Design And Implement Of Memory Control Invirtual Machines In A Consolidated Environments

P.SANTHI KUMARI, Y.SIVA KOTESWAR RAO, Dr.G.GURU KESAVA DAS

1Santhi Kumari , Pursuing M.Tech , Dept of CSE, Eluru college of Engineering and technology, Duggirala(V), Pedavagi(M), Eluru, India.
2 Guide Details, Y.Siva Koteswar Rao, M.Tech, Dept of CSE, Eluru college of Engineering and technology, Duggirala(V), Pedavagi(M), Eluru, India.
3 Hod Details,Dr.G.Guru Kesava Das,Phd, Dept of CSE, Eluru college of Engineering and technology, Duggirala (V), Pedavagi (M), Eluru, India.

Abstract—Through virtualization, multiple virtual machines (VMs) can coexist and operate on one physical machine. When virtual machines compete for memory, the performances of applications deteriorate, especially those of memory-intensive applications. In this study, we aim to optimize memory control techniques using a balloon driver for server consolidation. Our contribution is three-fold: (1) We design and implement an automatic control system for memory based on a Xen balloon driver. To avoid interference with VM monitor operation, our system works in user mode; therefore, the system is easily applied in practice. (2) We design an adaptive global-scheduling algorithm to regulate memory. This algorithm is based on a dynamic baseline, which can adjust memory allocation according to the memory used by the VMs. (3) We evaluate our optimized solution in a real environment with 10 VMs and well-known benchmarks (DaCapo and Phoronix Test Suites). Experiments confirm that our system can improve the performance of memory-intensive and disk-intensive applications by up to 500 and 300 percent, respectively. This toolkit has been released for free download as a GNU General Public License v3 software.

Index Terms–Virtual machine, server consolidation, memory control, global-scheduling

1 INTRODUCTION

VIRTUALIZATION has resurged as a result of cloud com-puting [1]. More and more applications are deployed into virtual machines (VMs) to multiplex a physical server. Although the resources of these VMs (such as CPU and memory) are isolated through virtual machine monitor (VMM) subsystems [2], [3], automatic control systems can reallocate the limited resources of the consolidated server dynamically, which can reduce the running time of applica-
tions and maximize resource utilization.

Automatic control systems for CPU devices have been widely researched [4], [5], [6], but time sharing for memory devices remains an open issue [7]. Normally, memory is statically allocated to each VM when the machine is booted, and memory size does not vary throughout the life cycle of the VM. When memory size requests exceed total physical memory, memory competition overhead increases exponen-tially, thus degrading the performances of applications. The automatic control of physical memory in virtualization is a bottleneck that increasingly limits the efficiency of the overall system.

Unlike in previous research, current studies on memory control in VMs face a minimum of three new challenges in the context of server consolidation:

Tools for automatic memory control at the applica-tion level require further investigation. To activate underlying mechanisms and to generate low-level interfaces, Xen [8], VMware [9], and KVM [10] have implemented page sharing, virtual hot plugs, and balloon drivers in the VMM.

1) However, these mechanisms and interfaces only focus on the underlying methods in kernel mode to resize the memory for an individual VM. They cannot specify which VM needs to reclaim/release its memory or how many pages it should take/give in a global perspective. Therefore, high-level tools in user mode are necessary to auto-matically collect memory usage from VMs, make global decisions and regulate their memory.

2) Memory scheduling algorithms need to be more adaptive to different scenarios, regardless of when the global memory is sufficient or insufficient. Each VM can submit a memory value, called committed memory, which will be used in the future. The memory state is sufficient if the sum of the committed memories of all VMs is smaller than the available memory of the physical machine. Otherwise, the memory state is insufficient. Our previous work [11] focused only on the sufficient state. Memory perspective. To dynamically allocate memory, Heo et al. [12] used control theory, but it is effective only in the sufficient state. Zhao et al.
[13] proposed a quick approximation algorithm to prevent total page misses from reaching a local mini-mum. To avoid this local minimum and to attain opti-mal performance, additional algorithms should thus be developed for general global scheduling.

3) The scale at which previous evaluations are per-formed is not coherent with the VM consolidation

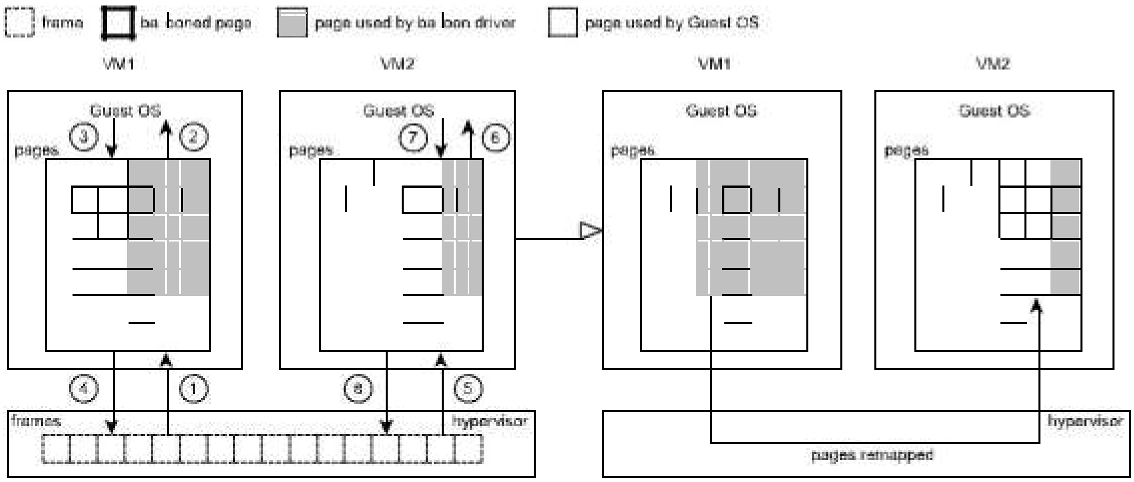


Fig. 1. Mechanism of the Xen balloon driver.

ratios used by large vendors. Although few cloud computing companies (Amazon EC2 [14] et al.) are willing to disclose the number of VMs they can host on a physical server, we conservatively estimate that one server contains at least 10 or 12 VMs [15], [16]. However, previous experiments are limited to a maximum of two or four VMs. These experiments also adopt workloads that are synthetic and traces-driven. Therefore, more tests should be conducted with additional VMs and real benchmarks.

In this study, we devisea lightweight framework based on the Xen balloon driver to control memory in the consoli-dation of multiple VMs. Our system is implemented in user space that does not interfere with VMM operation. For this framework, we propose a global-scheduling algorithm that runs on Domain0. This algorithm solves linear equations to obtain the global solution and adapts to sufficient and insuffi-cient states using dynamic baselines. Real-world bench-marks are adopted as workloads in our experiments, and 10 VMs are utilized.

