HyperDealer: Reference-pattern-aware Instant Memory Balancing for Consolidated Virtual Machines

Woomin Hwang, Yangwoo Roh, Youngwoo Park, Ki-Woong Park, and Kyu Ho Park
Computer Engineering Research Laboratory, EE
Korea Advanced Institute of Science and Technology
Daejeon, Korea
{*wmhwang,ywroh,ywpark,woongbak*}*@core.kaist.ac.kr, kpark@ee.kaist.ac.kr*

***Abstract*—Memory contention among consolidated virtual machines (VMs) creates the need for a memory balancing operation. In an attempt to provide a prompt memory balancing mechanism, we found problems with the retardation of memory transfer by the reclamation delay. The scheduling of the VMs generates the delay, and a conflicts of two reclamation policies between the guest OS and the hypervisor deteriorates it. As a remedy to these problems, we propose HyperDealer, which selects the victim page by applying reference patterns, reclaims the pages with hypervisor-level paging, and transfers those pages with ballooning of the guest OS. Our scheme eliminates the involvement of the victim VM in memory balancing and extends the dwell time of reclaimed pages in the reclaimed state. Consequently, HyperDealer significantly reduces the time taken to transfer memory with a low overhead and enhances the value of additional memory for the recipient VM. The experimental results of our scheme show that the application performance in the recipient VM is 11% more time-efficient and has a penalty which is 50% less than previous approaches.**

***Keywords*-memory balancing; virtual machine; consolidated; VM; reference pattern**

I. Introduction

As primary bases of cloud computing, virtualization technologies have been renewed as a new service hosting platform because it offers benefits of resource efficiency, cost saving, and ease of system management. For high resource utilization, multiple virtual machines (VMs) are consolidated and share the same hardware. Such a situation, a hypervisor controls all hardware resources while providing each guest OS with the illusion of a bare machine by virtualizing those resources. Although this separation is good for its essential objective, it also creates a need for cooperative management of physical memory between the hypervisor and the guest OS. Typically, the application performance on a guest is not proportional to the size of the memory allocated to each VM. Because each application has a different memory access locality and most of the free pages are consumed by a page cache for their own caching purposes. Furthermore, unlike CPU and I/O resources, which are sharable with a limited overhead, memory is much harder to share because it is allocated on large time scales.

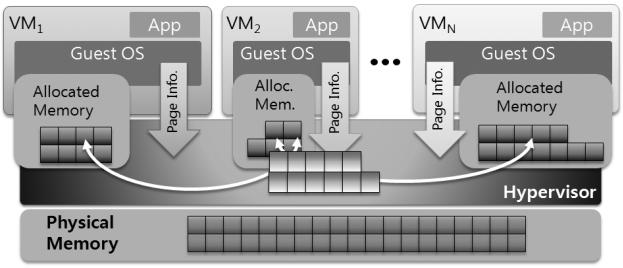


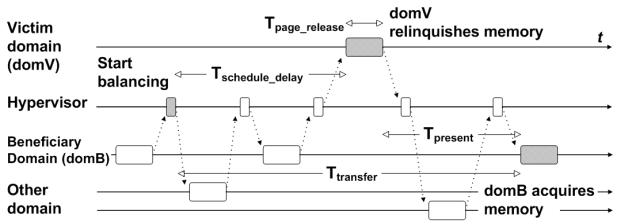
Figure 1. A Concept of memory balancing

Due to the memory pressure from a page cache's occupation, the *static memory partitioning* has a severe drawback as a result of the fixed boundary of the free memory in the system. Therefore, a conventional virtualization system enables the amount of physical memory to be extended or reduced to accommodate changes in memory requirements. Although *dynamic partitioning* [1], [2] and *hypervisor-level paging* [3], [4] provide a memory balancing scheme among multiple guests, they are still incapable of maximizing the value of additional memory for the recipient VM.

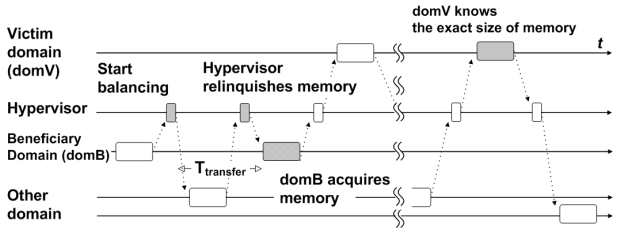
Dynamic partitioning is based on ballooning [1], which utilizes internal knowledge of the guest OS. Following the directives of the hypervisor, a balloon driver in each guest allocates memory by using the guest's own reclaiming algorithm and internal memory access knowledge. Those acquired page frames are returned to the hypervisor for allocation to a different guest. However, besides the explicit overhead of the size decision for its high complexity, we identified a fundamental limitation caused by the involvement of a victim guest OS, and that limitation drives the architecture of our instant balancing scheme.

