A Transparent Remote Paging Model for Virtual Machines

Haogang Chen, Yingwei Luo, Xiaolin Wang, Binbin Zhang, Yifeng Sun
Department of Computer Science and Technology, Peking University, P.R.China, 100871
Zhenlin Wang
Department of Computer Science, Michigan Technological University, Houghton, MI 49931, USA

Abstract

In virtual machine systems, with the increase in the number of VMs and the demands of the applications, the main memory is becoming a bottleneck for the application performance. To improve paging performance and alleviating thrashing behavior for memory-intensive or I/O-intensive virtual machine workloads, we proposed *hypervisor based remote paging*, which allows a virtual machine to transparently use the memory resource on other physical machines as a cache between its virtual memory and virtual disk device.

The goal of remote paging is to reduce disk accesses, which is much slower than transferring memory pages over modern interconnect networks. As a result, the impact of thrashing behavior can be alleviated since the average disk I/O latency is reduced. Remote paging also benefits some I/O intensive applications, resulting in a better-than-native performance.

Our remote paging model is totally transparent to the guest operating system as well as its applications, and is compatible with existing techniques like ballooning, hypervisor disk cache and page sharing. A combination of them can provide a more flexible resource management policy.

1 Introduction

In recent years, virtual machine(VM)[1, 2, 9] technologies are becoming more and more popular due to its support for resource encapsulation, hardware independency and easy manageability[5]. Most of current research focuses on how to multiplex hardware resources so that many VMs can run simultaneously on a single physical machine. As a result, resources available for a VM is usually restricted within a physical machine boundary.

In a virtual machine system, the memory resource is virtualized in the manner of partitioning. With the increase in the number of VMs and the demands of the applications, the main memory will become an extremely limited resource. If the memory requirements of a virtual machine exceeds the available amount, the VM may suffer severe performance degradation due to thrashing[13]. Various techniques have been developed to improve memory resource efficiency in virtual machine systems, including memory sharing[2], ballooning[19] and demand paging. These techniques can either reduce memory footprint or adjust memory allocation among VMs within the same physical machine. But none of them helps when all the VMs used up their memory.

Other existing work also aims at integrating hardware resources in a distributed environment using virtualization. For example, vNUMA[3] can create a virtual NUMA multiprocessor system over cluster of workstations. Taking the approach of distributed shared memory (DSM), vNUMA makes the memory resource on each node accessible for other nodes through a shared address space. Unfortunately, simulating shared address space in the system level imposed a significant overhead. The virtual machine monitor (VMM) has to maintain memory coherence among nodes, which is very expensive for write-shared pages[14]. More importantly, the unawareness of the underlying distributed architecture from the guest operating system (guest OS) can magnify the overhead, since it does not know how to exploit data locality.

It has been observed that performance thrashing on many server systems comes from *bursty requests*[7, 17], that is, memory requirements proliferate only for a short period of time. Generally, it is unnecessary to span the entire system over several machines, but better to temporarily "borrow" some memory from other physical machines to overcome the burst. In this way, most computation remains on the original machine, so locality is maintained.

This paper proposes *hypervisor based remote paging*. It allows a virtual machine to transparently use the memory resource on other physical machines as a cache between its virtual memory and virtual disk device. A paging request from a VM is intercepted by the hypervisor, and is preferably satisfied from remote machines (called *memory servers*). The goal of remote paging is to reduce disk accesses, which is much slower than transferring memory pages over modern interconnect networks.

Virtual machine systems can benefit from the remote paging in many ways. First, this model speeds up VM disk I/O transparently so that the impact of thrashing behavior can be alleviated. Second, some I/O intensive applications also benefit from remote paging, resulting in a better-than-native performance. Third, it can be used together with existing techniques like ballooning, hypervisor disk cache and page sharing, to provide a more flexible resource management policy. Additionally, our approach is more cost efficient than other methods, such as migrating and vNUMA, since it respects the principle of locality and does not require the remote memory server to have strong computation power.

