Improving Host Swapping Using Adaptive Prefetching and Paging Notifier

Wenzhi Chen, Huijun Chen, Wei Huang, Xiaoqin Chen, Dapeng Huang
College of Computer Science
Zhejiang University, Hangzhou, 310027, China
Email: chenwz@zju.edu.cn

ABSTRACT

In a virtualized system, the hypervisor may be forced to reclaim memory by swapping out pages of guest operating systems (OSes) when the regular memory balancing mechanisms, such as page sharing and ballooning, fail to revoke enough memory for reallocation purpose, which always leads to serious performance degradation. In this paper, we introduce Adaptive Swap Prefetcher (ASP) and Host Swapping Notifier (HSN), the effective and lightweight solutions to gracefully reduce the degradation in system performance when host swapping is triggered. ASP smartly prefetches more pages from the host swap file as long as the good spatial locality persists so as to reduce disk transfers. The guest OS will be notified by HSN when the hypervisor evicts pages, which then hides those pages from its memory reclamation routines to eliminate unnecessary guest swapping and to prevent the occurrence of double paging anomaly. Currently ASP and HSN are implemented in KVM, experimental results show that guest performance can be improved by a factory of 1.4x and 2x respectively using ASP and HSN.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*swapping*; C.4 [**Performance of Systems**]: Design studies

General Terms

Design, Performance

Keywords

Virtualization, Prefetching, Host Swapping, Double Paging, KVM

1. INTRODUCTION

Memory overcommitment [7], by which means the total memory of all guest operating systems (OSes) can exceed the maximum available memory of the hypervisor, is commonly adopted to increase the utilization of hardware resources and to run more guest OSes concurrently. Technologies, such as ballooning [9] and page sharing [9], are normally used to help balance memory usage. A premise to take full advantage of these assistive technologies is that there is enough memory in system to reclaim and redistribute, otherwise the hypervisor has to resort to the expensive swapping operations to force memory recycling. Theoretically, host swapping has shorter code path and fewer privilege changes to achieve better performance than guest swapping. However, deeper investigation explores some performance issues about host swapping, one is that it could result in the pages for guest OS scattering all over the swap file and degrade the spatial locality, which increases the IO seek time and IO amount and consequently decreases system performance [8]. Another problem is the so-called double paging anomaly [4], in which case the host OS has swapped out a page in the first place while the guest OS happens to pick on the same one to evict, resulting in an anomaly that this page gets faulted in and then paged out again immediately.

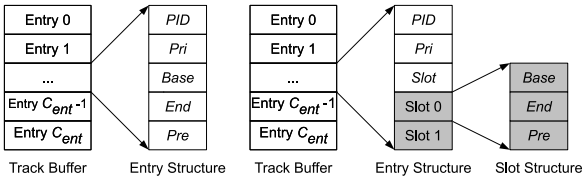
In this paper, we propose Adaptive Swap Prefetcher (ASP) and Host Swapping Notifier (HSN) to address the problems respectively.

2. RELATED WORK

Double paging anomaly was studied a long time ago, when Goldberg et al. revealed that an increase in the memory size of a virtual machine without a corresponding increase in real memory size can lead to this anomaly [4]. Then Ohmachi et al. proposed a new page replacement algorithm, PAWP/VMS, to prevent this anomaly and reduce the number of page fault interrupts [6]. Their method was based on the assumption that the size of LRU stacks are fixed and that both host and guest see the same LRU page sequences, which can not be satisfied by today's virtual machine systems. Chew et al. argued that a maximal-pool system can avoid the double paging anomaly [1], but their method is not well evaluated. A balanced approach, called Collaborative Memory Management (CMM) [7], that reaped the benefits of ballooning and host swapping, was introduced by Schwidefsky et al. to reduce overhead when memory is overcommitted. CMM basically maintains page states in both host and guest, and the host can readily discard pages that is not used or can be reconstructed from the guest, and then redistribute them. CMM requires a large effort to modify the guest code and coordinate host and guest, and the system wide overhead is not well studied.

The literature on OS level prefetching is comparatively

HPDC'10, June 20–25, 2010, Chicago, Illinois, USA.



(a) Simple Track Buffer (b) Buffer with Double Slots

Figure 1: A design of ASP track buffer and its improved variant.

very rich. Wu et al. redesigned the Linux readahead framework [2] by simplifying the legacy prefetching algorithm, which enjoys great performance improvements. By carefully studying the IO switch time of individual disks, the competitive prefetching can determine a most suitable prefetching depth to allow performance boost [5]. There are also adaptive asynchronous prefetching mechanisms [3, 10] that dynamically adjust the trigger distcance and prefetching depth to achieve better performance. Approaches that can significantly increase the predictive accuracy of prefetching by letting applications or compilers disclose hints are available as well, which due to page limits are not cited.