Decision Overhead To maximize systemwide performance by balancing memory, the hypervisor should figure out which VM requires memory and which VM gets the least useful memory. If we assume that $VM \in V$ can get a maximum of M pages, then a brute force search for the new memory size takes $O(M^{|V|})$ time and consequently reduces the merit of additional memory. Furthermore, the lack of guest OS information makes

978-0-7695-4130-3/10 $26.00 © 2010 IEEE
DOI 10.1109/CLOUD.2010.70



(a) Ballooning



(b) HyperDealer

Figure 2. Timeline for memory transfer

the working set size (WSS) inaccurate, thereby making the decision of memory balancing hard. For example, a working set may be overestimated with long sequential scans [5], [6].

Balancing delay A beneficiary VM can acquire additional memory only after one or more victim VMs relinquish the page frames to be transferred. As shown in Figure 2(a), unavoidable dependency from the intervention of the victim guest OS generates a *scheduling-induced memory balancing delay*. If the hypervisor decides that the beneficiary VM (domB) requires additional memory, it requests the victim VM (domV) to relinquish page frames. DomV spends its own CPU time, $T_{page\_release}$, for memory relinquishment when domV is scheduled after time $T_{schedule\_delay}$. DomB then receives those page frames on the next schedule after $T_{present}$. Unfortunately, the greater the number of victim VMs, the longer the duration of $T_{schedule\_delay}$; the sum of the $T_{page\_release}$ values is also greater, even if the hypervisor changes the scheduling order of the victim VMs. This delay can prevent domB from acquiring memory when domB scheduled before all domV completes memory relinquishment. If the hypervisor transfers memory before all required page frames are reclaimed, the total $T_{present}$ value increases because domB receives page frames multiple times. As a result, the memory transfer speed is slower, which means there is a reduction in the performance impact of additional memory for the recipient guest.

Unlike the ballooning method, hypervisor-level paging has no such overhead from the complex decisions and delays caused by the intervention of the guest OS. However, if a lack of memory access knowledge is visible to the guest OS,

References	1	2	3	2	4	2	5		2	6		7		8		9		10		11		1		5	
Page fault	*	*	*		*		*			*		*		*		*		*		*		*		*	
Guest OS	1	2	3	2	4	2		5	2		6		7		8		9		10		11		1		5	
		1	2	3	2	4		2	5		2		6		7		8		9		10		11		1	
			1	1	3	3		4	4		5		2		6		7		8		9		10		11	
					1	1		3	3		4		5		2		6		7		8		9		10	
Page fault	*	*	*		*		*			*		*		*		*		*		*		*		*	
Hypervisor (LRU only)	1	2	3	2	4	2	1	5	2	3	6	4	7	5	8	2	9	6	10	7	11	8	1	9	5	
		1	2	3	2	4	2	2	5	2	2	6	6	7	7	8	8	9	9	10	10	11	11	1	1	
			1	1	3	3	4	4	4	2	5	2	2	6	6	7	7	8	8	9	9	10	10	11	11	
					1	1				5	5	2	2	2	6	6	6	7	7	8	8	9	9	10	10	
Page fault	*	*	*		*		*			*		*						*						*	
Hypervisor (MRU for seq., LRU for others)	1	2	3	2	4	2	1	5	2	3	6	4	7	8		9		6	10	11		1		9	5
		1	2	3	2	4	2	2	5	2	2	6	2	7		8		8	8	10		11		1	1
			1	1	3	3	4	4	4	2	5	2	5	2		7		7	7	8		10		11	11
					1	1				5	5	2	5	2		7									

9 faults

5 faults

Figure 3. Example of policy mismatch

a double paging anomaly is generated [7]. That is, a decrease in the VM's real memory size without a decrease in the size of the memory of the VM generates a mismatch in two reclamation policies. This mismatch leads to a reduction of dwell time and a significant increase in the number of page faults.

Reclamation policy mismatch The mismatch of the two LRU-based page frame reclaiming policies reduces time so that the reclaimed pages remain reclaimed and the overhead of hypervisor-level paging is consequently increased. Most general-purpose guest OSs use LRU-like page frame reclaiming algorithms. As shown in Figure 3, if the hypervisor reclaims a page frame on the basis of the LRU strategy, the algorithm in the hypervisor makes the worst choice each time. The hypervisor must therefore recover page frames in a reclaimed state within a short time. This requirement reduces the dwell time of the page in the reclaimed state so that the algorithm reclaims another page and consequently increases the number of page faults.

