Hotplug or Ballooning: A Comparative Study on Dynamic Memory Management Techniques for Virtual Machines

Haikun Liu, *Member, IEEE*, Hai Jin, *Senior Member, IEEE*, Xiaofei Liao, *Member, IEEE*, Wei Deng, Bingsheng He, and Cheng-zhong Xu, *Senior Member, IEEE*

Abstract—In virtualization environments, static memory allocation for *virtual machines* (VMs) can lead to severe service level agreement (SLA) violations or inefficient use of memory. Dynamic memory allocation mechanisms such as ballooning and memory hotplug were proposed to handle the dynamics of memory demands. However, these mechanisms so far have not been quantitatively or comparatively studied. In this paper, we first develop a runtime system called U-tube, which provides a framework to adopt memory hotplug or ballooning for dynamic memory allocation. We then implement fine-grained memory hotplug in Xen. We demonstrate the effectiveness of U-tube for dynamic memory management through two case studies: dynamic memory balancing and memory overcommitment. With these two case studies, we make a quantitative comparison between memory hotplug and ballooning. The experiments show that there is no absolute winner for different scenarios. Our findings can be very useful for practitioners to choose the suitable dynamic memory management techniques in different scenarios.

Index Terms—Ballooning, memory management, memory hotplug, virtual machine, virtualization

1 Introduction

Virtualization provides a significant advantage for modern data centers to maximize physical resource utilization through server consolidation. Continuous advances of multi-core and I/O virtualization technologies [12], [18], [19], [36] have caused main memory to be a more valuable resource. It has become the primary capacity constraint for VM density and VM performance [25]. On the other hand, although many data intensive applications [26] have become memory hungry, they exhibit significantly different memory consumption behaviors in terms of memory footprint and temporal memory usage. Effective memory allocation among different VMs remains a challenging research problem.

In virtualization environments, most hypervisors typically allocate a fixed-size memory pool to each VM instance at boot. A VM may be also configured with a parameter such as *max_memory* to specify the VM's maximum memory capacity. We call it memory cap in this paper. As static memory allocation may lead to significant performance degradation or a waste of precious memory resource, ballooning is widely used in the state-of-the-art hypervisors such as VMware and Xen [7], [32] for dynamic memory management. It enables the virtual machine monitor (VMM) to reclaim underutilized memory from a lightly loaded VM and re-allocate it to overloaded VMs.

Although ballooning is widely used for VM memory resizing [11], [17], [39], it still has some limitations in several scenarios. In general, it is difficult to predict the exact demands of memory resource before a VM is created, so the maximum memory capacity is usually set by experience. However, in practice, the memory cap may become a risk of VM performance degradation when the applications exhibit drastic fluctuation of memory requirements. Ballooning is effective only when the scope of memory resizing do not exceed the VM memory cap. Otherwise, the VM needs to reboot for memory re-configuration. In high-availability systems, the cost of a reboot cycle for the sole purpose of adding system RAM is simply too expensive. Although a large setting of memory cap can mitigate the limitation of ballooning, it can never eliminate the constraint of memory cap. Consider the following scenario: when a VM's memory requirement has exceeded the maximum memory capacity of its host machine, the VM should be migrated to another host with larger memory capacity. However, as the VM's memory cap cannot exceed the memory capacity of its original physical host, ballooning cannot add more memory to the VM even if there are large amounts of spare memory in the new host. That means ballooning is constrained to the memory cap that originates from the VM's creation and persists in the VM's whole lifecycle. Another non-trivial shortcoming of ballooning is that it tends to reclaim memory that

- *H. Liu, H. Jin, X. Liao, and W. Deng are with Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: haikunliu@gmail.com, {hjin, xfliao}@hust.edu.cn, qqweideng@gmail.com.*
- *B. He is with the Department of Computer Science and Engineering, Nanyang Technological University, 635798 Singapore. E-mail: he.bingsheng@gmail.com.*
- *C. Xu is with the Department of Electrical and Computer Engineering, Wayne State University, 5050 Anthony Wayne Drive, Detroit, MI 48202. E-mail: czxu@wayne.edu.*

Manuscript received 24 Jan. 2014; revised 14 Apr. 2014; accepted 23 Apr. 2014. Date of publication 28 Apr. 2014; date of current version 8 Apr. 2015.
Recommended for acceptance by H. Frey.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2320915

is free in a guest OS and thus fragments the memory map of the guest OS. Moreover, as it is hard to estimate the size of applications' working set, ballooning may cause VM performance degradation and even crash the guest OS if it tries to steal too much memory from the guest.

Due to the limitations of ballooning, memory hotplug was proposed to expand VMs' memory capacity on the fly [3], [4]. Memory hotplug was originally developed to add or replace RAM for a physical machine. In virtualization environments, hotplugging can dynamically expand a VM's physical address space beyond the memory cap specified at boot, and thus can arbitrarily increase a VM's memory allocation without rebooting the VM. Furthermore, memory space added or removed by hotplugging is contiguous and large so that they do not cause external memory fragmentation. Due to these advantages, hotplugging is complementary to ballooning. However, the existing work [29] only discussed hotplugging with ballooning qualitatively and the discussion are at high level and abstract. There has been little attention paid to a quantitative and comparative study on these two techniques. Without these understandings, the system administrators may make wrong decisions on the alternatives. In this paper, we conduct a comprehensive comparison of hotplug and ballooning in terms of implementation details, performance overhead, memory fragmentation level, performance speedup of applications.

As current memory hotplugging for VMs only support coarse-grained memory addition (section-level) [3], we first design and implement memory addition/removal in both section and page levels. To make the comparison of hotplug and ballooning more clearly, we then develop a dynamic memory management runtime named U-tube in virtualization environments. U-tube provides a framework that can freely adopt hotplug or ballooning for dynamic memory allocation. We demonstrate the effectiveness of U-tube for dynamic memory allocation in two case studies: dynamic memory balancing and memory overcommitment. We implement U-tube in Xen and compare memory hotplug and ballooning in various system aspects. We find that memory hotplug is more complicated to implement than ballooning, and usually causes higher performance overhead in section-level memory control than ballooning. However, memory hotplug shows better performance in page-level memory control and less memory fragmentation than ballooning. Moreover, unlike ballooning, memory hotplug is not constrained to VM's memory cap, and thus offer better performance improvement to applications than ballooning.

The major contributions of this paper are summarized as follows:

1) We make a quantitative and comparative study of memory hotplug and ballooning by comparing their implementation details, performance overheads, memory fragmentation levels, performance speedup of applications. This study can benefit system administrators to better understand their strengths and weaknesses.
2) We implement a runtime system called U-tube in Xen. We show that U-tube is able to significantly improve the performance of VMs that suffer from insufficient memory allocation by leveraging dynamic memory balancing strategies. Furthermore, exploitation of memory overcommitment improves the memory utilization and expands a physical server's capacity modestly without compromising applications performance.

Organization. The remainder of this paper is organized as follows. Section 2 briefly introduces ballooning and memory hotplug. Section 3 describes the design and implementation of U-tube and its two typical applications. Section 4 presents the evaluation methodologies and experimental results. Section 5 describes the related work. Finally, we conclude in Section 6.

2 Background

In this section, we briefly introduce ballooning and Linux memory hotplug.

2.1 Ballooning

Ballooning mechanism has been used to manage memory by many hypervisors such as Xen [7] and VMware [32]. Ballooning relies on a special driver that resides in each guest OS and cooperates with the hypervisor to adjust a VM's memory size dynamically. The basic function of ballooning is to pass memory back and forth between the hypervisor and a guest OS. When the balloon inflates, the driver applies for memory from the guest OS and gives it to hypervisor. When the balloon deflates, the driver retrieves the loaned memory from the hypervisor and returns it to guest OS. Thus the VMM creates an illusion that there is more memory resource than the actually available memory. This feature is also known as memory overcommitment [22].

2.2 Memory Hotplug

Memory hotplug was first studied in Linux kernel development community [15], [28], [29]. The motivation of this technique is to expand system RAM capacity on demand without causing system downtime. To support this feature, the kernel needs to use SPARSEMEM memory model, which is an abstract of discontinuous memory mapping. SPARSEMEM logically divides physical memory into chunks of the same size. The chunk is called a section and its size is architecture-dependent. For example, x86-64 uses 128 MB while PowerPC uses 16 MB. The memory unit of adding/removing operation is one section.