The rest of this paper is organized as follows. In Section 2, we provide an overview of our memory control system and its implementation. We describe the memory scheduling algorithm in Section 3. The experimental results are pre-sented in Section 4. We discuss related studies in Section 5. Finally, we give concluding remarks and suggestions for future research in Section 6.

2 System Overview

In this section, we first review the principle of memory ballooning in Xen. We then propose our automatic memory control system based on the Xen balloon driver.

2.1 Background of Xen Balloon Driver

The ballooning mechanism aims to overcommit memory. In this process, physical memory can be allocated to all active domains, although the amount allocated is more than theto-tal physical memory in the system. In 2002, Waldspurger

[20] first introduced the “ballooning” mechanism for the VMware ESX Server. In 2003, Xen also implemented this mechanism to allocate memory from one domain to others [19]. As a result, memory from idle VMs or from domains

that use less memory can be committed to newly created VMs or to domains requiring additional memory.

Virtual memory in Xen decouples the virtual address space from the physical address space. The virtual memory in VMs and the physical memory in the actual machine are divided into pages and frames, respectively. The pages are addressed by their Guest Physical Frame Numbers, or GPFNs. The frames are addressed by their Machine Frame Numbers, or MFNs. Every VM has a physical to machine translation table, which maps the GPFNs to MFNs.

The Xen balloon driver resides in the domain but is controlled by the hypervisor [21]. Fig. 1 depicts its working pro-cess of inflation and deflation, with two VMs (VM1 and VM2) as examples. The left side of Fig. 1 represents the ini-tial page allocation of the VMs. The right side represents the remapped page allocation.

To inflate the balloon, first, the hypervisor sends an infla-tion request to the balloon driver in VM1 (phase **1**). Then, the balloon driver requests free pages from its Guest OS (phase **2**). After acquiring the pages, it records their corre-sponding GPFNs (phase **3**). It then notifies the hypervisor to replace the MFNs behind these GPFNs with “invalid entry”. Finally, the hypervisor puts these reclaimed MFNs on its own free list, which is given to VM2 (phase **4**).

To deflate the balloon, the balloon driver in VM2 receives a deflation request from the hypervisor (phase **5**). Then, the balloon driver releases pages to its Guest OS (phase **6**). If the Guest OS is allowed to increase its page numbers and if free frames are available (phase **7**), the hypervisor will allo-cate MFNs behind the GPFNs to increase the pages used by the Guest OS in VM2 (phase **8**).

Note that from VM1’s perspective, the ballooned pages appear to still be in use by its balloon driver. In fact, the frames behind these pages have been reclaimed by the hyper-visor and remapped to the ballooned pages in VM2.

2.2 Our Automatic Memory Control System

Fig. 2 shows our automatic memory control system based on Xen. This toolkit has been released for free download on GitHub [17] under a GNU general public license (GPL) v3.

Domain: A domain is a VM that is operating on a system. On boot, the Xen hypervisor activates the first

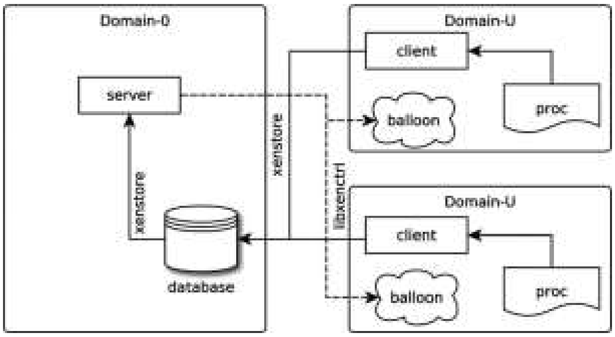


Fig. 2. Our automatic memory control system.

domain (Domain0), on which a Guest OS runs. Through Xen control tools, Domain0 is privileged to access the hardware and to manage other domains. These other domains are referred to as DomainUs and are unprivileged; they can thus run on any Guest OS that has been ported to Xen [18, 19].

Balloon: The Xen balloon driver is the basis of and supports our system of automatic memory control technically. We can thus focus on efficiently allocat-ing the memory pages across various domains.

XenStore: This is a hierarchical namespace shared between domains, which stores the running informa-tion of domains [22]. It also provides primitives to either read or write a key, enumerates a directory, and generates notifications when a key changes value. XenStore is categorized into three branches:

/vm—stores configuration information about domains;

/local/domain—stores information regarding the domain in the local node. Its key (<domid>/memory/target) contains the target page number of the domain;

/tool—stores information for various tools. XenStore can be accessed by virtual input/output (I/O) drivers using the in-kernel application programming interface (API) XenBus.

proc: A process file system is a virtual file system in the Guest OS layer that contains dynamic informa-tion related to kernel and system processes [25]. The directory /proc/meminfo stores the memory informa-tion. MemTotal is the total page number, MemFree is the size of unallocated pages, Buffers denotes the buffer size for files, and Cached is the size of the pages used by caches. The total free pages of the system includes MemFree, Cached, and Buffers. SwapTotal is the total size of swap memory, and SwapFree is the size of the free swap memory.

Libxenctrl: This is a C interface that can be called by libraries or applications in the domains to interact with the hypervisor. In this study, the following interfaces are the most relevant: xc_interface_open() and xc_interface_close() open and close a file handler named xc_handle, which is a /proc/xen/privcmd driver and used when applications want to make a hyper-call; xc_domain_setmaxmem() can set the maximal GPFNs of DomainUs; and xc_domain_set_pod_target() can set the target GPFNs of DomainUs.

Database: The database is hosted by Domain0 and functions in the application layer, which stores page information from DomainUs. Database contains the following records: 1) the target GPFNs of the domain, which are rooted in the /local/domain/**<**domid**>**/ memory/target of XenStore; 2) the total GPFNs of the domain, which is derived from /proc/meminfo/MemTotal; 3) the maximal GPFNs of used memory MemUsed, which is calculated as follows:

MemUsed ¼ MemTotal MemFree

Cached Buffers

where MemTotal, MemFree, Cached, and Buffers are obtained from /proc/meminfo.

Client: This collects memory information from DomainUs and periodically passes this information over to the Database. It is hosted by DomainUs and functions in the application layer. Memory and swap space information are gathered from the proc of DomainUs. These data are stored in Database with the APIs of XenStore. Client also collects the total and free MFNs of the physical machine.

Server: As the core of the system, it acquires memory information from Database. It resides in Domain0 and functions in the application layer. The schedul-ing algorithm of Server then determines the domain that requires additional pages, as well as the domain that provides these extra pages. The scheduling algo-rithm also calculates the optimal target pages for allo-cation to each domain. Finally, we invoke the API xc_domain_set_pod_target() in Libxenctrl to reset the target memory of the domains. According to the sys-tem state, different scheduling algorithms in Server may be utilized. These algorithms are discussed in detail in Section 3.