Our work inherits the ideas of previous studies on *remote paging* in multicomputer systems[6, 12]. Remote paging allows a node to use the memory on other nodes as its fast backing storage. When a node is out of memory, a set of pages will be sent and cached on remote memory servers, rather than directly going to the disk storage. The new caching layer introduced by remote paging architecturally lies between a node's local memory and its disk storage (usually on I/O nodes).

Our hypervisor-based remote paging can be entirely implemented within the VMM, without any modification to the guest OS. Moreover, we do not introduce another level of paging in the hypervisor. Victim pages are selected by the guest OS itself. A page swap in guest OS will result in disk I/O that can be intercepted by the VMM. Under the intervention of VMM, the paging request is transparently extended to remote memory servers. In other words, the remote cache can be considered as a second level cache of VM's virtual storage. Section 2 describes the characteristics and design choices of our hypervisor-based remote paging model, and investigates some cache policies for the remote cache. A hash-comparing based technique is proposed to support eviction-based remote cache in our work. Section 3 details our implementation of a prototype of our remote paging model in Xen hypervisor.

2 The Remote Paging Model

In most virtual memory systems, victim pages (i.e. the pages to be evicted) are selected by the paging routine itself. Therefore, an intuitive way to implement remote paging in virtual machine system is to initiate paging in the hypervisor. Because VMM controls the translation from guest-physical-address to the corresponding machine-address, this method will automatically extends VM's memory space to a remote machine, giving guest OS the illusion of a larger physical address space.

However, in virtual machine environment, introducing another level of paging raises several unique performance problems: First, VMM cannot wisely choose victim pages due to lack of information on page usages[19]. The second problem is *double paging*[9]. Because page replacement in the hypervisor is transparent to the guest OS, it is likely that the guest OS will choose to reclaim a page that has been paged out by VMM. This will cause the page to be read in just for writing back to the VM's swapping area again. The third one is *redundant write back*. If an unallocated page is selected by VMM as a victim page, it will unnecessarily write back the page content[10]. Even worse, when an access time-based (such as Least Recently Used, LRU) algorithm is used in hypervisor, such choice is very likely to happen because unallocated pages are usually inactive.

Some artifices can be used to walk around certain problems. For example, VMware ESX Server[19] uses a randomized page replacement algorithm to alleviate bad interaction with guest OS; Cellular Disco[10] avoids double paging by remapping guest disk blocks to hypervisor's swap area. However, these methods do not fundamentally solve the above problems. These counter-effects in all can likely offset the performance benefits gained by remote paging.

2.1 Basic Design

Rather than introducing another level of paging in the hypervisor, our design let the guest OS select victim pages itself, according to its native replacement policies. This idea is common with *ballooning*[19], which can eliminate the anomaly described earlier.

A page swap in the guest OS will result in disk I/O that can be intercepted by the VMM. The VMM maintains a fast cache on remote machines, called *memory servers*. A disk request that hits a cached block is directly satisfied from the remote memory server, which provides a lower latency than a real disk access. In this way, a new level of caching is transparently added into virtual machine's memory hierarchy. Figure 1 illustrates this design.

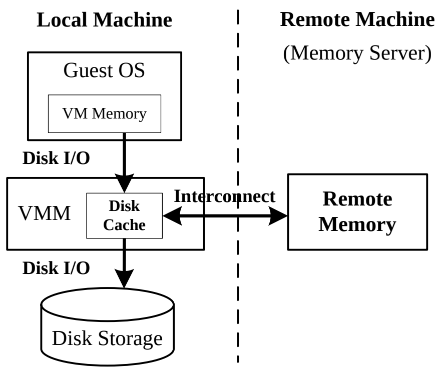


Figure 1: Basic design of hypervsior-based remote paging. In this paper, we assume the virtual machine uses the local storages on the same physical machine.