The operation of memory hotplug can be divided into physical and logical phases. Physical memory hotplug is responsible for communicating the hardware/firmware and preparing the environment for hotplugged physical memory. The firmware such as ACPI supports notification of connecting new memory to OS. Logical memory hotplug phase is responsible for changing memory state into available or unavailable for users. The amount of memory from a user's view is changed in this phase, which is also called online/offline operations.

We call adding/removing memory sections hot-add/hot-remove for short. Memory hot-add works when a physical DIMM is plugged. The firmware ACPI notifies the OS that a new range of memory address is available. The OS kernel initializes all memory in the DIMM as free pages and

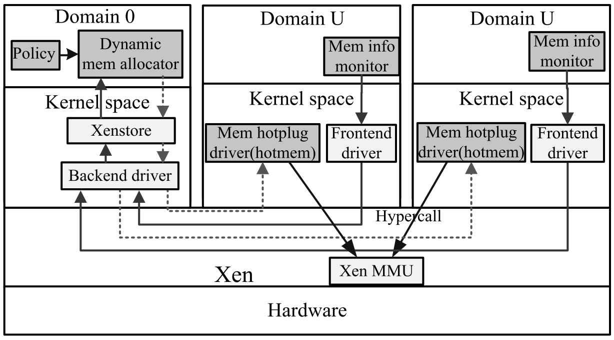

Fig. 1. System architecture of U-tube.

extends the *mem_map* data structure with one page table entry for each physical page. Finally the kernel adds the new memory into the allocator and makes it available for users. Memory hot-remove makes memory sections unavailable for users. Page migration [9] technique should be leveraged to move the used pages in the specified section to other sections, and then pages in the target section can be removed from the allocator. Page migration causes a small performance penalty to memory hot-remove. However, the sections removed are large and contiguous so they don't cause external memory fragmentation.

To support memory hotplug in Xen platform, Daniel Kiper provided a patch [3] which had been incorporated into Linux kernel. However, Daniel's approach only supports memory addition in coarse-grained sections, but cannot support memory removal. He also mentions that memory hot removal is quite complicated and cumbersome to implement. In the existing approach, the function of memory removal is achieved by ballooning. However, this approach may cause significant performance penalty, especially for reclaiming a large amount of memory in a heavily fragmented VM, as shown in our experiments. This paper improves the current hotplug implementation in various aspects. The major ones include 1) a real implementation of memory removal based on page migration; 2) fine-grained (page-level) memory addition/removal to support on-demand and lightweight memory allocation.

3 System and Implementation

For a quantitative and comparative study between ballooning and hotplug, we need a common platform to allow the integration of both techniques. That motivates us to develop U-tube, a dynamic memory management runtime on top of Xen. Since U-tube is developed on top of Xen, ballooning is naturally supported. We particularly focus on the details of memory hotplug implementation and dynamic memory allocation algorithms.

3.1 System Overview

Fig. 1 shows the system architecture of our U-tube prototype. It is composed of three main components: memory information monitor and memory hotplug driver that are deployed in each guest VM, and a dynamic memory allocator residing in the privilege domain (domain 0) for global decision making. The memory hotplug module is implemented as a loadable device driver in each guest OS. It coordinates the memory management unit (MMU) of guest OSes with underlying hypervisor to dynamically adjust VMs' memory allocations. U-tube provides a framework that can freely adopt hotplug or ballooning for dynamic memory allocation.

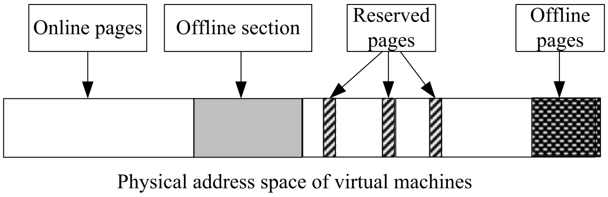

Fig. 2. A VM's memory footprint with hotplugged pages.

In each guest VM (domain U), there is a pseudo-device driver called *hotmem* deployed in kernel space. *Hotmem* driver coordinates with the hypervisor to allocate or reclaim physical memory to/from each guest OS. *Hotmem* driver extends or shrinks a VM's physical address space in coarse-grained sections through guest OS's memory hotplug interface, and then allocates/reclaims memory in fine-grained pages to/from the guest OS by invoking hypercalls exposed by hypervisor's memory management routines. We note that the *hotmem* driver can be replaced by a balloon driver if ballooning mechanism is used in U-tube framework. Each guest OS periodically collects the memory information statistics and sends it to domain 0 through *Xenbus*, which provides an interface for pseudo-devices including front-end and back-end drivers to communicate between domains.

In domain 0, the memory statistics are stored in a directory-like structure *Xenstore* and referenced by their domain ID. A global memory allocator periodically retrieves the memory information of each VM from the *Xenstore*, and resizes each VM's memory allocation based on two policies: dynamic memory balancing and memory overcommitment. If the global memory allocator finds some VMs' memory is under-allocated (as described in Section 3.3), U-tube uses the dynamic memory balancing algorithm to re-allocate memory for each VMs and balances their memory pressure (as described in Section 3.4.1). If the global memory utilization of the physical host exceeds a threshold (90 percent in our paper), then some VMs need to move to other hosts to release the memory pressure; otherwise, we continuously place more VMs on the host by using memory overcommit mechanism (as described in Section 3.4.2) until the whole memory utilization approaches to the threshold. In this way, U-tube achieves high memory utilization of host machines while does not compromise the applications performance.

3.2 Memory Hotplug

A memory page has several possible states in U-tube, as shown in Fig. 2. The online pages are usable memory and the others are unavailable for guest OSes. Offline pages are generated when *hotmem* driver dynamically adds memory sections to a VM but only a part of them are set as online, and others are set as offline. For example, if a VM needs to add 200 MB RAM and one section is 128 MB, then two sections should be added but 56 MB RAM is set as offline pages. Offline section and reserved pages are generated by removing memory from a VM in sections and pages respectively. In order to bring some unavailable memory back online, the priority from high to low is reserved pages, offline pages and offline section. Reserved pages are the first to be online because they fragment the available memory space. Offline pages become online from low memory address to high in the second order. At last, an offline section will be online only when the required memory space exceeds a whole section.

The following describes the details of memory hotplug implementation. U-tube uses an inter-domain communication mechanism to transfer the updates of a guest OS's memory information. This mechanism relies on the Xenbus/Xenstore functionalities. Xenbus provides a bus abstraction for virtual device drivers to communicate between domains. Xenstore is a filesystem-like database that is accessible for all domains. In general, management tools configure and control pseudo-devices by writing values into keys in Xenstore that trigger events in drivers. As to *hotmem* driver, the target memory size of the guest OS is stored in a key *memory/target* and *hotmem* driver registers a Xenbus watch *hotmem_watch* on it. When the value of the key changes, the watch immediately responds by executing the function *hotmem_set_new_target()* to adapt to the requested size. At this time, a worker thread *hotmem_process* () is created, and we need to check whether the new target is a reasonable value. If so, we set the target value in a global structure (protected by a mutex), and signal our *hotmem* worker to perform add/remove memory operations.

3.2.1 Add Memory

When we need to extend a VM's memory capacity, *hotmem* driver hot-adds new memory to the VM through the memory hotplug interface provided by guest kernel. It will allocate new page structures for the added memory. *Hotmem* driver gets memory from the hypervisor and enables the pages online incrementally. The following describes two phases of memory addition for VMs:

1) *Expand memory space in coarse-grained sections*. If the target size exceeds current memory capacity of the VM, the memory address space is not enough for extension. *Hotmem* driver extends memory address in sections using the interfaces provided by guest OS kernel. First, *hotmem* driver requests new memory address resource from guest kernel. The requests must be in sections. Second, memory mapping is initialized for the new address space and the necessary date structures such as *mem_map* are set up. Finally, mapping tables from physical frames to machine frames (p2m) for the new addresses are constructed. At this time, the new memory address space is ready but needs to wait for real memory allocation from the hypervisor.
2) *Allocate memory in fine-grained pages*. The former only support memory space expansion. At this time, the memory address space is sufficient but the physical frames aren't available. *Hotmem* driver invokes a hypercall (an interface provided by VMM for guest OSes) to claim real memory from the hypervisor and map the memory to the new addresses. If the hypervisor does not have enough memory, it should shrink some VMs to squeeze the requested capacity. After that, these memory addresses are added to the guest OS's memory allocator so as to make them available for applications. The memory is online only when the memory can be used by guest memory allocator, otherwise they are offline. In some cases, to increase a VM's memory capacity, *hotmem* driver only needs to set some reserved pages or offline pages to online state if these pages are sufficient to satisfy the requirement. The priority of changing the state of offline memory is first reserved pages and then offline pages, as described above.

