Automatic Memory Control of Multiple Virtual Machines on a Consolidated Server

Wei-Zhe Zhang, *member*, *IEEE*, Hu-Cheng Xie, and Ching-Hsien Hsu, *senior member*, *IEEE*

Abstract—Through virtualization, multiple virtual machines can coexist and operate on one physical machine. When virtual machines (VMs) 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%, 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 computing [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 applications and maximize resource utilization.

Automatic control systems for CPU devices have been widely researched [4—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 exponentially, 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:

(1) Tools for automatic memory control at the application 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 hotplugs*, and *balloon drivers* in the virtual machine monitor (VMM). 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 automatically 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 scheduling algorithms must be in a *sufficient* state, and *self-ballooning* was not observed with a global 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 minimum. To avoid this local minimum and to attain optimal performance, additional algorithms should thus be developed for general global scheduling.

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

- *W.-Z Zhang is with the School of Computer Science and Technology, Harbin Institute of Technology, China. E-mail: wzzhang@hit.edu.cn*
- *H.-C Xie is with the School of Computer Science and Technology, Harbin Institute of Technology, China. E-mail: xiehuc@gmail.com*
- *C.-H. Hsu is with the Department of Computer Science and Information Engineering, Chung Hua University, Taiwan. E-mail: chh@chu.edu.tw.*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing

driven. Therefore, more tests should be conducted with additional VMs and real benchmarks.

In this study, we devise a lightweight framework based on the Xen balloon driver to control memory in the consolidation 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 *insufficient* states using dynamic baselines. Real-world benchmarks 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 presented 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 the total 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 process of inflation and deflation, with two VMs (VM1 and VM2) as examples. The left side of Fig. 1 represents the initial *page* allocation of the VMs. The right side represents the remapped *page* allocation.

To inflate the balloon, first, the hypervisor sends an inflation 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 corresponding *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 allocate *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 hypervisor and remapped to the ballooned *pages* in VM2.

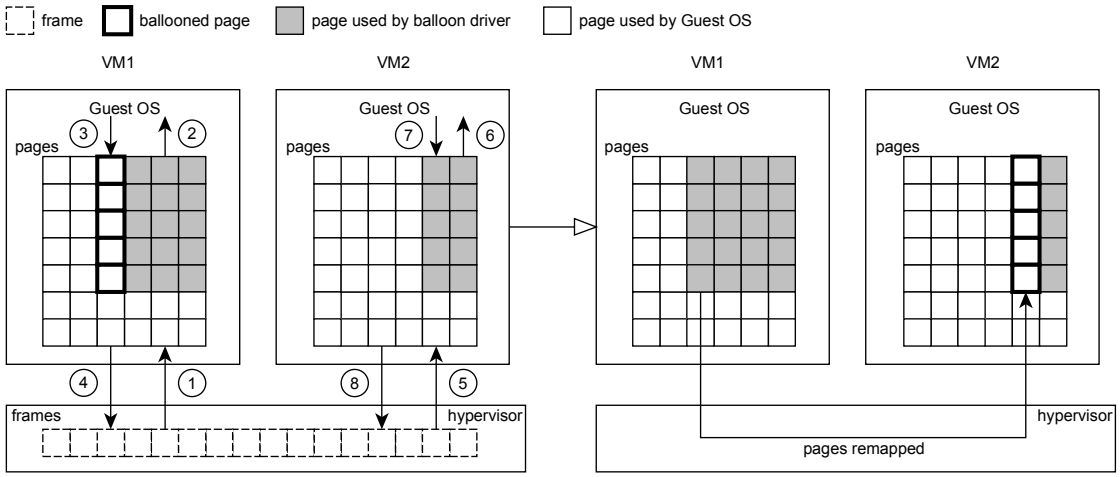


Fig. 1 Mechanism of the Xen balloon driver

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing

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.

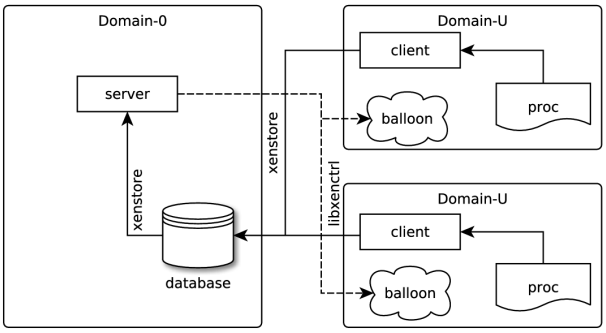


Fig. 2 Our automatic memory control system

- *Domain*: A domain is a VM that is operating on a system. On boot, the Xen hypervisor activates the first 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 allocating the memory *pages* across various domains.
- *XenStore*: This is a hierarchical namespace shared between domains, which stores the running information 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 information related to kernel and system processes [25]. The directory */proc/meminfo* stores the memory information. *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 hypercall; *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:

  $$MemUsed = MemTotal - MemFree - Cached - Buffers$$

  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 scheduling algorithm of *Server* then determines the domain that requires additional *pages*, as well as the domain that provides these extra *pages*. The scheduling algorithm also calculates the optimal target *pages* for allocation 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 system 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-scheduling* 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 memory. The *global-scheduling* algorithm is used to overcommit memory globally.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing

3.1 Global-scheduling Algorithm

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

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 i^{th} VM, $1 \leqslant i \leqslant n$
N_{ti}	Target memory allocated to the i^{th} VM, $1 \leqslant i \leqslant n$
F_i	Free memory of the i^{th} VM, $1 \leqslant i \leqslant n$
A_i	Used memory of the i^{th} VM, which equals to $(N_i - F_i)$, $1 \leqslant i \leqslant n$
A	$\{A_i \mid i = 1....n\}$
$\overline{A}$	$\overline{A} = \frac{1}{n}\sum_{i=1}^{n} A_i$