Despite the abundance of research on file system prefetching, no attention has been focused on swap prefetching due to its poor data organization. In this paper, the spatial locality of swap space is studied first before a prefetching policy is applied, which we will discuss in Section 3.1.2.

3. DESIGN AND IMPLEMENTATION

In this section, we present the design decisions and current implementation of Adaptive Swap Prefetcher (ASP) and Host Swapping Notifier (HSN) in KVM.

3.1 Adaptive Swap Prefetcher

3.1.1 Track Buffer

The key idea of our design of ASP is to keep track of each process/guest's swap-in activity. Each time the host OS (host for short) has to fault in pages for a guest OS (guest for short) or a process, ASP will first consult an entry in a track buffer (BUF) to determine how many pages to prefetch. The structure of BUF with maximum C_{ent} entries are shown in Figure 1a, where $Base$ and End are the locations in the swap partition of the last read, Pri is the priority of this entry, and Pre denotes the number of pages prefetched last time. Once an entry is picked up, a function is applied to figure out how many pages to prefetch.

3.1.2 Prefetching Function

The principle behind our prefetcher is that let the prefetching window grow smoothly when the good locality remains until a predefined limit is reached, and shrink quickly to the default prefetching size when the locality becomes poor. We define locality as the distance between two consecutive swap-in operations. More specifically, let $Base$ and End be the starting and ending point in the swap partition of last prefetching respectively, Tar be the place where the page needs to be faulted in locates, then the distance (DIS) of two successive swap-ins can be calculated as $DIS = Min(|Base - Tar|, |End - Tar|)$. If $DIS < MDIS$, which is a predefined threshold, it is recognized as a good locality. In our current implementation, $MDIS$ is set to 8, identical to the minimum number of prefetching pages. Ideally, the minimum prefetching size would be 0 or a small number to minimize disk transfer time, but considering the mechanical properties of magnetic disks, the minimum window size is adhered to the default Linux policy, which is 8 currently.

Providing the definition of locality, ASP can calculate how many pages to prefetch this time. 8 more pages are prefetched than last time when there shows a good locality, until a maximum prefetching limit (MPF) is reached, which is 32 pages as a good balance of prefetching and memory sharing. On the contrary, if the desired page content is located outside the $MDIS$ region, or no track entry can be found, we simply read ahead minimum number of pages.

3.1.3 Double Slot Prediction

Further investigation shows that when a guest is dominated by one process, the anonymous pages of those periodically waked up service routines may get swapped in and out thrashingly. Since the kernel threads share the PIDs of user space processes, ASP mistakes them for the regular processes, resulting in the confusion of entry information. Besides, as a guest is simply a regular process in the view of KVM, the sparsely spreading swapping activities make our prediction of locality less accurate.

So we improve the original proposal with double slot prediction as shown in Figure 1b, that is, each entry is split into two slots, each has its own $Base$ and End pointers, yet share the same PID and Pri. Another field, $slot$, is added to make ASP aware of which slot was used the last time. When calculating the locality, ASP tries first with the $slot$'s information, and then the other one. If neither indicates a good locality, both slots' parameters are reset to the default ones, and the default number of pages are prefetched, otherwise the corresponding slot's record is updated and more pages is prefetched until MPF is reached.

3.2 Host Swapping Notifier

The design of Host Swapping Notifier (HSN) is straightforward: A shared memory is established to notify the guest of guest frame numbers (GFNs) whose corresponding physical page frames have been swapped out or faulted in by the host. The guest periodically checks the shared memory and hides the pages that have been swapped out by the host, or rescues pages when they are faulted in from the host's swap space. By this cooperation between the host and the guest, the double paging anomaly can be eliminated.

3.2.1 Shared Memory

More precisely, when a guest boots up, it sets up a piece of shared memory, and then tells the host the start GFN and the size of the shared memory through a hypercall. The shared memory is depicted by Figure 2, which consists of three parts; first one is the notifier header, and the other two are the swap-out GFN buffer (OB) and the swap-in GFN buffer (IB) respectively. The header contains the information needed to manipulate the buffers, including the offsets of the buffers to the beginning of the shared memory, the size of each buffer, and the head and tail pointers, which the host updates the former and the guest the latter. The host answers the hypercall by mapping the shared memory

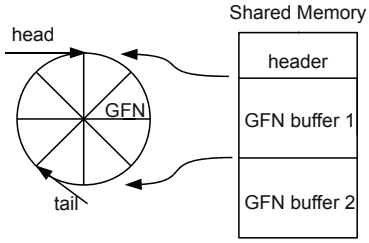


Figure 2: A design of shared memory. When the host produces a GFN in the buffer, it increases the head pointer by 1, while the guest increases the tail pointer by 1 when it consumes one. The buffer is empty when the head pointer equals tail pointer.

to a contiguous virtual address.

