2009 IEEE International Symposium on Parallel and Distributed Processing with Applications

REMOCA: Hypervisor Remote Disk Cache

Haogang Chen*, Xiaolin Wang*, Zhenlin Wang†, Xiang Wen*, Xinxin Jin*, Yingwei Luo* and Xiaoming Li*
*Peking University
{hchen, wxl, wxever, jxx, lyw, lxm}@pku.edu.cn
† Michigan Technological University
zlwang@mtu.edu

***Abstract*—In virtual machine (VM) systems, with the increase in the number of VMs and the demands of applications, the main memory is becoming a bottleneck of application performance. To improve paging performance for memory-intensive or I/O-intensive workloads, we propose the *hypervisor REMOte disk CAche* (REMOCA), which allows a virtual machine to use the memory resources on other physical machines as its cache between its virtual memory and virtual disk devices.**

The goal of REMOCA is to reduce disk accesses, which is much slower than transferring memory pages over modern interconnect networks. As a result, the average disk I/O latency can be improved. REMOCA is implemented within the hypervisor, by intercepting guest events such as page evictions and disk accesses. This design is transparent to the applications, and is compatible with existing techniques like ballooning and ghost buffer. Moreover, a combination of them can provide a more flexible resource management policy. Our experimental results show that REMOCA can efficiently alleviate the impact of thrashing behavior, and also significantly improve the performance for real-world I/O intensive applications.

I. Introduction

Virtualization can provide efficient resource encapsulation, hardware independency and easy manageability. Nowadays, it is widely used in data centers to consolidate multiple workloads into a single physical platform [1], [2]. In a virtual machine system, memory is partitioned among Virtual Machines (VMs). With the increase in number of VMs and the demands of applications, the main memory will become an extremely limited resource.

Various techniques have been developed to improve memory resource efficiency in virtual machine systems. For example, memory sharing [3] can discover and share identical memory pages in a physical machine, and thus reduce the total memory footprint; ballooning [1] allows the hypervisor (also known as *virtual machine monitor*, VMM) to dynamically adjust memory allocation among multiple VMs. However, none of these methods help when all the VMs are in tight memory situations. In such circumstance, the VM will have no choice but to page to its virtual disks, which may result in severe performance degradation [4].

It has been observed that performance thrashing on many server systems comes from *bursty requests* [5], [6], that is, memory requirements proliferate only for a short period of time. In these circumstances, it is unnecessary to reconfigure the physical system with larger memory, or to migrate the entire virtual machine to another host. To maintain the service locality and low cost, it is desirable to temporarily "borrow" some memory from other physical machines to overcome the burst.

In order to alleviate thrashing behavior for virtual machines, this paper proposes REMOCA, or *hypervisor remote disk cache*. It allows a virtual machine to transparently use the memory resources on other physical machines to form a cache between its virtual memory and virtual disk devices. A paging request from a VM is intercepted by the hypervisor, and is preferably satisfied from the remote cache. The goal of REMOCA is to reduce disk accesses, whose latencies are 1 to 3 magnitudes higher than transferring memory pages over modern interconnect networks.

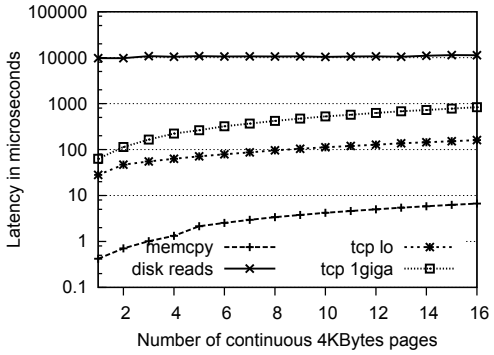

Fig. 1. Round-trip latency comparison between disk reads and network transfers.

Figure 1 compares the round-trip latency of requesting a block of contiguous data from the disk (disk reads) and over 1 Gbps Ethernet using TCP connections (tcp 1giga). When the data size is 4KB, the latency of 1 Gbps Ethernet is about 2 magnitudes lower than the disk. The access latency for the disk reads does not change notably when the requesting size increases because the disk access latency is dominated by the seek time. Also, the disk controller can optimize sequential accesses through prefetching. On the contrary, the latency of network transfers is independent of locations. Thus it increases linearly with the requested data size. As a result, the gap drops to 1 magnitude when the data size reaches 64 KB (16 pages). To separate the latency of data copy, protocol processing and the interconnection, we also measured the results of TCP latency over loopback interface (tcp lo) and the latency of memory copy (memcpy). The results also show that both of them increase linearly with the size of data.