The *global-scheduling* algorithm is given as follows:

Algorithm 1: *global-scheduling* algorithm
Input: N, n, N_i, A_i
Output: Nt_i

```
1.  while true do
2.  A ← Null
3.  for 1≤i≤n do
4.      N_i ← xs_read (/local/domain/VM_i/mem/total);
5.      F_i ← xs_read (/local/domain/VM_i/mem/free);
6.      A_i = N_i - F_i;
7.      AppendTo(A, A_i);
8.  end
9.  τ ← calculating_idle_memory_tax(A, f);
10. for 1≤i≤n do
11.     Nt_i ← solve_linear_equation(N_i, A_i, τ);
12.     xs_write(Nt_i, /local/domain/VM_i/mem/target);
13.     xc_domain_set_pod_target(VM_i, Nt_i);
14. end
15. sleep(interval);
16. 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 τ (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 (τ) is adopted from economic theory. It levies a high tax for idle memory on a VM that does not maximize 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 client can obtain resources based on its shares. To allocate space-shared resources in terms of proportional share, both randomized and deterministic algorithms are generated. If a client requests additional resources, the dynamic algorithm for min-funding revocation determines the client with the fewest shares and reallocates its resources [20, 24, and 25].

In practice, a VM with many shares acquires much memory, 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 ρ is:

$$\rho = \frac{S}{P(Q + k(1-Q))}$$

where the cost of idle pages is $k = 1/(1-\tau)$ for a given tax rate τ ($0 \le \tau < 1$). On one side, $\tau = 0$ specifies the pure allocation based on shares. On the other side, $\tau \approx 1$ specifies that the idle *pages* of all VMs can be reclaimed and allocated equally.

Supposing that in each VM, $S = 1$, $P = N_i$, and that the proportion of active sections in total memory is $Q_i = A_i / N_i$, the shares-per-page ρ_i of each VM is:

$$\begin{aligned} \rho_i &= \frac{1}{A_i + k(N_i - A_i)} \\ &= \frac{1}{A_i + \frac{1}{1-\tau}(N_i - A_i)} \\ &= \frac{1-\tau}{N_i - \tau \times A_i} \end{aligned} \quad (1)$$

3.3 Calculating Target Memory by Solving Linear Equations

Our system aims to guarantee identical shares-per-page ρ_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 supply. If the price of idle memory is high in VMs with insufficient 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 balance 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 re-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing

main equal.

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

$$\forall i, j \in (1...n), i \neq j : \frac{1-\tau}{N_{ti} - \tau \times A_i} = \frac{1-\tau}{N_{tj} - \tau \times A_j}$$

The linear equations are then expressed as follows:

$$\begin{cases} N_{t1} - \tau \times A_1 = N_{t2} - \tau \times A_2 \\ N_{t1} - \tau \times A_1 = N_{t3} - \tau \times A_3 \\ \quad ... \\ \quad ... \\ N_{t1} - \tau \times A_1 = N_{tn} - \tau \times A_n \\ N = \sum_{i=1}^{n} N_{ti} \end{cases} \quad (2)$$

We simplify Equation (2) as follows:

$$\begin{cases} N_{t1} + ... + N_{tn} = N \\ N_{t1} - N_{t2} = \tau \times (A_1 - A_2) \\ N_{t1} - N_{t3} = \tau \times (A_1 - A_3) \\ ... \\ N_{t1} - N_{tn} = \tau \times (A_1 - A_n) \\ N = \sum_{i=1}^{n} N_{ti} \end{cases} \quad (3)$$

Finally, Equation (3) can be simplified as follows:

$$\boldsymbol{AX} = \boldsymbol{B} \quad (4)$$

where $\boldsymbol{A}=\begin{pmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & -1 & 0 & ... & 0 \\ 1 & 0 & -1 & ... & 0 \\ & ... & ... & ... & \\ 1 & 0 & 0 & ... & -1 \end{pmatrix}$, $\boldsymbol{X}=\begin{pmatrix} N_{t1} \\ N_{t2} \\ N_{t3} \\ .... \\ N_{tn} \end{pmatrix}$, $\boldsymbol{B}=\begin{pmatrix} N \\ \tau(A_1 - A_2) \\ \tau(A_1 - A_3) \\ .... \\ \tau(A_1 - A_n) \end{pmatrix}$

Coefficient matrix A is ranked n, and the rank of a full rank matrix is equal to that of its augmented matrix. The unique solution provided by Equation (4) can be expressed as follows:

$$\boldsymbol{X}=\begin{pmatrix} \frac{N}{n} + \tau \times (A_1 - \overline{A}) \\ \frac{N}{n} + \tau \times (A_2 - \overline{A}) \\ \frac{N}{n} + \tau \times (A_3 - \overline{A}) \\ .... \\ \frac{N}{n} + \tau \times (A_n - \overline{A}) \\ \end{pmatrix}$$