In this paper, we present a hybrid approach that combines the strong points of the hypervisor-level paging and the ballooning method. With this approach, which utilizes the benefits from the consideration of the application characteristics but avoids their drawbacks, the memory balancing operation can be accelerated. The approach should be accomplished with minimal modification of the guest OS. We therefore propose HyperDealer: it utilizes hypervisor-level paging on the basis of the reference pattern of the page cache of guests and deploys ballooning to supply the recipient guest with additional memory. We monitor access to the page frames that belong to each guest's page cache, detect sequentially referenced clean pages at a filestream level. Those sequentially referenced clean pages are best candidates prior to other pages. The hypervisor steals them with the MRU strategy, which means, as shown in Figure 3, that there is a reduction in the number of page faults from memory stealing. The hypervisor transfers those page frames either through ballooning or through restoration of the stolen page frames. Later, if a victim guest OS is deprived of too much memory, the hypervisor requests explicit memory borrowing from the victim guest to make the victim guest know

the exact size of the memory the victim guest itself actually uses. As shown in Figure 2(b), HyperDealer consequently reduces the memory transfer delay and the number of page faults from memory stealing, and this reduction improves the response and throughput of the memory allocation requests for a memory recipient guest.

We implemented our scheme in Xen [8]. The experimental results show that the scheme can accelerate memory balancing and significantly increase the performance of VMs that suffer from insufficient memory allocation with only a slight overhead for the initially over-allocated VMs. The remainder of this paper is organized as follows: Section II describes our scheme in detail. Section III describes our implementation issues. Section IV compares our experimental results with the results of previous approaches. Section V discusses related works. Finally, Section VI presents our conclusions.

II. Reference pattern-based instant memory balancing

In this section, we present the basic design of our reference pattern-based hybrid memory balancing architecture, which includes a new victim page selection mechanism that ensures a low overhead.

A. Basic Design

Our balancing architecture has two primary design goals. First, to adapt the bursty and time-varying memory requirements of a variant workload, the scheme should try to remove any intervention of a guest OS on the page frame reclamation. Second, it should try to select victim page frames intelligently to reduce the performance penalty of a victim guest OS. To achieve these goals in HyperDealer, we designed the hypervisor to monitor page frames that are used as a page cache for all VMs and to reclaim the page frames through reference pattern-based paging. Those page frames are then allocated to the recipient VMs either directly or through ballooning.

Figure 4 illustrates the overall architecture of HyperDealer. Each guest OS passes on information about pages within its own page cache. The information specifies which page is inserted into, evicted from, and reused within the cache. At a machinewide level, the hypervisor maintains the relative age of each page frame that belongs to the page cache of any guest OS. The instances of memory access from the VMs to those pages are intercepted by the hypervisor and used to update the management data structure. Our pattern classification mechanism, which is described in Section II-B, causes page frames to be categorized into two types:*sequential references* and *unclassified references*. Two partitions accommodate those classified page frames separately. The management of each partition reflects the characteristics of the page frames that belong to each partition. For the partition that holds the sequential references, a *sequence* consists of page frames that are sequentially

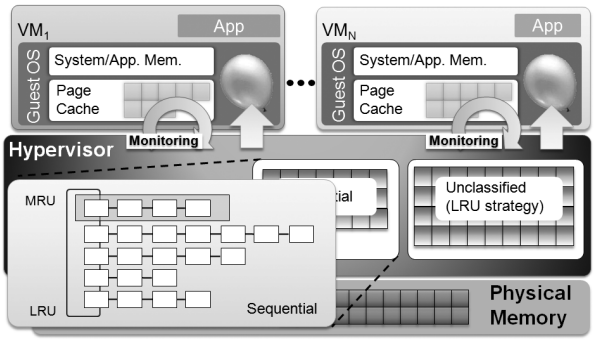


Figure 4. Overall architecture

referenced within a task-level filestream; All the sequences are arranged according to the MRU strategy. Within the partition that holds the unclassified references, we use the LRU strategy. If a page frame in the sequential partition is referenced again, the page frame is moved to the unclassified partition and the associated sequence is split into two new sequences with the same age.

The overall procedure of memory transfer is illustrated in Figure 2(b). When a page is requested for memory balance, for example, a least-valuable page frame is selected as the victim page on the basis of its reference pattern. The page frame is transparently reclaimed shortly before the hypervisor tries to schedule the beneficiary VM. Depending on the reason for the request, a different allocation mechanism is then applied to the beneficiary VM. If the reason is a paging-in event of a reclaimed page in the hypervisor, a victim page is directly allocated. However, if the request is the result of memory balancing, the hypervisor allocates the page to a guest OS through ballooning. Later, if a victim VM has stolen more memory than the threshold level, the hypervisor requests explicit memory borrowing from the VM. This type of borrowing reduces the disparity between the memory size that the victim guest OS knows about and the memory size that is actually allocated.