As memory hot-add in sections needs to prepare running environments for the new added memory, it would suffer more performance penalty than ballooning mechanism, which does not support capacity expansion beyond the memory cap. However, the memory added in sections is large and contiguous so they don't cause external fragmentation. Daniel's patch [3] only supports memory add in sections, the granularity is coarse in the sense that the memory added may exceed the requirement. In contrast, our implementation supports fine-grained memory extension that satisfies on-demand memory allocation. In practice, memory hot-add in sections is only needed when the memory requirement exceeds the VM's capacity and the available offline memory cannot satisfy it, otherwise, we only need to online the offline memory.

3.2.2 Reclaim Memory

When we need to reclaim memory from a VM for other uses, *hotmem* driver shrinks the VM's capacity and return memory to the hypervisor. The following describes two granularities of memory removal for VMs.

1) *Reclaim memory in coarse-grained sections*. If the memory to be removed is larger than the section size, *hotmem* driver needs to remove multiple memory sections. First, it traverses memory sections to find the removable sections. If a section contains some used pages that cannot be migrated (e.g., pages for kernel code and reserved pages), the memory section cannot be removed. Second, if a section is removable, all the pages in this section should be freed by migrating all allocated pages to other sections and removing free pages from the allocator. Once a memory section is completely empty, all references to it will be removed. At this time, the memory section can be safely removed because it is no longer referenced by the kernel MMU. At last, the *hotmem* driver returns the reclaimed memory to hypervisor.

   Note that not all pages are migratable. Migratable pages are anonymous pages and page caches in current Linux. If any of non-migratable memory is located in the target section for removal, the whole section cannot be removed. This poses a significant challenge for memory removal. Luckily, Linux memory allocator uses a flag called *_GFP_MOVABLE* to indicate whether a page is migratable or not at allocation time. Correspondingly, Linux MMU also provides a memory zone called ZONE_MOVABLE to partition memory between migratable and non-migratable pages. ZONE_MOVABLE is only usable by allocations that specify both *_GFP_HIGHMEM* and *_GFP_MOVABLE* flags. This keeps all non-migratable pages within a single memory partition while allowing migratable allocations to be satisfied by either partition. As all pages within ZONE_MOVABLE can be released by migrating or reclaiming, we always find a removable section from ZONE_MOVABLE and migrate pages within this zone. This avoids non-migratable pages and facilitates the page migration.

2) *Reclaim memory in fine-grained pages*. One approach to reclaiming memory in pages is similar to the ballooning driver. We can reclaim memory from the guest OS kernel in pages and sets these pages as reserved pages, which is no longer available to the guest OS. After that, it returns the memory to hypervisor by invoking a hypercall. However, reserved pages may fragment the VM's memory address space. Another approach that can avoid memory fragmentation is to reclaim offline pages from the available highest address to low address in a descending order. These offline pages can be returned to the hypervisor. This method causes less memory fragments than quasi-ballooning mechanism, which is used only when there are no offline pages available in the VM.

Note that memory hotplug in Linux does not support a real page-granularity memory removing because hotplugable physical memory scales in sections. However, fine-grained page removing can be achieved in virtualization environments because the memory resource is virtualized and can be partitioned to arbitrary sizes. Reclaiming memory in page granularity allows memory management routines to flexibly change a VM's memory capacity when the amount of memory to be reclaimed is less than one section. We also note that current implementation of memory hotplug in Xen does not support hot-unplugging. Memory removal is achieved by ballooning due to its simplicity. However, ballooning would fragment VMs' memory space. Moreover, it may cause significant performance penalty in memory removal, especially for reclaiming a large amount of memory in a heavily fragmented VM (see more in Section 4.2).

3.3 Memory Under-Allocation Detection

In a VM running dynamic workload, its memory requirement is often changing all the time. How much memory does each VM really need? Accurate detection of memory under-allocation determines the benefit of memory re-allocation. Monitoring the memory usage is a well-used solution. However, modern OSes such as Linux are greedy to use up all its available free memory as page cache, which can probably speed up the access of data from disk. Thus the memory utilization of a VM cannot completely reflect its real memory requirement. High memory utilization doesn't imply that the VM needs more memory because the memory used for page cache can be reclaimed or reused. Similarly, low memory utilization cannot reflect how much memory can be reclaimed without potential performance loss. Another approach to memory under-allocation detection is to monitor the paging I/O rates or major page fault rates[1] [23]. However, this technology may not accurately reveal the relationship between page fault rate and the memory requirement. Thus when a VM's memory is under-allocated, the page fault rate cannot accurately predict how much additional memory is required. On the other side, when a VM's memory is over-allocated, it is unable to hint how much free memory can be reclaimed for other VMs. Previous work has demonstrated the effectiveness of using page protection techniques to track page accesses at finer granularity. However, this approach without assistance of dedicated hardware usually results in an unacceptably high overhead [5], [39].

We combine memory utilization and paging I/O rates monitoring together to detect memory under-allocated VMs. We obtain memory usage information of each VM from guest OS kernel. For Linux OS, the *proc* file system provides detailed memory statistics that can be used for further analysis. The memory utilization can be inferred from */proc/meminfo* and the number of major page faults can be obtained from */proc/vmstat*. Memory utilization of a VM is calculated by the ratio of memory usage to total memory allocation. Note that the guest memory usage is defined as the total amount of memory actively used by guest OS and applications, including buffer or page cache memory that is in active use [8]. Previous study had demonstrated that applications performance may significantly degrade when memory utilization exceeds 90 percent [17]. Guided by this observation, we can experientially deem that the reasonable upper bound of memory utilization for good application performance is 90 percent. If such case occurs and the observed number of major page faults is continuously increasing, we should immediately increase the VM's memory allocation. However, the amount of memory requirement should be estimated by the changes of VM's working set size. We will describe the detailed estimation in the following section.

3.4 Memory Allocation Algorithms

In the following, we introduce two key applications of hotpluging or ballooning for dynamic memory management.

3.4.1 Dynamic Memory Balancing

Dynamic memory allocation among VMs is essential to improve memory management in a virtualized environment. In order to satisfy the memory requirement on-demand, we design dynamic memory balancing algorithm and implement it as a daemon process in domain 0. The algorithm uses a global coordinator to automatically balance memory load among all VMs.

We install a memory monitor in each VM. In each time window, the memory demand is written in Xenstore periodically. With the help of *Xenbus* callback function, it will immediately trigger a memory re-balancing by the global coordinator. Unlike the post-adjustment mechanisms

[1] A major page fault occurs when a page is not loaded in memory at the time the fault is generated. It adds disk latency to the interrupted program's execution thus is more expensive than a minor page fault.

such as feedback control [17], dynamic memory balancing can pre-allocate memory to a VM that has potential memory requirements before its memory becomes under-allocated, and thus can avoid applications performance degradation and improve memory utilization of all VMs on a physical host.

In U-tube, domain 0 is usually configured with a fixed amount of memory because this privilege domain should reserve sufficient memory to handle all VMs' I/O operations. The memory requirement of each VM can be partially determined by the changes of memory usage, based on the assumption that if the used memory increases in current time window, the VM is more likely to use more memory in the next time window. We denote $M_i^U(t)$ the memory size the VM i actually used at the time t, and $\Delta M_i^U(t)$ the change of used memory measured in the time window t, and $\Delta S_i(t)$ the change of swap space in the time window t. The global memory allocator can calculate the change of memory usage of each guest VM i by the following Equations:

$$\Delta M_i^U(t) = M_i^U(t) - M_i^U(t-1), \tag{1}$$

$$\Delta S_i(t) = S_i(t) - S_i(t-1). \tag{2}$$

We consider the increment of swap as memory requirement because swap is used for paging only when usable memory is very scarce. We predict the memory requirement of VM i in the next time window:

$$M_i^P(t+1) = M_i^U(t) + \lambda\big(\Delta M_i^U(t) + \Delta S_i(t)\big), \tag{3}$$

where λ represents the coefficient of memory increment. If $\Delta M_i^U(t) + \Delta S_i(t) > 0$, the value of λshould be set larger than 1. A relatively large λ can reduce the frequency of memory allocation but may lead to a waste of memory, especially when the memory utilization of a whole host is very high. We thus determine the value of λ by equation $\lambda = 1.2/U(t)$, where $U(t)$ denotes the global memory utilization at time t. If $\Delta M_i^U(t) + \Delta S_i(t) < 0$, the VM would keep the reclaimable memory until the memory balancing algorithm is triggered by some under-allocated VMs. This avoids unnecessary memory re-allocation among all VMs.