3 MEMORY SCHEDULING ALGORITHM

We have developed two scheduling algorithms for the Server: self-scheduling and global-scheduling. The self-schedul-ing algorithm was introduced and extensively verified in our previous study [11]. We focus on the global-scheduling algorithm in this section.

The self-scheduling algorithm is applied if the free frames in the physical machine can satisfy the total pages requested by all VMs. In this case, the self-scheduling algorithm can directly map MFNs to GPFNs through the balloon driver for each domain. It also deploys a driver in the hypervisor, which monitors available frames in the physical machine and will trigger the global-scheduling algorithm when the available frames run out.

The global-scheduling algorithm is utilized if the physical machine lacks free frames and cannot meet the total pages requested by all VMs. In this case, VMs compete for mem-ory. The global-scheduling algorithm is used to overcommit memory globally.

3.1 Global-Scheduling Algorithm

Table 1 summarizes the key notations that facilitate the discussion of this algorithm.

The global-scheduling algorithm is given as follows:

Algorithm 1. Global-Scheduling Algorithm

```
Input: $N, n, N_i, A_i$
Output: $Nt_i$
while true do
A   Null
for 1   i   n do
    $N_i$   xs_read (/local/domain/VM$_i$/mem/total);
    $F_i$   xs_read (/local/domain/VM$_i$/mem/free);
    $A_i$ ¼ $N_i$   $F_i$;
    AppendTo(A, $A_i$);
end
**t**   calculating_idle_memory_tax(A,   f);
for 1   i   n do
    $Nt_i$   solve_linear_equation($N_i$, $A_i$, **t**);
    xs_write($Nt_i$, /local/domain/VM$_i$/mem/target);
    xc_domain_set_pod_target(VM$_i$, $Nt_i$);
end
sleep(interval);
end
```

The global-scheduling algorithm is used in the Server program of Domain0. In this algorithm, two sub-procedures, calculating_idle_memory_tax() and solve_linear_equation(), are essential. First, the total and free memory sizes for each VM are acquired using XenStore. Second, the parameter **t** (idle memory tax) is computed using the function calculating_ idle_memory_tax(). By solving the linear equations, we calculate the target memory (Nt_i) allocated to each VM. Finally, the target memory for each VM is sent to the balloon driver using Libxenctrl interfaces to reallocate the memory pages.

3.2 Idle Memory Tax

Idle memory tax (**t**) is adopted from economic theory. It lev-ies a high tax for idle memory on a VM that does not maxi-mize its memory. Our system can force a VM under a high tax to pass memory pages to a VM under a lower tax.

In economic theory, shares represent the resource-rights owned by a client in proportional-share [20]; a cli-ent can obtain resources based on its shares. To allocate space-shared resources in terms of proportional share, both randomized and deterministic algorithms are gener-ated. If a client requests additional resources, the dynamic algorithm for min-funding revocation determines the cli-ent with the fewest shares and reallocates its resources [20], [24], and [25].

In practice, a VM with many shares acquires much mem-ory, much of which is idle. However, VMs with few shares have insufficient memory. To overcome this limitation, Waldspurger implemented a tax for idle memory in the VMware ESX Server [20]. This tax reclaims memory pages from a VM that does not maximize its memory and specifies the maximal fraction of idle pages that may be reclaimed from a VM. Min-funding revocation is extended for an adjusted ratio of shares-per-page. For a client with S shares and an allocation of P pages, of which a fraction Q are active, the adjusted ratio of shares-per-page **r** is:

$$\mathbf{r}\ ¼ \frac{\mathbf{S}}{\mathbf{PðQ\ þ\ kð1}_{\mathbf{QÞÞ}}};$$

TABLE 1

Summary of Key Notations

n	Number of VMs
N	Total memory of n VMs
f	Reserved free memory of VMs
N_i	Total memory of the ith VM, 1 i n
N_{ti}	Target memory allocated to the ith VM, 1 i n
F_i	Free memory of the ith VM, 1 i n
A_i	Used memory of the ith VM, which equals to $(N_i\;\; F_i)$, 1 i n
A	$\{A_i\; J\; i ¼ 1\cdots .n\}$ $_i$ 1 n
$\bar{A}$	$\bar{A}$ ¼ $\frac{1}{n}\sum_{i¼1} A_i$

where the cost of idle pages is **k ¼ 1=ð1 tÞ** for a given tax rate **t** (**0 t < 1**). On one side, **t ¼ 0** specifies the pure allo-cation based on shares. On the other side, **t 1** specifies that the idle pages of all VMs can be reclaimed and allocated equally.

Supposing that in each VM, **S ¼ 1**, **P ¼ N$_i$**, and that the proportion of active sections in total memory is **Q$_i$ ¼ A$_i$=N$_i$**, the shares-per-page **r$_i$** of each VM is:

$$
\begin{aligned}
r_i\; ¼ & \frac{1}{A_i\;\; þ\;\; kðN_i\;\; A_iÞ}\\
¼ & \frac{1}{A_i\;\; þ\;\; \frac{1}{1\;\; t}\; ðN_i\;\; A_iÞ}\\
¼ & \frac{1\;\; t}{N_i\;\; t\;\; A_i} :
\end{aligned}
\qquad (1)
$$

3.3 Calculating Target Memory by Solving Linear Equations

Our system aims to guarantee identical shares-per-page **r$_i$** for all VMs. We regard shares-per-page as the price of idle memory. In economics, price increases when supply cannot meet demand; we can thus reduce prices by increasing sup-ply. If the price of idle memory is high in VMs with insuffi-cient resources, we reduce it by allocating additional memory pages. In contrast, if the price is low for VMs with abundant idle pages, we can reclaim memory pages from these VMs and reallocate them to other VMs. We can bal-ance the allocation of memory pages by balancing the price of idle memory pages. As a result, the prices of idle memory pages in all domains remain equal.

Based on Equation (1), we can derive the following equation:

$$8\; i;\; j \overset{2}{\;} ð1:::nÞ;\; i \;¼6\; j : \frac{1\;\; t}{N_{ti}\;\; t\;\; A_i} ¼ \frac{1\;\; t}{N_{tj}\;\; t\;\; A_j} :$$

The linear equations are then expressed as follows:

$$
8\left\{
\begin{array}{l}
N_{t1}\;\; t\;\; A_1 \;¼\; N_{t2}\;\; t\;\; A_2\\
N_{t1}\;\; t\;\; A_1 \;¼\; N_{t3}\;\; t\;\; A_3\\
\quad t\; :::\\
N_{t1}\;\; t\;\; A_1 \;¼\; N_{tn}\;\; t\;\; A_n\\
N \;¼\; \sum_{i¼1}^{n} N_{ti}
\end{array}
\right. \qquad (2)
$$

below the reserved value, our system reclaims more memory pages from other VMs.

Dropping the memory uniformly. The value of **t** is set to 1.0 when the total available memory cannot satisfy the requests of all VMs. With this setting, all VMs can share the remaining memory available. The available memories of all VMs drop uniformly.

