MemFlex: A Shared Memory Swapper for High Performance VM Execution

Qi Zhang, Ling Liu, *Fellow, IEEE*, Gong Su, and Arun Iyengar, *Fellow, IEEE*

Abstract—Ballooning is a popular solution for dynamic memory balancing. However, existing solutions may perform poorly in the presence of heavy guest swapping. Furthermore, when the host has sufficient free memory, guest virtual machines (VMs) under memory pressure is not be able to use it in a timely fashion. Even after the guest VM has been recharged with sufficient memory via ballooning, the applications running on the VM are unable to utilize the free memory in guest VM to quickly recover from the severe performance degradation. To address these problems, we present *MemFlex*, a shared memory swapper for improving guest swapping performance in virtualized environments with three novel features: (1) *MemFlex* effectively utilizes host idle memory by redirecting the VM swapping traffic to the host-guest shared memory area. (2) *MemFlex* provides a hybrid memory swapping model, which treats a fast but small shared memory swap partition as the primary swap area whenever it is possible, and smoothly transits to the conventional disk-based VM swapping on demand. (3) Upon ballooned with sufficient VM memory, *MemFlex* provides a fast swap-in optimization, which enables the VM to proactively swap in the pages from the shared memory using an efficient batch implementation. Instead of relying on costly page faults, this optimization offers just-in-time performance recovery by enabling the memory intensive applications to quickly regain their runtime momentum. Performance evaluation results are presented to demonstrate the effectiveness of *MemFlex* when compared with existing swapping approaches.

Index Terms—Virtualization, Shared Memory, Memory Swap

1 Introduction

Virtualization is a core enabling technology for Cloud computing. Although virtualization has shown great success in dynamic sharing of hardware resources, dynamic VM memory consolidation remains a challenging problem for a number of reasons. First, existing dynamic memory balancing solutions do not address the problem of guest swapping. Memory intensive applications are characterized by unpredictable peak memory demands. Such peak demand can lead to drastic performance degradation, resulting in VM or application crashes due to out of memory errors. Dynamic memory consolidation is an important and attractive functionality to deal with peak memory demands of memory-intensive applications in a virtualization platform. Ballooning is a dynamic memory balancing mechanism for non-intrusive sharing of memory between host and its guest. However, it is hard to make decisions on when to start ballooning and how much memory ballooning is sufficient. The state of the art proposals typically resort to estimating the working set size of each VM at run time. Based on its estimated memory demands, additional memory will be dynamically added to or removed from the VM [1], [2], [3], [4]. However, accurate estimation of VM working set size is difficult under changing workloads [5]. Therefore, dynamic memory balancer may not discover in time that the VM is under memory pressure, or may not balloon additional memory fast enough. As a result, the VM under memory pressure may see more guest memory swapping events and more drastic performance degradation. Second, by virtualization design, when a virtualized host boots, it treats each of its hosted VMs as a process and allocates it a fixed amount of memory. Each guest VM is managed by a guest OS, independently (and unaware of the presence) of the host OS. Thus, even when the host has sufficient free memory, the guest VMs under memory pressure are unaware. Thus, any delay in dynamic memory balancing can cause the VM not be able to utilize the host idle memory in a timely fashion. As a result, VMs under memory pressure will experience increased guest swapping, which can lead to VM and applications to crash due to high latency induced timeout. Finally, due to the performance overhead incurred by page table scanning during page fault based swap-in, applications running on the VM are often unable to utilize the free memory that has been inflated to the VM fast enough due to poor performance of VM swapping-in operations.

In this paper, we argue that (i) efficient guest VM swapping can significantly alleviate the above problems and (ii) fast VM swapping is a critical component to ensure the just-in-time effectiveness of dynamic memory balancing. We design and implement *MemFlex*, a host-coordinated shared memory swapper for improving VM swapping performance in virtualized environments. There are a number of challenges for redirecting guest VM swapping to the host-guest shared memory area. First, the shared memory should be organized in a way that allowing multiple VMs to shared dynamically and access concurrently. Second, since there could be no more space in the shared memory to accommodate the swapped out pages from the VMs, *MemFlex* needs to enable VMs to interact with both shared memory and the traditional swap devices at the same time. Third, in order to achieve good utilization of the shared memory, a VM needs to proactively release its currently used shared memory as soon as it gains enough free memory from the host.

With these challenges in mind, we design *MemFlex*, a flexible shared memory swapper with three original contributions: (1) By redirecting the VM memory swapping to the host-guest shared memory swap partition, *MemFlex* avoids the high overhead of disk I/O for guest swapping as well as guest-host context switching, and enables the guest VM to respond fast to the newly ballooned memory and to quickly recover from severe performance degradation under peak memory demands. (2) To handle the situation of limited shared memory swap area due to insufficient available memory at the host, *MemFlex* provides a hybrid memory swapping model, which treats shared memory swap partition as the small primary swap area and the disk swap partition as the secondary swap area. This model enables fast shared memory based VM swapping whenever it is possible and a smooth transition to the conventional guest OS swapping on demand. (3) To address the problem of slow recovery of memory intensive workloads even after sufficient additional memory has been successfully allocated via ballooning, *MemFlex* provides a fast swap-in optimization to proactively swap-in the pages resident in the shared memory swap area. We implement the first prototype of *MemFlex* on KVM platform. Our experimental evaluation shows that *MemFlex* significantly improves the VM performance when it is under high memory pressure. *MemFlex* with *proactive swap-in* achieves two orders of magnitude performance improvement on VM memory swap-in after memory recharging via ballooning, and enables both VM and application execution to quickly regain their runtime

- *Q. Zhang and L. Liu are with the School of Computer Science, Georgia Institute of Technology, Atlanta, GA, 30332.*
  E-mail: {qzhang90, lingliu}@cc.gatech.edu
- *G. Su and A. Iyengar are with IBM T.J. Watson Research, Yorktown Height., NY, 10598*
  E-mail: {gongsu, aruni}@us.ibm.com

momentum.

The rest of the paper is organized as follows: We review the related work in Section 2 and describe the design of *MemFlex* in Section 3. Experimental evaluation results are reported in Section 4 and the paper is concluded in Section 5.