where $\overline{A} = \frac{1}{n}\sum_{i=1}^{n} A_i$

Finally, we obtain the following solution:

$$\forall i \in (1...n) : N_{ti} = \frac{N}{n} + \tau \times (A_i - \overline{A}) \quad (5)$$

The time complexity of the *global-scheduling* algorithm is determined by solving the linear equations. The complexity of constructing linear equations is denoted by $O(n)$, and the process of solving these linear equations is represented by $O(n^2)$. Finally, the time complexity of the algorithm is denoted by $O(n^3)$.

3.3 Calculating Idle Memory Tax based on the Dynamic Baseline

The selection of the value of idle memory tax (τ) is key to our system. In this section, we derive the function of idle memory tax based on a dynamic baseline.

Equation (5) determines the target memory of VMs. The target free memory (F_{ti}) after reallocation should correspond to the difference between target memory (N_{ti}) and currently used memory (A_i). To avoid swap space usage and to ensure the effectiveness of each VM, the target free memory should be greater than zero. In practice, the OS uses swap space as long as its free memory is below a specific threshold ξ_0. As a result, the solution of Equation (5) should guarantee that:

$$\forall i \in (1...n) : F_{ti} = N_{ti} - A_i \geq \xi_0 > 0$$

We then replace N_{ti} with Equation (5):

$$\frac{1}{n} - \tau \times \overline{A} - (1-\tau) \times \max(A) \geq \xi_0$$

The dynamic τ is expressed as follows:

$$\tau = \begin{cases} \dfrac{\xi_0 + \max(A) - 1/n}{\max(A) - \overline{A}} & \max(A) \neq \overline{A} \\ 0 & \max(A) = \overline{A} \end{cases}$$

where $\xi_0 = f/N$, f is the minimal free memory reserved, and N is the total memory of the VMs.

Fig. 3 compares the curves of static and dynamic τ, which vary with the currently used memory A_i. The curves indicate that only part of static τ (e.g., τ = 0.75 or 1.0) is above the dynamic baseline τ_b (τ_b denotes the baseline when $\xi_0 = 0$ and τ denotes the curve when $\xi_0 > 0$), except when $\tau = 0$. For example, the value of τ=0.75 is lower than that of τ_b after the two curves τ = 0.75 and τ_b intersect, thus resulting in a target free memory that is less than zero. In this process, swap space is used. If τ is statically set to zero, Equation (5) equalizes all of the target memory values. Therefore, memory balance is unaffected.

Fig. 3 also shows three advantages of the dynamic τ:

- *Reducing the overload of the scheduling system*: The value of τ is set to 0.0 when the memory used is small and does not use swap space. Our system is thus degraded so that it need not be regulated; as a result, scheduling system overload is reduced.
- *Reserving the free memory:* To improve the performance of VMs, dynamic τ can reserve some available memory. When the current free memory of VMs is below the reserved value, our system reclaims more memory *pages* from other VMs.
- *Dropping the memory uniformly*: The value of τ is set to 1.0 when the total available memory cannot satisfy the requests of all VMs. With this setting, all VMs can

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing

share the remaining memory available. The available memories of all VMs drop uniformly.

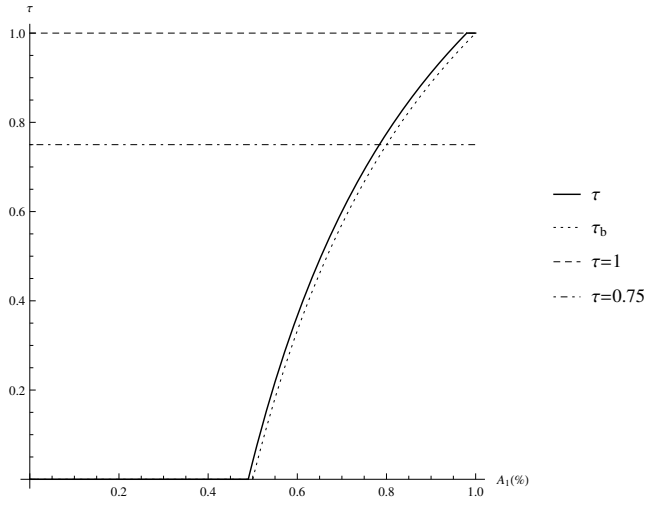


Fig. 3 Comparison of static and dynamic τ curves

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 reporting 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*, *sunflow*, *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 corresponding 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 profiles and 60 test suites. These tests range from traditional 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 profiles 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 (*Composite 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 frequency is 2100 MHz, its cache size is 2048 KB, and its memory size is 128 GB. The OS used is Ubuntu 12.04 LTS 64-bit, whose kernel is Linux 3.2.0-29-generic.