When the memory resource of a VM is under-allocated, U-tube resizes its memory and balances the memory pressure among all the VMs based on proportional memory allocation. Summing up each VM's memory requirement, the expected global memory utilization in a physical host becomes:

$$\overline{U}(t+1) = \sum_{i=1}^{n} M_i^P(t+1) \Big/ \sum_{i=1}^{n} M_i(t), \tag{4}$$

where $M_i(t)$ represents the total memory a VM owns, including the used portion and free portion at time t. Eq. (3) only considers the memory requirement locally. While considering the other VMs' demands, a fair memory allocation scheme that balances all VM's memory pressure can be represented as follows:

$$M_i^T(t+1) = M_i^P(t+1)/\overline{U}(t+1), \tag{5}$$

where $M_i^T(t+1)$ represents the target memory size that the allocator should assign to the VM *i*. The target memory size should be never below the value of a kernel parameter *Committed_AS*, an estimate of memory size guarantees that "out of memory" exception never occurs for the workloads.

Though memory adjustment can be done with marginal overhead, as shown in the performance evaluation section, we should avoid unnecessary memory allocation that doesn't benefit to a VM's execution. We set a threshold δ to determine whether a memory adjustment should be performed:

$$|M_i^T(t+1)/M_i(t) - M_i^U(t)/M_i(t)| \geq \delta. \tag{6}$$

That means memory adjustment occurs only when the expected increment of memory utilization is larger than the threshold δ, which is empirically set to 3 percent in U-tube.

3.4.2 Memory Overcommit

In a virtualized environment, server consolidation aims at maximizing resources utilization by placing several VMs on the same host. It is widely used in today's data centers to improve memory utilization, and to save total cost of ownership. U-tube provides a mechanism to maximize servers' capacity through memory overcommit. When the hypervisor does not have enough free memory to create a new VM, U-tube can reasonably shrink some VMs to squeeze the required capacity. Note that if the observed memory utilization of a VM exceeds 90 percent, U-tube would never take a risk to reclaim memory from this VM because of the potential performance degradation. Let $M^F(t)$ be the available free memory the hypervisor can provide, let M^R be the memory requirement of the new VM. Under the assumption that the new VM can be successfully created, we calculate the average memory utilization of existing resizable VMs by:

$$\overline{U}(t+1) = \left(M^R + \sum_{i=1}^{n} M_i^U(t)) \Big/ (M^F(t) + \sum_{i=1}^{n} M_i(t)\right). \tag{7}$$

If $\overline{U}(t+1)$ does not exceed 90 percent, then the new VM can be successfully created. At this time, U-tube adjusts the memory allocation of each VM that can be squeezed using balanced allocation as described in Eq. (5).

3.5 Implementation

The implementation of U-tube is based on Xen 4.2, with a guest Linux kernel 3.2. There are approximate 2,300 lines of code in U-tube implementation. The implementation of memory hotplug driver mainly includes two parts: 950 lines for a loadable pseudo-device driver in guest OS's kernel, which includes functions implementation of memory allocation/reclaim in sections and pages. The other 250 lines contribute to the modification of the memory management routines of Xen hypervisor. This part includes the hypercalls of initializing new memory resource to a VM. The implementation of memory allocation policies and memory allocator is about 1,100 lines within a daemon process running in domain 0. 600 lines contribute to the implementation of memory balancing

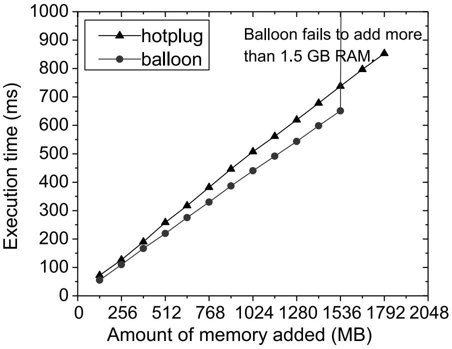

Fig. 3. The mean execution time of memory addition is linear with the amount of memory added by both hotplug and balloon driver.

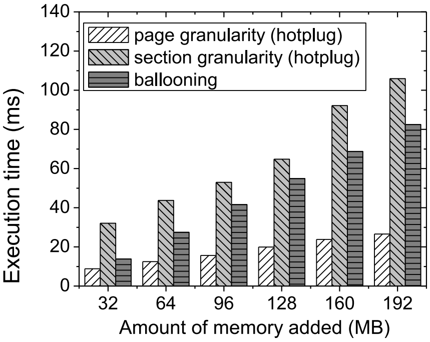

Fig. 4. The mean execution time of memory addition varies with the amount of memory added by section- and page-granularity hotplug.

algorithm and memory overcommitment algorithm. The remaining 500 lines are used for memory information monitoring and inter-domain communication.

4 Evaluations

In this section, we present the evaluation of U-tube with several benchmarks. We begin by introducing our experimental setup, and then compare memory hotplug with ballooning in terms of performance overhead, memory fragmentation level, performance speedup of applications.

4.1 Experimental Setup

We conducted all experiments on Dell PowerEdge1950 servers with two Intel quad-core Xeon E5450 3 GHz processors, 12 GB RAM, one 250 GB SATA hard disk, and 1 Gbit Ethernet interface. The host machine ran RHEL 5 distribution (x86-64) and the hypervisor was Xen 4.2 with Linux kernel 3.2. The privilege domain was configured to use one dedicated core. The guest OSes also ran RHEL 5 with Linux kernel 3.2. We evaluate U-tube using the following benchmarks:

TPC-C. It is an on-line transaction processing (OLTP) benchmark [1]. We used a public benchmark DBT-2 to simulate a complete computing environment where a number of users execute transactions against a database. The simulator and MYSQL database server all ran in a single VM. We configured 1,000 terminal threads and 500 database connections and ran the benchmark for 40 minutes. The workload shows moderate memory demand.

*SPEC CINT2006***.** It consists of a suite of applications for measuring compute-intensive integer performance [2]. The memory load varies dynamically when workload changes from one application to the other.

*DaCapo***.** It is a Java benchmark suite, which includes a set of real world applications with non-trivial memory loads [6]. We used JDK 1.6 as the applications execution environment.

4.2 Performance Overhead

To compare the performance overhead of memory hotplug with ballooning mechanism, we conducted experiments to measure memory addition/removal performances for both approaches.

At first, we conducted an experiment to measure the performance of coarse-grained memory addition in section. The VM to be resized was an idle Linux without any applications running on it. This assures there is no CPU resource contention that interferes with the *hotmem* driver and *balloon* driver. The VM was initialized with 512 MB RAM and its *max_memory* capacity is set to 2 GB. We increased the VM's memory in a 128 MB step-size and the hypervisor always have sufficient free memory to satisfy the requirement of hotplug and balloon drivers. Fig. 3 shows that the time cost of memory addition operations is linearly proportional to the amount of memory for both approaches. The *hotmem* driver costs 60 ms per section while ballooning costs 55 ms per 128 MB. Hotplug costs 9.1 percent more time than ballooning because it needs to extend the VM's memory address space first and then to initialize this portion of memory. Ballooning avoids such actions because they are done during the VM booting. However, the amount of memory added by ballooning cannot exceed the VM's memory cap. When we tried to expand the VM capacity up to 2 GB (1.5 GB memory is added to the VM), the *balloon* driver caused a non-response failure while *hotmem* driver worked well. This experiment demonstrates that memory hotplug shows higher applicability than ballooning.

To compare with the coarse-grained memory addition, we also evaluated the performance of fine-grained memory addition in pages. Assume there is sufficient offline memory for fine-grained control. As shown in Fig. 4, page-granularity hotplugging shows the least execution time as it only needs to change the state of offline memory. For ballooning, it needs to apply for memory from the underling hypervisor, reconstruct the P2M table, and thus pose much higher performance overhead. For comparison, we also measure the section-granularity hotplugging. We deliberately added memory sections to a VM and then only set a portion of them as online. For example, when we intended to increase 32 and 160 MB RAM to the VM, we should add one and two sections, respectively. However, due to the highest performance overhead, it is only used when the VM does not have enough address space for memory extension.