2 Related Work

Most of existing efforts have been dedicated to developing different host-guest coordination mechanisms along three threads.

The first thread is to introduce **host coordinated ballooning and host swap**. The balloon driver, proposed in 2002 [6], has been widely adopted in mainstream virtualization platforms, such as VMware [7], KVM [8], Xen [9]. Most of them embed a driver module into the guest OS to reclaim or recharge VM memory [5]. A fair amount of research has been devoted to periodic estimation of VM working set size because an accurate estimation is essential for dynamic memory balancing using the balloon driver. For example, VMware introduced statistical sampling to estimate the active memory of VMs [6], [10]. Alternatively, [11] builds and updates the page-level LRU histograms by having the hypervisor intercepting memory accesses from each VM and uses the LRU-based miss ratio to estimate VM memory working set sizes. [12] proposed to implement the page-level miss ratio estimation using specific hardware to lower the cost of tracking the VM memory access. However, [3], [13], [14] show that accurate VM working set size prediction is difficult under changing conditions. Host swap is a guest OS transparent mechanism [10] to deal with the shortage of host free memory by having host or hypervisor swapping out some inactive memory pages to host-specific disk swap partition, without informing the respective guest OS. However, such uncooperative host swapping can cause double paging [5]. VSwapper [5] tracks the correspondences between disk blocks and guest memory pages to avoid unnecessary disk I/O caused by uncooperative memory swapping between host and guest VMs. But it does not improve the VM page-level swap performance.

The second thread is the **Coordinated memory management**, which has centered on redesigning operating system (OS) to enable more efficient host-guest coordination. The transcendent memory (tmem) on Linux by Oracle and the active memory sharing on AIX by IBM PowerVM are the two representative efforts. For example, transcendent memory [16] allows the VM to directly access a free memory pool in the host. Frontswap [17] is a Linux kernel patch that using the tmem as a VM swap space, and it is currently working on Xen. In Frontswap, a hypercall has to be invoked for each swapped out page, and the swap-in operations depend on the page faults, which are quite costly. *MemFlex* is running on KVM, and it avoids the hypercall and page fault for the page swap-out and swap-in respectively. [18] shows that this pool of memory can be used by Guest OS to invoke the host OS services and by the host OS to obtain the memory usage information of the guest VM. [19] allows the applications that implement their own memory resource management, such as database engines and Java virtual machines (JVMs), to reclaim and free memory using application-level ballooning. However, most of the proposals in this thread rely on some serious changes to guest OS or applications, making it harder for wide deployment of the solutions.

The third thread is centered on complimentary techniques to improve **dynamic memory consolidation**, ranging from memory hotplug, collaborative memory management to remote memory swapping. Memory Hotplug [20], [21] was proposed to address the problem of insufficient memory or memory failing at both guest VMs and host. It refers to the ability to plug and unplug physical memory from a machine [20] without reboot to avoid downtime. [20] conducts comparative studies between balloon driver and memory hotplug. The results show that balloon driver has the advantage of easy implementation, since it can use the native MMU of the guest OS. But balloon driver cannot increase a VM's memory over its pre-defined cap, which is not a restriction for memory hotplug. Also, memory hotplug does a better job in avoiding memory fragmentation.. Collaborative memory management [22] proposed a novel information sharing mechanism between host and guests to reduce the host paging rate. In addition, several efforts have engaged in combining swapping to the local disk with swapping to the remote memory via network I/Os: [23] organizes the memory resource from all nodes of a cluster into one or more memory pools that can be accessed via high speed interconnect. It deals with memory overload by swapping the VM memory pages to network memory, and introduces an optimization to bypass the TCP/IP protocol by removing IP routing to improve the performance of swapping to the cluster-wide remote memory. Overdriver [24] distinguishes guest swapping caused by short term overloads from those due to long term overloads by detecting the reason of guest swapping. It uses the remote memory when guest swapping caused by short term overloads and migrates VMs if the swapping is instigated due to long-term overload.

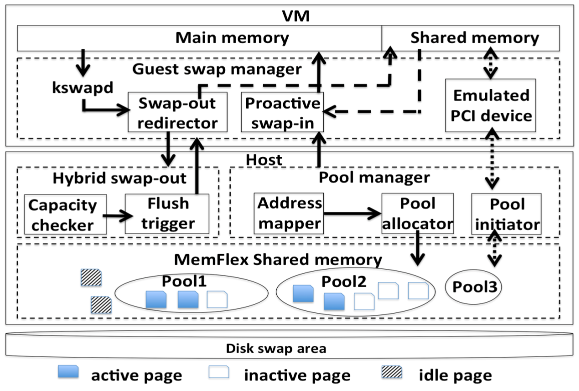


Fig. 1. MemFlex system overview.

In comparison, *MemFlex* promotes to use a small amount of host-coordinated shared memory as the fast swap partition and resort to conventional secondary storage based swap partition when there is insufficient free memory on the host. It is orthogonal but complimentary to using network memory for guest swapping. To the best of our knowledge, *MemFlex* is the first shared memory swapper with host coordinated two level memory swapping and fast proactive swap-in optimization.

3 MemFlex Design

The goal of *MemFlex* is to design and implement a shared memory based high performance guest VM swapping framework. Figure 1 gives a sketch of the *MemFlex* system architecture. Instead of swapping to the disk, *MemFlex* intercepts and redirects the swap-out guest memory pages to the guest shared memory swap space within the host memory region. This minimizes the high overhead of disk I/Os and alleviates the problem of uncooperative swapping. The shared memory swap partition is organized into multiple shared memory pools, one per VM, and each corresponds to a host-guest shared memory area. This design presents a clean separation and facilitates the protection of the shared memory swap area of a VM from unauthorized access by the other VMs on the same host. *MemFlex* optimizes the shared memory design with three unique features:

Dynamic memory management. A piece of shared memory is pre-allocated and shared by multiple VMs. The shared memory is equally divided into smaller chunks. Each chunk will be dynamically assigned to and revoked from a VM in terms of the VM's swap traffics. If there is no more room to allocate in the shared memory, the VM uses hybrid swap-out mechanism to accommodate the additional swapped out pages.