4 EVALUATION

4.1 Experimental Setting

To evaluate our system, we have adopted three types of benchmarks, as described below:

Mono [26] is a micro kernel benchmark, which is designed to verify the effectiveness and accuracy of our system. It shows the memory changes by report-ing the total, used, and free memory of regulated VMs over the entire time line. It operates in two phases. Given a memory range of low to high (low **<** high), Mono initially applies a low amount of memory pages in the first phase and then gradually increases the memory requests to high. During the second phase, it monotonically releases the memory pages in a range of high to low.

DaCapo (version 9.12) [27] is a Java benchmark suite used to manage memory and design computer architecture. It consists of a set of 14 open-source, real-world applications with non-trivial memory loads. For our tests, we select 13 applications among them, including CPU- (avrora, fop, jython, lusearch, pmd, sun-flow, tomcat, and xalan), memory- (h2, tradebeans, and tradesoap), and disk-intensive (eclipse and luindex) applications. The batik application is not stable in our platform. It crashes for each run and no correspond-ing data can be recorded.

Phoronix Test Suite (version 4.8.6; PTS) [28] is an automated platform for open-source testing and benchmarking. It contains more than 130 test pro-files and 60 test suites. These tests range from tra-ditional CPU, memory, and disk computing to emerging graphics processing unit, mobile device, and cloud computing. PTS is a multi-platform that is easy-to-use, with extensible architecture and support. It is not necessary to run all 130 test pro-files to verify our system because some profiles share similar memory usage behaviors. We select seven representative memory-consuming applications that cover small- (John-the-ripper, scimark2, and System-libxml2), medium- (pgbench), and large-scale (apache, compile-linux-kernel, and compress-7zip). John-the-ripper and scimark2 contain three (blowfish, traditional DES, and MD5) and five (Com-posite Monte Carlo, FFT, Sparse Matrix Multiply, Dense LU Matrix Factorization, and Jacobi Successive Over-Relaxation) subtests, respectively.

In our tests, a 64-core server that contains four 16-core AMD Opteron 6272 processors is utilized. Its CPU fre-quency is 2,100 MHz, its cache size is 2,048 KB, and its mem-ory size is 128 GB..

A Type-1 Xen hypervisor (version 4.1.2) is deployed in the server with full virtualization (HVM). The Guest OS in all of the VMs is the Ubuntu Server 12.10 without the X-win-dow system. Its kernel is Linux Ubuntu 3.5.0-17-generic. To prevent CPU contentions when multiple VMs are running, each VM is assigned a dedicated CPU core. By default, all VMs are initially allocated 512 MB to 1 GB memory without losing generality. The total memory size accessed by all of the VMs is restricted to 5 GB.

Our memory control system is implemented in C, including Server, Database, and Client. Server and Database are deployed in Domain0, whereas Client is implemented in DomainUs. An interval of 1 s is set for Client to collect memory information. An interval of 2 s is set for Server to balance the memory. To maintain the operations of Server and Client in real-time, the system call nice() enhances their priority. This system call reduces the latency of information collection and memory control.

Although Server checks the memory information stored in Database every 2 s, the memory need not be rescheduled each time. If we regulate the memory for small changes, memory is quickly fragmented. As a result, Server need not reschedule until the change in memory change is over 10 MB. To calculate idle memory tax and enhance perfor-mance, the minimal free memory reserved f is set within the range of 100 to 150 MB.

4.2 Validation of Two VMs Using Mono Benchmark

We aim to verify whether our system can successfully regu-late the memory of VMs.

Each VM is configured with a total and a reserved free memory (f) of 512 and 100 MB, respectively. Server runs on Domain0, whereas Clients and balloon drivers operate on both VMs. VM1 runs the micro kernel benchmark Mono, which gradually increases the memory from 50 to 500 MB before monotonically lowering it from 500 to 50 MB. VM2 is idle.

Fig. 4 shows that the memory changes in VM1 and VM2 undergo three phases. The total memory of VM1 remains unchanged when Mono increases the memory, although the sum of used and free memories does not exceed the initial total memory of 512 MB. In the first phase, idle memory tax **t ¼ 0**, and the balloon driver does not need to regulate the memory between VMs. When the memory used in VM1 exceeds 512 MB, our system then reclaims the free memory from VM2 and allocates it to VM1. As a result, the total memory of VM1 increases while that of VM2 decreases. Mono releases memory in the third phase, and the total memory sizes of both VMs gradually equalize.

This result confirms that our memory control system can effectively balance memory automatically.

4.3 Validation of 5 VMs Using the DaCapo Benchmark Suite

We investigate five VMs to verify whether our system can reduce the running time of the DaCapo benchmark suite.

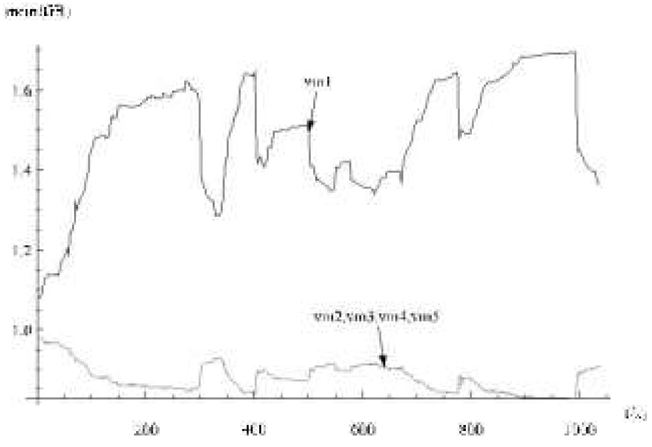


Fig. 7. Allocation of memory among five VMs when the memory control system is "on".

Therefore, we verify that our system can shorten the running times of applications with heavy workloads by reclaiming memory from other VMs and avoiding swap space usage.

4.4 Validation of 10 VMs Using Hybrid PTS and DaCapo Benchmarks

We examine 10 VMs to verify whether our system can shorten the running times of hybrid benchmarks com-posed of PTS and DaCapo.

When DaCapo was tested on five VMs previously, only one VM was in operation. In this test, we deploy different applications in all 10 VMs. DaCapo is a Java benchmark suite, whereas the benchmarks in PTS can be implemented in various applications. Moreover, PTS is convenient for test set-ups and result analysis. Therefore, we combine PTS with the DaCapo test suite in 10 VMs to verify the general scalability of our system.

We select three memory-intensive applications from DaCapo, namely, h2, tradebeans, and tradesoap, as well as seven memory-consuming applications from PTS, namely, apache, compile-linux-kernel, pgbench, John-the-ripper, scimark2, System-libxml2, and compress-7zip. These benchmarks cover memory-, CPU-, and disk-intensive applications, and their memory requests vary. Each benchmark runs on one domain, and each domain independently occupies one core. Table 2 depicts the details and distributions on 10 VMs.

