Reference Pattern-Aware Instant Memory Balancing for Consolidated Virtual Machines on Manycores

Woomin Hwang, *Student Member, IEEE*, Ki-Woong Park, *Member, IEEE*, and Kyu Ho Park, *Member, IEEE*

Abstract—Memory contention among consolidated VMs on the same hardware has created the need for repetitive memory balancing operations. In an attempt to provide a prompt memory balancing mechanism, we found problems with the retardation of memory reallocation by the reclamation delay. The scheduling of the VMs and their VCPUs generates the delay, the dirtiness of the candidate pages for balancing makes the delay fluctuated, and a conflict of two reclamation policies between the guest OS and the hypervisor deteriorates the application performance. As a remedy to these problems, we propose HyperDealer2 (HD2), which selects the victim pages based on the reference patterns of clean pages, reclaims them with hypervisor-level paging, and reallocates those pages with explicit ballooning of the recipient guest OS. HD2 eliminates the involvement of victim VMs in memory reclamation and extends the dwell time of reclaimed pages in the reclaimed state. Consequently, HD2 significantly reduces the time taken to reallocate memory with a low overhead and enhances the value of additional memory for the recipient VMs. The experimental results of HD2 show that the execution time of memory-intensive applications in the recipient VM is reduced by up to 50 percent in spite of less than 2 percent performance penalty.

Index Terms—Memory balancing, virtualization, consolidated, VM, reference pattern

1 Introduction

As the primary bases of cloud computing, virtualization technologies have played a key role as a new hosting platform. They offer benefits of efficient resource management, cost saving, and ease of system management. Virtual machine (VM) consolidation is a core scheme that increases resource utilization by sharing the same hardware. In such a situation, a hypervisor controls all hardware resources while providing each guest OS with the illusion of a bare machine by virtualizing them. However, misses in the resource management of a VM can degrade other VM's performance running with it. Among many shared resources, memory is one of the most sensitive resources in resource management for the VM-consolidated environment. Its available size and access speed greatly affects the application performance on each VM. The ideal memory size is one that makes the working set of applications fit with the VM's main memory allocation. If the system underestimates its memory requirements, the insufficient size of memory significantly degrades the system performance. In contrast, application performance is not proportional to the size of the memory, even with enough memory allocated to the VM.

- *W. Hwang and K.H. Park are with the Department of Electrical Engineering, KAIST, Daejeon, Korea. E-mail: wmhwang@core.kaist.ac.kr, kpark@kaist.ac.kr.*
- *K.-W. Park is with the Computer Hacking and Information Security Department, Daejeon University, Daejeon, Korea. E-mail: woongbak@dju.kr.*

Manuscript received 28 Oct. 2013; revised 26 June 2014; accepted 12 July 2014. Date of publication 17 July 2014; date of current version 5 June 2015. Recommended for acceptance by X. Sun.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2340854

In the consolidated VM environment, balancing memory among the VMs is a key challenge in maximizing system-wide performance. At any moment, each VM has its own amount of memory that is more than actually required to adapt various memory needs. Generally, not all of the memory is being actively used. It is because each application has different working set and they act with different access localities. However, unlike other resources such as CPU and I/O components, it is harder to do time-sharing among VMs. A guest OS has to preserve dynamically changing contents either on memory or in permanent storage. Furthermore, the guest OS can see only virtualized resources and has no physical memory information while the hypervisor can only see allocated resource information because detailed memory usage information is inside the guest OS. As a consequence, there is a need for cooperative memory balancing among the hypervisor and the guest OSes.

Due to the memory pressure from the page cache's occupation and dynamic memory requirements, static memory partitioning has a severe drawback as a result of the fixed boundary of the free memory in the system. Therefore, conventional virtualization systems enable 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 memory balancing schemes 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 from its own free memory pool. If there is not enough free memory, the driver utilizes the guest's own

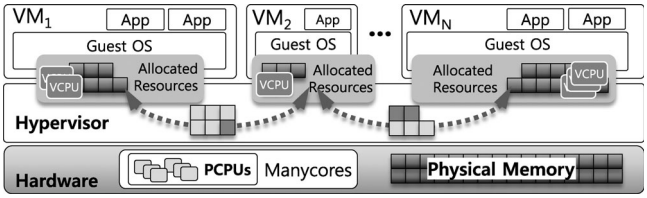


Fig. 1. MN-MATE [5] computing platform with manycores running consolidated VMs with memory balancing scheme.

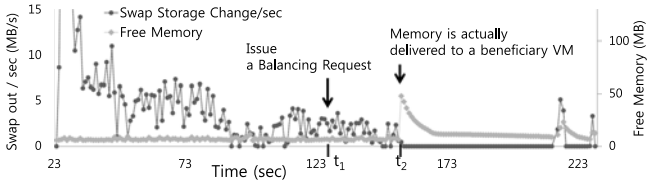


Fig. 2. An example of balloon-based memory reallocation delay with 16 VMs. Late arrival of additional memory dissipates chances to reduce swap overheads.

reclaiming algorithm and internal memory access knowledge. Those reclaimed page frames are returned to the hypervisor for allocation to different guests.

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 its VCPU. That limitation drives the architecture of our instant balancing scheme. A beneficiary VM can acquire additional memory only after one or more victim VMs relinquish their page frames to be reallocated. Such dependency delays balloon-based memory balancing operations, thus depreciating the reallocated memory in the recipient VM. Fig. 2 shows a sample balloon-based memory reallocation with 16 VCPU-busy VMs. Additional memory arrives late to the beneficiary VM due to the memory reallocation delay. This makes the beneficiary VM lose chances to reduce unnecessary swap overheads generated during $t_2 - t_1$. Unlike the ballooning, hypervisor-level paging has no such delays caused from the dependency. Instead, there exists a mismatch of the two LRU-based page frame reclamation policies on both the hypervisor and the guest OS. The mismatch makes the hypervisor choose the worst candidate whenever selecting page frames to reallocate.

In this paper, we present HyperDealer2 (HD2), a hybrid approach that combines the strong points of the hypervisor-level paging and the ballooning method. We accelerate the memory balancing operation based on the application's characteristics while avoiding drawbacks. Our architecture adapts a hybrid approach of two schemes: 1) a hypervisor-level paging on the basis of the reference pattern to the page cache in guest OSes and 2) a ballooning to supply recipient guests with additional free memory. We based our scheme in monitoring access to the page frames that belong to the page cache of all guest OSes. The hypervisor detects memory reference patterns on the monitored clean pages at a filestream level. Those sequentially referenced clean pages are the best candidates for reallocation prior to other pages. During the memory reallocation procedure, the hypervisor steals page frames with the MRU strategy to avoid policy mismatch, which will be described in Section 3. It reduces the number of page faults from memory stealing. The hypervisor reallocates 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 page frames, the hypervisor requests explicit memory borrowing from the guest. The borrowing makes the victim guest OS know the exact size of the physical memory it actually uses. HD2 reduces the memory reallocation delay and the number of page faults from memory stealing. This reduction consequently improves the response and throughput of the memory allocation requests for a memory recipient guest.

This study is an extension of our previous work [6], in which we focused on finding problems of memory balancing with a small number of cores. Our objective in this study, however, is to devise a full-fledged non-obstructive memory reallocation scheme and integrate the overall components transparently in MN-MATE [5], our resource management system for manycores and a hybrid memory hierarchy, as shown in Fig. 1.

We implemented our scheme in Xen [7]. The experiment results showed 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: In Section 2, we present some background. Section 3 describes several motivations behind our work. Section 4 describes our scheme in detail. Section 5 describes our implementation issues. Section 6 compares our experiment results with the results of previous approaches. Section 7 discusses related works. We conclude this paper in Section 8.

2 Background

In this section, we introduce a hypervisor briefly, then we present the memory balloon, which is used for explicit memory reallocation.

2.1 Hypervisor

Hypervisor is a software that allows multiple guest OSes to run on the same hardware host concurrently. With paravirtualization approach, it can minimize overhead from virtualization through a *hypercall*, a software trap to a hypervisor. With full virtualization, it can work with the support of CPUs including Intel VT-x [8] or AMD-V [9] extensions. Guest domains are initialized with the preconfigured size of main memory. The hypervisor, however, can reallocate allocated memory to each VM at runtime.

2.2 Memory Balloon

In the consolidated VMs environment, a guest OS can acquire some memory to enhance application performance running on it and it can also donate some redundant memory to other memory-thirsty guest OSes. After the movement, the VM should perform as if it has been configured with the changed memory size. Waldspurger [1] introduced an important memory reclamation, transfer of ownership, and supply a technique called *memory ballooning* among multiple VMs in the VMware ESX Server. Fig. 3 illustrates this procedure.

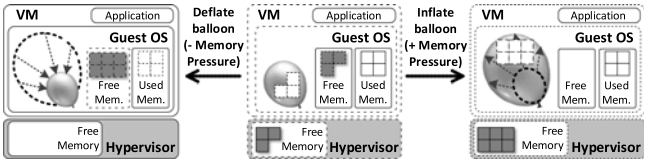


Fig. 3. Basic operations of the balloon driver [1].

Each guest OS uses a special kernel module, the *balloon*, as a device driver or kernel service. When the hypervisor tries to reduce the size of allocated physical memory to a guest OS, the balloon *inflates*. It increases memory pressure in the guest OS by requesting free memory. Such pressure then triggers its own reclamation algorithm in the memory management system. The balloon pins unused memory inside the guest OS and lets the hypervisor use them. This works when the available memory size is decreased. Similar to inflation, the balloon *deflates* to retrieve memory and to give the memory back to a guest OS when the hypervisor tries to increase the size of memory for the VM. Consequently, the balloon gives the guest OS an illusion that the driver occupies memory, even though the physical memory is under the control of other VM.