Our remote paging model lies between the virtual machine's memory and its virtual storage devices. More specifically, if the guest OS also has its own buffer cache, the remote cache becomes a second level cache of the VM's virtual storage. There are many studies focusing on low level buffer cache management[4, 20]. Some recent work also discusses how to make use of second level buffer cache in virtual machine systems[13, 15]. However, our remote paging model has its own characteristics, which result in some unique design choices:

- Our design should use as little memory as possible on the local machine. Since our goal is to improve thrashing behavior, consuming too much memory in the VMM is meaningless.
- The remote cache lies neither on the same machine with its upper level (the VM memory), nor on the same machine with its lower level (the disk storage). In other words, the overhead of cache data exchange in either direction is non-trivial.
- The distribution of disk cache on different physical machines introduced reliability risk. Some means must be taken to to ensure data persistence for fault tolerance.

In the next two subsections, we will show how these characteristics affect the design choices of our remote paging model.

2.2 Page Placement Policy

Page placement policy defines when a block should be placed into the cache. There are two common page placement policies: *access-based* placement and *eviction-based* placement. We have implemented both.

Access-based placement Access-based placement places a missed block into the cache when it is being accessed. Implementing access-based placement is straitforward. For a read, the missed request is passed to the storage device as usual. As soon as the block data is ready, it is copied to a send buffer built in the hypervisor, waiting to be sent to the memory server soon. Meanwhile, the guest OS is acknowledged about the I/O completion. For a write, the data is copied to the send buffer, and the guest OS is immediately acknowledged. The data will be actually written to the storage device sooner or latter, depending on the write policy, which we will discuss in the section 2.3.

Eviction-based placement Eviction-based placement puts a block into the cache when it is evicted from the upper level cache (the buffer cache in the guest OS in our case). Previous studies[4, 20] have showed that eviction-based placement is more suitable for lower-level buffer cache, since it can provide a better hit rate. However, adopting eviction-based caching into our design faces two major challenges.

First, VMM should decide from where the evicted block is loaded. The block data can either be loaded from disk, or directly from the guest memory. Although both methods require the page content to be transfered to the memory server, loading from disk introduces additional disk accesses, thus result in a higher miss penalty. In contrary, if the guest's storage resides on another machine (e.g. NFS server), reloading data from the storage is worthwhile, because data exchange can be done by memory server and the file server, without the intervention of local machine (i.e. the one running guest OS).

Second, it is difficult to detect buffer cache eviction of guest OS in a transparent way. Chen *et al.*[4] presented a tracking table based approach that can detect evictions at reuse time. In their approach, the hypervisor tracks storage block associates with each memory page, and updates the tracking table at every disk access. A block number mismatch when updating means a previous eviction.

However, in virtual machine environment, a memory page may be reclaimed by the guest OS for the purposes other than buffer cache. In this case, tracking table based approach will fail to detect the eviction since no further disk access will be made on this page. For example, a clean buffer cache page can be recycled silently, then serves as the heap memory for an application. Even worse, the application may latter modify the its content. Thus, when the same page is reused as buffer cache again, it may contain obsolete data which is inconsistent with the storage block recorded in the VMM, making it impossible to load the page content from guest memory.

Geiger[13] improves the accuracy of tracking table based approach by monitoring other activities from guest OS such as page faults, but its correctness is not absolutely guaranteed. Lu *et al.*[15] walk around the above problems by modifying the guest OS to explicitly notify the hypervisor when a clean page is evicted (before reuse). However, their method contradicts our transparency goal.

Since our purpose is to reduce disk accesses, imposing additional I/O operations in the VMM is not a wise choice. In our model, a page always enters the remote cache from guest memory. To avoid data inconsistency, we only admit those pages whose contents are identical with the corresponding disk blocks. Unlike [15], we do not modify the guest OS to achieve this. When a disk block is accessed, a hash code is calculated according to the page content. This hash code is recorded in the tracking table, indexed by the page frame number (pfn). When an eviction is detected by the hypervisor, another hash code is generated for the the evicted page, using the same algorithm. By comparing the code with the previous one stored in the tracking table entry, we can determine whether the evicting page is dirty, as illustrated by Figure 2.

