it's
already loaded.\n", seqNum, info->VMNum, pageNum);
#endif

        Page *page = pageTable[pageNum].page;

        if (page == manager->head)
        {

        }
        else
        {
                if (page == manager->tail)
                {
                        page->previous->next = NULL;
                        manager->tail = page->previous;
```

```
                }
                else
                {
                        page->previous->next = page->next;
                        page->next->previous = page->previous;
                }

                manager->head->previous = page;
                page->previous = NULL;
                page->next = manager->head;
                manager->head = page;
        }

        info->pageMisses[seqNum] = 0;
    }
    else // Else, replace the last page of LRU list.
    {
#ifdef DEBUG
        printf("This is No.%d page access of VM-%d. Load page %d into memory and access
it.\n", seqNum, info->VMNum, pageNum);
#endif

        Page *replace = manager->tail;

        // If the page is used before, reset the correspond item in page table.
        if (replace->isUsed != 0)
        {
                pageTable[replace->pageContent].page = NULL;
                info->pageMisses[seqNum] = 1;
        }
        else
        {
                info->pageMisses[seqNum] = 0;
        }

        replace->isUsed = 1;
        replace->pageContent = pageNum;
        pageTable[pageNum].page = replace;
        manager->tail = replace->previous;
        manager->tail->next = NULL;
        replace->previous = NULL;
        replace->next = manager->head;
        manager->head->previous = replace;
        manager->head = replace;
    }
}
```

```
/*The process of VM.*/
void processVM(void *arg)
{
   int i;
   VMInfo *info = (VMInfo *)arg;

   for (i = 0; i < info->accessCount; i++)
   {
         sem_wait(info->VMSem);

         access_page(info, i);

         sem_post(info->hypervisorSem);
   }
}
```

Appendix F: VM.h

```
/**
*
* VM.h
* The structure of VMInfo.
* @author Shuye Wu
* @date 05/18/2011
*
*/

#include<semaphore.h>
#include<stdio.h>
#include<stdlib.h>
#include"page.h"

#ifndef VM_H
#define VM_H

/*The struct of VM information.*/
typedef struct VMInfo
{
   int VMNum;                 // The number of VM.
   sem_t *VMSem;              // VM must wait for this semaphore to process one page access.
   sem_t *hypervisorSem;      // Hypervisor must wait for this semaphore to process dynamic
balancing.
   PageManager *pageManager;    // The page manager of VM.
   int accessCount;      // The count of page access.
```

```
    int *accessSeq;                 // The page access sequence.
    PageTableItem *pageTable;    // The local page table of VM.
    int *pageMisses;      // The array of flags indicate whether page access miss.
} VMInfo;

typedef struct SeqInfo
{
    int seqCount;
    int pageCount;
    int *seqs;
} SeqInfo;

void processVM(void *);

#endif
```

Appendix G: strategies.c

```
/**
*
* strategies.c
* The file of dynamic balancing strategies.
* @author Shuye Wu, Hayley Wu
* @date 05/27/2011
*
*/

#include"page.h"
#include"VM.h"

typedef struct VMInfoItem
{
    VMInfo *info;
    int misses;
    struct VMInfoItem *previous;
    struct VMInfoItem *next;
} VMInfoItem;

void dynamic_balance_1(int, VMInfo*, int, int);
void dynamic_balance_2(int, VMInfo*, int);

void dynamic_balance(int strategyNum, int VMCount, VMInfo *infos, int seqNum, void *data)
{
    int frequency;
    switch (strategyNum)
```

```
    {
        case 1:
            break;

        case 2:
            frequency = *((int *)data);
            dynamic_balance_1(VMCount, infos, seqNum, frequency);
            break;

        case 3:
            dynamic_balance_2(VMCount, infos, seqNum);
            break;

        default:
            break;
    }
}

void dynamic_balance_1(int VMCount, VMInfo *infos, int seqNum, int frequency)
{
    VMInfoItem *items = (VMInfoItem *)malloc(sizeof(VMInfoItem));
    int i, j;

    items->previous = NULL;
    items->next = NULL;

    // If there is not enough accesses to calculate the miss ratio, just return.
    if ((seqNum + 1) % frequency != 0)
    {
        return;
    }

    for (i = 0; i < VMCount; i++)
    {
        int misses = 0;
        for (j = seqNum; j > seqNum - frequency; j--)
        {
            misses += infos[i].pageMisses[j];
        }

        VMInfoItem *item = (VMInfoItem *)malloc(sizeof(VMInfoItem));
        item->info = &infos[i];
        item->misses = misses;

        VMInfoItem *last = items, *current = items->next;
        while (current)
```

```
        {
                if (current->misses > item->misses)
                {
                        break;
                }
                last = current;
                current = current->next;
        }
        last->next = item;
        item->previous = last;
        if (current)
        {
                current->previous = item;
        }
        item->next = current;
}

VMInfoItem *tmp1 = items->next, *tmp2 = items->next;
while (tmp2->next)
{
        tmp2 = tmp2->next;
}

while (tmp1 != tmp2 && tmp1->previous != tmp2)
{
        if (tmp1->misses == tmp2->misses)
        {
                break;
        }