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-window 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 performance, the minimal free memory reserved *f* is set within the range of 100 MB to 150 MB.

4.2 Validation of Two VMs using Mono Benchmark

We aim to verify whether our system can successfully regulate 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 MB to 500 MB before monotonically lowering it from 500 MB 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 $\tau = 0$, and the balloon driver does not need to regulate the memory between VMs. When the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing

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.

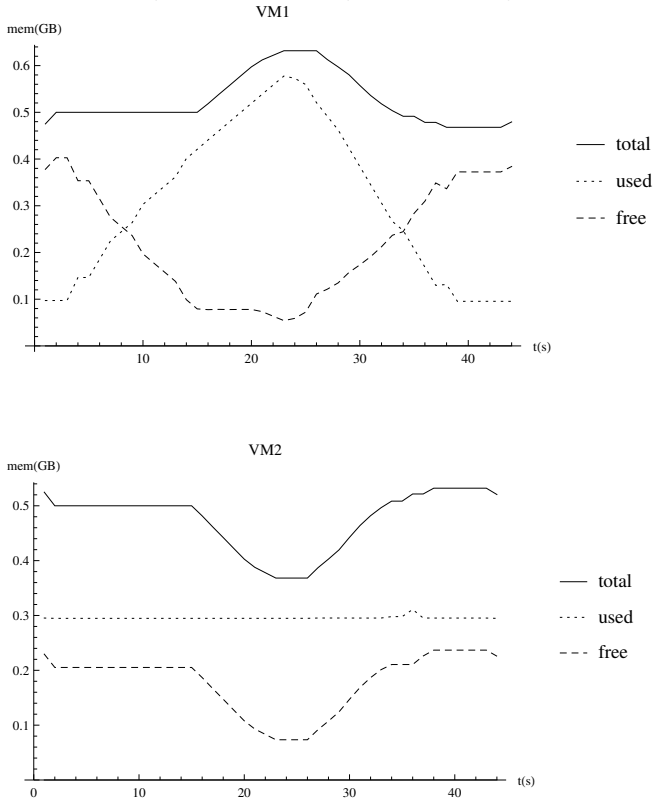


Fig. 4 Automatic control of memory in two VMs using the Mono benchmark

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.

Five VMs are configured with 1 GB of initial available memory and 150 MB of reserved free memory (*f*). *DaCapo* runs on only one VM with a workload that allows for an additional 800 MB of memory. The other four VMs are idle. When the VM that is running *DaCapo* is out of memory, our system reclaims memory from the other VMs.

Fig. 5 displays the running times of 13 applications from the *DaCapo* benchmark suite when our system of automatic memory control is either "*on*" or "*off*." A logarithmic y-axis is used to reduce a wide scale to a more manageable size. We also provide median confidence intervals for each plotted value by running the test for 20 times. The median confidence interval is represented by a box and whisker chart. The top and the bottom of the box indicate a 75% and 25% percentile running time, respectively, and a line in the middle of the box indicates the median.

When our system is "*off*", the running time (median value) of the *h2* application is more than six times longer than when the system is "*on*." Similarly, the running times of the *eclipse, tradebeans*, and *tradesoap* applications are two or three times longer. The running times of the *fop*, *jython*, *pmd*, and *tomcat* applications are also slightly longer. However, the running times of the *avrora*, *luindex*, *lusearch*, *sunflow*, and *xalan* applications are slightly shorter, but the decrease is no more than 8%. Note that most of the applications displaying significant performance differences are either memory- or disk-intensive.

Additionally, when our system is "*off*", the differences between the upper and lower fences are nearly triple (*h2* application). When our system is "*on*", their running times remain stable and the differences are no more than 15% (*h2* application).

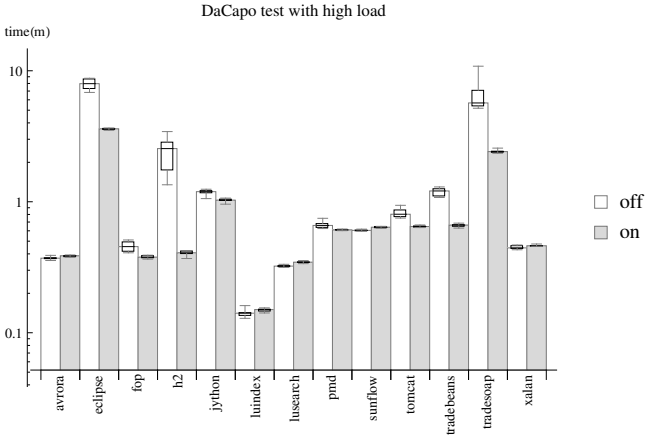


Fig. 5 Comparison of the running time of the DaCapo benchmark suite when the memory control system is either "*off*" or "*on*"

The short running time for most applications during system activation is attributed to the fact that our system does not use swap space. Fig. 6 exhibits swap space usage with or without memory control by the VM running *DaCapo*. When the memory control is "*off*", nearly 500 MB of swap space is used. However, no swap space is used when our memory control is "*on*". Additionally, for the same reason, when our system is "*on*", the applications do not suffer frequent access to the swap space, their running time thus can remain stable and the confidence intervals are narrow. Instead, when our system is "*off*", the applications show huge performance fluctuations because of frequent swap space accesses. Note that the running times of the *avrora*, *luindex*, *lusearch*, *sunflow*, and *xalan* applications for the "*off*" case are slightly shorter than the "*on*" case. The reason for this is two-fold: 1) these applications are all CPU-intensive, which are not very sensitive to memory fluctuations but compete for CPU cycles; 2) when our system remaps *pages* to other VMs, the balloon driver and the hypervisor will take up some CPU cycles to reclaim *GPFNs*, scrub them by filling with zeros for security reasons, detach the *MFNs* behind these *GPFNs*, and associate the *MFNs* with the *GPFNs* in new VMs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing