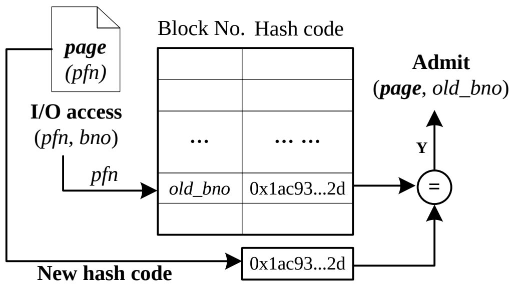


Figure 2: The modified tracking table with hash code. An unchanged hash code means the page still hold the data of the block recorded in the tracking table entry.

However, this approach does have limitations. Although, by carefully choosing a hash algorithm, the probability of hash collision for non-identical pages is negligible[18], it is not yet risk-free. Some work[11] argues that compare-by-hash is inapplicable when data integrity is crucial. Moreover, this method does not guarantee every page evicted by guest OS to be admitted into the cache, thus has negative effects on hit ratio. We are investigating other means to overcome these limitations.

2.3 Reliability Considerations

The distribution of disk cache in the remote paging model also introduces reliability or fault-tolerance problems. A faulty memory server can cause data loss. Previous work proposes several solutions. For example, a *mirroring* approach[12] replicates a caching page on multiple machines. The cost of mirroring is that it requires more physical machines and multiple page transfers. A *parity* approach[16] is similar to the well-known RAID technique, which keeps checksum values on remote server instead of mirrored contents. Both of these methods are designed for clusters but too expensive in our environment.

We address the problem by applying different write policies on the disk cache. In our model, the hypervisor level disk cache can be either *write-back* or *write-through*.

To implement a write-through cache, the VMM commits a intercepted write operation to the physical device in parallel with transferring its content to the remote cache. The guest is notified only *after* the content is actually written to the the physical device to ensure data persistency. Write-through policy does not introduce additional latency, since the transferring to the remote cache is fully overlapped with the normal disk write.

However, write-through policy does impose I/O overhead. With write-through, it is also impossible to improve the latency of disk writes by using remote cache. This is not a problem since in most guest OS, disk writes usually overlap with the computation and will not lead to guest suspension. For example, the Linux performs write-back of its in-memory disk cache in a background task. In contrast, disk reads usually bound the application because applications cannot continue without the requested data. The fact that most I/O intensive application is bounded by disk reads rather than writes means that improving latency on disk reads is more crucial in our design.

Obviously, when reliability is not a significant concern, the write-back policy should be used to improve performance. A write-back cache does not commit block content to physical storage until a dirty block is evicted from remote cache. At that time, the page content is transferred back to the local machine, and the VMM initials disk writes to commit the change.

2.4 Memory Server Design

The architecture of memory server is quite flexible. It can be a dedicate machine running the memory service, or a machine running both the service and other virtual machines. In the later case, if the VMM supports ballooning, the memory service can dynamically change the physical memory share with other VMs. Actually, the service acts like another type of “balloon” who can steal memory across physical machine boundary.

The memory service is in charge of storing block contents of the paging machine, responding to its page in/out requests, as well as handling cache eviction when the server runs out of space. A cache block for a particular virtual block device can be indexed by the block number.

The memory service controls its own cache replacement policy, without the regard for the paging machine. However, what happens after the victim block is selected differs from the write policy. If write-through is used on the paging machine, the server can simply discard the evicting cache block since its content should be identical with the remote storage. Under write-back policy, however, the server should be notified about the dirty status of its cached blocks, so that it initiates a transfer when the evicting block is dirty.