Second, we measured the performance of memory removal in two VMs. One was an idle Linux whose memory image consists of a large quantity of free memory. The other one was running OLTP application TPC-C for a long time, and thus its memory were mostly used as page cache and seriously fragmented. Both VMs were initially booted with 2 GB RAM. We decreased its memory in a 128 MB step-size. Fig. 5 shows the experimental results. When we remove

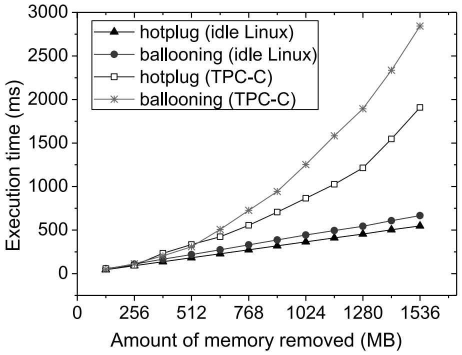

Fig. 5. The mean execution time of memory removal varies with the amount of memory removed by hotplug and balloon driver in both light-loaded and heavily-loaded VMs.

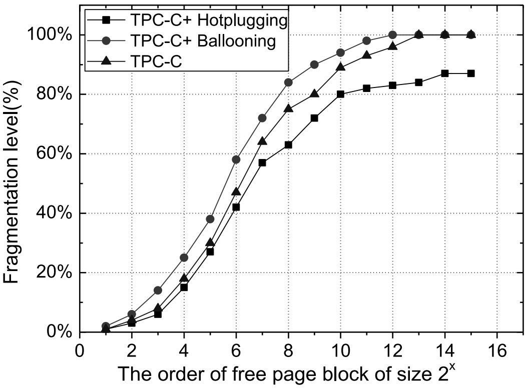

Fig. 7. The memory fragmentation level affected by hotplugging and ballooning.

memory from the idle VM, hotplug driver costs 46 ms per section while ballooning costs 56 ms per 128 MB. Hotplugging shows 21.7 percent better performance than ballooning. In contrast, when hotplug and balloon driver remove memory from the heavily-loaded VM running TPC-C benchmark, the time cost of memory removal significantly increases compared to that in the idle Linux. The performance overhead of hotplug mainly comes from page migration and paging a portion of memory to disk when the VM has not enough free memory for removal. The overhead of ballooning depends on the guest OS's memory allocation algorithm and the layout of memory footprint. When a large amount of memory is removed from the VM, to squeeze a large and continuous region of memory, ballooning would trigger the guest kernel MMU to reclaim memory and swap pages to disk. Because hotplugging performs page migration in memory, and thus it is much faster than ballooning which should swap pages to disk. Memory hotplug reduces 30.4 percent response time compared to ballooning, especially when they remove a large region of memory, as shown in Fig. 5.

To compare with the coarse-grained memory removal, we also evaluated the performance of page-granularity memory removal in an idle VM. Assume there is sufficient offline memory for fine-grained control. As shown in Fig. 6, page-granularity hotplugging is much faster than ballooning as it only needs to return the offline memory to the hypervisor. However, if we deliberately removed memory in sections, oblivious to the available offline memory, the overhead of section-granularity hotplugging is usually higher than ballooning. The reason is that the overhead of hotplugging is linear to the number of sections removed, while the overhead of ballooning is linear to the amount of memory removed.

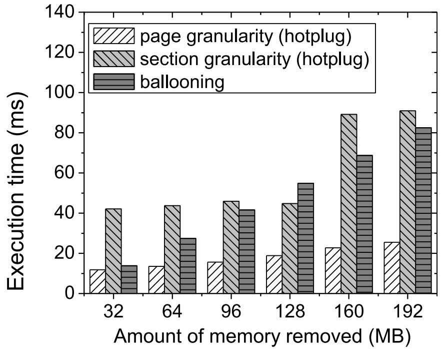

Fig. 6. The mean execution time of memory removal varies with the amount of memory removed by section- and page-granularity hotplug in a light-loaded VM.

4.3 Memory Fragmentation

We also conducted an experiment to evaluate the impact of ballooning and hotplugging on memory fragmentation, which affect the size of large object that can be allocated. We measured the *memory fragmentation level* of a VM with 1 GB RAM running TPC-C application for three times. During the second and third runs, we conducted 50 times of memory allocation/reclaim using ballooning and hotplugging mechanism, respectively. In each operation, we first added 128 MB RAM to the VM and then reclaimed it one minute later. In each test, we measured the memory fragmentation level, which is defined in the following formula [13]:

$$Fraglevel = \frac{TotalFreePages - \sum_{i=j}^{n} 2^i * k_i}{TotalFreePages},$$

where 2^n is the largest free page block that can be allocated, i is the order of pages, j is the order of desired allocation and k_i is the number of free page blocks of size 2^i.

We calculate the memory fragmentation level referring to different sizes of free memory block. The free page information is collected from /proc/buddyinfo. The experimental results demonstrate that the fragmentation level affected by hotplugging is less than ballooning, especially when we refer to large free page blocks, as shown in Fig. 7. As the ballooning tends to remove memory that is already free thus usually fragments the pseudo-physical memory map of the VM, causing the fragmentation level higher than the case of non-intrusive execution of TPC-C. In contrast, hotplugging is able to allocate/reclaim contiguous regions of memory and thus it avoids fragmentation of the memory map. Moreover, removing memory with hotplugging often leads to pages migration that is able to eliminate memory fragmentation.

4.4 Improvement of Application Performance and VM Density

We ran benchmarks TPC-C, SPEC CINT2006, DaCapo and a suite of mixed workloads to evaluate the performance

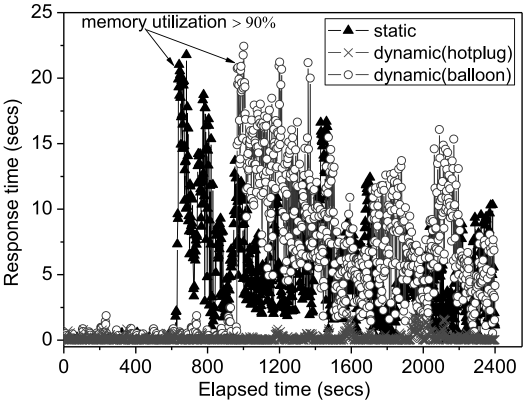

Fig. 8. The mean response time of TPC-C transactions varies with memory utilization using static and hotplug-based, balloon-based dynamic memory allocation.

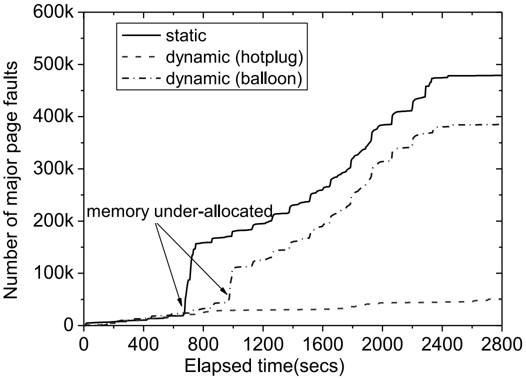

Fig. 9. The number of major page faults generated when TPC-C is running with static and hotplug-based, balloon-based dynamic memory allocation.

improvement due to dynamic memory allocation, and VM density improvement due to memory overcommitment. We observed that most of applications in our experiments show relatively small working set. To simply simulate memory overload scenarios, if not expressly stated in the following, each VM we used to run the applications was initialized with 512 MB RAM and one virtual CPU, and 50 percent memory capacity extension. That means the parameter *max_memory* is configured as 768 MB. We set a 5-second time window to collect the memory usage information of each VM and balance VMs' memory pressure by making a tradeoff between runtime overhead and the sensitivity of memory adjustment.

We conducted each experiment on four hosts with 12 GB RAM. We allocated 0.5 GB RAM to domain 0 thus each host has 11.5 GB spare memory to accommodate VMs. We created 23 VMs with initial 0.5 GB RAM on the target host that we evaluated, and some VMs were created on other two host for increasing the target host's load if its memory was under-allocated. Another resourceful host was provisioned for receiving VMs migrated from the target host if its memory was over-allocated. When the VMs were ready, we started the workload in all VMs concurrently, and monitored the target host's memory load to determine VM migrations. To simulate the scenario of memory overcommitment, we continuously placed VMs to the target host one by one until it had no space to accommodate more VMs. That reveals the memory utilization of all VMs on the target host reach to 90 percent and the hypervisor does not have spare memory. Meanwhile, if the total memory utilization of target host exceeds 90 percent as the load increases, U-tube moved some VMs to the reserved host to guarantee the application performance. We measured the application performance improvement and VM density using both hotplug-based and balloon-based dynamic memory allocation approaches using the same policy (as described in Sections 3.4 and 3.5).