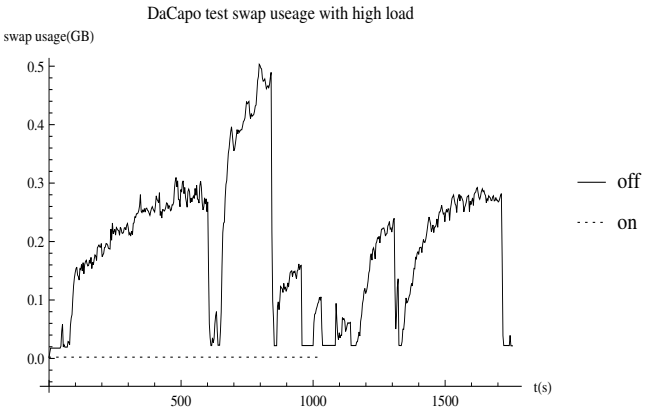


Fig. 6 Comparison of swap space usage by the DaCapo benchmark suite when the memory control system is either "*off*" or "*on*"

In terms of memory allocation, Fig. 7 indicates that when VM1 (which runs *DaCapo*) is out of memory, our system can reclaim memory from the other VMs. Given that the initial memory sizes of the other idle VMs are equal, the reclaimed memory size is similar and their curves overlap.

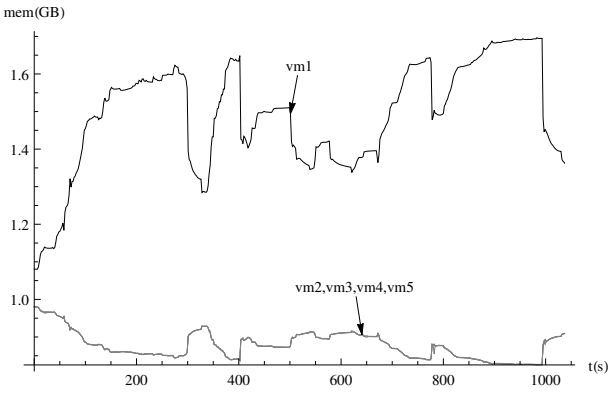


Fig. 7 Allocation of memory among five VMs when the memory control system is "*on*"

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 composed 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*

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.

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	161.528
	FFT	36.278	36.7225
	Sparse Matrix Multiply	420.94	441.843
	Dense LU Matrix Factorization	923.814	953.023
	Jacobi Successive Over-Relaxation	430.304	430.375
System-libxml2		171762	178210
compress-7zip		1361.75	1480.25

The benchmark *John-the-ripper* contains three subtests (*blowfish*, *traditional DES,* and *MD5*) whereas *scimark2* has six (*Composite*, *Monte Carlo*, *FFT*, *Sparse Matrix Multiply*,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing

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 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 *compress-7zip*) likewise obtain higher scores. The scores of the *John-the-ripper* and *scimark2* are just slightly higher. However, the scores of *apache, compile-linux-kernel, tradebean*, and *System-libxml2* drop by approximately 10% when the system is "*on*".

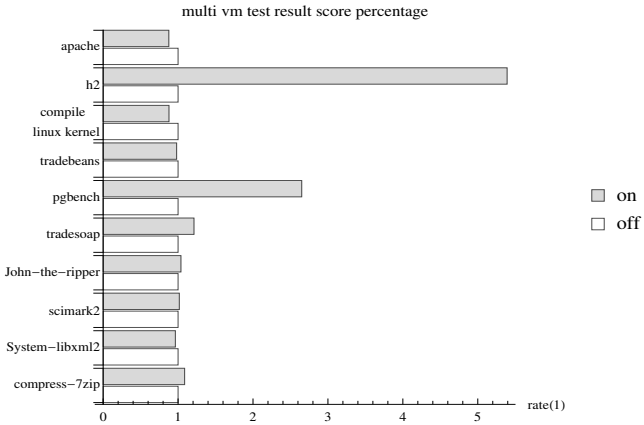


Fig. 8 Comparison of the scores of *PTS* & *DaCapo* hybrid benchmarks when memory control system is "*off*" or "*on*"

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

Fig. 9 a), Fig. 9 c), and Fig. 9 i) 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.

Fig. 9 b), Fig. 9 f), and Fig. 9 j) 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 underlying database to benchmark the *daytrader* via a SOAP to a Geronimo backmend. *Compress-7zip* measures the file compression with *p7zip* to test the processor and memory. These benchmarks are all memory-intensive applications and frequently 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.

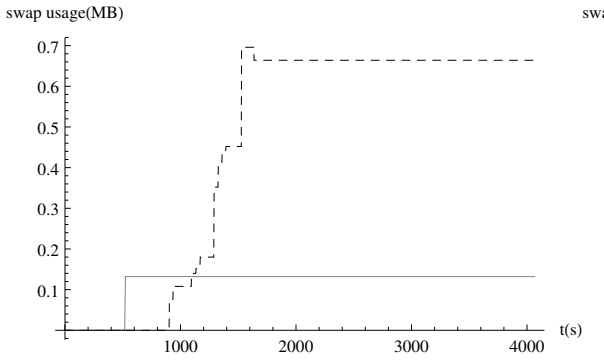
Fig. 9 d) shows the swap space usage for the *tradebean* benchmark. *Tradebean* is also memory-intensive, which uses *h2* as the database to benchmark the *daytrader* via a JavaBeans to a Geronimo backend. However, at most of the time, its swap space usage is approximately 20 MB and keeps stable except 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 ξ_0. In this test, the value of ξ_0 happens to be between 15 MB and 20 MB, which means that enough free memory is still available. As a result, when our system is "*on*", its performance improvement is not obvious because of the stable and small swap space usage below the threshold ξ_0.

Fig. 9 e) shows that *pgbench* occupies surprisingly less swap space than expected, compared to its triple performance 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 benchmark, uses disk caches for caching data, and rarely relies on swap space usage to avoid double paging. Additionally, VMs frequently access the swap space when they are out of memory. Hence, many I/O operations are intensified. These functions interfere with the normal I/O operations 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 performances of disk-intensive applications when the system is out of memory.

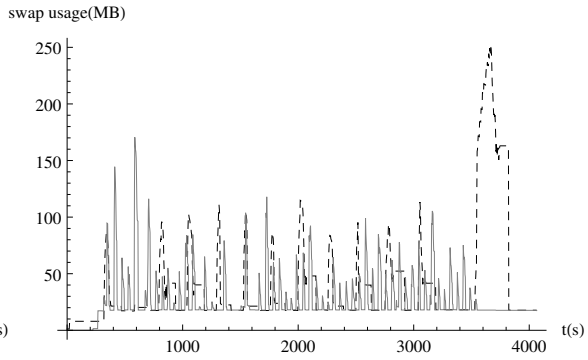
Fig. 9 g) and Fig. 9 h) show that the swap space usage for *John-the-ripper* and *scimark2* is nearly zero no matter when our system is "*on*" or "*off*". *John-the-ripper* is a password cracker, and *scimark2* is for scientific and numerical computing and includes Fast Fourier Transform, Jacobi Successive Over-relaxation, Monte Carlo, Sparse Matrix Multiply, and dense LU matrix factorization benchmarks. Both are CPU-intensive 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.