3 Motivation

Memory balancing is a consecutive procedure of finding the least-valuable pages from VMs not being pressed for memory, making free memory by sacrificing found pages, and reallocating them to memory-thirsty VMs. It is an efficient method of enhancing system performance by increasing memory utilization. Though there are two representative balancing approaches, the ballooning and the hypervisor-level paging, they have several drawbacks. Besides the high complexity of the balancing decision, ballooning induces memory reallocation delay among the VMs from the intervention of guest OSes. With hypervisor-level paging, the hypervisor undergoes delays due to the mismatch between reclamation policies on each level. We will explain four drawbacks in more detail. We also abbreviated victim VM to domV and beneficiary VM to domB, respectively.

3.1 Decision Overhead from Information Asymmetry

To balance memory among consolidated VMs, we have to select victim VMs (domVs), beneficiary VMs (domBs), and the size of the memory to move, as well as the page frames to be reallocated. The hypervisor performs this based on information about each candidate page frame's memory value, a relative need for staying its data on memory. However, information asymmetry about memory value makes those decisions more complex and less accurate. Each guest OS has no information about the relative values of its own memory compared with other VMs. The hypervisor has no knowledge about OS-internal memory usage though it can see all memory on the hardware. This makes the hypervisor misjudge the memory requirements of each guest OS. For example, the hypervisor may overestimate the working set size of each VM with long sequential scans [10], [11].

Even if each guest OS exports the required knowledge to the hypervisor, it takes too much time to select the proper

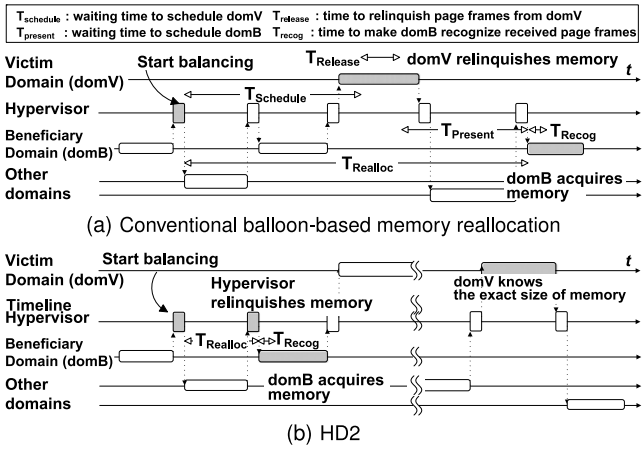


Fig. 4. Example timelines for memory reallocation.

memory size of each VM. For example, assume that V is a collection of VMs running on the same hardware. If a domB $\in V$ can get a maximum of M pages, then a brute force search for the new memory size takes $O(M^{|V|})$ time for each VM. The calculation overhead delays the decision of domVs and the amount of memory donation.

Periodic and incremental memory adaptation could be an alternative solution for the decision overhead. The adaptation iteratively reallocates a fixed amount of memory among VMs based on the monitored needs until the results meet balance goals. Such approximation method may make the decision simpler. However, it does not relieve performance degradation from the internal memory reclamation of the guest OS. If the memory requirement bursts within a short time, added memory is not enough to cover immediate memory requirements until the iteration finishes. The longer the period the hypervisor reallocates memory iteratively, the more the internal memory reclamation procedure runs. It reduces the number of page frames to be reallocated despite the need for more memory and the performance degradation from the internal reclamation of the VM.

3.2 Memory Reallocation Delay among VMs

The time delay to deliver additional memory among the participating VMs attenuates performance benefits from additional free memory. The victim guest OS should explicitly donate free memory to the hypervisor by wasting own timeslices. The domB cannot acquire memory during its scheduled time if it is scheduled earlier than the donation of the victim VM. These create a scheduling dependency among memory donor VMs and recipient VMs, which generates memory reallocation delay, $T_{Realloc}$.

Fig. 4a illustrates a one-way memory reallocation for memory balancing based on ballooning [1], [2]. Assume that three VMs are running on the same hardware and the hypervisor tries to reallocate memory from a domV to a domB. The elapsed time of a memory reallocation, $T_{Realloc}$, can be decomposed into four periods: $T_{Schedule}$, $T_{Release}$, $T_{Present}$, and T_{Recog}. After the decision of memory reallocation, a domV consumes own time quantum, $T_{Release}$, to relinquish the allotted size of memory to the hypervisor after $T_{Schedule}$ has passed until scheduled. The hypervisor then allocates them to the domB when it is

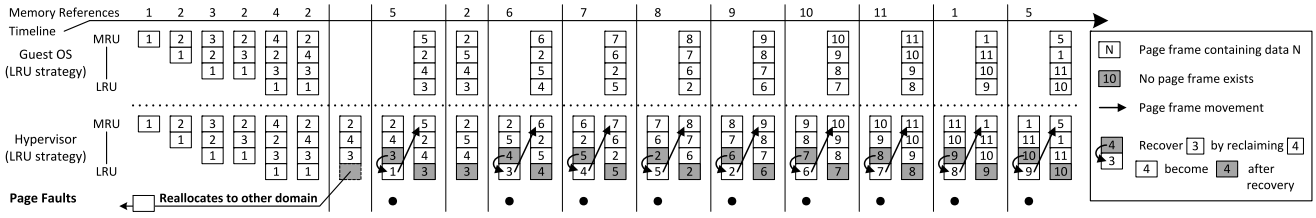


Fig. 5. An example of page frame reclamation policy mismatch between the hypervisor and a guest OS. After transfer of an LRU-positioned page frame, every reclamation trial of the guest OS generates a page fault. To handle the page fault, the hypervisor should recover the page frame by reclaiming the next LRU-positioned page frame. It is worst choice of the hypervisor because the reclaimed page is the next candidate for reclamation of the guest OS.

scheduled for the first time since the hypervisor took control of the donated memory, which takes $T_{Present}$. Finally, the domB recognizes the memory at the cost of time T_{Recog}, which is negligible.

Writes to permanent storage during reclamation increases $T_{Release}$. Usually, a guest OS utilizes most of the free page frames as a page cache for an indefinite period of time until the OS needs more free memory [12]. If a domV has not enough free pages to reallocate, the hypervisor cannot took control of the designated amount of free memory until the guest finishes its own page frame reclamation procedure and *inflates* own balloon. However, flushing dirty pages in the page cache generates writes to permanent storage. The swapping out of pages also delays the reclamation.

There are three factors that significantly increase $T_{Schedule}$: the number of busy VCPUs, the number of domVs, and the number of busy tasks in a domV.

First, a higher number of busy VCPUs delays the allocation of PCPUs to domVs. From the fairness-oriented characteristic of the hypervisor scheduler, more active VCPUs of all VMs than the physical cores generate a longer delay for a VCPU to be scheduled than a PCPU. Because the balloon-based memory donation sacrifices its own time, it delays the execution of the reclamation procedure by any VCPU of a domV. Second, the number of domVs affects the temporal distribution of donated memory arrival to the hypervisor. On securing free memory to reallocate, each domV transfers them to the hypervisor. If some domVs are scheduled later and transfer the memory to the hypervisor, the memory will arrive sporadically. Therefore, domB receives memory fragments by consuming multiple T_{Recog}. The late arrival of additional memory makes the domB initiate its own reclamation procedure, which delays the application performance. Third, more busy tasks in a domV also increase the $T_{Release}$ and $T_{Schedule}$. In a domV, memory donation operation has at most equal priority than other normal tasks. The OS scheduler may allocate VCPU timeslices after other active tasks run long enough. If other tasks run out of VCPU timeslices, the domV has to wait until the next VCPU allocation.

3.3 Page Frame Reclamation Policy Mismatch between the Hypervisor and a Guest OS

Hypervisor-level paging [1], [13], [14] is a reclamation mechanism of page frames based on paging by the hypervisor. After swapping out the data from a page frame, the hypervisor reloads the data into memory only if a page fault occurs, which is the same operation as paging in the native OS. It provides no intervention by the owner guest OS to make free page frames for memory balancing. However, similar paging policies operating each at the hypervisor and the guest OS generate a mismatch. The mismatch is known as a double paging anomaly [15] in a system running a paged OS under a paged hypervisor. Most guest OSes use an LRU-like policy for their own page frame reclamation. If the hypervisor uses the LRU-like victim selection strategy, the reclaimed pages are likely located in the LRU position of the guest OS management list. Those reclaimed pages may face an increased risk of access by the guest OS within a short time, which generates additional overheads.

Fig. 5 illustrates an example of the memory reclamation policy mismatch based on Mattson's stack distance algorithm [16]. In this example, we assume that the hypervisor reallocates a page frame at the LRU position after the guest OS fills the pages with data, whose variants are widely used in many OSes [17]. If a guest OS loads data 5 to memory, it tries to reclaim the least-valuable page frame with data 1 to make a free page frame for the new data. It generates a page fault. The hypervisor should handle this by recovering the page with another free page frame. The hypervisor gets it by reclaiming the next LRU-positioned page frame with data 3, which is depicted as the curved arrow in the figure. After recovery, the guest OS moves the page with the data 5 to the MRU position on the list.

Here, the recovery procedure incurs two additional storage access: 1) A write to swap out data on a target page frame to be sacrificed and 2) A read to restore data of the accessed page. Repeated reclamation of guest OS generates chained reclamation and recovery operations until the guest OS stops reclaiming its own memory. Consequently, an LRU-based selection of the hypervisor induces: 1) The reclaimed page staying shorter; 2) Other page faults propagated to follow guest OS accesses. If multiple page frames are reallocated, the propagated overheads are accumulated and degrade the performance. With multiple victim VMs, the selected next least-valuable page is likely to have the same owner VM for their own localities, thus also accumulating overheads.