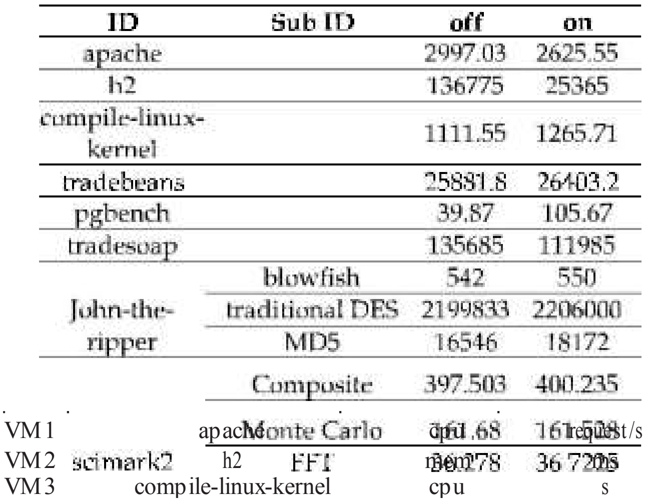
TABLE 2

Information on and Distributions of Hybrid PTS and DaCapo Benchmarks on 10 VMs

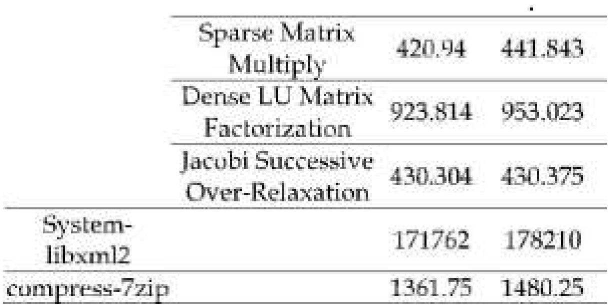
ID	Name	Type	Unit
VM1	apache	cpu	request/s
VM2	h2	mem	ms
VM3	compile-linux-kernel	cpu	s
VM4	tradebeans	mem	ms
VM5	pgbench	disk	transaction/s
VM6	tradesoap	mem	ms
VM7	John-the-ripper	cpu	real C/S
VM8	scimark2	cpu	Mflops
VM9	System-libxml2	cpu	ms
VM10	compress-7zip	cpu/mem	MIPS

TABLE 3

Average Scores of Hybrid PTS and DaCapo Benchmarks on 10 VMs when the Memory Control System is either "off" or "on"



ID	Sub ID	off	on
apache		2997.03	2625.55
h2		136775	25365
compile-linux-kernel		1111.55	1265.71
tradebeans		25881.8	26403.2
pgbench		39.87	105.67
tradesoap		135685	111985
John-the-ripper	blowfish	542	550
	traditional DES	2199833	2206000
	MD5	16546	18172
scimark2	Composite	397.503	400.235
	Monte Carlo	161.68	161.508
	FFT	36.278	36.7205



	Sparse Matrix Multiply	420.94	441.843
	Dense LU Matrix Factorization	923.814	953.023
	Jacubi Successive Over-Relaxation	430.304	430.375
System-libxml2		171762	178210
compress-7zip		1361.75	1480.25

Each of the 10 VMs is configured with 512 MB of initial available memory and 100 MB of reserved free memory (f). We run the tests 20 times and average the scores regardless of the activation of memory control. Table 3 displays the final scores of the benchmarks.

The benchmark John-the-ripper contains three subtests (blowfish, traditional DES, and MD5) whereas scimark2 has six (Composite, Monte Carlo, FFT, Sparse Matrix Multiply, Dense LU Matrix Factorization, and Jacobi Successive Over-Relaxation). The scores of John-the-ripper and scimark2 can thus be normalized as the geometric means of their subtests.

Fig. 8 compares the final scores (the higher score and the better performance) of these benchmarks with or without memory control. The score of the h2 application

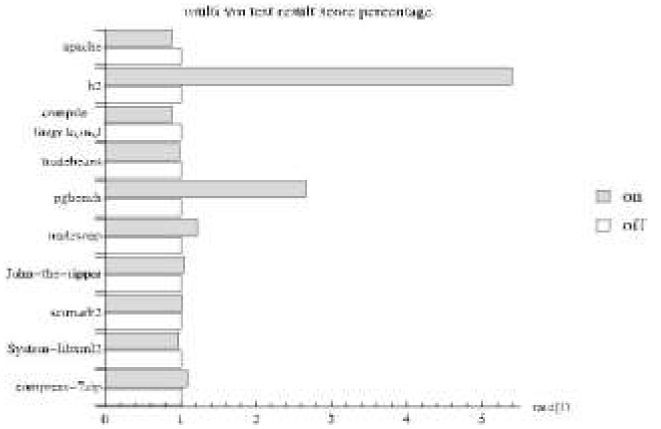


Fig. 8. Comparison of the scores of PTS & DaCapo hybrid benchmarks when memory control system is “off” or “on”.

is more than five times higher when the system is “on” than when it is “off.” Similarly, the score of pgbench is also nearly three times higher when the system is “on”. Other memory-intensive applications (e.g., tradesoap and com-press-7zip) likewise obtain higher scores. The scores of the John-the-ripper and scimark2 are just slightly higher. How-ever, the scores of apache, compile-linux-kernel, tradebean, and System-libxml2 drop by approximately 10 percent when the system is “on”.

Fig. 9 shows swap space usage when the memory control is either “off” (dotted lines) or “on” (solid lines).

Figs. 9a, 9c, and 9i show that the swap space usage for the benchmarks apache, compile-linux-kernel, and System-libxml2 is small (no more than 4 MB) when the system is “off”. Apache benchmarks the HTTP server, compile-linux-kernel measures the time to build the Linux 3.1 kernel, and System-libxml2 records the time to parse a random XML file. Because they are all CPU-intensive applications and not sensitive to the memory size, their performance does not improve when our system is “on”. The extra overload incurred by page remapping and cleaning slightly degrades their performance.

Figs. 9b, 9f, and 9j show that the swap space usage for the benchmarks h2, tradesoap, and compress-7zip is large and up to 250, 100, and 150 MB, respectively, when the system is “off”. The benchmark h2 accesses the memory frequently by using a JDBCbench-like in-memory database to simulate many banking transactions. Tradesoap uses h2 as the under-lying database to benchmark the daytrader via a SOAP to a Geronimo backmend. Compress-7zip measures the file com-pression with p7zip to test the processor and memory. These benchmarks are all memory-intensive applications and fre-quently access the swap space with huge fluctuations. When the memory control is activated, the swap space usage quickly decreases to approximately 25 MB. Therefore, their performance is greatly improved.