2.5 Memory Server vs. Live Migration

It is necessary to compare remote paging with *live migration*[5]. Although both of remote paging and live migration require the cooperation of more than one physical machines, our remote paging model is more suitable for many scenes.

In the case of bursty requests[7], resource requirements proliferate only for a short period of time. It is not worthy to migrate the *entire* system to another machine, and soon moving it back. By contrast, remote paging provided transparent utilization of remote memory, while keeping majority of resources on the local machine.

More importantly, migration not only shifts machine status but also computation tasks to the target machine. So the target machine is expected to have at least the same computation power (e.g. CPU and bus power) as the original one. But in the case of remote paging, the only requirements for memory servers are sufficient free memory and adequate network bandwidth. This is more cost efficient, since modern off-the-shelf PCs usually meet such requirements.

Additionally, remote paging also gives the opportunity to use distributed memory resources on multiple machines. But it is pretty hard for live migration to break the physical machine boundary.

In summary, live migration and remote paging are targeted at different goals. Live migration aimed to load balancing and quality of service (QoS), while remote paging targets at bursty request and low-cost resource integration. We also suggest that the combination of the two may lead to a more flexible resource management solution.

3 Prototype Implementation

We have implemented a prototype of remote paging model on Xen virtual machine monitor (version 3.1.0). The prototype consists of two modules: *local module* and *memory server module*.

Since every disk access from guest OS passes through the back-end block driver, we modify it to implement the local module. Important data structures maintained in the module includes: a cached block list, which keeps track of all disk blocks currently cached by the remote machine, with their dirty status; a tracking table with hash code, which is consulted to detect page evictions; and a send buffer and a receive buffer, which are used to exchange page content with the remote server module.

Both the access-based and eviction-based placements described in section 2.2 are implemented in the prototype. For the eviction-based placement, SHA-1[18] is used to calculate the hash code. On the memory server, we simply use LRU as the cache replacement algorithm: a newly accessed block is added into the end of cached block list, and an evicting block is always at the head of the list. For reliability and convenience, a write-through policy is used. The memory server controls the cache replacement on its own. There is no interaction between the memory server and the local machine when a replacement occurs.

The memory overhead of our implementation is low. On a 64-bit machine, both cache block entry and tracking table entry occupy 28 bytes. For a machine with 1GB physical memory and 1GB remote cache, the data structures consume about 15MB RAM on the local machine (assume 4KB page size). The send buffer and receive buffer use additional (but fixed amount of) pages, depending on their capacities. In practice, we found that 4MB of send buffer and 2MB of receive buffer is adequate.

Our page transferring protocol starts with a sequence of block headers, describing the block addresses to be transferred or retrieved. For transfer, the headers are followed by corresponding page contents. This protocol allows bursty transferring of multiple pages to improve throughputs.

Besides cache replacement policy and protocol handling routines, implementing the memory server module is strait-forward. In our implementation, it is a user level process who allocates pinned memory from OS to form the cache. To support fast looking up of block data for a given block number, a hash table is also maintained.

4 Preliminary Evaluation

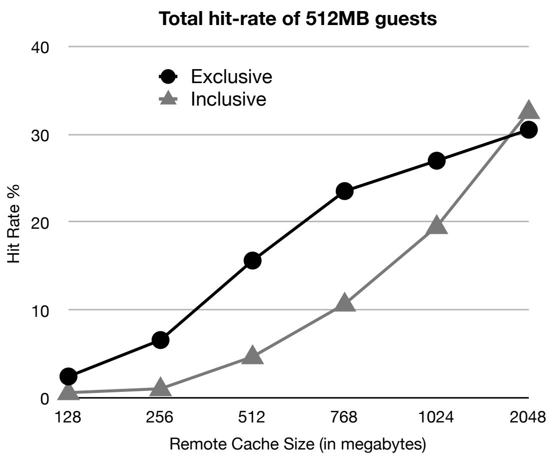
We did some preliminary evaluation and studied on the correctness and cache hit rate on our prototype of remote paging model described in section 3. The goal of the evaluation is to prove the feasibility of our idea. At the time of writing, we are still improving the prototype, and it is not yet strong enough to perform a complete performance evaluation.