We can see the effect of policy mismatch more clearly with the dwell time of reclaimed pages. A dwell time indicates a time period that a page stays in the reclaimed state by the hypervisor. It ends when the owner OS of the page tries to access data on the page or reclaim the page. Page reclamation takes effect as if there is an additional page frame in the system while the page remains reclaimed. A longer dwell time of a reclaimed page increases the effect of an additional page frame to the system. The mismatch of two policies, however, incurs the worst-case selection of the

hypervisor, like that shown in the example. It brings about a short dwell time of each reclaimed page. In other words, the shorter the dwell time of pages the hypervisor reclaims and reallocates, the more the hypervisor has to perform reclamation and recovery operations. It makes the hypervisor perform two more successive storage operations to maintain the merits of one additional page frame during the same period of time. An increased number of storage accesses directly degrades the system's performance.

3.4 Capacity Disparity of Victim VMs

With hypervisor-level paging, there emerges a difference between the amount of memory that a guest OS knows and the amount of memory the guest OS actually uses. We named the difference as *capacity disparity*. A page implicitly reclaimed by the hypervisor has to be restored on the handling page fault from the reaccess. The recovery procedure makes the owner VM blocked until a free page frame is prepared and the content is reloaded from its original location. A disparity of the page, therefore, represents a potential overhead of performance degradation.

The more capacity disparity a VM has, the higher the possibility of performance degradation the VM suffers. If a VM with high capacity disparity tries to allocate more memory, it first consumes its own free memory. Then it tries to reclaim used pages, including already reclaimed pages by the hypervisor. As we observed in Section 3.3, those hypervisor-reclaimed pages are highly likely located around the least valuable position of the memory management list. It may initiate recovery procedures within a short time. If a VM has high capacity disparity, an accumulated number of reclaimed pages generate consecutive free-page making and swap-in operations, causing performance degradation.

4 Reference Pattern-Aware Instant Memory Balancing

In this section, we present instant memory balancing architecture based on reference pattern awareness. We first clarify our design goals and describe the basic designs of the system. We then describe our new memory balancing scheme, including a new victim page frame selection policy, that ensures low balancing delay and low decision overhead.

4.1 Design Goals

The primary purpose of memory balancing is to enhance the system performance by utilizing memory efficiently when some VMs consume lots of memory. To make the balancing more effective without problems in Section 3, our memory balancing architecture is based on the following design goals. First, it should be able to balance system-wide memory needs and the memory redundancy of each VM promptly with low overhead. Second, it should have a lower delay in memory reallocation procedure to maximize the values of additional memory to the domB. Third, it should try to minimize the performance penalty from the wrong selection of reclaimed page frames. The candidates should be page frames that mostly will not be used.

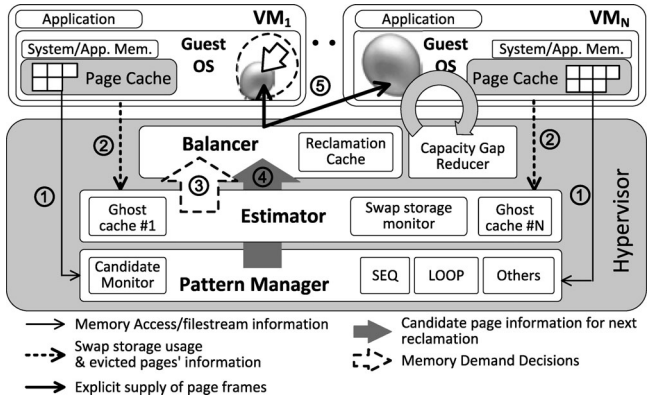


Fig. 6. Overall architecture and procedure of HD2.

4.2 Basic Design

To achieve our design goals for effective memory balancing among VMs, the hypervisor need to decide: 1) When the reallocation occurs; 2) how many page frames are needed to move; 3) which page frames are to be sacrificed; and 4) how to transfer the ownership of the page frames. We primarily excluded the interventions of the victim guest OS from the memory balancing procedure. To determine these, we design six major components in HD2: an estimator, a candidate monitor, a pattern manager, a balancer, a reclamation cache, and a gap reducer. Fig. 6 illustrates the overall architecture of HD2.

The Estimator assesses which VM needs more memory compared with each other. Finding a need to balance memory supplies among the VMs is the first step to reduce performance degradation from an imbalanced memory allocation. We targeted the memory requirements for both dynamic memory and the page cache for all guest OS, which is not shown to the hypervisor. To get them, the estimator monitors the swap storage usage and tracks the hit ratio change of their page cache according to the increase in cache size. The estimator decides the size of additional memory for each VM based on the collected demand information. The final step is that the hypervisor uses the differences of estimated memory usage as the memory needs of each VM.

The Candidate Monitor distinguishes page frames that belong to page caches and then eavesdrops on accesses to them. In HD2, each guest OS passes on status information about the pages belonging to its own page cache to the hypervisor. The information includes which page is inserted into, evicted from, and reused within the cache. The candidate monitor maintains an integrated pool of the informed page frames. It reduces monitoring overhead by restricting target memory to page frames in the pool.

In our HD2, we target page frames that belong to the page cache to avoid several overheads. First, tracking access to all pages in a guest OS generates prohibitive management cost. Second, we can reduce additional storage I/O when a target page frame is reclaimed by the hypervisor. If a reclaimed page is reaccessed, it is the responsibility of the hypervisor to recover the page. If the page is clean, there is no need for the hypervisor to generate extra I/O to save the data because the original copy is located in the storage. If dirty, the cold page will most likely be flushed to the storage when it faces eviction from the page cache.

The Pattern Manager finds access patterns to the candidate page frames and manages them based on the detected sequence. To find access patterns of the candidate page frames, the pattern manager eavesdrops the file access of all tasks. With the combination of references to the candidates and its per-task file-level access information, the manager finds access patterns from the access to multiple page frames. The page frames involved in a pattern are tied together as a *sequence* that has access pattern as an attribute. The pattern manager classifies sequenced page frames into three categories: *sequential*, *loop*, and *unclassified*. Management details are explained in Section 4.5.

The Balancer changes the memory allocation of all VMs with hybrid balancing scheme based on the decisions from the estimator. It marks VMs as beneficiary if and only if the VM needs more page frames than the predefined threshold. The balancer reclaims selected page frames managed by the pattern manager to make free page frames and takes ownership of them. It then transfers the ownership to the domBs and makes them recognize additional memory by triggering their balloon drivers. If the balancer consumes most of the candidates, it uses a ballooning scheme.

The Reclamation Cache is a fixed size LRU-managed memory cache that cancels out recovery overhead from the wrong decision, like the Linux swap cache [12]. It manages a predefined number of page frames where it starts with free page frames. When a page frame is marked as reclaimed by the balancer, the page frame is inserted into the MRU position of the reclamation cache without clearing data. Instead, an LRU-positioned page frame in the cache is reclaimed and reallocated to the domB. If a page frame is reaccessed within the cache, the hypervisor lazily reclaims another page frame and refills the cache. Consequently, pending data removal on the about-to-be-reclaimed page frames for a short time can make the hypervisor reduce the amount of data recovery from reaccess to the page frame just reclaimed.

The Capacity Gap Reducer is a module to reduce the capacity disparity of the domVs between the guest OS knows and it actually uses. It manages a metadata list of the reclaimed pages for each VM. It triggers *inflation* of the balloon of the target domV and then restores the reclaimed pages of the VM with the page frames from the inflation.

4.3 Instant Memory Balancing

Memory balancing is a sequential procedure of memory acquisition from victim VMs and memory supply to beneficiary VMs. To take advantage of previous approaches while removing drawbacks, our HD2 reclaims target page frames with hypervisor-level paging and presents them with the *deflation* of the ballooning, which is a hybrid method of previous approaches. The overall procedure of hybrid memory balancing is illustrated in Fig. 6.

1. The Candidate Monitor in the hypervisor tracks access to candidate page frames, ①, and the Pattern Manager detects reference patterns of page frames based on filestream information and tracking history.
2. The Estimator determines relative memory demands based on information from guest OS, ②, and criteria described in Section 4.6.
3. On receiving balancing decision, ③, the Balancer makes free page frames based on the selection policy on candidate page frames, ④. It unmaps selected page frames from their page table entry (PTE) and sends them to the reclamation cache. The same number of page frames in the cache is cleared and becomes ready for reallocation to the domB. The balancer subsequently triggers *deflation* of the balloon driver, ⑤, to make beneficiary guest OS recognize the supplied free page frames. The *balloon* in the domB initially has pinned pages that are not mapped to the physical page frames. It makes the guest OS have provisional pages for additional physical page frames. The hypervisor then maps the presented physical page frames to the spare pages so that it can recognize additional memory. The hypervisor finally *deflates* the balloon in the domBs so that the balloon unpins a designated number of pages and relinquishes their control to guest OS. The beneficiary guest OS holds power over the released page frames.
4. The Capacity Gap Reducer is initiated asynchronously whenever the capacity disparity between the memory size a guest OS knows and the size the guest OS actually owns exceeds a predefined threshold.

Basically, hypervisor-level paging makes the reclamation procedure available even if the victim is not in the running state. Unlike the ordinary paging mechanism, hypervisor-controlled paging in HD2 has no swap out operation of data on the target page frame. The paging in HD2 targets a clean page of the page cache in the victim guest OS so that the original copy of the data is located in the permanent storage. As a result, the hypervisor consumes less time to steal free page frames.