Hybrid swap-out. When the amount of shared memory cannot hold all the memory pages swapped out from a VM, *MemFlex* uses the shared memory swap area as the fast primary swap partition and automatically resort to secondary storage swap partition. This feature improves the overall guest swapping performance even when there is insufficient shared memory resource.

Proactive swap-in. Once a VM is recharged with sufficient free memory via a balloon driver, it needs to swap-in the pages from the shared memory swap area back to its main memory. *MemFlex* improves swap-in performance from two aspects: (i) the overhead of swap-in is reduced from disk I/O to page table operations for those pages residing in the shared memory swap area, and (ii) instead of demand paging from the guest VM, *MemFlex* enables VM to proactively swap in the pages from the shared memory, minimizing the costly page faults, and effectively eliminating the problem of slow recovery of application performance upon the addition of sufficient memory to the VM under pressure.

3.1 Host Memory based VM Swapping

There are two alternative ways to implement the host-coordinated memory swapping: (i) swapping to host ramdisk and (ii) swapping to host-guest shared memory.

Ramdisk based approach improves efficiency of traditional VM swapping by swapping to host memory instead of the VM disk image. This is achieved by building a ramdisk from the host memory and mounting this ramdisk as the swapping area of the guest VM. This approach is simple and requires no guest kernel modification, but it has some inherent performance limitations. Although the ramdisk is residing in the host memory, it is still used by the guest VM as a block device. From the perspective of guest VMs, writing to the ramdisk is the same as writing to its disk though faster. As a result, certain overheads that are applicable to disk I/O will also apply to ramdisk I/O. Concretely, a disk I/O request from the application running on a guest VM will first go through the guest OS kernel before the request and the I/O data are copied from the guest kernel to the host OS kernel; this represents a data transfer between the host user space and the host kernel space. Hence, when a guest VM swaps its memory pages into a mounted ramdisk, there are two major sources of overhead: the frequent context switch and the data copying between the host user space and host kernel space. We use Perf tool [25] to evaluate the performance overhead of ramdisk-based swapping by letting a VM swap 512MB memory into a ramdisk provided by its KVM host. It shows that (i) 7.86% CPU cycles are spent on *copy_user_generic_string()*, which is a kernel function that copies user generated data into kernel space; and (ii) 7.50% CPU cycles are spent on context switching between the guest VM and the host. Although ramdisk based swapping may outperform the traditional disk-based swapping (baseline), these overheads limit the efficiency of ramdisk based swapping when compared to the *shared memory based swapping* (see detail in Section 4).

Shared memory based approach. *MemFlex* presents a shared memory based swapping approach. Instead of mounting a host resident ramdisk to the guest VM, in *MemFlex*, a memory region provided by the host is mapped into the guest VM address space and used as the host-guest shared swapping area for the guest VM. Figure 1 illustrates the workflow of shared memory based swapping implemented on the KVM platform. As a part of the system initialization, A shared memory region with a pre-configured size is initialized. The shared memory area is divided into multiple adaptive *pools*, and each pool is corresponding to a specific VM. A pool manager is working in the host, which has two functionalities: (i) maintaining the mapping between the page offset in the VM swap area and the address in the corresponding shared memory pool, and (ii) dynamically adjusting the size of each VM pool. The pages in the shared memory are categorized into three types: *active* pages refer to ones being used by the VMs as their swapping destination, *inactive* pages are those allocated to some pool but has not being used yet, and *idle* pages indicate those shared memory pages that do not belong to any pool, as shown in Figure 1. The *kswapd* is a default kernel daemon, which is responsible for memory page swap-in and swap-out. The swap redirector intercepts and redirects the swap in/out pages from/to the VM shared memory area. The proactive swap-in handles both on-demand swap-in and batch swap-in.

Compared with swapping to a block device, such as hard disk or ramdisk, the shared memory swapping mechanism in *MemFlex* has a number of advantages. *First*, by intercepting all the memory swapping traffic in the guest VMs and redirecting them to the shared memory regions between the host and its guest VMs, both VM and host file system can be skipped in VM memory swap and no block I/O needs to be carried out. *Second*, by eliminating or minimizing the disk I/O traffic caused by VM swapping, *MemFlex* also helps alleviate the performance interference between memory intensive VMs and disk intensive VMs. *Third*, with *MemFlex*, the cost of double paging can be largely reduced. Even though double paging [6], [26], [27] may still happen in the presence of host swap in *MemFlex*, its cost will be reduced from two block device I/O operations (one by reading the page from the host swap area to the VM memory, and the other by writing the page from the VM memory to the VM swap area) to only one block device I/O operation, which is the cost of a regular VM swap-in.

3.2 Dynamic Shared Memory Management

Each VM has its own shared memory swap area, which is allocated from the global shared memory. In order to achieve proportional shared memory allocation, the global shared memory is divided into smaller chunks, and each chunk will be allocated and revoked from a VM according to the demand from its swap traffic.

Figure 2 illustrates the working of dynamic shared memory management in MemFlex: two categories of addresses are maintained for the shared memory swap area of each VM: pseudo addresses and a shared memory addresses. The pseudo addresses are continuous. It is in accordance to the offset of the swap area that is used by the operating system to interactive with the swap partitions. The shared memory addresses are the offsets of the shared memory that the swapped-out pages will be actually written to. Mappings between the pseudo addresses and the shared memory addresses are maintained by *MemFlex* to translate from the one to the other. There is also a bitmap at the very beginning of the global shared memory to indicate which chunks are free and which chunks are in use. Each bit in the bitmap correspondent to a chunk in the shared memory. Since multiple VMs could be able to request shared memory chunks concurrently, *MemFlex* also maintains a global lock to

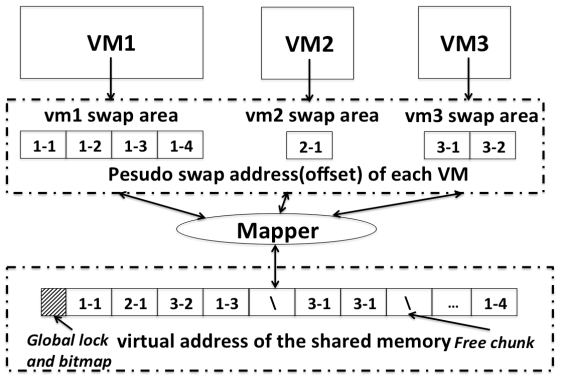


Fig. 2. Dynamic shared memory management.