Fig. 8 shows that the response time of one type of TPC-C transactions (*new order*) varies at different phases of execution using static, hotplug-based and ballooning-based dynamic memory allocations. Fig. 9 shows the corresponding major page faults, which is an important performance metric to evaluate how well the memory requirement is satisfied. When the benchmark began, the VM exhibited almost linear increase of memory requirement for the upcoming transactions. We found that the response time increases significantly when the memory utilization went beyond 90 percent. Once the VM experiences significant memory pressure, it needs to reclaim memory for new transactions by paging some used memory to disk. This results in significant increase of major page fault, as shown in Fig. 9. Compared to the static memory allocation, Hotplug and balloon reduce the major page faults by 89 and 20 percent, respectively. For static memory allocation, the response time of TPC-C transactions and the VM's major page faults significantly increased at 642 seconds when the 512 MB memory is used up. For ballooning-based dynamic memory allocation, the bursty increase of response time and major page fault occurred at 986 seconds when the memory requirement exceeded the VM memory cap. In contrast, in U-tube, memory under-allocation never occur because hotplug is not constrained to the VM's memory cap. The VMs' memory utilization is relatively high but is below the threshold for memory extension. Otherwise, some VMs would be migrated to other host to release the memory pressure.

Table 1 shows the statistics of TPC-C performance and VM density in a fully-utilized host using different memory allocation strategies. Compared to the balloon-based dynamic allocation, hotplug is able to reduce the total mean response time by 92 percent and increase the throughput by 93 percent. Especially, both hotplug-based and balloon-based memory overcommitment can expand the host capacity by 35 percent when the VMs' memory is over-allocated at the beginning. Then the VM density is continuously decreasing with the increase of memory required by the VM workloads, and finally the minimum VM density of

TABLE 1
TPC-C Performance and VM Density

Mem alloc policy	Mean Resp. Time(secs)		Throughput (trans/min)	VM density (min, max)
	Total	Mem Util ≥ 90%		
Static	7.36	9.42	176.4	(23, 23)
Hotplug	0.43	—	434.3	(13, 31)
Balloon	5.28	9.68	224.6	(16, 31)

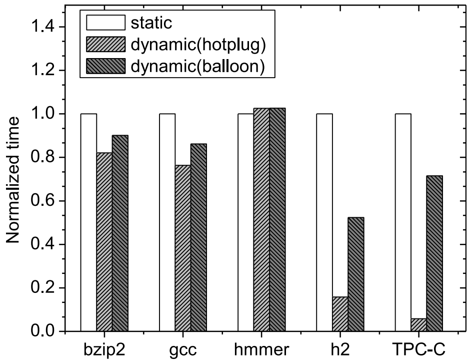

Fig. 10. Application performance improved by hotplug-based and balloon-based dynamic memory allocation.

hotplug is less than balloon because much more memory is allocated to each VM. This reveals that hotplug shows more applicability in memory extension than balloon.

For SPEC CINT2006 and DaCapo, we also observed that memory hotplug always perform better than ballooning and static memory allocation policies. Here, we only show the normalized performance and major page faults of typical applications in Fig. 10 and 11, respectively. For the selected applications, *bzip2, gcc* (both in SPEC CINT2006 suite) and *h2 (*DaCapo) are memory-intensive, and *hmmer (*SPEC CINT2006) shows small working set and is not sensitive to memory allocation. The performance of *bzip2, gcc, h2,* and *hmmer* were measured in execution time, and *TPC-C* was measured in mean response time. For memory-intensive applications such as *bzip2*, hotplugging and ballooning reduce the application execution time by 18 and 10 percent compared to static memory allocation, respectively. Correspondingly, hotplug and ballooning can significantly reduce the number of major page faults by 84.1 and 65.5 percent, respectively. The other memory over-allocated applications such as *hmmer* show 1∼4 percent increase of execution time due to the cost of memory usage monitoring and re-allocation. For DaCapo, memory hotplug can reduce the execution time of *h2* program even by 85 percent. Correspondingly, the number of major page faults can be significantly reduced by up to 99 percent. However, for ballooning, the VM's memory cap causes a large number of major page faults during *h2*'s execution. As too much time was wasted for handling major page faults, the performance of *h2* significantly degraded by using static and balloon-based memory allocation.

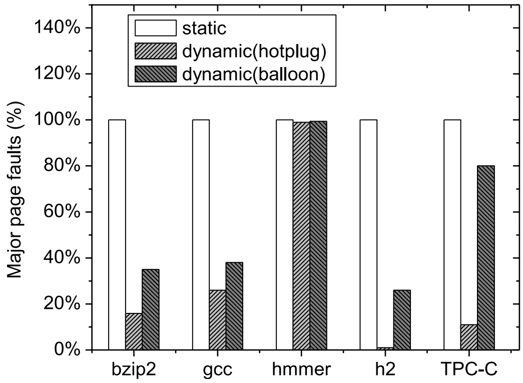

Fig. 11. The major page faults reduced by hotplug-based and balloon-based dynamic memory allocation.

TABLE 2
Summary of Comparison between Memory Hotplug and Ballooning

	Memory hotplug	Ballooning
Implementation	Complicated	Easy
Dependence	hotplug driver of guest OSes	MMU of guest OSes
Constraints	No	Memory cap
Performance overhead	Fine-grained control is lower	Coarse-grained control is lower
Memory fragmentation	Mitigate	Exacerbate
VM performance improvement	Best	Moderate

4.5 Summary of Comparison

Finally, we give a summary of the detailed comparison between hotplugging and ballooning, as shown in Table 2. A definite advantage of ballooning is that it can directly use the native MMU of the guest OS, and thus facilitate the ballooning implementation. However, memory expansion cannot go beyond a guest OS's memory cap configured at booting time. The memory cap poses a significant limitation in VMs capacity extension. Furthermore, the ballooning relies on the buddy system of guest MMU, and thus fragments the pseudo-physical memory map of the guest OS when it inflates. In contrast, memory hotplug can add/remove whole sections at a time, avoiding memory fragmentation of VMs. The significant advantage of hotplug is that it is able to expand a VM's capacity beyond its memory cap in the fly. The better scalability of hotplugging always means better application performance. However, the overhead of coarse-grained section control is usually higher than ballooning.

4.6 Fairness of Dynamic Memory Balancing

We compare the policy of dynamic memory balancing (Section 3.4) with VMware memory management. VMware EXS server uses proportional-share algorithm combining with *idle memory tax* mechanism to allocate memory to each VM [32]. In the cloud environment, this is related to economic fairness [35], which is important for the monetary cost and performance of multi-tenant systems. To present the actions of memory dynamic allocation more clearly, we only co-locate two VMs on a single host. One runs DaCapo and another runs TPC-C. Both VMs are configured with 1 GB RAM. The other RAM are allocated to domain 0 and cannot be deprived by the two VMs. Such setting implies that the two VMs can share 2 GB RAM at most. In our experiment, we set the tax rate as 75 percent and start reclaiming memory when the percentage of free memory drops below 10 percent.

Fig. 12 shows the memory utilization of two VMs using two different policies. For U-tube, as shown in Fig. 12a, the two VM show almost the same memory utilization all the time. The policy is similar to the principle of U-tube in which liquid can always balance in two columns automatically. For the policy of VMware, the two VM shows

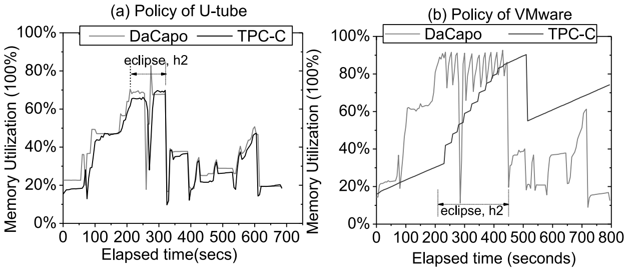

Fig. 12. Memory utilization of two co-located VMs running DaCapo and TPC-C. Sub-figure (a) and (b) use memory management policies of U-tube and VMware, respectively.

significant difference of memory utilization. When memory utilization of the VM running DaCapo reaches 90 percent, idle memory is reclaimed from the VM running TPC-C and reallocated to the under-allocated one. This operation is performed many times during the execution of *eclipse* and *h2*, and thus significantly increase the application execution time. Unlike the post-adjustment mechanisms, U-tube avoid the memory under-allocations and improve the application performance. Moreover, U-tube is also immune to the prediction error of memory requirements.