3.2.2 GFN Tracking and Notifying

Once the shared memory is established, the host and guest continue to execute as usual, making their own decisions to select victim pages and do swapping, with addition that HSN will examine every paging operation and guest will regularly check the shared memory. When host swapping happens, HSN has to tell whether the page belongs to some guest. If it does, HSN then puts the guest frame number (GFN) into one of the buffers in the shared memory. When the guest gets scheduled, it will notice that there are pending GFNs in the buffers, and starts to handle them. For pages that have been paged out by the host, the guest will check their states to make sure that they can be hidden from memory reclamation procedures. If nothing is wrong, the guest hides the corresponding page, otherwise the guest just continues to handle the next one. By this means, the memory reclamation procedures of the guest OS will not see the pages that have already been evicted by the host, and have to choose other pages to steal, which probably still exist in physical memory. As a result, the double paging anomaly can be avoided. As for the pages that are faulted in by the host, all the guest needs to do is simply unhide them.

It should be noted that since the hidden pages are not backed up by physical memory while still occupy it from the guest's sight of view, processes may get killed brutally when the guest is out of memory (OOM). However, considering the performance gain and the controllability of OOM, we still believe it is worth adopting HSN.

4. EVALUATION

For our test we use a server equipped with 4-core Intel Q9300 CPU, 4GB of memory and 320GB 7200RPM SATA disk. Memory of both host and guest is limited to simply the tests, and it is overcommitted as well to better evaluate our approaches.

4.1 Evaluation of ASP

To evaluate ASP, we turn off the swapping in the guest OS to exclude the influence of double paging, which is evaluated separately. Then we deploy the SpecJBB benchmark on three guests, each of 320MB of memory and 230MB Java heap. The result is shown in Table 1, from which we can conclude that the performance increase can be obtained by around 18% using our ASP. Table 1 also indicates that the disk scheduler has an impact on system performance. By changing the default CFQ scheduler to Anticipatory, system performance increases by 6%.

Table 1: Impact of ASP and Disk Scheduler on System Performance in Terms of SpecJBB Score

Hypervisor	Disk scheduler		
	CFQ	Anticipatory	Impr.
KVM	3417	3632	6.3%
ASP patched	4045	4287	6.0%
Impr.	18.3%	18.0%	-

Table 2: Impact of Prefetching Size on Swap Cache Hit Rate

Page Cluster	No. of prefetching pages					Hit rate
	8	16	24	32	Total	
3	15642	-	-	-	15642	92.2%
4	-	18031	-	-	18031	83.3%
5	-	-	-	21852	21852	71.2%
ASP	7490	6296	2021	2963	18770	90.6%

To explain how ASP works, we implement a tracing tool, MMTRACE, to record some useful information when benchmarks are running, including the number of pages swapped out and in, swap cache hits, etc. Swap cache hit rate is an important indicator of prefetching effectiveness, and is calculated using $\frac{N_{hit}}{\sum_{i=1}^{N_{in}} PN_i - N_{in}}$, where N_{in} and N_{hit} denote the number of swap-in activities and swap cache hits respectively, and PN denotes the number of pages swapped in each time. As indicated by Table 2, swap cache hit rate is inversely proportional to the prefetching size (which can be modified by changing the value of $/proc/sys/vm/page-cluster$ and is evaluated as $2^{page-cluster}$), and more irrelevant pages are faulted in as the prefetching size increases. By using ASP, only those pages related to the faulted ones are likely to be prefetched, leading to less IO transfers and higher swap cache hit rates.

4.2 Evaluation of HSN

We first illustrate how HSN solves the double paging anomaly by assigning a same workload on different system setups, which we abbreviate as follows:

- KVM-1200 — a 1200-MB guest running on a 512-MB host with guest swapping disabled.
- KVM-768 — a 768-MB guest running on a 512-MB host with guest swapping enabled.
- HSN-768 — a 768-MB guest running on a 512-MB host with guest swapping enabled and HSN patched.

Each guest runs a Sysbench benchmark with BLOCK_SIZE set to 1GB; host swapping is enabled in all cases.

The breakdown of disk transfers recorded on the host side are shown in Figure 3, from which we can have a deep insight of how HSN works. Because system memory is overcommitted in all cases, a minimum number of pages need to be swapped out to successfully execute the benchmark. This minimum number can be expressed as the total blocks written out when only the host swapping is enabled, that is, the value of *Disk-out* in the case of KVM-1200. When guest memory is reduced and guest swapping is enabled, the

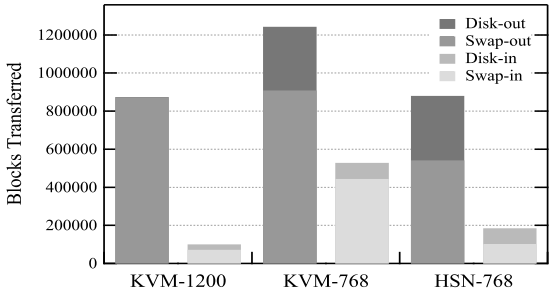


Figure 3: A comparison of disk transfers recorded on the host side for the Sysbench workload in different system setups. Blocks swapped out and in by the guest can be expressed approximately as *Disk-out* − *Swap-out* and *Disk-in* − *Swap-in* respectively.