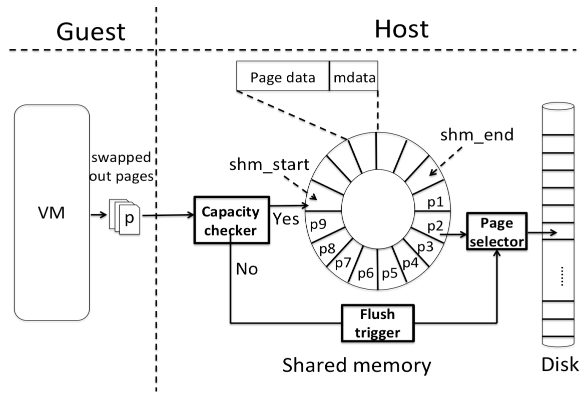


Fig. 3. Hybrid (disk and shared memory) swap-out.

guarantee a VM's exclusive access and update of the bitmap. At the very beginning, a single piece of shared memory chunk will be allocated to a VM as its shared memory swap area. Additional memory chunks will be added if the size of current chunks is not big enough for the swap traffic from the VM. The *hybrid swap-out* mechanism, which will be discussed in section 3.3, is used to deal with the case when there is not enough chunks to be allocated from the global shared memory. Since a VM can get more memory by utilizing a balloon driver [6], a separate thread is maintained for each VM to check whether there is enough memory to swap-in all the pages that are current in the shared memory chunks. If there is, the VM will proactively swap in those pages and return the chunks to the global shared memory.

3.3 Hybrid Swap-out

Hybrid Swap-out is designed to handle the situation when the size of shared memory is not large enough to hold all the swapped pages from a VM. To enable the shared memory as the fast primary swap partition and smooth transition to the disk swap area as the secondary choice, *MemFlex* employs a hybrid guest swap model with both shared memory and the disk. VM swapped out pages that cannot be kept in the shared memory are written to its disk swap area. There are a number of strategies to design the hybrid swap-out model. For instance, if we maintain the timestamp or access frequency of memory pages, we can use such information to prioritize the swap-out pages that should be kept in the shared memory. Clearly, this selective swap-out to disk mechanism is optimized for keeping frequently accessed pages in shared memory at the cost of maintaining the timestamp or access frequency of swap-out memory pages. To simplify the implementation and minimize the overhead of hybrid swap-out, in the first prototype of *MemFlex*, we implement the following two light weighted mechanisms for hybrid swap-out:

Most recent pages to disk. This approach is straightforward but naive. In this approach, memory pages swapped out of the VM will be written into the shared memory first. When the shared memory is full, the newly swap-out pages from the VM will be written to the VM disk swap area. The advantage of this approach is its simplicity in both design and implementation. However, the disadvantage is also obvious: most of the pages stored in the shared memory are older than the pages in the disk swap area. According to the principle of locality, programs tend to reuse data near those they have used recently. Therefore, the possibility of accessing the recent swapped out data is larger than that of fetching an 'older' swapped out page, and using the disk swap area to store the most recent swapped out pages may seriously impact the performance of a large number of applications. Also the benefit of using shared memory as the swap area will diminish as the number of pages stored on disk area is increasing.

Least recent pages to disk. This approach puts older pages to the disk swap area when the shared memory swap area is full, enabling the most recent VM swapped out pages to be kept in the shared memory. One way to organize the shared memory swap area is to use a ring buffer as shown in Figure 3. We maintain a *shm_start* pointer and a *shm_end* pointer pointing to the swapped pages with the smallest and largest offset in the buffer respectively. If a page fault comes with an offset between these two pointers, all the accesses can be served from the shared memory. Otherwise, the conventional disk based swap path will be invoked. In this design, a separate working thread is standing by and ready to be triggered to flush the pages from the shared memory to the disk swap area. In order to parallel the disk I/O and page swap operations, the working thread starts flushing the page when the shared memory is partially full, say $m\%$ full. In the first prototype of *MemFlex*, we set the **least recent pages to disk** method as the default configuration for our hybrid swap-out model. According to [28], for the benchmark workloads such as Eclipse and SPECjbb, more than 50% of the swap-out pages stay in the swap storage on average for less than 20% of the total execution time of the workloads before being swapped in, and more than 75% of the swap-out pages stay in the swap storage for less than 28% of the total execution time. These statistics about resident time of swap-out pages are consistent with the general principle of locality and indicate that keeping the most recently swapped out pages in shared memory is more effective for hybrid swap-out than the most recent pages to disk method.

3.4 Proactive Swap-in

After a VM gets sufficient additional memory from the balloon driver, the VM needs to proactively swap in all the pages that are previously swapped out to the shared memory. This mechanism has two advantages. First, it reduces the number of page faults which occur in the near future. The memory paging itself is quite expensive. According to [29], it takes about an extra 1000 to 2000 CPU cycles to handle even a minor page fault, namely the required page already exists in the page cache. Otherwise, it will take even longer CPU waiting time to bring the page back into the VM memory from the secondary swap storage. The second advantage of our proactive swap-in optimization is the improved efficiency of shared memory utilization, because as soon as the proactive swap-in is completed, the corresponding shared memory swap

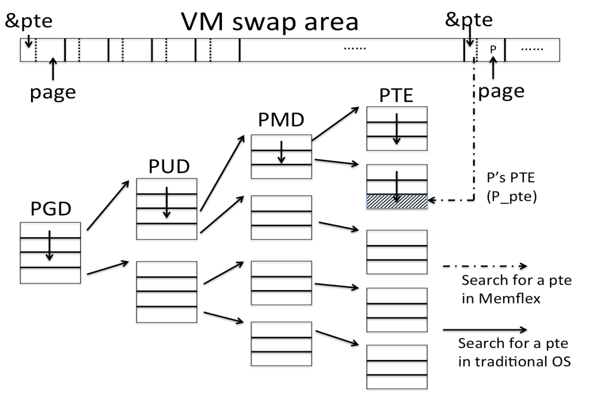

&pte
VM swap area
&pte
page
page
PGD
PUD
PMD
PTE
P's PTE (P_pte)
Search for a pte in Memflex
Search for a pte in traditional OS