We evaluate the above two polices in terms of fairness, which is defined as the Min-Max Ratio (MMR) of memory utilization between the two applications, namely U^{min}/U^{max}. As shown in Fig. 13, the gray line shows the fairness of memory allocation using policy of VMware. The MMR significantly fluctuates with the variation of memory load. In contrast, the results of U-tube are more stable and all approximates to one. There are only numbers of pulses when the working set of applications significantly changes. In summary, the memory allocation policy of VMware achieves 61 percent fairness on average, while U-tube can achieve even up to 90 percent fairness on average.

5 Related Work

There are many works focusing on memory dynamic management in virtualization environments, such as page sharing, memory mapped I/O [21], [40], ballooning [7], [14], [32], hotplug [27]. One of the most important techniques is ballooning. It is widely used by many hypervisors for dynamic memory management [7], [32].

There are several works on dynamic memory allocation using ballooning mechanism [31], [38], [39]. Memory Balancer (MEB) [39] is a system designed for dynamic memory management based on ballooning. MEB monitors the memory usage of each virtual machine, predicts its memory need using Least Recently Used (LRU) histogram, and periodically adjusts a VM's memory using ballooning mechanism. MEB tracks normal memory accesses by revoking user access permission and trapping them into VMM as page faults, and constructs a LRU histogram to predict memory need of each VM. Although this approach provides a reasonably accurate prediction, it causes considerable performance penalty due to the costly memory access tracking. Xiao, et al. proposed a more efficient approach to memory demand prediction by leveraging an exponentially weighted moving average (EWMA) scheme [38]. Bayllocator [31] employed Bayesian networks to predict memory requirements, and provided proactive dynamic memory allocation based on ballooning. These works primarily focused on memory demand prediction. They are orthogonal to the theme of this paper, but can be complementary to our work.

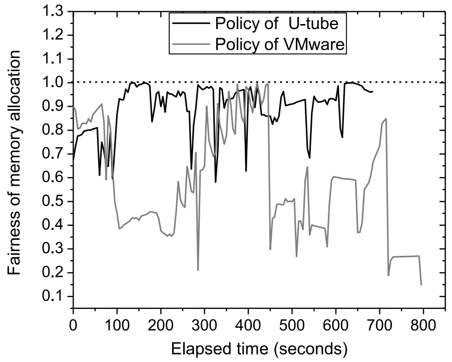

Fig. 13. The fairness of memory allocation achieved by memory management policies of U-tube and VMware.

Another important concept of ballooning is memory overcommitment. It can significantly increase the number of VMs that can be hosted in a single physical server. Ginkgo [11] is a memory overcommitment framework that leverages ballooning mechanism to dynamically adjust memory capacity of each VM in cloud computing environment. A feedback control method for dynamic memory allocation was proposed in [17], which also employed ballooning mechanism for VM memory resizing. In comparison with the feedback strategy, U-tube uses a dynamic memory balancing algorithm to pre-allocate memory to a VM that has potential memory requirements, and thus avoids application performance degradation.

Although ballooning can improve VMs' memory usage through dynamic allocation, it may impact the performance of applications which manage their own memory, such as databases and Java runtimes. Salomie et al. motivated by this problem and proposed an application-level ballooning mechanism for server memory overcommitment [30].

The following introduces other dynamic memory management techniques that are different from ballooning. Transcendent Memory [24] provides sharable memory pools among different VMs to manage the physical memory. The hypervisor collects idle memory from different VMs and manages it as one or more physical memory pools. A guest OS can indirectly access this memory with a well-defined API which imposes a carefully-crafted set of rules and restrictions. From the perspective of a guest OS, a memory pool appears to be a fast RAM disk with nondeterministic and varying size. A guest OS may use a memory pool as an extension to its memory, thus reduce disk I/O and improve performance. However, the management of memory pools is very complicated. SwapBypass [21] is somewhat similar to transcendent memory. It is a proof-of-concept symbiotic service implemented in Palacios VMM. SwapBypass imposes SymCall to re-consider swap decisions made by a symbiotic Linux guest, adapting to guest memory pressure. Although a page in the VM may be swapped out when the guest is experiencing high memory pressure, SwapBypass can still keep the page in memory and mark it available in the shadow page table. In this way, SwapBypass allows a guest to access the swapped pages at main memory speeds and use more physical memory than it was initially allocated. Overdriver [33] was proposed to expand VM memory capacity through both VM live migration and network memory techniques, but introduced much performance overhead in terms of network traffic and latency. These works all impose indirect memory accessing mechanisms to expand VMs' memory capacity, and thus introduce an additional latency compared to native memory accessing. In contrast, the memory added by hotplugging can be directly accessed by guest OSes, and thus does not cause performance degradation.

This paper mainly focuses on the comparison within a single machine. Recently, resource management on multi-tier applications are emerging in the cloud environment [16], [20]. It is out of the scope of this paper to revisit different memory management techniques in those environments.

6 Conclusion

In virtualization environments, it remains challenging to effectively manage the main memory resource. Dynamic memory allocation mechanisms such as ballooning and hotplug were proposed to handle the dynamics of memory demands. However, so far there is no quantitative comparison between these two mechanisms. In this paper, we first develop a runtime system called U-tube, which can adopt hotplug or ballooning for dynamic memory allocation freely. We then propose dynamic memory balancing and memory overcommitment algorithms to manage memory resource for VMs dynamically. Finally, we make a quantitative and comparative comparison between hotplug and ballooning in terms of implementation details, performance overhead, memory fragmentation level, performance speedup of applications. This study can benefit system administrators to better understand strengths and weaknesses of the two approaches, and thus make better decisions on the alternatives in different scenarios.

There are a number of extensions for U-tube in the future. First, it is interesting to study U-tube in NUMA environments, such as the issues of memory locality and load balancing. Second, releasing the constraint of memory cap may introduce security concerns because a malicious VM may continuously request memory and finally exhaust all memory resource on a host. This is a general security issue and there are numbers of works on memory leak detection [10], [37]. It could be interesting to address this issue through memory allocation policies, such as max-min fairness. Third, the memory hotplug technique provides an opportunity for rank-aware RAM power saving [34].

Acknowledgments

This work was supported by NSFC under grants No. 61300040, 61272408, 61322210, the Fundamental Research Funds for the Central Universities under grant No. 13MS87, and a startup grant from Nanyang Technological University (NTU), Singapore. The work was partly done when Haikun was visiting NTU. The source code of U-tube can be found from sourceforge website (https://sourceforge.net/p/liquidmem).

References

[1] TPC-C, 2012, [Online]. Available: http://www.tpc.org/tpcc

[2] SPEC CPU2006, 2006, [Online]. Available: http://www.spec.org/cpu2006/CINT2006/

[3] Memory hotplug for Xen, 2011, [Online]. Available: https://lkml.org/lkml/2011/3/28/108

[4] VMware vSphere hot-add, 2011, [Online]. Available: http://www.petri.co.il/vsphere-hot-add-memory-and-cpu.htm

[5] R. Azimi, L. Soares, M. Stumm, T. Walsh, and A. D. Brown, "PATH: Page access tracking to improve memory management," in *Proc. Int. Symp. Memory Manage.*, Oct. 2007, pp. 31–42.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. ACM Conf. Object-Oriented Program., Syst., Lang., Appl.*, Oct. 2006, pp. 169–190.

[7] 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," in *Proc. ACM Symp. Oper. Syst. Principles*, Oct. 2003, pp. 164–177.

[8] K. Colbert and R. Venkatasubramanian, "Understanding host & guest memory usage and related memory management concepts," *VMWORLD*, 2007.

[9] R. Chandra, S. Devine, A. Gupta, and M. Rosenblum, "Scheduling and page migration for multiprocessor compute servers,," in *Proc. ACM Conf. Archit. Support Program. Lang. Oper. Syst.*, Oct. 1994, pp. 12–24.

[10] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proc. ACM Int. Conf. Program. Lang. Des. Implementation*, Jun. 2007, pp. 480–491.

[11] A. Gordon, M. R Hines, D. D Silva, M. Ben-Yehuda, M. Silva, and G. Lizarraga, "Ginkgo: Automated, application-driven memory overcommitment for cloud computing," in *Proc. RESoLVE: Runtime Environm./Syst., Layering, Virtualized Environ. Workshop*, 2011.