4.4 Filestream-Aware Reference Pattern Detection

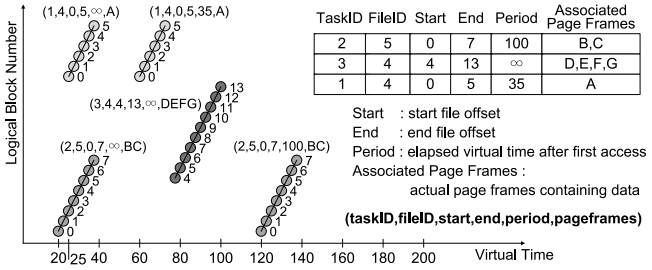
As a preparation to select page frames that have the longest estimated dwell time, the hypervisor detects reference patterns of candidate page frames based on filestream-awareness. We proposed a two-phase reference pattern detection and page frames association. In the first phase, the hypervisor distinguishes tasks and its filestreams. It then detects a filestream-level data access pattern and associated page frames.

4.4.1 Filestream Distinguishment

A filestream is an access stream to a file by a task through a file descriptor. The task accesses multiple files by creating multiple filestreams identified by the file descriptor. To distinguish each filestream, the hypervisor differentiates each task by recognizing the changes of the page table used in [18]. The guest OS invokes a hypercall to notify arguments of some system calls related to the file operations stored in the permanent storage. A combination of an identified task and notified arguments of system calls enable the hypervisor to distinguish each filestream.

4.4.2 Filestream-Aware Sequence Detection

The pattern detector detects the reference patterns from accessing each filestream and creates sequences with the



TaskID	FileID	Start	End	Period	Associated Page Frames
2	5	0	7	100	B,C
3	4	4	13	∞	D,E,F,G
1	4	0	5	35	A

Fig. 7. Examples of sequential and loop reference patterns and a sequence management table.

associated page frames. A sequence is an ordered collection of page frames that chronological references to them form a pattern. We concentrated on two data reference patterns: sequential and loop. A reference is categorized as a sequential reference if it accesses a contiguous position of a filestream. Contiguous and non-repeated sequential references to a filestream create a sequential reference pattern if the number of references is greater than a predefined threshold. Associated page frames form a sequence and are considered *sequential* type. A reference is categorized as a loop reference if it is also a sequential reference and is part of an already detected sequence. Associated page frames change its type as *loop*. We classified associated page frames as sequential if two or more consecutive read system calls access more than two blocks contiguously. To manage a sequence, six-tuple information was collected and used to detect data reference patterns from each distinguished filestream. A tuple contains a key composed of a task descriptor and a file descriptor. Each tuple has the following values: a start offset, an end offset, a loop period, and the physical address of associated page frames.

Fig. 7 shows an example of the sequential and loop reference pattern and a sequence management table showing gathered information, including associated page frames. 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 an infinite loop period. The page frames that contain several blocks of files with fileID 4 and fileID 5 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. Each entry in the table represents a *sequence*.

4.5 Reference Pattern-Aware Victim Page Selection

The pattern detector manages sequences based on their detected reference pattern. Sequences fall into two categories: sequential and loop. Page frames that are not included in any sequence are managed in the *unclassified* category.

Sequential. A sequential category collects sequences that consist of page frames that store data blocks which are contiguously accessed. If a page has a sequential reference pattern, it will not be accessed again until the page is evicted from the page cache. Sequences in the *sequential* category are arranged in the most-recently-used (MRU) order. Whenever a new sequence is created or a sequence is lengthened, the sequence moves to the MRU position of the category list. If some part of a sequence has a different access pattern, the sequence is split into subsequences according to the pattern. All subsequences then update its own location to the corresponding category.

Loop. If a *sequential* sequence is accessed repeatedly, the sequence is categorized as *loop*. Because it is accessed sequentially during the first iteration of the repeated accesses, its pattern is correctly detected when the second iteration is initiated. A sequential sequence changes its location to the loop category if it is accessed again. In the *loop* category, a sequence is a sequential sequence with a reference period. Sequences in the *loop* category are arranged in the LRU order. A sequence moves its location to the MRU position whenever the sequence is accessed sequentially.

Unclassified. All candidate page frames that are included in any page cache, but not included in any sequence, are ordered with the LRU strategy in the unclassified category. If several page frames form a sequence, they move to the sequential category.

In any selection the balancer makes, sequential sequences always come first. The balancer reclaims page frames that belong to a sequence in the MRU position of the management list for sequential sequences. If the number of page frames required is less than the target sequence has, the balancer reclaims page frames from the start position of the sequence. The loop category is the next candidate only after there are no more sequential sequences to reclaim. Unlike the sequential sequences, the balancer selects sequences in the LRU position of the management list for its periodic access. Selection criteria within a sequence is the same as sequential sequences. If all sequences are reclaimed, the page frames in the LRU position of the *others* categories become candidates for reclamation.

Note that we used two approaches when reclaiming sequences. One is to make the balancer reclaim the same number of page frames as requested. The other is to make the balancer reclaim the entire sequence if it is necessary to reclaim a page frame in a sequence.

4.6 Decision of Beneficiary VM and Its Memory Needs

The estimator shown in Fig. 6 evaluates the need for additional memory based on two values: 1) dynamic memory requirement and 2) additional page cache demands. We estimate them from swap storage usage and ghost cache.

Estimating the dynamic memory requirement of a VM starts from monitoring the swap storage usage of a time period like [2]. The estimator then calculates the average value of all VMs' swap storage usage collected during the last period. Dynamic memory requirement is calculated by the difference between the estimated need and the average need. The estimator also evaluates additional page cache demand of a VM with the ghost cache described in Section 5.3. The estimator decides additional cache needs N during the same time period from the index number i_N of the ghost cache where the index has the maximum derivative of changes to index i_{N+1}, which exceeds the threshold. Finally, the total memory demand of a domB is the sum of estimate for the dynamic memory requirement and for page cache. The estimator marks a VM as beneficiary if and only if the VM is estimated to have more memory requirement than the average. We use 100 ms as the time period and the

average number of all ghost cache hits as the ghost cache threshold. We also use 32 as a triggering threshold for the sensitivity of the Balancer.

Note that HD2 makes the hypervisor only monitor relative needs for memory growth. The memory demands of VMs selected as the beneficiary determine the number of page frames to be moved. The owner VMs of the reclaimed page frames become victims after the balancer's pattern-based victim selection.

4.7 Reducing Overhead from Capacity Disparity

Applying hypervisor-controlled paging creates a gap of memory capacity between that a guest OS knows and that the VM actually uses. This gap, called capacity disparity, has potential overheads on reaccess to the reclaimed pages. On trying to access a reclaimed page, the hypervisor should recover the page with a free page frame and data in it. Securing a free page frame may generate another reclamation of a page frame. Restoring data in the page frame is completed only after a data read from a permanent storage. This results in additional delay.

We applied two techniques to reduce recovery overhead from the capacity disparity of victim VMs. For both techniques, the hypervisor maintains per-VM lists, *reclaimed_lists*, to track reclaimed pages. First, the capacity gap reducer distinguishes the source of access to the target page and does not initiate data recovery if the access is by the page frame reclamation algorithm in a guest OS trying to reclaim the page. The second technique is to make the victim guest OS know its actual memory usage, named reverse-ordered ballooning. On deciding to reduce the disparity of a victim VM, the hypervisor *inflates* the balloon in the victim guest OS enough to compensate the disparity. The hypervisor then restores the reclaimed pages in the *reclaimed_list* of the VM. The hypervisor executes this when a VM is not involved in balancing or in idle state for 10 seconds.

5 Implementation

We have implemented our prototype system for the Xen [7] 4.1.1 hypervisor and Linux 3.9.6 dom0 and guest OS. All components were implemented in the Xen with some hypercall invocations in the Linux kernel. In the following subsections, we describe some implementation issues of our scheme in detail.

5.1 Monitoring Page Cache Access

Monitoring access to the candidate page frames composed of all VMs' page cache is a combination of distinguishing page caches and access detection to them. To distinguish page frames that belonged to a page cache, we explicitly used hypercalls, even though they are available without any modification of the guest OS [19]. Whenever a page is inserted into or evicted from a page cache, a *hd2_page_cache_op()* hypercall is called to inform the hypervisor of a change in the page's position. In order to detect memory access events that are usually transparent to the hypervisor, we borrow minor page faults from [2], [20], [21]. With minor page faults, the hypervisor traps target page frame access intentionally by granting the highest access privilege to the pages of concern. As access to the page table entry requires the highest privilege, non-privileged access to the target page is trapped to the hypervisor. It can make the hypervisor recognize access to the concerned page frames. Consequently, the hypervisor can monitor all access to page frames used in any page cache of all VMs.

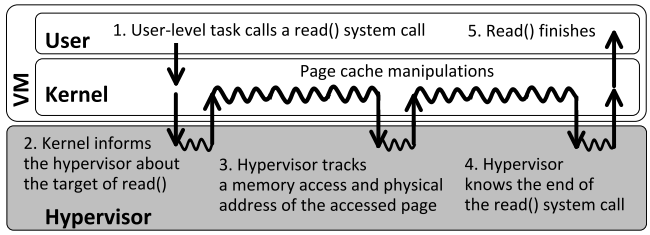


Fig. 8. Overall procedure of the event correlation scheme.

5.2 Page Frame Association with Event Correlation

Filestream-level access pattern is collected information of a consecutive series of file access events informed by the guest OS. However, all file information from guest OS has no physical page frame information because they are not visible to the guest OS. We applied event correlation to associate the detected access patterns with the actually accessed target page frames in the hypervisor.