4.1 Experimental Platform

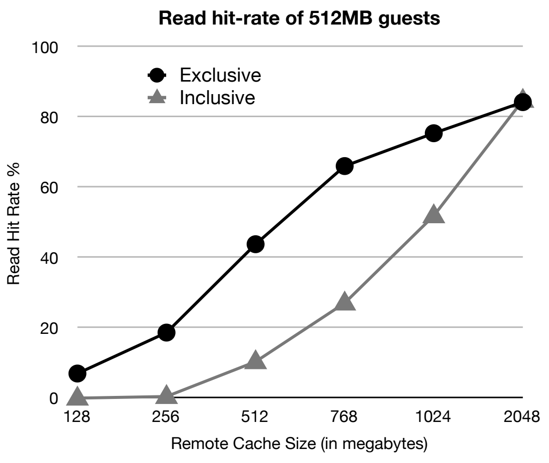
Our experimental platform consists of two PCs. One is the local paging machine and the other runs the memory server. Each of the PCs has an Intel Core 2 Duo 6300 1.86 GHz processor, 2 GBytes main memory and a Seagate ST3160811AS 160 GBytes SATA hard disk. We installed Xen-3.1.0 on the local machine, patched with our local module implementation. Its driver domain (Domain 0) is configured with 512MB RAM, and the guest domain for testing is set to various memory sizes, as will describe below. Both the Domain 0, guest domain and the remote memory server runs SUSE Linux Enterprise Server 10 linux distribution, with kernel version 2.6.18. The two machines are connected to a D-Link DES-1016D 1 Gbps switch by Realtek RTL8168B network adaptors on both sides.

4.2 Kernel Compile Benchmark

In this benchmark, we compile the linux kernel (version 2.6.18) in the guest domain, using gcc version 4.0.1 and an all-yes-config (make allyesconfig). The allyesconfig leads all modules to be compiled and included in a single kernel images. As a result, the total working set is about 2.1 GBytes, which far exceeds the configured guest memory of 512 MBytes, leading to guest paging.



(a)



(b)

Figure 3: Remote cache hit rate of the kernel compile benchmark for various remote cache sizes and two different cache placement policies. The second figure decouples disk reads from the total disk accesses.

Figure 3 shows the hit rate of the remote cache in the kernel compile benchmark for various remote cache sizes and two different cache placement policies. From the top figure (a), we can see that the total hit rate improves as the size of the remote cache increases, and the exclusive cache generally out performs the inclusive one. The result confirms some previous researches in that exclusive remote cache can make more efficient use of the second-level storage cache.

However, the hit rate showed in Figure 3 (a) is relative low. Even when we have 2048 MBytes of remote cache, which is equivalent to an infinite cache for our workload, the hit rate is only 32.7%. The reason is that most disk accesses in kernel compiling are disk writes that generate object files. These writes result in large amount of inevitable compulsory misses. Meanwhile, only the reuse of these object files happens in the linking stage contribute to cache hits.

As described in Section 2.3, we should only focus on read hit rate because our write-through policy does not provide performance improvement for disk writes. Figure 3 (b) decouples disk reads from the total disk accesses to provide an insight on the potential performance gains of the remote paging model. From Figure 3 (b), we can see that exclusive remote cache has significant advantages over inclusive one for relatively small cache sizes. When the remote cache size ranges from 256, 512 to 768 MBtyes, the read hit rate for the exclusive cache is 18.7%, 43.8% and 66.1% respectively, or 38.7, 4.2 and 2.4 times higher respectively than the inclusive one.

4.3 Quicksort Benchmark

The kernel compile benchmark examines how the remote cache behaves when the guest OS scans through regular disk files in a sequential manner. We also runs *quicksort* micro-benchmark to show the efficiency of remote paging model on the swap area.