Fig. 4. Proactive swap-in v.s. Baseline swap-in.

area will be released for use by other concurrent applications. Traditional OS provides two mechanisms to swap in pages from the disk swap area: *page faults* and *swapoff*. It is known that relying on the page faults to swap in the pages from the swap disk is expensive, and sometimes results in unacceptable delay in application performance recovery. Therefore, *MemFlex* implements the proactive swap-in by extending the *swapoff* mechanism. Concretely, *swapoff* is an OS syscall that enables to swap-in pages from the disk swap area in a batched manner. Compared with the page fault mechanism, *swapoff* allows larger amount of pages be swapped in during a shorter period of time. Once all the pages are swapped in, these pages can be accessed without any page faults. However, directly applying *swapoff* will not provide the speed-up required for fast application performance recovery for a number of reasons.

The *swapoff* syscall first checks whether there is enough free memory in the VM to hold all the currently swapped out pages, and does nothing if there is insufficient free memory. Otherwise, *swapoff* brings all the pages from the swap area to the main memory of the VM. However, this process is extremely time consuming, because the sysycall *swapoff* traverses the swap area and swaps in each page one by one. Figure 4 illustrates the process: each time when a page is swapped in by the syscall *swapoff*, it needs to locate the corresponding page table entry (PTE) and update it. If a swapped out page is a shared page, then multiple PTEs need to be located and updated. Furthermore, in order to locate each PTE, *swapoff* has to scan the whole page table from the very beginning, compare each PTE with that of the page to be swapped in, until a match is found. The process of finding a corresponding PTE is annotated in Figure 4. In order to swap in the page P, which only has a single PTE (P_pte), the OS has to start from the page global directory (PGD), and traverse through all the PUD, PMD, and compare with all the PTEs until the P_pte is found. This design of *swapoff* is adopted in the traditional OS for two reasons: (1) The purpose of the *swapoff* syscall is to remove a specified swap device, which is assumed not to be invoked very frequently, thus a long delay can be tolerated. (2) The operation of swapping-in a page generally consists of two parts: reading the page from the swap area and updating the corresponding PTEs. The conventional swap area is a slow secondary storage, such as magnetic disk. Compared with the time spent on disk I/O wait, the cost of traversing the system wide page table for each page is relatively small. However, these reasons are no longer true in *MemFlex*. Reading a memory page from shared memory is much faster than reading a memory page from disk. Thus, compared with swapping in from the shared memory, the cost of page table scanning becomes a significantly portion of the entire cost of swapping-in a page from the shared memory swap area. Also this page table traversal cost increases as the size of the page table becomes larger. Our idea for designing an efficient implementation of *proactive swap-in* mechanism is to maintain some meta data in the swap area, which can assist *MemFlex* to quickly locate the corresponding PTEs in the page table. Concretely, when a page is swapped out, MemFlex will keep the address of the PTEs related to this page together with the swapped-out page. Recall Figure 4, the address of the corresponding PTE is kept as the metadata in front of the swapped out page in the shared memory. For each page of 4K bytes, its meta data only takes up 4 bytes. Thus, the cost of keeping this metadata in the swap area is around only 1/1000 of the total size of the swapped out pages. For those shared pages which have multiple PTEs, we allocate a specific area in the shared memory as *PTE store*. In this case, the first byte of the meta data specifies the number of PTEs related to this page, while the lasts three bytes is an index pointing the first related PTE in the *PTE store*. When a page is swapped in, *MemFlex* is able to quickly locate the PTE(s) that needs to be updated by referring to this metadata without the need to scan the page table. The time spent on accessing the PTE of a page to be swapped in from the shared memory is only one time memory access, and it will not increase as the size of the system wide page table grows.

3.5 Interaction with kswapd

Kswapd is a kernel daemon, which is responsible for swapping memory pages. *MemFlex* intercepts the memory swap out/in traffics by instrumenting two critical kernel functions: ”swap_writepage()” and ”swap_readpage()”. *MemFlex* redirects the swap traffics from traditional disk based swap partition to the shared memory swap area by modifying the behaviors of these two functions. By default, the operating system (OS) treats the disk based swap partition in terms of pages. Thus *MemFlex* uses the same offset for the shared memory area as those previously used by the OS to track where the pages are swapped out to. ”struct swap_info_struct” is the data structure that the OS uses to represent each swap device. *MemFlex* adds another integer field ”is_shm” to indicate whether or not the swap partition is a shared memory area. Thus later on, when a swap device is chosen by the OS, *MemFlex* refers to this field to decide whether the contents of a swapped page should be copied to/from the shared memory or written to/read from disk. Since the shared memory that is allocated to a VM may not be virtually continuous, *MemFlex* maintains a mapping between the swapped out offsets and the actual virtual addresses in the shared memory that the pages are written to. *MemFlex* translates one address to another whenever a page needs to swapped out to or swapped in from the shared memory area . The mappings is maintained by *MemFlex* and updated when an additional shared memory chunk is added to a VM.

3.6 Security Support

MemFlex provides the support for security of the global shared memory that are accessed by multiple VMs by utilizing a number of system-level mechanisms. For example, it enforces that when a shared memory chunk is being released by a VM, it should be cleared first before making it available in the global shared memory pool. This ensures that when the *MemFlex* shared memory manager allocates this chunk to another VM latter on, this chunk is empty and all information from the previous VM have been erased. In the second release of *MemFlex*, the support of the VM trust group will be provided as another complimentary mechanism to allow those VMs that are mutually trusted to form a trust group. We allocate different global shared memory regions to different trust groups and prohibit the VMs in one trust group to access the shared memory region that is allocated to another trust group. This guarantees that swapped-out pages of a VM will not be accessed by another untrusted VM. One mechanism to enforce the VM trust group is to utilize encryption. For example, each VM can generate an exclusive key that is recorded by *MemFlex*. All swapped-out pages in the shared memory will be encrypted by the key. This encryption powered trust group offers stronger protection than using the VM trust group alone. As the overhead of encryption and decryption continues to drop, this mechanism can be deployed in practice.