The basic idea is that a read system call has a causal relationship with the following access to page frames in the page cache in the same process context. A read system call is a blocking operation that waits for the data to be read into the specified buffer. Not any operation could progress in the same process context during the system call. As a result, the system call is the only source that causes the page frame access until the call returns, and so does the caller task identified by the CR3 register. We borrowed the task identification technique from [18].

Fig. 8 illustrates the detailed procedure of the event correlation scheme. It starts from informing the event of consecutive file access and target information, including a file descriptor, an offset in the file, and the length of the target data via an *inform_event_op()* hypercall. From the page cache monitoring, access to the page frames in the page cache are trapped to the hypervisor. With information from two entities, and on the basis of our idea, the hypervisor associates a read event with the accessed page frame addresses. By detecting per-task reference patterns, the hypervisor forms pattern sequences, which are represented as six-tuple entries in Fig. 7.

5.3 Ghost Cache with Stack Distance Algorithm

To estimate the memory requirement of page cache for the storage I/O of a VM, we implemented a profiler and *ghost cache*, which were inspired by [19], [22]. It is based on Mattson's stack distance algorithm [16], which is widely used for cache size estimation. The algorithm is based on the inclusion property of the LRU replacement strategy for a size-limited cache. That is, the content of any LRU-based cache with size N is a subset of the contents in a cache with size larger than N.

Fig. 9 illustrates the concept of the ghost cache. A ghost cache manages a fixed number of metadata entries with LRU strategy where each entry is indexed by its distance from the MRU position. The metadata of the evicted page

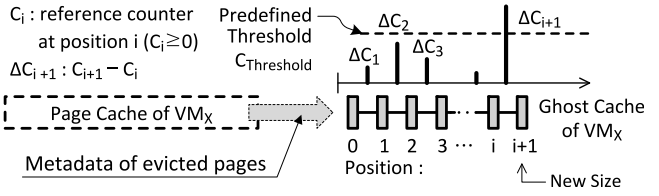


Fig. 9. A basic concept of the ghost cache to measure page cache memory requirement. The hypervisor gives i page frames to the VM_X if $\Delta C_i > C_{Threshold}$.

from the page cache is inserted into the MRU position of the ghost cache. It includes storage location information of the data in which the evicted page is contained. Entries are evicted from the LRU position if the cache size exceeds a predefined capacity. At each position i of the ghost cache, there is a reference counter, C_i. Every time there is a miss in the page cache, the hypervisor detects and looks them up in the ghost cache. If an access touches a block information in a metadata at the ith position of the ghost cache, the hypervisor increases the C_i counter and removes the entry from the cache. If the access ends up being a miss on both the page cache and the ghost cache, the hypervisor increases the C_{i+1} counter. The hypervisor considers that the page cache requires $i+1$ more page frames when the value of ΔC_{i+1} exceeds a predefined positive threshold $C_{Threshold}$.

6 Performance Evaluation

In this section, we describe the experimental evaluation of our prototype implementation. HD2 is implemented on an HP DL165G7 server with 24 cores (two AMD Opteron 2.0 GHz 12-Core processors) and 32 GB memory. By default, all guest VMs are initially allocated 512 MB of memory and configured with 2 GB memory as their highest possible memory allocation. HyperDealer2 and Balloon-based balancing are designated by HD2 and BLN, respectively. Memory usage profiles of the benchmarks can be found in [23], [24].

6.1 Runtime Overhead

Runtime overhead comes from four possible sources: 1) estimation of additional memory demand with ghost cache and swap storage usage tracking, 2) the minor page faults for tracking page frames in the page cache, 3) the detection of reference patterns, 4) and data reload from permanent storage because of the reaccess to the reclaimed pages.

To evaluate it, we performed experiments with our scheme turned on but no memory reallocation. Table 1 shows the normalized execution time of several benchmarks. The result shows that only three of SPEC benchmarks and three I/O benchmarks suffer from 2 percent of performance degradation with our scheme. It is because they access data on page cache more, thus triggering a trap to the hypervisor and pattern detection. We will discuss the data reload overhead in Section 6.4 for its dependency on pattern detection accuracy and page cache characteristics.

6.2 Reclamation Delay

In this subsection we evaluate average one-time memory reclamation delay, $T_{Schedule} + T_{Release}$ of the two balancing methods to analyze the effect of the victim VM's state, the number of victim VMs, the number of VCPUs, and the workload on the VCPUs.

TABLE 1
Normalized Runtime of Benchmarks with Detection Overhead

Benchmarks	Time	Benchmarks	Time
400.perlbench	100.4%	462.libquantum	100.8%
401.bzip2	100.1%	464.h264ref	100.0%
403.gcc	100.0%	471.omnetpp	100.0%
429.mcf	100.0%	473.astar	100.3%
445.gobmk	100.0%	Tiobench(seq. read)	100.9%
456.hmmer	100.0%	Tiobench(ran. read)	100.2%
458.sjeng	100.0%	Dbench	101.7%

6.2.1 Effect of the Internal State of The DomVs

To analyze the effect of the target VMs' state to the part of reclamation delay, $T_{Release}$, we measured the average reclamation delay of pages from a VM in different situations. In this experiment, a victim VM is in an 'idle' state when the reclamation procedure incurs no data flush to the storage and there are no I/Os disturbing the reclamation operation. Otherwise, the domV is in a 'busy' state. Table 2 shows the result. When the domV is in an 'idle' state, the balloon driver reclaims clean pages only. Otherwise, the balloon driver in the domV reclaims dirty pages with a data flush to permanent storage. Excluding the execution time of the balloon, the average one-time reclamation delay increases slightly as the number of reclaimed pages increases. With a 'busy' domV, every reclamation initiated by the balloon generates a storage write event. Intensive write I/Os within a short time cause contention, which makes the reclamation procedure longer. Therefore, the time delay increases rapidly. In contrast, our scheme shows a small and almost invariant delay because it sacrifices the same number of clean pages in the page cache, where the contents on those pages can be discarded without flush operations.

6.2.2 Effect of the Number of DomVs

The effect of the number of domVs to the reclamation delay can be shown by measuring the acquisition time of a fixed number of pages from various number of domVs. Fig. 11 shows the experiment result. Each domV is assigned to relinquish a smaller number of pages as the number of domVs increases. With the ballooning method, more domVs have a more parallelized effect on the reclamation procedure. It reduces the total elapsed time to transfer control of

TABLE 2
One-Time Reclamation Delay of Pages from a domV

Methods (HD2 : HyperDealer2, BLN : Ballooning) (ms)		Number of Pages 100	1000	10000
HD2		0.1	0.7	6.4
BLN	domV w/ idle state all target pages are clean pages	9.4	12.6	40.5
	domV w/ data flush to storage all target pages are dirty pages	53.1	674.9	1601.2

The Delay greatly depends on the I/O state of the target domV.

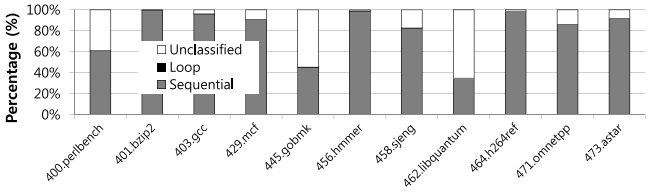


Fig. 10. Detected patterns of selected benchmarks.

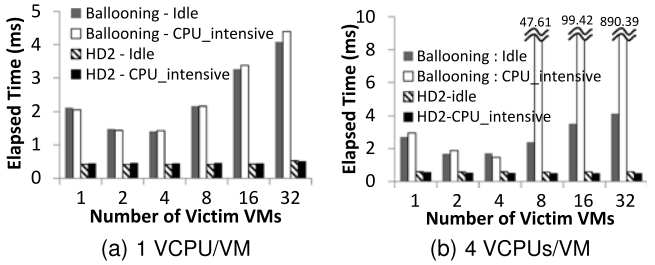


Fig. 11. Elapsed time to reclaim 1,024 pages from domVs with the designated number of VCPUs to the hypervisor.

a designated number of page frames to the hypervisor. However, overhead from $T_{Schedule}$ countervails the benefit from the parallelization, thus increasing reclamation time irrelevant to the workload. If a domV does not handle the memory balancing request during its scheduled time, the hypervisor has to wait for the next schedule of the domV. The more domVs, the longer all domVs complete memory relinquishments, which leads to an increase in $T_{Schedule}$.

Unlike the BLN reclamation procedure, HD2 does not spend timeslices allocated to the domVs. The hypervisor directly chooses pages to be reclaimed based on own reclamation policy. It makes the reclamation procedure independent of the number of domVs and their schedulability, which makes memory balancing more scalable.

6.2.3 Effect of the Number of VCPUs

The next experiment is to analyze the effect of the number of VCPUs and its state to the reclamation delay. With the idle state of VCPUs, there are little effects of the number of VCPUs on the memory reclamation, even more than the number of PCPUs. Figs. 12a and 12b show the results. It is because only a VCPU is activated to respond to the memory request with the ballooning method. The proposed method performs a reclamation procedure in the hypervisor, which is independent of the schedule of the domVs. Therefore, the elapsed time only depends on the number of page frames to be reallocated on both methods.

However, if domVs with multiple VCPUs are in VCPU-busy states, the memory reclamation from them takes longer. Figs. 12c and 12e show the results where each domV runs the same number of CPU-intensive tasks as the number of VCPUs each domV has. With ballooning, an increase in the number of VCPUs with a fixed number of domVs have little effect on the reclamation procedure, especially if the total number of VCPUs does not exceed the number of PCPUs in the machine. If there are more VCPUs than PCPUs in the machine, the delay increases. From the effect of an increased number of VCPUs, the hypervisor waits for another VCPU of the target domV to be scheduled if the scheduled VCPU for the domV does not handle the reclamation procedure. It therefore increases the $T_{Schedule}$. We can see the same result in Fig. 11, where an increase in the number of domVs also increases the number of VCPUs. The noticeable delays appearing with more than 32 busy VCPUs are mainly due to the total number of VCPUs exceeding the total number of PCPUs, i.e. 24.