In our page frame management, we concentrate on clean page frames that belong to a page cache because of the volatility of the clean pages. Generally, guest operating systems, such as Linux, attempt to use any available memory for their own caching purposes. As a result, only a small amount of memory is left free; others contain content stored in permanent storage. Because the cached non-dirty content can be rebuilt by doing a read operation from its storage location, the guest can tolerate the loss of the page content. Thus, if the hypervisor tries to reclaim those pages, there is no need to swap out page content to its own swap device and dual-swapping can be avoided. We don't consider zero-pages of the guest OS as a candidate for reclamation because the counterproductive effect of doing so. In general, a guest OS must maintain a small number of free pages so that it can respond immediately to a memory allocation request.

A guest OS can handle a request for memory allocation if it uses the prepared free pages and replenishes them later. However, a hypervisor that tries to reclaim the free pages for the benefit of having no swap-out or reload offers the guests false information about memory availability. Finally, the hypervisor delays frequent instance of small memory allocation. Furthermore, it blocks the execution of the process until the guest kernel or the hypervisor uses its own reclaiming algorithm to make up for a deficiency of free pages; this step propagates delays for all applications in the same victim domain.

Hence, we restrict our monitoring to page frames that are used as a page cache and choose clean pages as a victim. This process requires no swap storage area for the hypervisor, no additional management cost, and no data flush overhead. For the policy on the usage of zero-pages, we use the same explicit management policy as the one proposed in [2].

B. Reference pattern-based victim selection

The long dwell time of a reclaimed page staying with no access is an important factor for ensuring the victim guest OS has a low penalty. The victim guest OS gets a low penalty when a page stays in the reclaimed state for a long period. However, as mentioned in Section I, the short dwell time from the reaccess under the LRU strategy repeatedly provokes the reclamation of another page that is accessed again.

To ensure a page stays in the reclaimed state for longer, the hypervisor selects victim pages on the basis of each page's reference pattern. To detect reference patterns and associated page frames, we distinguish the tasks and then detect the filestream-level sequentiality on the basis of the task. We distinguish each task by recognizing changes in the page table used in [9]. We intercept read system calls issued from a distinguished task and associate them with the touched page frames. The hypervisor then collects the associated 6-tuple information, which consists of a task descriptor, a file descriptor, a start offset, an end offset, a loop period, and the address of associated page frames. A reference is categorized as a sequential reference after the same task makes a given number of consecutive references. The touched pages are then considered to be a sequential reference. Sequential references have no loop period, whereas loop references have a loop period represented by a number of page cache references between touches. Information about page frames that contain touched data is required because the memory transfer unit handles memory at the level of the page unit.

Figure 5 shows an example of sequential and loop references. In the figure, the page frames that contain a file with fileID 4 referenced by a task with taskID 3 constitute a sequential reference because they have infinite loop period. The page frames that contain files with fileID 4 and fileID 5

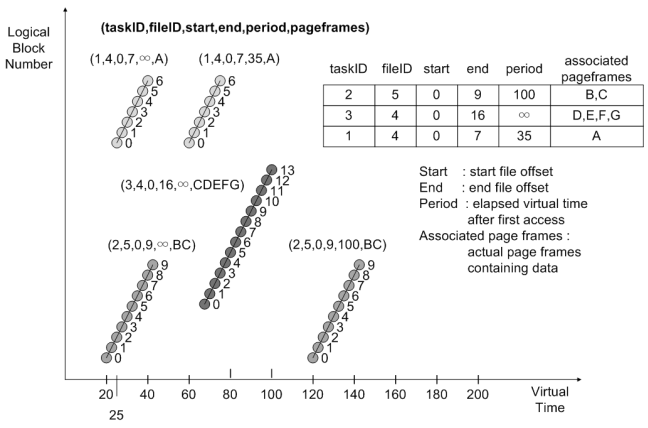


Figure 5. Examples of sequential and loop pattern detection

referenced by task 1 and task 2 are loop references with periods of 35 and 100, respectively. A page frame that contains data about more than two reference patterns belongs to the sequence with the last reference.

Victim page selection is based on the reference pattern of the candidate page frames. Sequentially referenced pages are never re-referenced. Therefore, the MRU selection strategy is used for a longer dwell time in a reclaimed state and consequently reduces the delay from a policy mismatch. The hypervisor selects victim page frames from within the partition that holds the sequential references. The page frames of the partition that holds unclassified references are selected with the LRU order after all the sequentially referenced pages are reclaimed.

C. Reverse-ordered ballooning

As described in Section I, the main reason for the balancing delay is the ordered execution of inflation and subsequent deflation of balloon drivers. We remove the functional dependency by replacing the inflation of the balloon with the reclamation in the hypervisor. This replacement reverses the execution order of the ballooning. In our design, the inflation of the balloon in a victim guest OS is performed either in the idle time of the victim VM or when the stolen page frames of the guest OS exceed the threshold. However, the deflation occurs whenever the hypervisor balances memory in such a way that inflation follows a number of deflations. Therefore, reverse-ordered ballooning creates a dependency between the reclamation in the hypervisor and the deflation in a beneficiary VM. This process creates freedom from the schedule of the victim VM and reduces the memory transfer delay.