4 Evaluation

Experiments Setup. Most of our experiments are conducted on an Intel Xeon based server provisioned from a SoftLayer cloud [30] with two 6-core Intel Xeon-Westmere X5675 processors, 24GB DDR3 physical memory available for the guest VMs, 1.5 TB SCSI hard disk, and 1Gbit Ethernet interface. The host machine runs Ubuntu 14.04 with kernel version 4.1.0, and uses KVM 1.2.0 with QEMU 2.0.0 as the virtualization platform. Four VMs are simultaneously running on the host. The guest VMs run Ubuntu 14.04 with kernel version 3.14.4. Each VM is assigned with two vCPUs. For the set of experiments reported in this paper, we first run the Redis server on VM1 and then start another memory intensive application with high VM memory demand of varying intensity. The memory utilizations of VMs are monitored at an interval of 5 seconds to trigger the balloon driver when necessary. We compare the following three cases in most of the experiments. **HDD case:** Each VM is assigned with up to 6GB memory. **MemFlex case:** Each VM is assigned with up to 5.5GB memory, while the remaining 2GB memory is reserved for the host as the available shared memory area. **Ramdisk case:** Similar to the *MemFlex* case, except that instead of using shared memory, the available 2GB memory is organized as a ramdisk, which is mounted to the VMs.

4.1 Overall Performance of MemFlex

In this section, we use Redis to evaluate the effectiveness of MemFlex on Redis workloads. A Redis server is running on VM1, which is pre-loaded with 5GB data in memory, while the other 3 VMs are idle. Different workloads generated by YCSB [31] are executed in a remote machine, which is connected to VM1 as a client. We let the client run for about 10 second, and then start the memory-intensive applications in the VM where the Redis server is running, which demands 7GB memory. This increases the total memory demand of VM1 to 12GB. In the *HDD* case, the balloon driver will move 2GB memory from each of the remaining three idle VMs (VM2, VM3 and VM4) to VM1, while in the *MemFlex* case and the *Ramdisk* case, the balloon driver will move a total of 4.5GB to VM1 with 2GB memory from VM2 and 2.5GB from VM3, while the reserved shared memory is used by VM1 as its swap area. We measure the throughput as well as the latency of the Redis server, and compare the Redis server performance among the three cases. The YCSB workloads used in the experiments include *Insert, Read, Update, and Scan*.

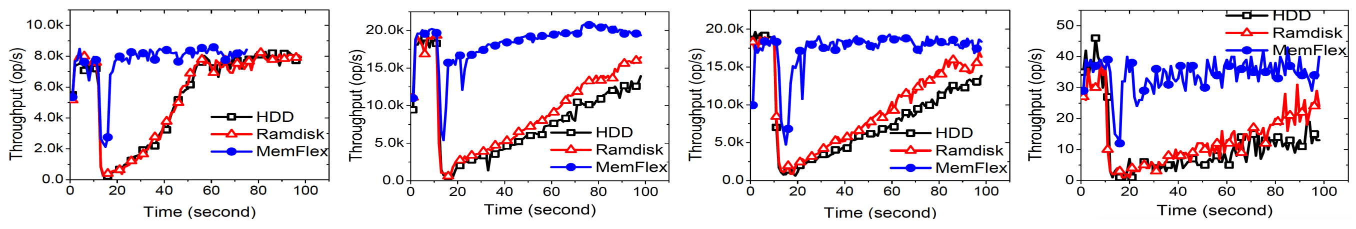
Figure 5 displays the results of how the Redis server performs in terms of throughput. We make several interesting observations. First, the Redis throughput drops significantly at around the 10th second in all the cases, no matter which workload is running. This is due to the execution of the memory-intensive applications. Since there is not enough free memory in VM1 when the memory-intensive applications start. However, VM1 cannot get free memory from the balloon driver immediately, which causes VM1 to swap out some of its memory pages, causing serious degradation of the Redis performance. We also observe that the performance in *MemFlex* also drops at the 10th second, this is because the memory-intensive applications also consume CPU cycles to allocate the large amount of memory when it starts to run, which results in severe CPU competition within VM1. Second, in both the *HDD* case and the *Ramdisk* case, the performance of all workloads can recover from the performance drop, but very slowly. The slow performance recovery indicates that after VM1 gets enough free memory from the balloon driver, the swap-in process becomes the dominating bottleneck. When VM1 needs to access some page that has previously been swapped out, a page fault will be triggered and the page needs to be read back from the swap device to the memory of VM1 through disk I/O. Both page fault and disk I/O are very expensive operations. In comparison, for the *MemFlex* case, the Redis performance recovers significantly faster. For the *Read* workload in Figure 5(c), its throughput drops from 17,813 OP/sec to 7,049 OP/sec at the 10th second, but it only takes about 5 seconds for the throughput to be fully recovered in *MemFlex* whereas it will take more than 80 seconds for the *Ramdisk case* and much longer for the *HDD case* to fully recover to the throughput performance prior to the launch of the memory-intensive applications. In short, MemFlex responds to the sudden memory pressure much faster and more effectively with the reserved shared memory in the host, compared to the HDD scenario, where all host available memory is pre-allocated equally to each VM, instead of reserving some host memory, and the response to memory pressure solely relies on the Balloon driver being periodically triggered or manual controlled by system admin. It also shows that *MemFlex* compliments the Balloon driver capability effectively through hybrid swap-out and fast shared memory swap-in.

It is worth noting that the swap device *Ramdisk* is also using the host memory for swap, but its performance, though slightly higher than that of the *HDD* case, is substantially lower than *MemFlex*. This experimentally proves our analysis in Section 3.1: when the swap area is mounted to a VM as a block device, the overhead of block I/O processing and the context switch between the VM and the host is dominating the efficiency of VM swapping. Thus simply changing the VM swap device from disk to ramdisk in host memory will not guarantee sufficient VM performance improvement. This observation further validates the superiority and originality of the *MemFlex* design.