The quicksort benchmark reads 1 GBytes of random integers from the disk, then sort them in application's heap memory, using qsort() function provided by standard C library. Without regards to the initial data loading phase, the working set size of this benchmark is about 1 GBytes. If the working set is larger than the physical memory granted to the guest OS (768 MBytes in our experiment), application will suffer severe thrashing behavior by accessing its swap partition constantly. Unlike kernel compile benchmark, the access pattern on the swap partition is quite arbitrary and exhibits some temporal locality.

The result showed in Figure 4 indicates that our remote paging model performs even better under this workload. When the remote cache size ranges from 256, 512 to 768 MBtyes, the exclusive cache produces read hit rate at 70.2%, 88.9% and 99.5% respectively. This means that with 512 MBytes of remote cache, nearly 90 percent of paging request can be satisfied from remote cache, without actually reading the physical disk.

The efficiency of the remote cache in this benchmark lies in the fact that on a swap area, a data block is never read before being written to. This property eliminates all compulsory read misses on the remote cache. Moreover, since a swap area usually does not impose data persistency requirement, we can aggressively adopt write-back policy to improve the write latency.

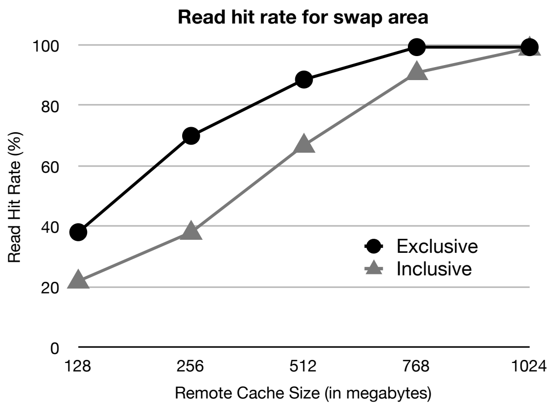


Figure 4: Remote cache hit rate of the quicksort benchmark for various remote cache sizes and two different cache placement policies. The guest is configured with 768 MBytes RAM and the working set size of this workload is roughly 1 GBytes.

5 Related Work

Our work is inspired by *remote paging* ideas presented in[6, 12], which are originally designed for multicomputers. Remote paging model can take advantages of the fast interconnect in multicomputer systems to improve the paging performance on a node, by using memory on other nodes as its fast backing storage. Some work shows that remote paging over low bandwidth interconnection (like Ethernet) may speed up the execution time for real application as well[6, 16]. With the even widen performance gap between switched network and magnetic disks, remote memory model remains a cost effective way to improve system performance.

Dahlin *et al.* extends this idea and proposes *cooperative caching* for file servers[8], which coordinate the file caches of many client machines to form a more effective overall file cache. It is shown that cooperative caching significantly reduced disk accesses and improved file system response time.

Most previous work aims at multicomputers, and is requires changes to the operating system. Our hypervisor-based remote paging enables virtual machines to utilize memory resources on other physical machines, without modifying the guest OS. This transparent fashion provides a more extensive application scope, and also introduces new design challenges on eviction page selection and cache placement policies.

Chen *et al.*[4] have showed that an eviction-based lower-level cache can provide higher hit ratio than access-based one. They also presented a tracking table based approach that can detect evictions at reuse time, without modifying client software. In a virtual machine environment, however, the guest OS does not usually use a dedicated buffer cache, making it difficult to precisely detect guest evictions.

Some recent works also investigates how to support eviction-based cache in VMM. Geiger[13] improves the accuracy of tracking table based approach by monitoring other activities from guest OS such as page faults. Their approach can be integrated into our design. Lu *et al.*[15] proposes *hypervisor exclusive cache* to support miss ratio curve prediction in the hypervisor. But their solution for eviction detection is not fully transparent to the guest OS.