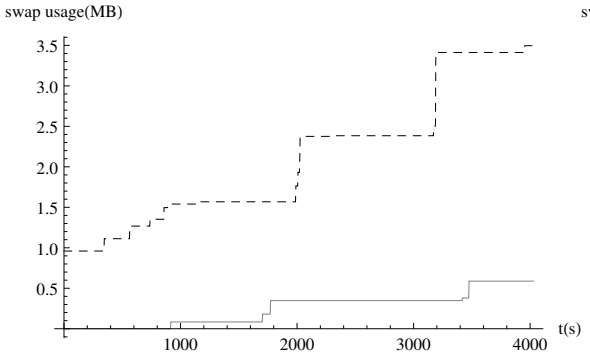
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing



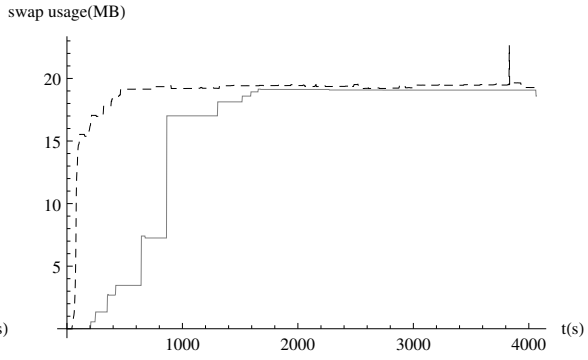
a) *apache* swap usage



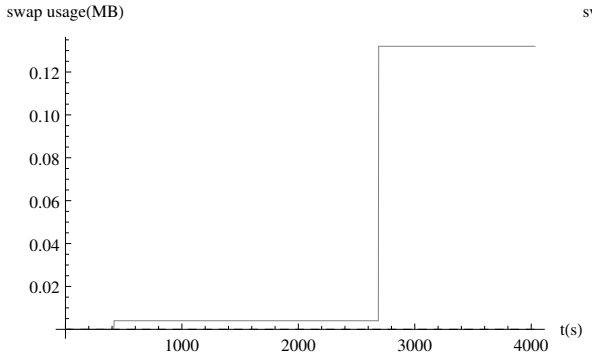
b) *h2* swap usage



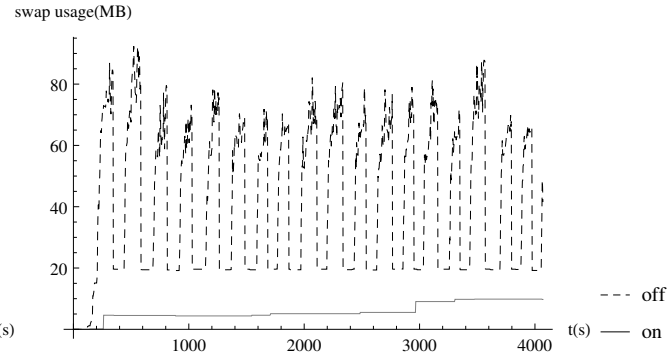
c) *compile linux kernel* swap usage



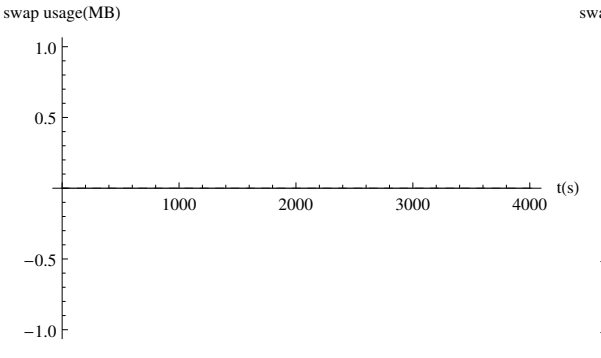
d) *tradebeans* swap usage



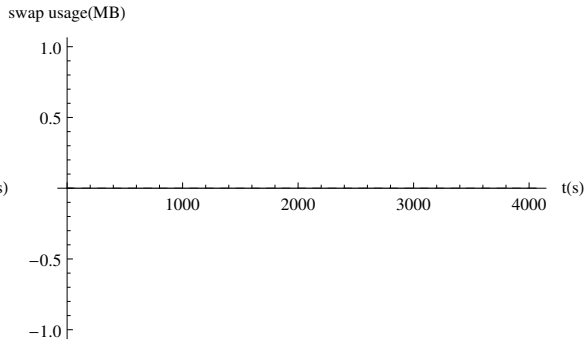
e) *pgbench* swap usage



f) *tradesoap* swap usage

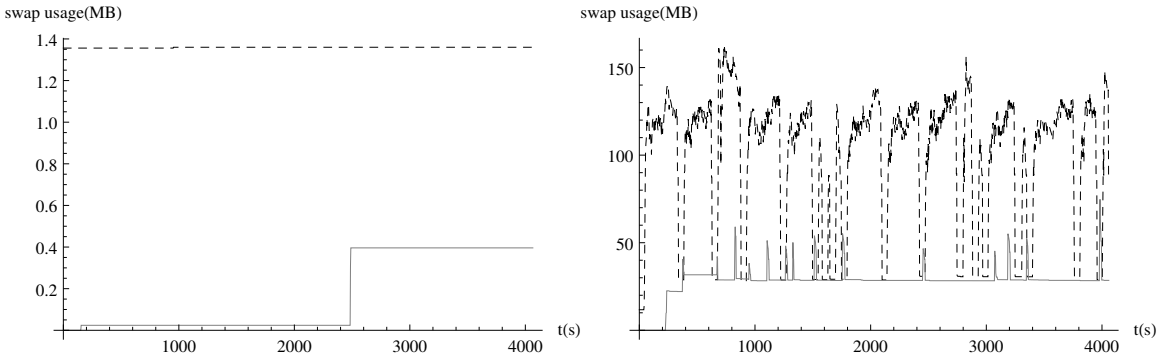


g) *John-the-ripper* swap usage



h) *scimark2* swap usage

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing



i) *System-libxm12* 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*"

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*.

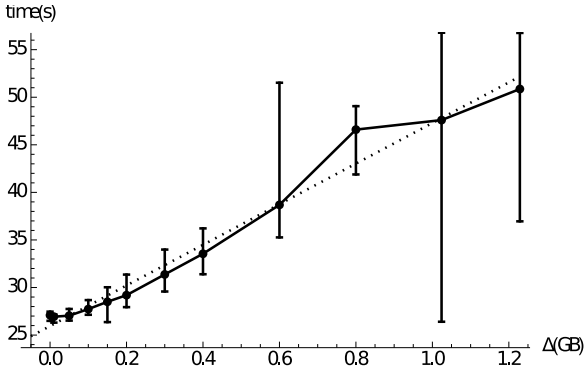
Each of ten VMs is configured with 12 GB of initial available memory and 300 MB of reserved free memory (*f*). On one VM, we deploy a CPU-intensive benchmark (*sunflow*) or a memory-intensive one (*h2*) together with a special tool we developed, which can continue claiming memory at adjustable rates.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

This test occurs in the following way: First, we run *sunflow* or *h2* in the target VM while our tool, resides in the same VM, sends requests to boost page exchanges by modifying 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 different rates by running the test for 20 times.

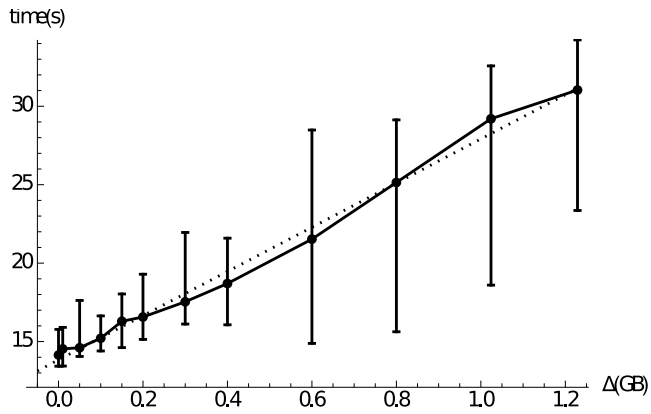
Fig 10. a) shows that the running time of *sunflow* increases with the memory allocation rate. For example, its running time with 600 MB allocation rate is nearly 1.5 times as high as with 0 MB. When the allocation rate is up to 1.2 GB, its running time is nearly 2 times as high as with 0 MB. This means that our system incurs extra costs for the CPU-intensive 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 10. b) 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 introduced for the applications.

In summary, the overloads of our memory control system linearly increase with the memory allocation rates. In terms of memory-intensive applications, their performance improvement by balancing the memory and avoiding the swap space usage, can outperform the performance degradation, which is caused by remapping and scrubbing *pages* if we carefully control the memory allocation rate.



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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2014.2378794, IEEE Transactions on Cloud Computing

5 Related Work

Memory control has been extensively studied in the context of VMMs. Modern VMMs save memory using these four 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 identical and/or similar content. Cellular Disco [29] first proposed 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 comparison 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 deteriorates 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 inline. 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 analyzed the advantages and disadvantages of the *virtual hotplug* and of balloon drivers. The *virtual hotplug* complements 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 application 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 system 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 periodically 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 performance. However, our system is lightweight and can be completely incorporated into user space without interfering 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 limited resources, e.g., two and four VMs, whereas our system 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 physical 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 *sufficient* and *insufficient* physical memory. Moreover, the experimental 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 existing 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 components 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 optimize 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 perfor-