III. Implementation

We implemented a prototype system for Xen 3.3.1 and Linux 2.6.18. All components were implemented in a Xen hypervisor, with some hypercall invocations implemented in

the Linux kernel. The hardware platform is x86-64 architecture, where the guest OS runs in privilege level 3. The following subsections describe the specific implementation of our scheme. First, we describe the implementation in relation to OS information exploitation, such as tracking page cache access. We then use the event correlation scheme to describe the data read.

A. Monitoring of page cache access

The task of monitoring access to the page frames of all the VM's page cache is a combination of page cache distinguishment and access detection to those page frames. To distinguish page frames belonged to a page cache, we use hypercall explicitly though they are available without any modification of the guest OS [10]. When a page is inserted into or evicted from a page cache, a hypercall is called to inform the hypervisor of a change in the page's position. For detecting memory access event which are usually transparent to the hypervisor, we borrow minor page fault from [2], [15], [16]. With minor page fault, the hypervisor traps target page frame accesses intentionally by granting highest access privilege to the pages of concern. As access to the page table entry (PTE) requires highest privilege, non-privileged access to the target page is trapped to the hypervisor. Therefore, the hypervisor can recognize accesses to the concerned page frames.

B. Page read inference with event correlation

To detect filestream-level memory access pattern in the hypervisor, we correlate file access information with physical memory address.At a filestream-level, memory access pattern is a consecutive series of the read or write requests to the page cache from a task. Any file information is inside the guest OS, a read or write request is performed by a corresponding system call. However, the series give no information about the machine address of the touched page frames where the data is located. In addition, the information is not exploited outside the OS; the information reveals which page frame contains the file data of a specific offset. Therefore, the reference to the cached file data contains no information about the physical page frames because it works on the basis of a virtual address space of the guest OS.

To resolve this problem, we intercept some system calls and correlate event that connects the reference to the data and the touched page frame information. A read system call includes required information as arguments such as a file descriptor, an offset in the file, and the length of the target data. The hypervisor distinguishes each file by the file descriptor because of its uniqueness in a task; it also detects consecutive instances of access by the offset and the length of read data in the specified file. The event correlation scheme, which is described in Figure 6, associates page frames touched by the read system call with the sequence in which the read system call belongs.

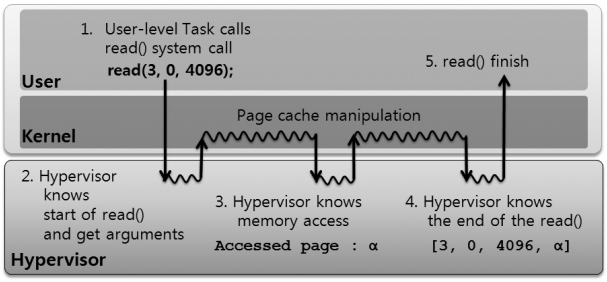


Figure 6. Intercepting page cache read based on file and task-awareness

The start of the read system call is initially intercepted by the hypervisor to get arguments of the read request. Access to the page frame in the page cache is trapped by the hypervisor, and the hypervisor consequently knows where the data read of the current task are located in the page frame. This behavior is possible because the data read usually blocks the operation, which means that the specified task's next event are stalled by the read request. The hypervisor is notified whenever a read operation is completed. By chaining consecutive events for the same task, the hypervisor can infer the relation between the file-level access and its physical memory location. In the original Linux, system calls are made by triggering the *int 0x80* interrupt line. In the original Xen hypervisor, a system call is implemented as a direct trap for performance improvement. In our Linux implementation on an x86-64 system, a system call is trapped by the hypervisor because the guest OS runs on privilege level 3. We simply add argument acquisition codes. For the notification of the completion of a read operation, we add hypercall.

IV. Performance Evaluation

In this section, we describe the experimental evaluation of our prototype implementation. Our scheme is implemented on an Intel server with 4 cores (two Intel Xeon 3.40 GHz Dual-Core processors) and an 8GB memory. By default, all the VMs are initially allocated 256MB of memory and configured with 512MB as their highest possible memory allocation. The beneficiary VM is executed first to minimize the value of $T_{present}$ whenever the page frames requested to be transferred are ready. For the dynamic partitioning, that is ballooning, we also change the scheduling order of the victim VM with an interrupt to make the requested page frames relinquish immediately, thereby minimizing the value of $T_{schedule\_delay}$.