It is obvious that busier VCPUs intensify the competition among all VCPUs in all VMs for PCPU timeslice allocation. The hypervisor scheduler tries to share time fairly among the increased number of VCPUs for all VMs. Any VCPU in a victim VM, therefore, is allocated a smaller timeslice after a longer waiting time. It generates a longer $T_{Schedule} + T_{Release}$, despite multiple PCPUs improving parallel execution of all victim VMs' ballooning operation.

The situation gets worse if the victim VM performs page frame reclamation and makes I/O to flush data to storage. Figs. 12f and 12h shows a remarkable increase of reclamation delay where an I/O performing task co-exists with CPU-intensive tasks. Flushing dirty pages to the permanent storage during the reclamation procedure increases $T_{Release}$. Application I/O requests interfere flushing operation of the guest OS, thus making the delay unpredictable. As a consequence of the longer $T_{Schedule}$ and $T_{Release}$, reclamation time using ballooning technique increases as the number of busy VCPU for victim VMs increases.

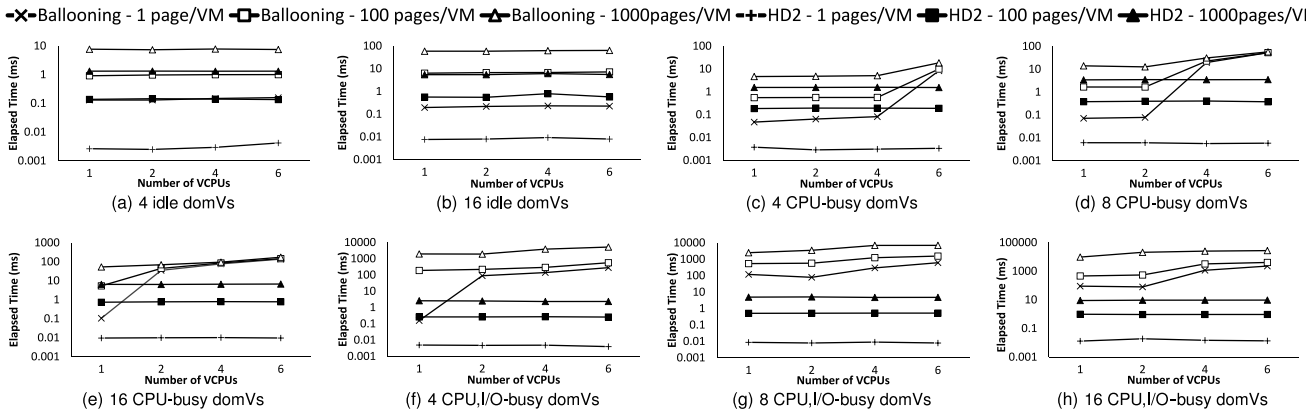


Fig. 12. Average elapsed time of one-time page frame reclamation for a beneficiary VM according to the number of VCPUs in victim VMs (domVs). Proposed Method uses LRU victim selection strategy.

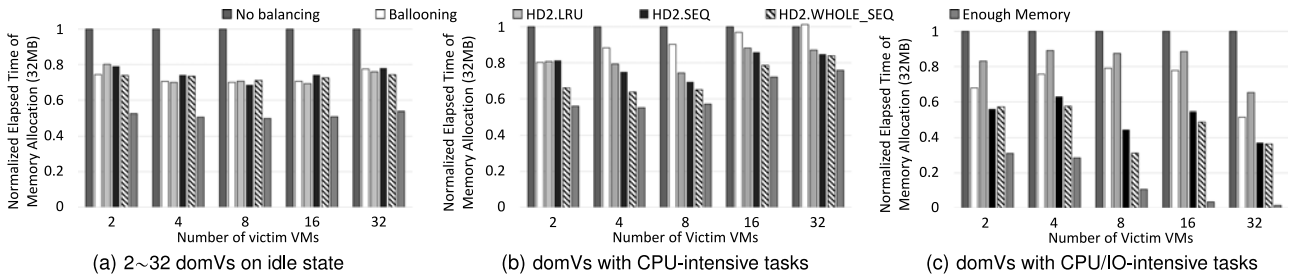


Fig. 13. Performance effect of the memory reallocation speed measured by the elapsed time of reclamation for a beneficiary VM according to the number of victim VMs with four VCPUs/VM.

Unlike the BLN, the proposed method shows little variances in delay in spite of storage I/O and busy CPU states of victims. Figs. 12c and 12h support the analysis with a small balancing delay. From the properties of the paging, hypervisor-level paging makes the reclamation procedure independent of the scheduling of the victim VM and its task taking charge of the reclamation. It is sufficient for the hypervisor to handle memory balancing just before each domB is scheduled. It results in a negligible $T_{Schedule}$. In addition, two policies restrict the occurrence of drawbacks from the hypervisor-level paging. Reclaiming clean pages avoids generating data flush to the permanent storage. Pattern-based victim selection restricts the reaccess of the reclaimed page frames. These make $T_{Release}$ sensitive only to the number of page frames to reclaim. Consequently, the hypervisor performs balancing with a small delay, even with busier VCPUs of the victims.

6.3 Effect of the Memory Reallocation Speed

The primary purpose of memory balancing is to enhance memory allocation speed by reducing time to make free memory in a guest OS. To evaluate the effect of memory reallocation speed on the memory allocation performance, we measured the elapsed time of memory allocation with multiple domVs. In the experiment, a domB performs a memory allocation job while domVs run warmup I/O tasks, followed by CPU-intensive tasks, in the SPEC CPU2006 benchmark. Each memory allocation trial asks for 32 MB of free memory allocation and fills random data on it. In an idle state, each domV does nothing and has enough free memory. In an busy state, each domV runs an instance of modified Tiobench [25] benchmark generating read and write requests equally, and (the number of VCPU-1) instances of *445.gobmk*, which makes CPU busy with a small memory footprint. The modified Tiobench benchmark generates read and write requests equally where each type of request is composed of sequential and random patterns equally. Each VM starts with 512 MB of main memory and 2 GB of maximum main memory capacity. In particular, each VM has 1.5 GB of memory with two victim VM cases to supply enough memory to reallocate.

Fig. 13 a shows the elapsed time of memory allocations in domB with idle domVs. Almost all cases show similar performances, regardless of the change in the number of idle domVs and balancing methods. Because there are little active tasks in each domV, the VCPUs performing the reclamation procedure of each domV are in competition with each other. It schedules the reclamation procedure to the PCPU after a small $T_{Schedule}$. Enough free memory in each domV makes them respond to the balancing request with a small $T_{Release}$. This results in a similar reclamation delay despite the varying number of VCPUs and a different method is applied to the balancing.

In a busy state of the domVs with the BLN, however, an increased delay from more domVs reduce the effect of additional memory on the domB. The results of the BLN is shown in Figs. 13b and 13c. As for the delay, the $T_{Schedule}+T_{Release}$ is longer, while a designated amount of free memory arrives in domB much later from when it is requested. This results in little reduction of internal memory reclamation overhead of the domB, which makes the domB lose chances to reduce overhead that arose from memory thirst. Even with 32 domVs, the domB acts as if there was no balancing system.

Compared with the BLN, the proposed method greatly reduced memory allocation speeds. To distinguish the effect of pattern-based victim page selection, we compared the proposed method, denoted as SEQ, with a traditional LRU selection strategy. The SEQ in Fig. 13 shows the worst case results as continuous I/O make pages in the page cache reused fast. Compared to the BLN, memory allocation takes little time with both LRU and SEQ-based balancing. Irrelevance to the state of the domVs and their number depletes the $T_{Schedule}$ and the $T_{Release}$. Based on this, the time gap widens as the number of domVs increase.

The difference between two methods consists in the position where the algorithm selects page frames. Unlike LRU, SEQ selects from the MRU position among sequences. Both cases have the same result if there are no new I/Os incurring the reuse of pages at the LRU position of the page cache, which rarely occurs. A long dwell time from sporadic I/O adds little delay to the reclamation procedure. Details of dwell time are explained in Section 6.4. More I/O make the dwell time of all reclaimed pages shorter. In particular, the dwell time is extremely shortened under the LRU-based reclamation policy. A shorter dwell time is the result of a more frequent occurrence of page frames reuses. This incurs more reclamation to restore already reclaimed pages. Therefore, an aggravated iteration of reclamation delays balances the speed, thus providing less benefit to the domB than with SEQ policy.

In addition, we can get more accelerated results with the WHOLE_SEQ policy. WHOLE_SEQ is an aggressive balancing method that reallocates all page frames in a victim sequence at a time. Because page frames in a sequence are highly likely to be access together. By using this scheme, the

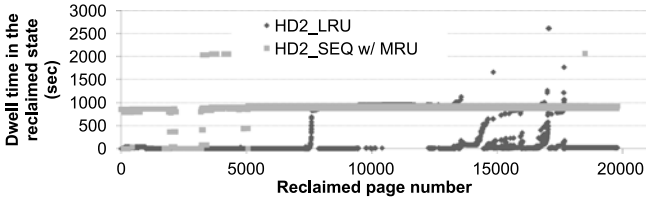


Fig. 14. Dwell time of the reclaimed pages in the reclaimed state in domVs.