Fig. 9d shows the swap space usage for the tradebean benchmark. Tradebean is also memory-intensive, which uses h2 as the database to benchmark the daytrader via a Java-Beans to a Geronimo backend. However, at most of the time, its swap space usage is approximately 20 MB and keeps stable except the initial phase. As mentioned in Section 3.3, the Guest OS uses swap space as long as its free memory is below a specific threshold $_0$.

In this test, the value of $_0$ happens to be between 15 and 20 MB, which means that enough free memory is still available. As a result, when our system is “on”, its performance improve-ment is not obvious because of the stable and small swap space usage below the threshold $_0$.

Fig. 9e shows that pgbench occupies surprisingly less swap space than expected, compared to its triple perfor-mance improvement. It runs the same sequence of SQL commands in multiple concurrent database sessions on PostgreSQL. It is an I/O-intensive, mostly-on-disk bench-mark, uses disk caches for caching data, and rarely relies on swap space usage to avoid double paging. Addition-ally, VMs frequently access the swap space when they are out of memory. Hence, many I/O operations are intensi-fied. These functions interfere with the normal I/O opera-tions of pgbench, thus seriously affecting its performance.

When our system is “on”, swap space usage decreases and pgbench performance improves, thus suggesting that frequent swap space requests seriously affect the per-formances of disk-intensive applications when the system is out of memory.

Figs. 9g and 9h show that the swap space usage for John-the-ripper and scimark2 is nearly zero no matter when our system is “on” or “off”. John-the-ripper is a password cracker, and scimark2 is for scientific and numerical computing and includes Fast Fourier Transform, Jacobi Successive Over-relaxation, Monte Carlo, Sparse Matrix Multiply, and dense LU matrix factorization benchmarks. Both are CPU-inten-sive applications. Therefore, our system can not help to improve their performance by ballooning pages.

In summary, our system is scalable and can greatly enhance the performance of memory- and disk-intensive applications.

4.5 Costs of Automatic Memory Control System

We aim to verify the costs of our memory control system incurred by remapping and scrubbing pages.

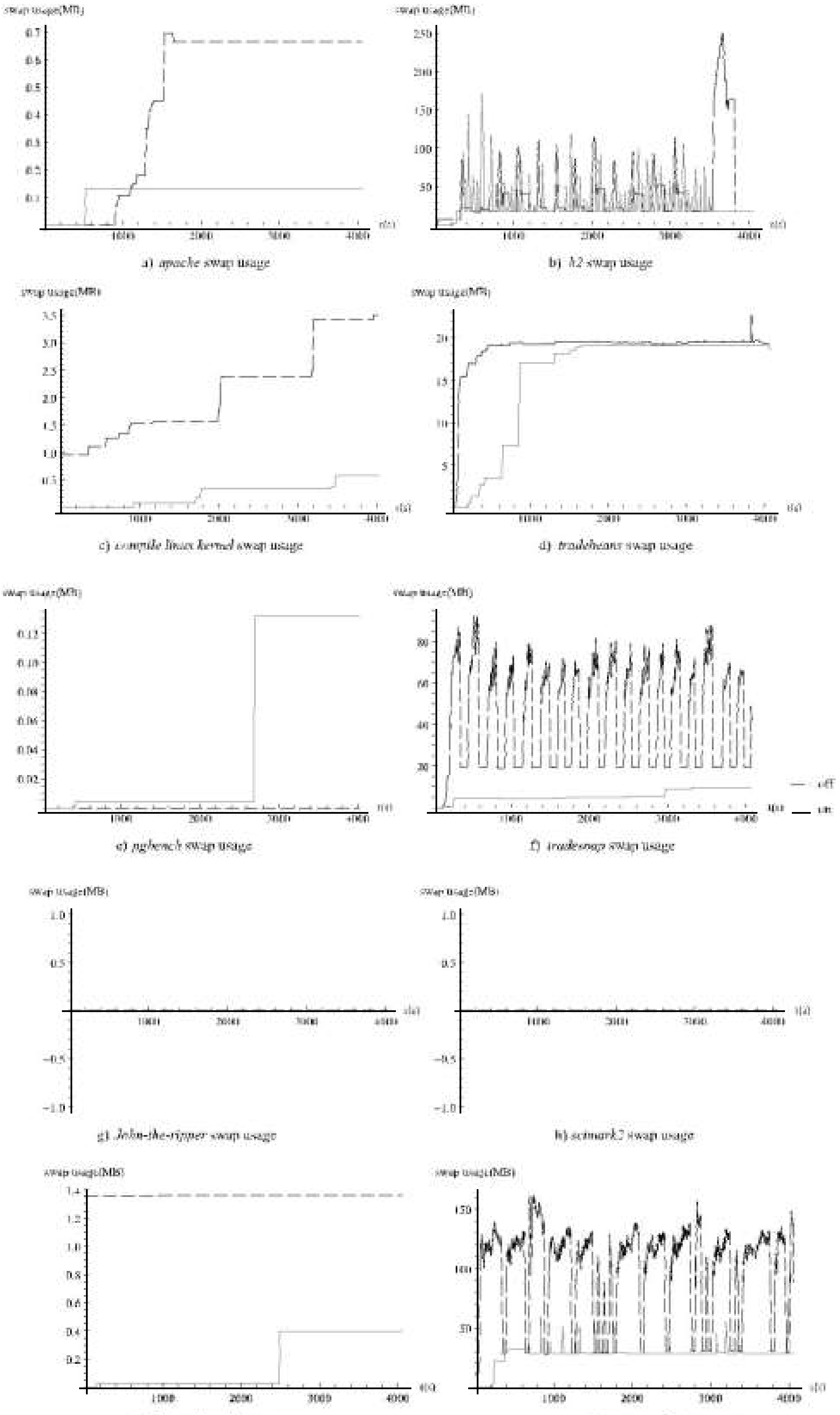
Each of 10 VMs is configured with 12 GB of initial avail-able memory and 300 MB of reserved free memory (f). On one VM, we deploy a CPU-intensive benchmark (sunflow) or a memory-intensive one (h2) together with a special tool we developed, which can continue claiming memory at adjustable rates. The other nine VMs are idle.

This test occurs in the following way: First, we run sun-flow or h2 in the target VM while our tool, resides in the same VM, sends requests to boost page exchanges by modi-fying the key /local/domain/<domid>/memory/free_mem in Xenstore. The rates of the requests are 0 MB (“off” case), 50 MB, 100 MB, 150 MB, 200 MB, 300 MB, 400 MB, 600 MB, 800 MB, 1 GB and 1.2 GB per second. Then, the server in Domain0 is triggered at the corresponding rate to reclaim pages from the other nine VMs. Finally, the running time of sunflow or h2 at different rates is recorded to compare with the running time when our system is off (0 MB). We use median confidence intervals for the running time at differ-ent rates by running the test for 20 times.