A. Overhead of the Access Monitoring

The runtime overhead of our scheme mainly comes from the minor page faults of page frames in the page cache, the detection of reference patterns and the reloading of data from permanent storage. To evaluate the runtime overhead of our scheme, we measure the execution time of the Tiobench

	Normalized Throughput	
	Non-monitored	HyperDealer
Tiobench (seq. read)	1	0.9908
Tiobench (random read)	1	0.9980
Dbench	1	0.9834

Table I
RUNTIME MONITORING OVERHEAD

	Page acquisition time (μs)
Ballooning	8.345
HyperDealer	0.600

Table II
ELAPSED TIME OF A PAGE RELEASE TO TRANSFER

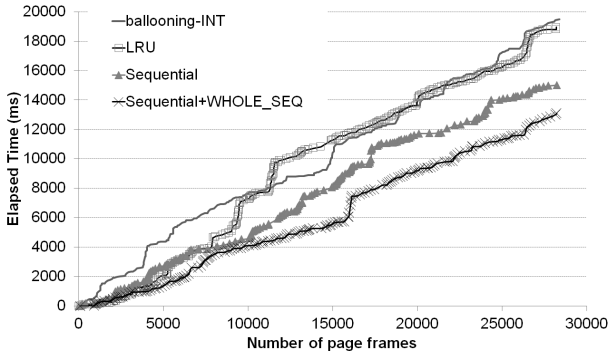


Figure 7. Performance effect of memory transfer speed

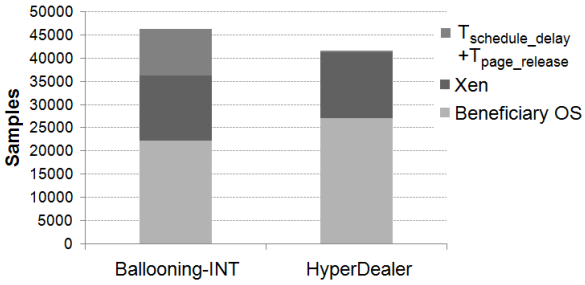


Figure 8. Decomposed execution time of memory transfer

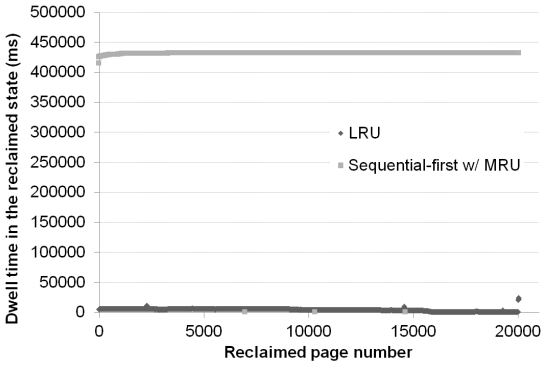


Figure 9. Dwell time in the reclaimed state according to the victim selection strategy

(version 0.3.3) [11] benchmark with our scheme turned on but without any memory reallocation. As shown in Table I, the average monitoring overhead is negligible.

B. Effect of the memory transfer speed

To evaluate the effect of memory transfer speed to the application performance, we first assign memory allocation job to a VM, while other 3 VMs perform sequential and random read in Tiobench benchmark. Then experiments are performed to measure memory transfer speed. We use each VM's swap usage as a criterion of memory balancing.

Table II shows the result of average reclamation time of a page from a victim VM using *inflate* of ballooning and our scheme. Ballooning takes much time compared with HyperDealer because it requires victim domain's involvement consuming $T_{page\_release}$ to acquire free memory. Therefore, it imposes restrictions on elaborate memory balancing for its delay proportional to the number of balancing trials.

Figure 7 shows the elapsed time of memory allocation in beneficiary VM while the numbers of page frames are transferred as a result of memory balancing. It presents the effect of memory transfer speed and victim selection policy on the application performance. Faster transfer of page frames enlarge the amount of free memory when the guest OS requires them. As a result, it reduces the number of time-consuming memory reclamation trials if no free memory exists. Therefore, the guest OS can respond faster to the memory allocation request of user application. In addition, if we reclaim and restore a sequence instead of a page, we can get more accelerated result indicated as *Sequential+WHOLE_SEQ*. Because a sequence has more page frames, and data can be easily recovered with less number of long access to the permanent storage.

We can see how much the victim guest's involvement affects to the memory transfer speed in Figure 8. The graph shows a ratio among the delay caused by $T_{page\_release}$ and minimized $T_{schedule\_delay}$ from victim VM, execution time of beneficiary VM and the hypervisor. The delay makes the beneficiary VM acquire memory slowly; hence the VM consumes more CPU time to do its own job, including self-reclamation inside the guest OS. HyperDealer has little dependency on the delay; however, as a result of the fast memory transfer, the OS of the beneficiary VM takes longer to handle additional memory, particularly with regard to managing and cleaning new pages.