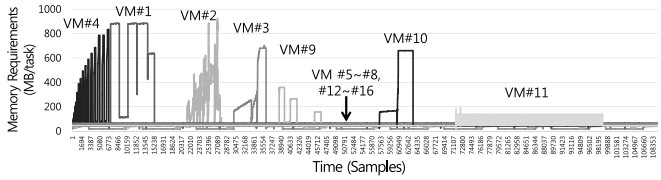


Fig. 15. Memory loads scenario on each VMs.

hypervisor can supply more memory preemptively to domBs against memory shortage in the near future. If reaccess to the reclaimed page occurs due to wrong pattern detection, those pages can be recovered with one or two read operations, which minimize restoration overhead of the contents. We applied the WHOLE_SEQ policy with 64 pages of maximum single reallocation in Fig. 13. In the experiment, preemptive memory reallocation is iteratively performed in response to the sharp increase of memory needs so that the domB can make provisions for the following memory requests. It brings a more stable, but accelerated, memory allocation performance of the domB. This policy is good to handle a sharp-increasing memory requirement of guest OSes.

6.4 Effectiveness of Pattern Detection: Dwell Time

Sacrificing sequentially referenced pages prior to sacrificing unclassified pages affects the performance. We compare the dwell time of reclaimed pages in the reclaimed state to show the effectiveness of the pattern detection. Detected reference patterns of the benchmarks are shown in Fig. 10. The results are shown in Fig. 14. Like the swap operation of an OS, reclamation of the hypervisor is more effective if the hypervisor could select pages that will not be reaccessed longer. The purpose of the pattern detection in the reclamation procedure is to classify candidate pages by its reaccess possibility. Because higher detection accuracy generates less number of pages with short dwell time, more pages with longer dwell time indicates better choices of the hypervisor with the pattern detection. In the experiment, each of the three VMs runs two instances of 445.gobmk, while a VM executes sequential file reads and another VM runs 401.bzip2. We turned off the capacity disparity reduction scheme.

The result shows that many pages reclaimed with the LRU policy in the hypervisor has shorter dwell time than the page frames by the SEQ policy. Reclamations by the guest OS triggers recovery of the pages reclaimed recently, which is caused by the policy mismatch. Some pages reclaimed with LRU has similar dwell time but it is due to low reclamation rate of the domVs. The page frames reclaimed by SEQ, on the contrary, show a longer dwell time except some pages. Those exceptions are from mass reallocation of memory, which reclaim pages in sequences that are located near the LRU position. It shows similar performance as the LRU policy in the worst case. Those long dwell time also shows that the detection scheme works well.

6.5 Impacts on Application Performance

Fig. 15 illustrates two memory load scenarios to analyze the performance impact of memory balancing methods on running applications hosted by multiple VMs. The configuration of the scenarios in Fig. 15 is shown in Table 3. In the first scenario, each of the four VMs run two copies of a memory-intensive benchmark at different times and run two copies of a CPU-intensive benchmark for the rest of the experiment period. Seven different memory-intensive benchmarks were run in the second scenario with 16 VMs. Before starting, the page cache of each guest OS is warmed up with I/O workloads with a clean:dirty ratio of 1:1. Each VM starts with a page cache of approximately 413 MB after the warm-up. HD2 runs with the SEQ reclamation policy.

We can see remarkable performance enhancements on memory-intensive tasks while other CPU-intensive tasks show negligible performance degradation. Table 4 shows the experiment results. With 403.gcc having dynamically changing memory requirements on VM #2 with eight VMs running, we acquire 51 percent of performance improvement compared with the ballooning procedure. With HD2, reclaiming clean pages in the page cache simply discards

TABLE 3
Workload Configurations of the Memory Load Scenario

VM#	Workloads		VM#	Workloads	
	Mem.-intensive	CPU-intensive		Mem.-intensive	CPU-intensive
1	401.bzip2	445.gobmk	9	473.astar	465.tonto
2	403.gcc	445.gobmk	10	450.soplex	445.gobmk
3	400.perlbench	456.hmmer	11	436.cactusADM	444.namd
4	447.dealII		12		482.sphinx3
5		465.tonto	13		435.gromacs
6		482.sphinx3	14		445.gobmk
7		435.gromacs	15		482.sphinx3
8		444.namd	16		456.hmmer

Warming-up workload for each guest OS : clean:dirty=1:1, equally distributed, sequential reads/writes of 10 pages on average

TABLE 4
Normalized Execution Time of Benchmarks with Consolidated VMs Memory load scenario is depicted in Fig. 15.

(a) Performance with 8 VMs					(b) Performance with 16 VMs								
VM#	Benchmark	Methods			VM#	Benchmark	Methods			VM#	Benchmark	Methods		
		BLN	LRU	HD2_SEQ			BLN	LRU	HD2_SEQ			BLN	LRU	HD2_SEQ
1	401.bzip2	1	1.63	0.71	1	401.bzip2	1	1.35	0.83	9	473.astar	1	1.05	0.96
	445.gobmk	1	1.02	1.00		445.gobmk	1	1.00	1.00		465.tonto	1	1.00	1.00
2	403.gcc	1	1.47	0.49	2	403.gcc	1	1.47	0.86	10	450.soplex	1	1.39	0.89
	445.gobmk	1	1.03	0.99		445.gobmk	1	1.00	1.01		445.gobmk	1	1.00	1.00
3	400.perlbench	1	1.60	0.49	3	400.perlbench	1	1.61	0.50	11	436.cactusADM	1	4.42	0.93
	456.hmmer	1	1.00	1.00		456.hmmer	1	0.97	1.00		444.namd	1	1.00	1.00
4	447.dealII	1	2.33	0.93	4	447.dealII	1	0.97	0.90	12	482.sphinx3	1	0.98	0.98
5	465.tonto	1	1.01	1.00	5	465.tonto	1	1.00	1.00	13	435.gromacs	1	1.00	1.00
6	482.sphinx3	1	1.01	0.96	6	482.sphinx3	1	0.99	1.01	14	445.gobmk	1	1.00	1.01
7	435.gromacs	1	1.02	1.00	7	435.gromacs	1	1.01	1.00	15	482.sphinx3	1	1.00	1.00
8	444.namd	1	1.00	1.00	8	444.namd	1	0.97	0.98	16	456.hmmer	1	0.97	0.97

(LRU : HyperDealer2 w/ LRU policy, SEQ : HyperDealer2 w/ SEQ policy, BLN : Ballooning).

data, thus generating no data flush. This results in a shorter $T_{Release}$, which leads to shorter $T_{Realloc}$.

Results with the LRU policy support the effectiveness of HD2. With LRU, the recovery procedures degrade the application performance in spite of the same $T_{Schedule}$. The memory demand of the domB triggers its own reclamation operations along with memory reallocation by the hypervisor. Candidates for reclamation by the hypervisor are likely to be the candidates for reclamation inside the guest. This generates memory reclamation and reallocation by the hypervisor followed by a recovery procedure in the hypervisor. The delay from the repetitive recovery due to the previously reallocated memory fluctuates and aggravates the application performance significantly. This leads to a chained reclamation and recovery process, thus delaying the memory reallocation procedure. HD2, on the other hand, generates fewer recovery procedures for reallocated memory due to the long dwell time and the periodic capacity gap reducing operations. Fast memory reallocation can make the guest OS respond following memory allocation requests with the received page frames. This reduces the number of following swap out operations of the domB, which consequently accelerates the application performance.

The BLN takes more time to reallocate the same number of page frames than the HD2. A data flush is required into permanent storage if the target pages are dirty or if they are dynamically allocated by tasks, which increases $T_{Release}$. This causes the domB to lose its opportunity to cancel subsequent swap out operations with the donated page frames, worsening the performance.

A 14 percent performance improvement of 403.gcc is noted when 16 VMs are ran together. The small performance improvements with 16 VMs originate from the long $T_{Schedule}$ due to there being at least 32 busy VCPUs on the 24 PCPUs. In addition, the relatively small memory demand induces less frequent memory balancing events. From the results shown in Fig. 12, balancing with a smaller number of pages makes a smaller difference in the performance results between the two methods. While memory-intensive tasks accelerated, CPU-intensive tasks had little performance degradation independent of their running location. It is from their relatively small memory requirements, which make them act only as domVs.

7 Related Work

Contemporary studies have presented solutions that enhance machine-wide performance in the VM-consolidated environment. They focused on an increase of available memory for each VM to enhance system performance.

In Cellular Disco [3], each cell can borrow memory from other cells that are rich in free memory. The cell is defined as a fault containment unit in a cluster composed of several physical machines, and the authors proposed resource balancing policy among those cells. Our scheme, however, is for multiple VMs within a cell with single physical machine.

Memory hotplugging [26] can change the amount of memory on demand by installing or removing new physical DIMMs. With the balloon, it can allocate more memory than the initial configuration for a VM. However, it generates the same scheduling-induced delay because the balloon controls reallocation of the memory. Having a fixed and large size of memory unit can generate additional swapping and flushing of page cache, thus degrading the performance.

In Hypervisor Exclusive Cache [27], the hypervisor manages some of the VM's memory 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, achieves the same effect as changing the VM's memory allocation. A newly allocated memory is either inserted into the VM through a balloon or utilized as an exclusive cache in the hypervisor. Because the tracking of the 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.

Transcendent Memory (Tmem) [28] 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 tried 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 that require a large size page cache. However, each VM must maintain a sufficiently large private cache for a correct decision. It also cannot respond to the needs of memory-intensive but non-I/O-intensive tasks like [27].