Fig. 10a shows that the running time of sunflow increases with the memory allocation rate. For example, its running time with 600 MB allocation rate is nearly 1.5 times as high as with 0 MB. When the allocation rate is up to 1.2 GB, its running time is nearly 2 times as high as with 0 MB. This means that our system incurs extra costs for the CPU-inten-sive application because of page remapping and scrubbing. However, the median line of the running time at different rates is linear, and the slope is relatively small.

Fig 10b shows that the the median line of the h2 running time is also linear and has the same slope as that of sunflow. The overload for the memory-intensive application is the same as that for the CPU-intensive one. Additionally, the confidence intervals for both applications increase with the memory allocation rates. This means that if we use higher rates to exchange the pages, more fluctuations are intro-duced for the applications.

In summary, the overloads of our memory control sys-tem linearly increase with the memory allocation rates. In terms of memory-intensive applications, their performance improvement by balancing the memory and avoiding the



a) *apache* swap usage

b) *h2* swap usage

c) *compile linux kernel* swap usage

d) *tradebeans* swap usage

e) *pgbench* swap usage

f) *tradesoap* swap usage

g) *John-the-ripper* swap usage

h) *scimark2* swap usage

i) *System-libxml2* swap usage

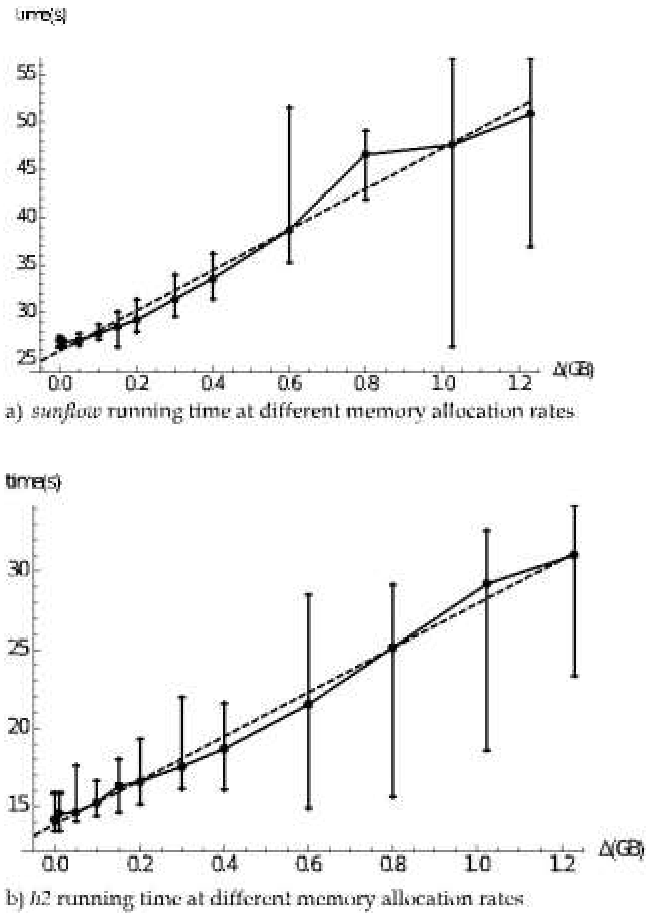
j) compress-*7zip* swap usage

Fig. 9. Comparison of swap space usage by hybrid PTS and DaCapo benchmarks when the memory control system is either "off" or "on".

swap space usage, can outperform the performance degra-dation, which is caused by remapping and scrubbing pages if we carefully control the memory allocation rate.

5 RELATED WORK

Memory control has been extensively studied in the context of VMMs. Modern VMMs save memory using these four



a) *sunflow* running time at different memory allocation rates

b) *h2* running time at different memory allocation rates

Fig. 10. Comparison of running time for CPU-intensive and memory-intensive applications at different memory allocation rates.

main techniques: page sharing, virtual hotplug, live migration, and balloon driver.

Page sharing. Memory can be saved by periodically detecting and sharing the pages of all guest VMs with iden-tical and/or similar content. Cellular Disco [29] first pro-posed this technique by exchanging pages between the physical memory and the disk partition of guest VMs. Sugerman et al. [30] and Waldspurger [20] implemented the process of virtual memory exchange with page sharing in the VMWare Workstation and ESX Server. Waldspurger

[20] further developed page sharing based on the compari-son of page content with consistent hashing. Gupta et al.

[31] integrated page sharing, compression, and patching to utilize more virtual memory. In our system, the page sharing method is applied orthogonally. Performance thus deterio-rates when the scanning of similar pages to and from the disk at a high rate uses increased memory from the CPU and the paging guest.

Virtual hotplug. A virtual hotplug either enhances or reduces memory by deceiving the interfaces of memory management in the Guest OS. This hotplug mimics the inclusion of a physical memory module that is dual in-line. First, a new memory address group is conveyed to the kernel. The kernel then enables the new memory. Schopp et al. [21], [32] researched the principles and implementations of the virtual hotplug in-depth and ana-lyzed the advantages and disadvantages of the virtual hotplug and of balloon drivers. The virtual hotplug comple-ments our approach; the hotplug to add memory has already been integrated into the mainline kernels. As a result, the kernels can utilize additional memory that has not been used to boot them. However, the hotplug to remove memory is separated from the mainline because this hotplug often fails to remove entire sections.

Live migration. To avoid application performance degradation, live virtual machine migration can move a running virtual machine between different physical machines without disconnecting the client or application [34]. Live migration is complementary to our system at different scopes of a data center. Our system aims to solve the memory competition on a consolidated server when some VMs are idle or use less memory, although some VMs are lack of free memory. However, in a heavily consolidated server where all of the VMs are highly utilized, our system cannot improve appli-cation performance, while live migration can move some VMs to other physical machines and reduce the memory burden of the current server.

Balloon driver. The research on the memory control sys-tem based on the Xen balloon driver is most relevant to our study. Zhao et al. [26] proposed a Xen-based memory balancer (MEB) that can predict memory requirements by monitoring memory usage. Memory is then periodi-cally reallocated using this balloon driver. However, our system has three significant advantages over MEB. First, MEB modifies the VMM kernel to intercept memory access and monitor memory usage. This process generates heavy additional overloads and deteriorates VMM perfor-mance. However, our system is lightweight and can be completely incorporated into user space without interfer-ing with VMM operation. Second, MEB uses a quick approximation algorithm to prevent total page misses from reaching a local minimum. Our system determines the optimal allocation of global memory by introducing dynamic baselines and solving linear equations. Finally, MEB verifies the effectiveness of the algorithm using lim-ited resources, e.g., two and four VMs, whereas our sys-tem can scale up to 10 VMs.

Heo et al. [12] used control theory to dynamically allocate memory on Xen VMs. This system is also lightweight and is implemented in user space, as with our system. However, Heo's system is more limited than our system in two ways. First, feedback can be controlled effectively only if the phys-ical memory can accommodate all of the memory requests. Our system can determine the allocation by solving linear equations with the dynamic baseline, which fits both suffi-cient and insufficient physical memory. Moreover, the exper-imental setup of Heo's system is more specific than that of ours. Our system uses real benchmarks, e.g., DaCapo and PTS, on 10 VMs, whereas Heo's system loads two VMs with synthetic and traces-driven work. Users are most concerned with the running times of the benchmarks, although these benchmarks merely record response time and throughput as metrics in operation.