mances of disk-intensive applications by limiting the swap space usage of applications in other VMs. For example, the running time of the *pgbench* application is decreased to one-third of its original time.

- Our system is scalable and suitable for various applications. In our experiments on the system, the number of VMs is extended from two or five to 10. The system can also accommodate pure memory-, memory-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 scheduling system overload, balance the free memory, and lower memory uniformly.
- Our system also hints at the use of the task dispatcher to balance resource usage in cloud environments with multiple physical machines; when the cloud dispatcher schedules tasks for physical machines, it should deploy different types of applications 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 applications 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

We thank the anonymous reviewers' comments, which are all valuable and very helpful for revising and improving our paper, as well as being important guiding significance to our research.

This work is 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] Fox, Armando, Rean Griffith, 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. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS 28 (2009): 13.

[2] Smith, James Edward, and Ravi Nair. "Virtual machines: versatile platforms for systems and processes." Elsevier, 2005.

[3] Gupta, Diwaker, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. "Enforcing performance isolation across virtual machines in Xen." In *Middleware* 2006, pp. 342-362. Springer Berlin Heidelberg, 2006.

[4] Padala, Pradeep, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. "Adaptive control of virtualized resources in utility computing environments." In *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 289-302. ACM, 2007.

[5] Zhang, Weizhe, Hongli Zhang, Huixiang Chen, Qizhen Zhang, and Albert MK Cheng. "Improving the QoS of Web Applications across Multiple Virtual Machines in Cloud Computing Environment." In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pp. 2247-2253. IEEE, 2012.

[6] Zhang, Weizhe, Hui He, Gui Chen, and Jilong Sun. "Multiple Virtual Machines Resource Scheduling for Cloud Computing." *Applied Mathematics & Information Sciences* 7, no. 5 (2013).

[7] Magenheimer, Dan. "Memory overcommit... without the commitment." *Xen Summit (2008)*: 1-3.

[8] Xen, the powerful opensource industry standard for virtualization. http://www.xenproject.org/

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

[10] KVM. Kernel Based Virtual Machine. http://www.linux-kvm.org/page/Main_Page

[11] Zhang, Weizhe, Tao Cheng, Hui He, and A. M. K. Cheng. "LVMM: A lightweight virtual machine memory management architecture for virtual computing environment." In *Uncertainty Reasoning and Knowledge Engineering (URKE), 2011 International Conference on*, vol. 1, pp. 235-238. IEEE, 2011.

[12] Heo, Jin, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. "Memory overbooking and dynamic control of Xen virtual machines in consolidated environments." In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on*, pp. 630-637. IEEE, 2009.

[13] Zhao, Weiming, Zhenlin Wang, and Yingwei Luo. "Dynamic memory balancing for virtual machines." *ACM SIGOPS Operating Systems Review* 43, no. 3 (2009): 37-47.

[14] Amazon Elastic Computing Cloud (EC2). http://aws.amazon.com/ec2

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

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

[17] Xen Memory Management System. https://github.com/wzzhang-HIT/xenmm/

[18] Ian Pratt. Xen status report. Cambridge, UK: University of Cambridge, 2005

[19] Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the art of virtualization." *ACM SIGOPS Operating Systems Review* 37, no. 5 (2003): 164-177.