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 working set size estimation. The domVs then inflate the balloon to release memory, and the domBs subsequently deflate the balloon to make the guest OS control the reallocated memory. MEB utilizes its own memory reclaiming policy with regards 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 storage in accordance with the guest OS's policy. Furthermore, the balloon's inflation and subsequent deflation can cause a scheduling-induced balancing delay. Hence, the ballooning is unsuitable for applications with dynamically changing memory requirements. Feedback-directed ballooning [29] also mentioned a similar solution with the same weak points.

CMM [4] attempts to address the issue of double paging [15] 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. However, unlike the PowerPC, monitoring of all memory access in hardware without any additional CPU protection privilege, such as Intel CPUs, significantly degrades performance [20], [21]. Our scheme limits the overhead by monitoring only page frames of a page cache that belongs to all VMs without any hardware assistance.

For the VM scheduling-related issue, Govindan et al. [30] tried to enhance the communication performance degradation caused by VM scheduling-induced delays in network communication. We found that a similar delay occurs in the conventional memory reallocation procedure and proposed a solution to reduce the delay.

The need for reference pattern detection was addressed in [22], [31], [32], [33], [34], [35], [36], [37] for victim selection of the page reclamation in OS. The authors detected and classified the sequential and loop reference patterns from instances where the disk cache is accessed and then a victim page is selected for reclamation on the basis of each pattern's marginal gain. Although this paper used a detection scheme for detecting reference patterns, we focused on the performance effect of memory balancing among consolidated VMs, especially with regard to the reduction in the victim VM's overhead.

[38], [39], [40] talked about memory balancing decisions among multiple victims. But they used conventional methods to move memory among VMs and did not consider balancing drawbacks on a manycore environment. They also did not cover the memory requirements of the page cache in a guest OS for I/O performance improvement.

This study is a part of the MN-MATE [5], [41], a resource management system for manycores and a hybrid main memory hierarchy of the on-chip DRAM, off-chip DRAM, and off-chip NVRAM to work as a cloud node. The primary purpose of the MN-MATE is to enhance system performance while saving energy by balancing CPU cores and various memories among VMs and by managing allocated resources inside each VM.

8 Conclusion

As virtualization technology consolidates more guest OSes 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 memory-thirsty virtual machines. In this paper, we provided a full-fledged non-obstructive memory reallocation scheme enhanced with the reference pattern-based victim selection and hypervisor-level reclamation. We proposed HyperDealer2, which makes balancing operation free from the involvement of victim VMs and extends the dwell time of reclaimed pages in the reclaimed state. Consequently, HyperDealer2 significantly accelerated the balancing operation with a low overhead, thereby increasing the effect of additional memory on the beneficiary VM. Our monitoring mechanism incurs less than 2 percent of performance degradation. Nevertheless, the experimental results shows that our proposed scheme reduces the execution time of memory-intensive applications by up to 50 percent.

Acknowledgments

This work was supported by the Ministry of Knowledge Economy, South Korea, under Project No. 10035231-2010-01.

References

[1] C. A. Waldspurger, "Memory resource management in VMware ESX Server," in *Proc. 5th Symp. Oper. Syst. Design Implementation*, 2002, pp. 181–194.

[2] W. Zhao and Z. Wang, "Dynamic memory balancing for virtual machines," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Envir.*, 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 *Proc. Ottawa Linux Symp.*, 2006, pp. 321–336.

[5] K. H. Park, Y. Park, W. Hwang, and K.-W. Park, "Mn-mate: Resource management of manycores with DRAM and nonvolatile memories," in *Proc. High Perform. Comput. Commun.*, 2010, pp. 24–34.

[6] W. Hwang, Y. Roh, Y. Park, K.-W. Park, and K. H. Park, "Hyperdealer: Reference-pattern-aware instant memory balancing for consolidated virtual machines," in *Proc. IEEE 3rd Int. Conf. Cloud Comput.*, 2010, pp. 426–434.

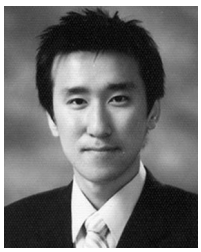
[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proc. 19th ACM Symp. Oper. Syst. Principles*, 2003, pp. 164–177.

[8] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[9] *Amd64 virtualization codenamed "Pacifica" technology: Secure virtual machine architecture reference manual*, AMD, Sunnyvale, CA, USA, May 2005.

[10] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *Proc. 11th Int. Conf. Very Large Data Bases*, 1985, pp. 127–141.

[11] G. M. Sacco and M. Schkolnick, "Buffer management in relational database systems," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 473–498, 1986.
[12] D. Bovet and M. Cesati, *Understanding the Linux Kernel*. Sebastopol, CA, USA: Oreilly & Associates Inc, 2005.
[13] VMWare, "Understanding memory resource management in VMWare ESX server," White Paper, 2009.
[14] Microsoft. Hyper-v dynamic memory overview. (2012). [Online]. Available: http://technet.microsoft.com/en-us/library/hh831766.aspx
[15] R. P. Goldberg and R. Hassinger, "The double paging anomaly," in *Proc. Nat. Comput. Conf. Expo.*, 1974, pp. 195–199, 2012.
[16] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.
[17] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2007.
[18] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2006, p. 1.
[19] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the buffer cache in a virtual machine environment," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 14–24, 2006.
[20] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "CRAMM: Virtual memory support for garbage-collected applications," in *Proc. 7th USENIX Symp. Oper. Syst. Des. Implementation*, 2006, pp. 103–116.
[21] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," *SIGPLAN Notices*, vol. 39, no. 11, pp. 177–188, 2004.
[22] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proc. 15th ACM Symp. Oper. Syst. Principles*, 1995, pp. 79–95.
[23] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.
[24] J. L. Henning, "Spec cpu2006 memory footprint," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 84–89, Mar. 2007.
[25] M. Kuoppala, "Tiobench-Threaded I/O bench for Linux," http://sourceforge.net/projects/tiobench/, 2000.
[26] J. H. Schopp, K. Fraser, and M. J. Silbermann, "Resizing memory with balloons and hotplug," in *Proc. Ottawa Linux Symp.*, 2006, pp. 313–320.
[27] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache," in *Proc. USENIX Annu. Tech. Conf.*, 2007, pp. 1–15.
[28] D. Magenheimer, "Transcendent Memory on Xen," *Xen Summit*, Feb. 2009.
[29] D. Magenheimer, "Memory Overcommit... without the commitment," *Xen Summit*, Jun. 2008.
[30] 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 *Proc. 3rd Int. Conf. Virtual Execution Envir.*, 2007, pp. 126–136.
[31] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references," in *Proc. 4th Conf. Symp. Oper. Syst. Design Implementation*, 2000, p. 9.
[32] 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 *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2000, pp. 286–295.
[33] C. Gniady, A. R. Butt, and Y. Charlie, "Program-counter-based pattern classification in buffer caching," in *Proc. 6th Conf. Symp. Opear. Syst. Design Implementation*, 2004, p. 27.
[34] 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 Trans. Comput.*, vol. 51, no. 7, pp. 793–800, Jul. 2002.
[35] F. Zhou, R. von Behren, and E. Brewer, "AMP: Program context specific buffer caching," in *Proc. USENIX Annu. Tech. Conf.*, 2005, p. 20.
[36] B. S. Gill, L. Angel, and D. Bathen, "Amp: Adaptive multi-stream prefetching in a shared cache," in *Proc. 5th USENIX Symp. File Storage Technol.*, 2007, p. 26.
[37] Y. Zhu and H. Jiang, "Race: A robust adaptive caching strategy for buffer cache," *IEEE Trans. Comput.*, vol. 57, no. 1, pp. 25–40, Jan. 2008.
[38] C. Min, I. Kim, T. Kim, and Y. Eom, " Vmmb: Virtual machine memory balancing for unmodified operating systems," *J. Grid Comput.*, vol. 10, pp. 69–84, 2012.
[39] Y. Niu, C. Yang, and X. Cheng, "Dynamic memory demand estimating based on the guest operating system behaviors for virtual machines," in *Proc. IEEE 9th Int. Symp. Parallel Distrib. Process. Appl.*, 2011, pp. 81–86.
[40] Q. Zhu and T. Tung, "A performance interference model for managing consolidated workloads in QoS-aware clouds," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, Jun. 2012, pp. 170–179.
[41] K.-H. Park, S. K. Park, W. Hwang, H. Seok, D.-J. Shin, and K.-W. Park, "Resource management of manycores with a hierarchical and a hybrid main memory for MN-mate cloud node," in *Proc. IEEE 8th World Congr. Serv.*, 2012, pp. 301–308.



Woomin Hwang received the BS degree in school of electrical engineering from Korea University in 2004, and the MS degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 2006. He is currently working toward the PhD degree in the Department of Electrical Engineering at KAIST. His research interests include virtualization, resource management, cloud computing, and embedded systems. He is a student member of the IEEE.

Ki-Woong Park received the BS degree in computer science from Yonsei University in 2005, and the MS and PhD degrees in electrical engineering from the KAIST in 2007 and 2012, respectively. He is an assistant professor in the computer hacking and information security department at Daejeon University. He worked as a researcher at National Security Research Institute in 2012. His research interests include system security issues for a real cloud and mobile computing systems. He is a member of the IEEE and ACM.

Kyu Ho Park received the BS degree in electronics engineering from Seoul National University, Korea in 1973, the MS degree in electrical engineering from KAIST, Korea in 1975, and the DrIng degree in electrical engineering from the Universite de Paris XI, France in 1983. He has been a professor of the Department of Electrical Engineering, KAIST, Korea, since 1983. His research interests include computer architectures, file systems, storage systems, and parallel processing. He is a member of the KISS, KITE, Korea Institute of Next Generation Computing, IEEE and ACM.