Sacrificing sequential references prior to sacrificing unclassified pages affects performance. To see how this effect compares with the ordinary LRU strategy, we compare the dwell time of a reclaimed page in a reclaimed state according to the reclamation method. The result of the comparison are shown in Figure 9. All the victim VMs execute the

Tiobench benchmark. Maintaining a long dwell time is important for a reclaimed page in a reclaimed state. The long dwell time means that the reclaimed page cause no unnecessary page fault or data reload from the permanent storage-both those factors degrade performance. The results confirm that the victim selection policy for the hypervisor-level paging is effective. However, because of the policy mismatch, a page that is reclaimed by the LRU victim selection policy in the hypervisor is soon reclaimed by the guest OS. The quick recovery of the reclaimed pages results in a short dwell time. On the other hand, HyperDealer selects sequentially referenced pages first and reclaims those pages prior to others on the basis of the MRU strategy. Although the inaccuracy of the sequential reference detection scheme generates a short dwell time, a near maximum dwell time of pages reclaimed by HyperDealer is much longer that of pages reclaimed by the LRU policy.

This measurement is taken in a situation where reclamation in a page cache of the guest OS is actively performed by the Tiobench benchmark. If the workload fluctuates, the difference in the dwell time from the policy mismatch is increased. The large difference lowers the overhead of the victim VM as a result of the page fault of the reclaimed pages.

V. Related Work

Contemporary studies present solutions that enhance machine-wide performance in the VM-consolidated environment.

In Cellular Disco [3], each cell can borrow memory from other cells that rich in free memory. However, the cell is defined as a fault containment unit in a cluster composed of several physical machines, and the authors propose resource balancing policy among those cells. Our proposed memory reallocation policy is for multiple VMs within a cell, especially single physical machines. However, they did not address this approach.

Waldspurger [1] introduces an important memory transfer technique called *memory ballooning*, which is used in the VMWare ESX Server. Each guest OS uses a special kernel driver, a *balloon driver*, which *inflates* when memory is allocated from the kernel and is subsequently released to the hypervisor. Similar to *inflate*, the driver also *deflates* when memory is requested from the hypervisor and subsequently released to the guest kernel. The balloon driver gives the guest OS the illusion that the driver occupies memory when the memory is under the control of another VM. Because of this behavior, the memory allocation of a VM can be changed dynamically. However, the inflation and deflation of a balloon driver sacrifices the CPU time of the victim guest OS. Moreover, the inflation in the victim guest OS must precede the deflation in the beneficiary guest OS. As a result, there is a delay in the memory transfer.

In Hypervisor Exclusive Cache [12], the hypervisor manages each VM's memory portion as a second level exclusive cache and tracks its LRU-based miss ratio curve. By changing the size of each VM's hypervisor-level cache, it achieves the same effect as changing the VM's memory allocation. Newly allocated memory is either inserted into the VM through a balloon driver or used as an exclusive cache in the hypervisor. Because the tracking of the miss ration curve starts when the memory size needs to be mediated and the hypervisor requires some additional memory references in the exclusive cache, there may be a long lag between the time for a change and the time for completing the balance. Furthermore, a VM that executes a memory-intensive but non-I/O-intensive job cannot measure the memory requirement correctly.

Transcendent Memory (Tmem) [13] also proposed a hypervisor-level second-chance page cache for each guest domain in a physical machine. With Tmem, a hypervisor collects fallow memory and wasted guest memory and then uses it as a per-VM private page cache. The authors try to balance memory by implicitly mediating the size of a private page cache with a global LRU queue. The scheme is effective for balancing I/O-intensive applications with a large size page cache, but each VM must maintain a sufficiently large private cache for a correct decision. A page copy operation from every eviction of the cache content generates some overhead, and every data read should check the caches on both layers. Furthermore, because Tmem is not directly accessible and a page copy operation is required, it cannot respond to the memory need of user-level applications.

Memory Balancer (MEB) and Collaborative Memory Management (CMM) are highly relevant for our research. MEB [2] decides the proper memory size of each VM through WSS estimation. The victim VMs then inflate the balloon driver to release memory, and the beneficiary VMs subsequently deflate the balloon driver to make the guest OS control the acquired memory. MEB utilizes its own memory reclaiming policy with regard to the guest OS's selection of victim pages. However, the policy also causes a swap out or data flush of dirty pages to the corresponding storage location in accordance with the guest OS's policy. Furthermore, the balloon driver's inflation and subsequent deflation cause a scheduling-induced balancing delay; hence, the ballooning is unsuitable for applications with dynamically changing memory requirements. If multiple VMs exist, the increased delay prolongs the memory contention, and consequently depreciates the longer, thus depreciate the additional memory in the beneficiary VM. Feedback-directed ballooning [14] also mentioned a similar solution with the same weak points.

CMM [4] attempts to address the issue of double paging [7] and the overhead of moving memory by ballooning in the hosted Linux environment of System z. In CMM, the guest VM gives the hypervisor hints that help page frames

to be selected and reclaimed in a more intelligent way. The hints contain each page frame's state about which pages are being used and which pages are reclaimable with little penalty. The hypervisor can then select the victim pages with the internal knowledge of the guest OS. On the other hand, monitoring of all memory access in hardware without any additional CPU protection privilege, such as Intel CPUs, significantly degrades performance [15], [16]. Our scheme limits the overhead by monitoring only page frames of a page cache that belongs to all VMs without any hardware assistance. The policy mismatch problem is not addressed in this paper; it will require a large number of modifications of the guest OS.