        VMInfo *missVM = tmp2->info, *nonMissVM = tmp1->info;
        Page *page = nonMissVM->pageManager->tail;

        nonMissVM->pageTable[page->pageContent].page = NULL;
        page->isUsed = 0;
        nonMissVM->pageManager->tail = page->previous;
        nonMissVM->pageManager->tail->next = NULL;

        missVM->pageManager->tail->next = page;
        page->previous = missVM->pageManager->tail;
        page->next = NULL;
        missVM->pageManager->tail = page;

        tmp1 = tmp1->next;
        tmp2 = tmp2->previous;
```

```
#ifdef DEBUG
        printf("Deallocate one page from VM-%d and allocate this page to VM-%d.\n",
nonMissVM->VMNum, missVM->VMNum);
#endif
   }

#ifdef DEBUG
   for (i = 0; i < VMCount; i++)
   {
        int j = 0;
        Page *page = infos[i].pageManager->head;
        while (page)
        {
                j++;
                page = page->next;
        }
        printf("VM-%d has %d pages.\n", i, j);
   }
#endif
}

void dynamic_balance_2(int VMCount, VMInfo *infos, int seqNum)
{

        // Count total number of page misses across all VMs
        int i, j;
        int totalBonusPages = PAGE_COUNT - (VMCount * MIN_PAGE_PER_VM);
        int VMMissedSum[VMCount];
        int totalPageMisses = 0;

        for (i=0; i<VMCount; i++)
        VMMissedSum[i] = 0;

        for(i=0; i<VMCount; i++)
        {
        for(j=0; j<=seqNum; j++)
        VMMissedSum[i] = VMMissedSum[i] + infos[i].pageMisses[j];

#ifdef DEBUG
        printf("Total number of page missed at this checkpoint for VM-%d  is %d\n", i,
VMMissedSum[i]);
#endif

        totalPageMisses = totalPageMisses + VMMissedSum[i];
```

```
        }

#ifdef DEBUG
        printf("Total number of page missed across all VMs at this checkpoint is-%d\n",
totalPageMisses);
        printf("There are a total number of %d bonusPages to be distributed\n",
totalBonusPages);
#endif

        if (totalPageMisses == 0)
        {
#ifdef DEBUG
        printf("No page misses, no balancing required\n");
#endif
        return;
        }

        int bonusPageRemaining = totalBonusPages;
        int VMStartPages[VMCount];

        for (i=0; i<VMCount; i++)
        VMStartPages[i] = 0;


        for (i=0; i<VMCount; i++)
        {
        if (totalPageMisses != 0)
        {

        VMStartPages[i] =  (int)(((float)VMMissedSum[i])/totalPageMisses * totalBonusPages);
        bonusPageRemaining =   bonusPageRemaining - VMStartPages[i];

        }
        }

        if(bonusPageRemaining)
        {
        // distribute bonusPageRemaining among the VMs randomly
        int targetVM = rand() % VMCount;
        VMStartPages[targetVM] = bonusPageRemaining + VMStartPages[targetVM];

        }

#ifdef DEBUG
        for (i=0; i<VMCount; i++)
        {
```

```
        printf("Allocate %d extra pages to VM-%d\n", VMStartPages[i], i);
        }
#endif

        // New allocation of pages for each VM
        int newPageAllocation[VMCount];
        for (i=0; i<VMCount; i++)
        {
        newPageAllocation[i] = VMStartPages[i] + MIN_PAGE_PER_VM;
#ifdef DEBUG
        printf("VM-%d will get %d pages\n", i, newPageAllocation[i]);
#endif
        }

        //Get number of current pages for each VM
        int pageCount[VMCount];
        int takeAwayPages[VMCount];
        for (i=0; i<VMCount; i++)
        {
        pageCount[i] = 0;
        takeAwayPages[i] = 0;

        Page *page = infos[i].pageManager->head;
        while (page)
        {
        ++pageCount[i];
        page = page->next;
        }
        takeAwayPages[i] = newPageAllocation[i] - pageCount[i];
#ifdef DEBUG
        printf("VM-%d has %d pages.  ", i, pageCount[i]);
        printf("Need to add/remove %d page(s) from VM-%d\n", takeAwayPages[i], i);
#endif

        }



  Page *pages = (Page *)malloc(sizeof(Page)), *current = pages;
        pages->next = NULL;
        pages->previous = NULL;


  // Start adding/removing pages
  for (i = 0; i < VMCount; i++)
        {
```

```
                for(j = takeAwayPages[i]; j < 0; j++)  // add pages
                {
                Page *tmp = infos[i].pageManager->tail;
                tmp->previous->next = NULL;
                if (tmp->isUsed == 1)
                {
                        infos[i].pageTable[tmp->pageContent].page = NULL;
                }
                infos[i].pageManager->tail = tmp->previous;
                tmp->next = NULL;
                tmp->isUsed = 0;
                tmp->pageContent = -1;

                current->next = tmp;
                tmp->previous = current;
                current = tmp;
                }
        }

        for (i = 0; i < VMCount; i++)
        {
                Page *tmp = pages->next, *end = pages;
                for(j = 0; j < takeAwayPages[i]; j++) //remove pages(s) from the head
                {
                end = end->next;
                }
                pages->next = end->next;

                if (takeAwayPages[i] > 0)
                {
                infos[i].pageManager->tail->next = tmp;
                tmp->previous = infos[i].pageManager->tail;
                infos[i].pageManager->tail = end;
                }
        }


} // end dynamic_balance_2
```

Appendix H: strategies.h

```
/**
*
* strategies.h
* The head file of dynamic balancing strategies.
```

```
* @author Shuye Wu
* @date 05/27/2011
*
*/

#include"page.h"
#include"VM.h"

#ifndef STRATEGIES_H
#define STRATEGIES_H

void dynamic_balance(int, int, VMInfo*, int);


#endif
```