Table 3: Impact of HSN on Benchmark Performance

Running Sysbench and SpecJBB separately		
	Sysbench (seconds)	SpecJBB (score)
KVM	51.4	4364
HSN	32.6	5492
Running Sysbench on two guests concurrently		
	Guest 1	Guest 2
KVM	96.9s	99.2s
HSN	44.8s	49.6s

amount of host swapping, namely *Swap-out*, should reduce as well since the guest will help reclaim memory. However, despite the help provided by guest (which can be calculated as *Disk-out* − *Swap-out*), *Swap-out* is not significantly reduced in the case of KVM-768, which is a good indicator of double paging. On the contrary, the close values of *Disk-out* in KVM-1200 and HSN-768 imply that double paging is mostly eliminated at the help of HSN, since no more disk transfers are involved. However, although HSN can guide the guest not to select the pages that have been evicted by the host, the guest still needs to steal from other pages to reclaim memory to run the benchmark, that is why KVM-768 and HSN-768 have the same differences between *Disk-out* and *Swap-out*.

We evaluate HSN by first running Sysbench and SpecJBB separately to trigger double paging, the performance boost is around 36% and 26% respectively, as indicated by the first part of Table 3. To further explore the potential of HSN, we design a workload with heavier double paging. In this experiment, two guests, each with 300MB of memory, are hosted by hypervisors with 512MB of memory. Both guests run a Sysbench benchmark with BLOCK_SIZE set to 500MB concurrently to trigger double paging. The execution time is record in the second part of Table 3. As the result shows, HSN can cut the runtime by half, which means a much graceful performance degradation when the system is under heavy memory pressure.

5. CONCLUSIONS AND FUTURE WORK

To gracefully reduce the performance degradation in virtualized environments, we propose the adaptive prefetcher and GFN notifier to help relieve the pain caused by host swapping. Experimental results based on our KVM implementation show that both solutions can reduce the performance degradation when the system is busy host swapping.

There are several ways we can improve our design. The kernel and KVM module will be studied more carefully to simply the implementation of HSN so that the guest kernel won't be affected. The impact of ASP will be examined further to eliminate cache pollution and prefetch wastage, which are common problems of prefetching.

6. ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China under grant no. 60970125 and National Grand Fundamental Research 973 Program of China under grant no. 2007CB310900.

7. REFERENCES

[1] K. Chew and A. Silberschatz. On the avoidance of the double paging anomaly in virtual memory systems. 1992.

[2] W. Fengguang, X. Hongsheng, and X. Chenfeng. On the design of a new linux readahead framework. *SIGOPS Oper. Syst. Rev.*, 42(5):75–84, 2008.

[3] B. S. Gill and L. A. D. Bathen. Amp: Adaptive multi-stream prefetching in a shared cache. In *FAST '07: 5th USENIX Conference on File and Storage Technologies*, pages 185–198, Berkeley, CA, USA, 2007. USENIX.

[4] R. P. Goldberg and R. Hassinger. The double paging anomaly. In *AFIPS '74: Proceedings of the May 6-10, 1974, national computer conference and exposition*, pages 195–199, New York, NY, USA, 1974. ACM.

[5] C. Li, K. Shen, and A. E. Papathanasiou. Competitive prefetching for concurrent sequential i/o. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 189–202, New York, NY, USA, 2007. ACM.

[6] K. OHMACHI, T. NISHIGAKI, and S. TAKASAKI. Analysis of pawp/vms: Paging algorithm to prevent double paging anomaly in virtual machine systems. *Journal of information processing*, 4(2):55–60, 19810715.

[7] M. Schwidefsky, H. Franke, R. Mansell, H. Raj, D. Osisek, and J. Choi. Collaborative memory management in hosted linux environments. In *OLS '06: 2006 Ottawa Linux Symposium*, pages 313–328, 2006.

[8] D. Su, W. Chen, W. Huang, H. Shan, and Y. Jiang. Smartvisor: towards an efficient and compatible virtualization platform for embedded system. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pages 37–41, New York, NY, USA, 2009. ACM.

[9] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[10] Z. Zhang, A. Kulkarni, X. Ma, and Y. Zhou. Memory resource allocation for file system prefetching: from a supply chain management perspective. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 75–88, New York, NY, USA, 2009. ACM.