Recent studies [33] utilized application-level ballooning (ALB) on Xen VMs. However, the concept of "application-level" in ALB as presented by these studies is quite different from our definition. In the literature, ALB extends the exist-ing ballooning technique to applications that manage their own memory in consolidated VMs. It must modify both the Linux kernel and the Xen balloon driver. However, our

ALB operates in user space without altering kernel compo-nents and interfering with VMM operation.

6 CONCLUSION

In this study, we devise a system for automatic memory control based on the balloon driver in Xen VMs. Researchers can download our toolkit, which is under a GNU GPL v3 license, for free. Our system aims to opti-mize the running times of applications in consolidated environments by overbooking and/or balancing the memory pages of Xen VMs. Unlike traditional methods, such as MEB, our system is lightweight and can be completely integrated into user space without interfering with VMM operation. We also design a global-scheduling algorithm based on the dynamic baseline to determine the optimal allocation of memory globally. We evaluate our optimized solution to memory allocation using real workloads (DaCapo and PTS) that run across 10 VMs. Some key findings are listed below:

Our system significantly improves the performances of memory-intensive applications. For example, the running time of the h2 application is reduced to a quarter of its original time.

Our system significantly enhances the performances of disk-intensive applications by limiting the swap space usage of applications in other VMs. For exam-ple, the running time of the pgbench application is decreased to one-third of its original time.

Our system is scalable and suitable for various appli-cations. In our experiments on the system, the num-ber of VMs is extended from two or five to 10. The system can also accommodate pure memory-, mem-ory-intensive, and CPU-intensive applications, as well as a combination of memory-, CPU-, and disk-intensive applications.

Our global-scheduling algorithm is adaptive. The dynamic baseline of this algorithm can limit schedul-ing system overload, balance the free memory, and lower memory uniformly.

Our system also hints at the use of the task dis-patcher to balance resource usage in cloud environ-ments with multiple physical machines; when the cloud dispatcher schedules tasks for physical machines, it should deploy different types of appli-cations to VMs on one physical machine. Specifically,

a maximum of one disk-intensive application should be released along with disk- or memory-intensive applications. However, automatic memory control should be activated if many memory-intensive appli-cations are run on one physical machine.

In addition to memory devices, we plan to extend our system to CPU and I/O devices in the future.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers' comments, which are all valuable and very helpful for revising and improving our paper, as well as being important guiding sig-nificance to our research.This work was supported by the National Basic Research Program of China (973 Program) under grant No. 2011CB302605, the National Science Foundation of China (NSFC) under grant No. 61173145, and also the Doctoral Program of Higher Education of China under grant No. 20132302110037.

REFERENCES

[1] A. Fox, R. Griffin, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "Above the clouds: A Berkeley view of cloud computing," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2009-28, p. 13, 2009,

[2] J. E. Smith and R. Nair, Virtual Machines: Versatile Platforms for Systems and Processes. Amsterdam, The Netherlands: Elsevier, 2005.

[3] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in Xen," in Proc. ACM Int. Conf. Middleware, 2006, pp. 342–362.

[4] P. Padala, G. S. Kang, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resour-ces in utility computing environments," ACM SIGOPS Oper. Syst. Rev., vol. 41, no. 3, pp. 289–302, 2007.

[5] W. Zhang, H. Zhang, H. Chen, Q. Zhang, and A. M. K. Cheng, "Improving the QoS of web applications across multiple virtual machines in cloud computing environment," in Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum, 2012, pp. 2247–2253.

[6] W. Zhang, H. He, G. Chen, and J. Sun, "Multiple virtual machines resource scheduling for cloud computing," Appl. Math. Inf. Sci., vol. 7, no. 5, pp. 2089–2096, 2013.

[7] D. Magenheimer, "Memory overcommit. . . without the commitment," Xen Summit, pp. 1–3, 2008.

[8] Xen, the powerful opensource industry standard for virtualiza-tion. (2013). [Online]. Available: http://www.xenproject.org/

[9] VMware virtualization software for desktops, servers & virtual machines for public and private cloud solutions. (2013). [Online]. Available: http:// www.vmware.com

[10] KVM. Kernel Based Virtual Machine. (2013). [Online]. Available: http://www.linux-kvm.org/page/Main_Page

[11] W. Zhang, T. Cheng, H. He, and A. M. K. Cheng, "LVMM: A lightweight virtual machine memory management architecture for vir-tual computing environment," in Proc. Int. Conf. Uncertainty Reasoning Knowl. Eng., 2011, vol. 1, pp. 235–238.

[12] J. Heo, X. Zhu, P. Padala, and Z. Wang, "Memory overbooking and dynamic control of Xen virtual machines in consolidated environments," in Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage., 2009, pp. 630–637.

[13] W. Zhao, Z. Wang, and Y. Luo, "Dynamic memory balancing for virtual machines," ACM SIGOPS Oper. Syst. Rev., vol. 43, no. 3, pp. 37–47, 2009.

[14] Amazon Elastic Computing Cloud (EC2). (2013). [Online]. Avail-able: http://aws.amazon.com/ec2

[15] Amazon Data Center Size. (2012). [Online]. Available: http:// huanliu.wordpress.com/2012/03/13/amazon-data-center-size/

[16] Anatomy of an Amazon EC2 Resource ID. (2009). [Online]. Avail-able: http://www.jackofallclouds.com/2009/09/anatomy-of-an-amazon-ec2-resource-id/

[17] Xen Memory Management System. (2014). [Online]. Available: https://github.com/wzzhang-HIT/xenmm/

[18] I. Pratt, Xen Status Report. Cambridge, U.K.: Univ. Cambridge, 2005.

[19] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," ACM SIGOPS Oper. Syst. Rev., vol. 37, no. 5, pp. 164–177, 2003.

[20] C. A. Waldspurger, "Memory resource management in VMware ESX server," ACM SIGOPS Oper. Syst. Rev., vol. 36, no. SI, pp. 181–194, 2002.

[21] J. H. Schopp, K. Fraser, and M. J. Silbermann, "Resizing memory with balloons and hotplug," in Proc. Linux Symp., 2006, vol. 2, pp. 313–319.

[22] XEN. (2013). XenStore Reference [Online]. Available: http://wiki. xen.org/wiki/XenStoreReference

[23] M. Wilding and D. Behman, Self Service Linux. Apogeo Editore, 2006.

[24] C. A. Waldspurger and W. W. Weihl, "An object-oriented frame-work for modular resource management," in Proc. 5th Int. Work-shop Object-Orientation Oper. Syst., 1996, pp. 138–143.