Figure 6 displays the latency measured by the *Insert, Read, Update and Scan* workloads. Similar to the observations by Figure 5, the execution of the memory-intensive applications at around 10th second introduces significant performance degradation to the Redis server, which causes the sudden latency increase at that time. As the balloon driver inflates more memory at 5 seconds interval, the performance of Redis server in both *HDD* and *Ramdisk* case can recover, but at much slower speed than that of the *MemFlex* case. A tiny bump of the latency of the Redis server is observed in the *MemFlex* case, due to the execution of the second memory intensive application on the VM running Redis, and it recovers quickly, in just a few seconds, thanks to the design of *MemFlex* shared memory swapper.

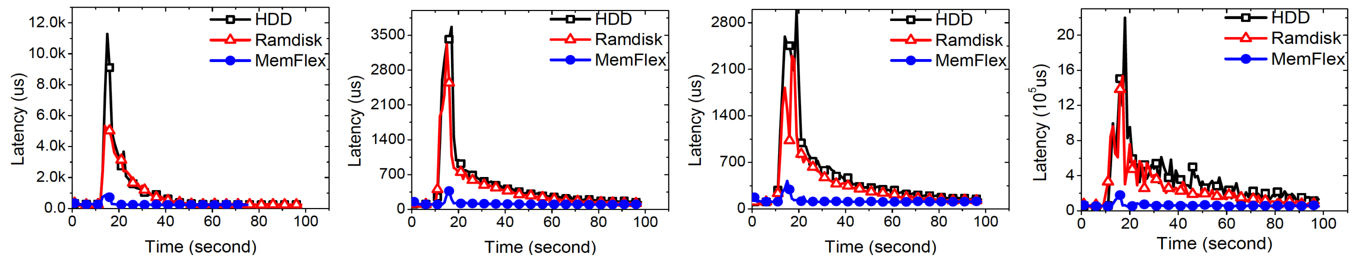
4.2 Micro Performance of MemFlex

In this section, we evaluate the effectiveness of the *proactive swap-in* mechanism in *MemFlex*. The purpose of *proactive swap-in* is to let the VM quickly swap in those previously swapped



(a) Insert (b) Update (c) Read (d) Scan

Fig. 5. Throughput of Redis server measured by YCSB workloads.

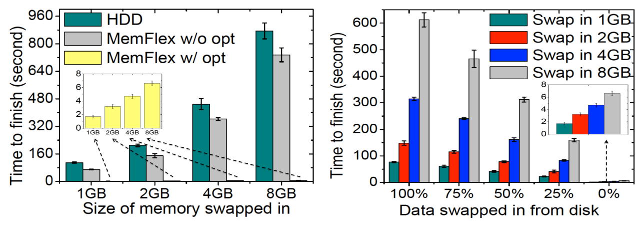


(a) Insert (b) Update (c) Read (d) Scan

Fig. 6. Latency of Redis server measured by YCSB workloads.

TABLE 1
Time (nanoseconds) spent on *Page read* and *PTE update* when swapping in 2GB data.

	Overall statistics(sec)		Per page statistics(ns)						
	Total time	s.d.	Page read	s.d.	update	s.d.	Others	s.d.	Total
HDD	212.96	10.49	9,931.20	190.80	292,318.40	8,318.57	98,772.20	921.61	401,021.80
MemFlex w/o opt	150.19	3.25	8,192.80	212.15	277,498.00	3,032.23	1,294.00	71.64	286,984.80
MemFlex w/ opt	3.46	0.03	5,153.40	90.42	416.00	29.83	475.60	12.47	6045.00



(a) Shared memory swapin (b) Hybrid Swap in

Fig. 7. Total time spent on swapping in.

out memory pages resident in the shared memory as soon as the VM gets sufficient free memory. This has two advantages: it reduces the number of expensive page fault operations in memory accesses and it quickly frees up the shared memory swap area for other applications.

We start a Redis server on VM1 with 6GB main memory and 2GB swap area. A client then loads 6GB data to the Redis server VM to fully fill the memory. Then we start the memory-intensive applications to demand 2GB memory from the VM. This will trigger 2GB of the Redis data to be swapped out to the VM1 swap area. Finally, we terminate the memory-intensive applications so that the 2GB memory in VM1 will be released and *MemFlex* will proactively swap in the 2GB swapped out memory pages from the shared memory swap area back to VM1. Various swap related statistics are measured.

Recall Section 3.4, the process of swapping-in a page consists of three parts: *Page read*, *PTE update* and *others*. Table 1 shows how much time is spent on each part in order to swap-in a page from the swap area under three cases: (1) *HDD swap-in*, (2) *MemFlex without (w/o) proactive swap-in optimization (MemFlex w/o opt)*, and (3) *MemFlex with (w/) proactive swap-in optimization (MemFlex w/ opt)*. We make three observations from Table 1. First, in the *HDD swap-in* case, the time spent on*Page read* and *PTE Update* occupies 2.48% and 72.89% of the total swap-in time. This is partly because the existence of swap cache and partly because in order to swap-in a single page, the OS needs to scan the page table to find the corresponding page table entries that it needs to update, which is very time consuming, and this cost will increase as the size of the page table increases. Second, compared to the *HDD swap-in* case, the average page swap-in time in the *MemFlex w/o opt* case is improved by 28.44% from 401,021.80 nanoseconds (ns) to 286984.80 ns. By zooming into the time spent on each part, we find that this improvement is mostly due to the decrease of ”Others”, which include the wait time of reading a page from disk. This also validates our analysis in Section 3.4 that when using the shared memory for VM swap traffic instead of disk swap, the *PTE Update* becomes the major bottleneck for swap-in events. With the help of *proactive swap-in*, the *PTE Update* time is significantly decreased from 277,498.00 ns to 416.00 ns. Because in this case, the OS swaps out not only memory pages but also some metadata, which can help the OS to quickly identify which page table entries need to be updated during the swap-in process. Thus the cost of scanning the page table is avoided.