[12] A. Gulati, A. Merchant, and P. J. Varman, "mClock: Handling throughput variability for hypervisor IO scheduling," in *Proc. USENIX Symp. Oper. Syst. Des. Implementation*, Oct. 2010, pp. 437–450.

[13] M. Gorman and P. Healy, "Measuring the Impact of the Linux Memory Manager," in *Proc. Libre Softw. Meeting*, 2005.

[14] M. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proc. ACM Int. Conf. Virtual Execution Environ.*, Mar. 2009, pp. 51–56.

[15] D. Hansen, M. Kravetz, B. Christiansen, and M. Tolentino, "Hotplug Memory and the Linux VM," in *Proc. Linux Symp.*, Jul. 2004, pp. 278–294.

[16] D. Huang, B. He and C. Miao, "A survey of resource management in multi-tier web applications," *IEEE Commun. Surv. Tutorials*, vol. PP, no. 99, pp. 1–17, Jan. 2014, DOI: 10.1109/SURV.2014.010814.00060

[17] 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 Symp. Integr. Manage.*, Jun. 2009, pp. 630–637.

[18] M. Kesavan, A. Gavrilovska, and K. Schwan, "Differential virtual time (DVT): Rethinking I/O service differentiation for virtual machines," in *Proc. ACM Symp. Cloud Comput.*, Jun. 2010, pp. 27–38.

[19] D. Le and H. Wang, "An effective memory optimization for virtual machine-based systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 10, pp. 1705–1713, Oct. 2011.

[20] H. Liu and B. He, "VMbuddies: Coordinating live migration of multi-tier applications in cloud environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. PP, no. 99, pp. 1–14, Apr. 2014. DOI:10.1109/TPDS.2014.2316152

[21] J. Lange and P. A. Dinda, "SymCall: Symbiotic virtualization through VMM-to-guest upcalls," in *Proc. ACM Int. Conf. Virtual Execution Environ.*, Mar. 2011, pp. 193–204.

[22] D. Magenheimer, "Memory overcommit... without the commitment," in *Proc. Xen Summit*, Jun. 2008, pp. 1–3.

[23] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the buffer cache in a virtual machine environment," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Oct. 2006, pp. 14–24.

[24] D. Magenheimer, "Transcendent memory on xen," in *Proc. Xen Summit*, 2009, pp. 1–3.

[25] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazi`eres, S. Mitra, A. Narayanan, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for ramclouds: Scalable high-performance storage entirely in DRAM," *ACM SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, Dec. 2009.

[26] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat, "M3R: Increased performance for in-memory hadoop jobs," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1736–1747, Aug. 2012.

[27] S. S. Pinter, Y. Aridor, S. Shultz, and S. Guenender, "Improving machine virtualization with 'Hotplug Memory'," in *Proc. 17th IEEE Int. Symp. Comput. Archit. High Perform. Comput.*, Oct. 2005, pp. 168–175.

[28] J. Schopp, D. Hansen, M. Kravetz, H. Takahashi, I. Toshihiro, Y. Goto, K. Hiroyuki, M. Tolentino, and B. Picco, "Memory hotplug redux," in *Proc. Linux Symp.*, Jul. 2005, pp. 151–174.

[29] J. Schopp, K. Fraser, and M. J. Silbermann, "Resizing memory with balloons and hotplug," in *Proc. Linux Symp.*, Jul. 2006, pp. 305–311.

[30] T. I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," in *Proc. ACM Eur. Conf. Comput. Syst.*, Apr. 2013, pp. 337–350.

[31] E. Tasoulas and H. Haugerund, "Bayllocator: A proactive system to predict server utilization and dynamically allocate memory resources using bayesian networks and ballooning," in *Proc. USENIX Large Installation Syst. Admin. Conf.*, Dec. 2012, pp. 111–121.

[32] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proc. Symp. Oper. Syst. Des. Implementation*, Dec. 2002, pp. 181–194.

[33] D. Williams, H. Weatherspoon, H. Jamjoom, and Y. H. Liu, "Overdriver: Handling memory overload in an oversubscribed cloud," in *Proc. ACM Int. Conf. Virtual Execution Environ.*, Mar. 2011, pp. 205–216.

[34] D. Wu, B. He, X. Tang, J. Xu, and M. Guo, "RAMZzz: Rank-aware DRAM power management with dynamic migrations and demotions," in *Proc. ACM/IEEE SuperComput.*, Nov. 2012, p. 32.

[35] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou, "Distributed systems meet economics: pricing in the cloud," in *Proc. USENIX Workshop Hot Topics Cloud Comput.*, Jun. 2010, pp. 6–6.

[36] C. Xu, S. Gamage, P. N. Rao, A. K. lou, R. R. Kompella, and D. Xu, "vSlicer: Latency-aware virtual machine scheduling via differentiated-frequency CPU slicing," in *Proc. ACM Symp. High-Perform. Parallel Distrib. Comput.*, Jun. 2012, pp. 3–14.

[37] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 115–125, Sep. 2005.

[38] Z. Xiao, W. Song, and Q. Chen, "Dynamic resource allocation using virtual machines for cloud computing environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 6, pp. 1107–1117, Jun. 2013.

[39] W. Zhao and Z. Wang, "Dynamic memory balancing for virtual machines," in *Proc. ACM Int. Conf. Virtual Execution Environ.*, Mar. 2009, pp. 21–30.

[40] J. Zhu, Z. Jiang, Z. Xiao, and X. Li, "Optimizing the performance of virtual machine synchronization for fault tolerance," *IEEE Trans. Comput.*, vol. 60, no. 12, pp. 1718–1729, Dec. 2011.

Haikun Liu received the PhD degree from the Huazhong University of Science and Technology. He is currently a research fellow at the School of Computer Engineering, Nanyang Technological University. He received the outstanding doctoral dissertation award in Hubei province, China. His current research interests include virtualization technologies, cloud computing, and distributed systems. He is a member of the IEEE.

Hai Jin received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST) in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering at HUST in China. He is currently the dean of the School of Computer Science and Technology at HUST. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He was at the University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of National 973 Basic Research Program Project of Virtualization Technology of Computing System. He is a member of the Grid Forum Steering Group (GFSG). He has co-authored 15 books and published more than 400 research papers. His research interests include computer architecture, virtualization technology, cluster computing and grid computing, peer-to-peer computing, network storage, and network security. He is named the steering committee chair of several International Conferences, such as GPC, APSCC, FCST, and ChinaGrid and is a member of the steering committee of CCGrid, NPC, GCC, ATC, and UIC. He is a senior member of the IEEE and a member of the ACM.

Xiaofei Liao received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), China, in 2005. He is currently a professor in the School of Computer Science and Engineering at HUST. He was a reviewer for many conferences and journal papers. His research interests are in the areas of system software, P2P system, cluster computing and streaming services. He is a member of the IEEE and the IEEE Computer society.

Wei Deng received the PhD degree from the Huazhong University of Science and Technology. He received the Best Paper Nominee from IEEE CloudCom 2012, the scholar travel grant of ICDCS 2013, and the Mircrosoft Fellowship Nominee. His research interests focus on cloud computing, datacenter networking, green computing, smart grids, modeling and optimization.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science from the Hong Kong University of Science and Technology (2003-2008). He is an assistant professor in the Division of Networks and Distributed Systems, School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing, cloud computing, and database systems. He has been awarded with the IBM PhD fellowship (2007-2008) and with NVIDIA Academic Partnership (2010-2011).

Cheng-Zhong Xu received the PhD degree from the University of Hong Kong in 1993. He is currently a tenured professor of Wayne State University and the director of the Institute of Advanced Computing and Data Engineering, Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences. His research interests include parallel and distributed systems and cloud computing. He has published more than 200 papers in journals and conferences. He was the Best Paper Nominee of 2013 IEEE High Performance Computer Architecture (HPCA), and the Best Paper Nominee of 2013 ACM High Performance Distributed Computing (HPDC). He was on a number of journal editorial boards, including *IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Cloud Computing, Journal of Parallel and Distributed Computing* and *China Science Information Sciences*. He received the Faculty Research Award, Career Development Chair Award, and the President's Award for Excellence in Teaching of WSU. He also received the "Outstanding Oversea Scholar" award of NSFC. He is a senior member of the IEEE. For more information, visit http://www.ece.eng.wayne.edu/∼czxu.