6 Conclusion

In this paper, we proposed *hypervisor based remote paging*, which allows a virtual machine to transparently use the memory resource on other physical machines to meet the demands of its applications. The remote paging model takes advantages of the ever increasing speed gap between disk accesses and the data transfer over modern interconnect networks, providing the VM a lower latency of virtual disk access.

Our work differs from previous researches on remote memory access in that by doing the work in VMM layer, our method is fully transparent to the operating system and its software. Rather than introducing another level of paging in the VMM, the borrowed memory on the remote memory server acts like a cache between the VM's virtual memory and its virtual disk device. This avoids many anomalies imposed by multi-layered, uninformed resource management. But it also introduced some design challenges.

We discussed about various design decisions in our remote paging model. For eviction-based placement policy, we investigated several method to transparently detect page eviction from the guest operating system. We proposed a hash code aided tracking table approach to detect clean eviction pages at the reuse time. We also compared different write policies in our model, presenting the trade-off in performance, reliability and implementation complexity for both write-back and write-through remote caches.

We have implemented a prototype of remote paging model on Xen virtual machine monitor. Both the access-based and eviction-based placements are included in the local module of the prototype. The prototype uses write-through policy to ensure data persistency. A simple memory server is also implemented, but it currently only supports a single virtual machine as its client.

The prototype is able to run a SUSE 10 linux distribution (kernel version 2.6.18) with remote disk cache working correctly. Our prototype implementation and preliminary evaluation proves the feasibility of our idea. At the time of writing, we are still improving the prototype, and it is not yet strong enough to perform a complete performance evaluation.

7 Acknowledgement

This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No.2007CB310900, National Science Foundation of China under Grant No. 90718028, National High Technology Research 863 Program of China under Grant No.2008AA01Z112, MOE-Intel Information Technology Foundation under Grant No.MOE-INTEL-08-09, and HUAWEI Science and Technology Foundation under Grant No.YJCB2007002SS.

References

[1] 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. *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[2] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[3] M. Chapman and G. Heiser. Implementing transparent shared memory on clusters using virtual machines. *Proc. 2005 USENIX Techn. Conf.*

[4] Z. Chen, Y. Zhou, and K. Li. Eviction Based Cache Placement for Storage Caches. *Proc. of the USENIX Annual Technical Conf.*, pages 269–282, 2003.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, May 2005.

[6] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. *Proceedings of the USENIX Summer Conference*, pages 127–135, 1990.

[7] M. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: evidence and possiblecauses. *IEEE/ACM Trans. on Networking*, 5(6):835–846, 1997.

[8] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: using remote client memory to improve file system performance. *Proceedings of the 1st conference on USENIX 1994 Operating Systems Design and Implementation Proceedings*, pages 19–19, 1994.

[9] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[10] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Trans. on Computer Systems (TOCS)*, 18(3):229–262, 2000.

[11] V. Henson. An analysis of compare-by-hash. *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 13–18, 2003.

[12] L. Iftode, K. Li, and K. Petersen. Memory servers for multicomputers. *Compcon Spring'93, Digest of Papers.*, pages 538–547, 1993.

[13] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. *ACM SIGOPS Operating Systems Review*, 40(5):14–24, 2006.

[14] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[15] P. Lu and K. Shen. Virtual Machine Memory Access Tracing With Hypervisor Exclusive Cache. *Proceedings of the USENIX Annual Technical Conference*, pages 29–43, 2007.

[16] E. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. *Proceedings of the Annual Technical Conference on USENIX 1996 Annual Technical Conference table of contents*, pages 15–15, 1996.

[17] C. Ruemmler and J. Wilkes. *UNIX Disk Access Patterns*. Hewlett-Packard Laboratories, 1992.

[18] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, 2002.

[19] C. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(si):181, 2002.

[20] T. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. *Proc. of the USENIX Annual Technical Conf.*, pages 161–175, 2002.