For the VM scheduling-related issue, Govindan et al. [17] tried to solve the performance degradation caused by scheduling-induced delays in network communication. Each application spends a large amount of time waiting for a chance to communicate because it must wait until the CPU scheduler chooses it among the competing co-located applications. As a result, the dependency on the scheduling order causes the delay. We found that a similar delay in the memory reallocation procedure makes the beneficiary miss the opportunity to maximize the effect of donated memory.

The need for reference pattern detection is addressed in [18]–[21] for victim selection of the page reclamation in OS. The authors detect and classify the sequential reference pattern and the loop reference pattern from instances where the disk cache is accessed and then select a victim page for reclamation on the basis of each pattern's marginal gain. Although this paper uses a detection scheme for detecting reference patterns, we focus on the performance effect of memory balancing among consolidated VMs, especially with regard to the reduction in the victim VM's overhead.

VI. Conclusion

As virtualization technology consolidates more guest OSs into single hardware units, resource management has become a key issue. Although memory balancing can reduce memory contention among virtual machines, slow memory balancing degrades the effectiveness of additional memory in the suffering virtual machines. In this paper, we provide a full-fledged non-obstructive memory transfer scheme enhanced with the reference pattern-based victim selection and hypervisor-level reclamation. We propose HyperDealer, which makes balancing operation free from the involvement of a victim VM and extends the dwell time of reclaimed pages in the reclaimed state. Consequently, HyperDealer significantly accelerates the balancing operation with a low overhead, thereby increasing the effect of additional memory on the beneficiary VM. The experimental results show that our scheme outperforms other approaches by 11% in terms of time efficiency and by 50% in terms of mitigating the overhead. In future works, we plan to adapt our scheme to the memory requirements of the I/O workload.

References

[1] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *Proceedings of Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, Dec. 2002.

[2] W. Zhao and Z. Wang, "Dynamic memory balancing for virtual machines," in *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2009, pp. 21–30.

[3] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum, "Cellular disco: resource management using virtual clusters on shared-memory multiprocessors," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 154–169, 1999.

[4] M. Schwidefsky, H. Franke, R. Mansell, D. Osisek, H. Raj, and J. Choi, "Collaborative Memory Management in Hosted Linux Systems," in *Proceedings of the 2006 Ottawa Linux Symposium*, 2006.

[5] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *VLDB '1985: Proceedings of the 11th international conference on Very Large Data Bases*, 1985, pp. 127–141.

[6] G. M. Sacco and M. Schkolnick, "Buffer management in relational database systems," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 473–498, 1986.

[7] R. P. Goldberg and R. Hassinger, "The double paging anomaly," in *AFIPS '74: Proceedings of the national computer conference and exposition*. ACM, 1974, pp. 195–199.

[8] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM, 2003, pp. 164–177.

[9] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: tracking processes in a virtual machine environment," in *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2006.

[10] ——, "Geiger: monitoring the buffer cache in a virtual machine environment," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 14–24, 2006.

[11] M. Kuoppala, "Tiobench - Threaded I/O bench for Linux," 2002.

[12] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache," in *ATC'07: Proceedings of the 2007 USENIX Annual Technical Conference*. USENIX Association, 2007, pp. 1–15.

[13] D. Magenheimer, "Transcendent Memory on Xen," Xen Summit, 2009.

[14] ——, "Memory Overcommit... without the commitment," Xen Summit, June 2008.

[15] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," *SIGPLAN Not.*, vol. 39, no. 11, pp. 177–188, 2004.

[16] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "CRAMM: Virtual Memory Support for Garbage-Collected Applications," in *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[17] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co.: communication-aware CPU scheduling for consolidated xen-based hosting platforms," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007, pp. 126–136.

[18] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "Towards application/file-level characterization of block references: a case for fine-grained buffer management," in *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM, 2000, pp. 286–295.

[19] J. Choi, S. H. Noh, S. L. Min, E.-Y. Ha, and Y. Cho, "Design, implementation, and performance evaluation of a detection-based adaptive block replacement scheme," *IEEE Transactions on Computers*, vol. 51, pp. 793–800, 2002.

[20] F. Zhou, R. von Behren, and E. Brewer, "AMP: Program Context Specific Buffer Caching," in *ATEC '05: Proceedings of the 2005 USENIX Annual Technical Conference*, 2005, pp. 371–374.

[21] Y. Zhu and H. Jiang, "Race: A robust adaptive caching strategy for buffer cache," *Computers, IEEE Transactions on*, vol. 57, no. 1, pp. 25–40, Jan. 2008.