[20] Waldspurger, Carl A. "Memory resource management in VMware ESX server."*ACM SIGOPS Operating Systems Review* 36, no. SI (2002): 181-194.

[21] Schopp, Joel H., Keir Fraser, and Martine J. Silbermann. "Resizing memory with balloons and hotplug." In *Proceedings of the Linux Symposium*, vol. 2, pp. 313-319. 2006.

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

[23] Wilding, Mark, and Dan Behman. Self service Linux. Apogeo Editore, 2006.

[24] Waldspurger, Carl A., and W. W. Weihl. "An object-oriented framework for modular resource management." In *Object-*

Orientation in Operating Systems, 1996., Proceedings of the Fifth International Workshop on, pp. 138-143. IEEE, 1996.

[25] Waldspurger, Carl A., and William E. Weihl. "Lottery scheduling: Flexible proportional-share resource management." *In Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, p. 1. USENIX Association, 1994.

[26] Zhao, Weiming, Zhenlin Wang, and Yingwei Luo. "Dynamic memory balancing for virtual machines." *ACM SIGOPS Operating Systems Review* 43, no. 3 (2009): 37-47.

[27] The DaCapo Benchmark Suite. http://www.dacapobench.org/

[28] The Phoronix Test Suite. http://www.phoronix-test-suite.com/

[29] Govil, Kinshuk, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. "Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors." In *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, pp. 154-169. ACM, 1999.

[30] Sugerman, Jeremy, Ganesh Venkitachalam, and Beng-Hong Lim. "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor." In *USENIX Annual Technical Conference, General Track*, pp. 1-14. 2001.

[31] Gupta, Diwaker, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. "Difference engine: Harnessing memory redundancy in virtual machines." *Communications of the ACM* 53, no. 10 (2010): 85-93.

[32] Schopp, Joel, Dave Hansen, Mike Kravetz, Hirokazu Takahashi, Toshihiro IWAMOTO, Yasunori Goto, Hiroyuki Kamezawa, Matt Tolentino, and Bob Picco. "Hotplug memory redux." In *Linux Symposium*, p. 151. 2005.

[33] Salomie, Tudor-Ioan, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. "Application level ballooning for efficient server consolidation." In *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 337-350. ACM, 2013.

[34] Clark, Christopher, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. "Live migration of virtual machines." In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273-286. USENIX Association, 2005.

Weizhe Zhang is currently a professor in the School of Computer Science and Technology at Harbin Institute of Technology, China. His research interests are primarily in parallel computing, distributed computing, cloud and grid computing, and computer network. He has published more than 100 academic papers in journals, books, and conference proceedings. He is a member of the IEEE.

Hucheng Xie received a bachelor's degree from the School of Computer Science and Technology at Harbin Institute of Technology, China in 2013. He is currently a master's candidate at Harbin Institute of Technology. His research interest involves virtualization techniques for cloud computing.

Ching-Hsien Hsu is currently a professor in the Department of Computer Science and Information Engineering at Chung Hua University, Taiwan. His research interests are primarily in parallel and distributed computing, cloud and Grid computing, P2P computing, RFID, services computing, and smart homes. He has published more than 150 academic papers in journals, books, and conference proceedings. He was awarded the annual outstanding research award in 2005, 2006, 2007, and 2010, and a distinguished award in 2008 for excellence in research from Chung Hua University. Dr. Hsu serves on a number of journal editorial boards. He has edited more than 20 international journal special issues as a guest editor and has served many international conferences as a chair or committee member. He serves on the executive committee of the IEEE Technical Committee on Scalable Computing (TCSC) and is a senior member of the IEEE.