Next, we measure the total time spent on swapping-in different amount of memory pages under different scenarios. Figure 7(a) shows that in both *HDD swap-in* case and *MemFlex w/o opt* case, the total time spent on swap-in will increase dramatically with the increase of the total amount of memory that needs to be swapped in. However, with *proactive swap-in*, the total time consumed by the same swap-in events is two orders of magnitude shorter than the *HDD swap-in* case and the *MemFlex w/o opt* case. Also as the size of the memory to be swapped-in grows, the time spent on swap-in also grows though slowly. For instance, in the *MemFlex w/ opt* case, swapping in 1GB memory needs 1.8 seconds, while it takes only 6.5 seconds to swap in 8GB memory. Figure 7(b) shows the swap-in performance when the swap area is a mix of shared memory and disk. Compared with the *HDD swap-in*, even if all the data is swapped in from the disk swap area, *MemFlex* saves 30% swap-in time compared to the *HDD swap-in* in Figure 7(a). Also, as the portion of on-disk data decreases, the total swap-in time also decreases.

5 Conclusions

We have presented the design of MemFlex, a highly efficient shared memory swapper. This paper makes three original contributions. First, *MemFlex* can effectively utilize host idle memory by redirecting the VM swapping traffic to the host-guest shared memory swap area. Second, *MemFlex* hybrid memory swapping model promotes to use the fast shared memory swap partition as the primary swap area whenever possible, and smoothly transits to the conventional disk-based VM swapping scheme on demand. Third but not the least, *MemFlex* proactive swap-in optimization offers just-in-time performance recovery by replacing costly page faults with an efficient swap-in implementation. We evaluate *MemFlex* using a set of well-known applications and benchmarks and show that *MemFlex* offers up to two orders of magnitude performance improvements over existing memory swapping methods. We have implemented the first prototype of *MemFlex* on the KVM platform (https://sites.google.com/site/gtmemflex/). We are planning to deploy *MemFlex* on the Xen platform by utilizing the relevant kernel functions and kernel data structures provided in Xen hypervisor.

Acknowledgement

The first two authors performed this work under the partial support by the National Science Foundation under Grants CNS-1115375, NSF 1547102, SaTC 1564097, and Intel ISTC on Cloud Computing.

References

[1] R. Birke, L. Y. Chen, and E. Smirni, "Data centers in the wild: A large performance study," *Technical Repo*, no. Z1204-002, 2012.
[2] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," *Communications of the ACM*, vol. 53, no. 10, pp. 85–93, 2010.
[3] M. R. Hines, A. Gordon, M. Silva, D. Da Silva, K. D. Ryu, and M. Ben-Yehuda, "Applications know best: Performance-driven memory overcommit with ginkgo," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*. IEEE, 2011, pp. 130–137.
[4] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: exploiting page sharing for smart colocation in virtualized data centers," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2009, pp. 31–40.
[5] N. Amit, D. Tsafrir, and A. Schuster, "Vswapper: A memory swapper for virtualized environments," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 349–366.
[6] C. A. Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.
[7] M. Rosenblum, "VMware's virtual platform," in *Proceedings of hot chips*, vol. 1999, 1999, pp. 185–196.
[8] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.
[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.
[10] "Understanding Memory Resource Management in VMware vSphere 5.0," *VMware Technical White Paper, August 2011*.
[11] W. Zhao, Z. Wang, and Y. Luo, "Dynamic memory balancing for virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 37–47, 2009.
[12] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ACM SIGOPS Operating Systems Review*, vol. 38, no. 5. ACM, 2004, pp. 177–188.
[13] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: monitoring the buffer cache in a virtual machine environment," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 14–24.
[14] P. Lu and K. Shen, "Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache," in *Usenix Annual Technical Conference*, 2007, pp. 29–43.
[15] P. Sharma and P. Kulkarni, "Singleton: system-wide page deduplication in virtual environments," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 15–26.
[16] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel, "Transcendent Memory and Linux," in *Proceedings of the Linux Symposium*, 2009, pp. 191–200.
[17] "Frontswap," *https://www.kernel.org/doc/Documentation/vm/frontswap.txt*.
[18] J. R. Lange and P. Dinda, "Symcall: Symbiotic virtualization through vmm-to-guest upcalls," in *ACM SIGPLAN Notices*, vol. 46, no. 7. ACM, 2011, pp. 193–204.
[19] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 337–350.
[20] H. Liu, H. Jin, X. Liao, W. Deng, B. He, and C.-z. Xu, "Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1350–1363, 2015.
[21] J. H. Schopp, K. Fraser, and M. J. Silbermann, "Resizing memory with balloons and hotplug," in *Proceedings of the Linux Symposium*, vol. 2, 2006, p. 313319.
[22] M. Schwidefsky, H. Franke, R. Mansell, H. Raj, D. Osisek, and J. Choi, "Collaborative memory management in hosted linux environments," in *Proceedings of the Linux Symposium*, vol. 2, 2006.
[23] U. Deshpande, B. Wang, S. Haque, M. Hines, and K. Gopalan, "MemX: Virtualization of cluster-wide memory," in *Parallel Processing (ICPP), 2010 39th International Conference on*. IEEE, 2010, pp. 663–672.
[24] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon, "Overdriver: Handling memory overload in an oversubscribed cloud," in *ACM SIGPLAN Notices*, vol. 46, no. 7. ACM, 2011, pp. 205–216.
[25] "Perf," *https://perf.wiki.kernel.org/index.php/Tutorial*.
[26] R. P. Goldberg and R. Hassinger, "The double paging anomaly," in *Proceedings of the May 6-10, 1974, national computer conference and exposition*. ACM, 1974, pp. 195–199.
[27] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum, "Cellular disco: resource management using virtual clusters on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 229–262, 2000.
[28] P. Zhang, X. Li, R. Chu, and H. Wang, "HybridSwap: A scalable and synthetic framework for guest swapping on virtualization platform," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE, 2015, pp. 864–872.
[29] A. Omondi and S. Sedukhin, *Advances in Computer Systems Architecture: 8th Asia-Pacific Conference, ACSAC 2003, Aizu-Wakamatsu, Japan, September 23-26, 2003, Proceedings*. Springer Science & Business Media, 2003, vol. 2823.
[30] "Softlayer," *http://www.softlayer.com*.
[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.