Our work inherits the ideas of previous studies on *remote*

978-0-7695-3747-4/09 $25.00 © 2009 IEEE
DOI 10.1109/ISPA.2009.27

paging in multicomputer systems [7], [8]. Adopting remote paging into a virtual machine environment introduces new challenges because there are two independent resource management layers. To avoid unpredictable interactions with the guest OS, we choose a cache design rather than introducing another level of paging in the hypervisor. We also use the exclusive cache policy [9], [10] to improve the efficiency of our hypervisor remote disk cache — REMOCA only caches the pages that are evicted from the guest OS.

Virtual machine systems can benefit from REMOCA in many ways. First, it speeds up the VM disk I/O so that the impact of thrashing behavior can be alleviated. Second, some I/O intensive applications can also benefit from remote caching, resulting in a better-than-native performance. Third, it can work together with existing techniques such as ballooning, ghost buffer [11], [12] and miss ratio curve prediction [4], providing a more flexible resource management policy. Lastly, our approach is cost efficient — it neither relies on specialized interconnections nor requires the remote memory server to have strong computation power.

II. The design of REMOCA

In the following section, we will present the basic design of REMOCA, then describe our choices on the cache organization, the placement policy and the write policy. We will also enumerate possible design options of the memory server, and illustrate how the client VMM and the memory server interact through the REMOCA protocols.

A. Overview

In REMOCA, the VMM maintains a fast disk cache on remote machines, i.e. *memory servers*. The cache architecturally lies between the virtual machine's memory and its virtual storage devices. Since most guest OSes also have the ability to cache recently accessed disk blocks in their own memory (known as *guest page cache*), our remote disk cache can be considered as a second level cache in the VM's virtual storage hierarchy.

When a disk access misses in the guest page cache, the guest OS will try to swap-in the requested disk blocks. A page swap in the guest OS will result in disk I/Os that will be intercepted by the VMM. If a disk I/O hits our remote cache, it is directly satisfied from the remote memory server, which provides lower latency than a real disk access. In this way, a new level of caching is transparently added into the virtual machine's memory hierarchy.

Figure 2 illustrates this design. Note that the hypervisor only manages the cache index structures on the local machine, while the actual cache content resides remotely. Although the general idea is simple, REMOCA has some unique characteristics that may lead to special design choices:

Exclusive cache placement: the guest OS will do its own page cache management as well. Caching a block that has already been cached in the guest memory is inefficient. However, achieving exclusiveness in REMOCA is complicated because the behavior of guest page cache is a black box to the hypervisor. We will discuss the exclusive cache placement further in Section II-C.

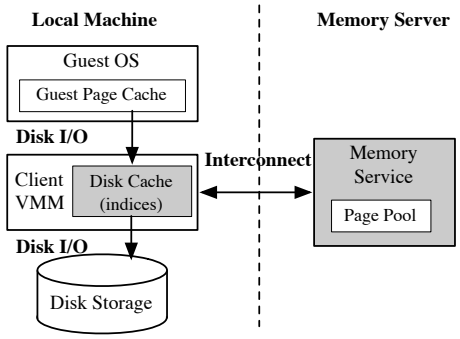

Fig. 2. The basic design of REMOCA. The shaded portion shows the key components. It consists of a local module running in the client VMM and a memory service running on another machine.

Load cache content from the guest memory: in conventional cache hierarchy, data usually enters the cache from the lower-level storage. But in REMOCA, loading cache content from the disk is very impractical since it introduces extra I/O overheads. Therefore, we always load cache data from the upper-level storage (the guest page cache).

Use write-through to enforce data persistence: the distribution of disk cache on different physical machines introduces reliability risk. In general cases, REMOCA uses write-through policy to ensure data persistence and provide fault tolerance. In Section II-D, we will further discuss trade-offs on write policies.

B. Cache Organization

A cache block in REMOCA is identified by a *block number* (*bno*), which consists of a physical device id and a sector number. REMOCA always manages its cache block in a page granularity. Any fragmented or misaligned disk I/Os from the guest OS are ignored by REMOCA and passed to the disk.

In our remote disk cache, all cache blocks are organized into a queue. A newly admitted disk block is appended to the queue tail. When the cache gets full, the victim block is selected from the head. Because the guest OS will append the newly accessed block into its own LRU queue, we also remove a block from our remote cache when it is hit. In this way, our hypervisor remote disk cache and the guest page cache can form a unified LRU cache.

C. Exclusive Cache Placement

Previous studies [9], [10] show that the exclusive cache design can provide a better hit ratio for lower-level storage caches. To make the remote cache more efficient, we adopt the exclusive cache design: REMOCA admits a disk block into the remote cache when it is evicted from the guest page cache.

Unfortunately, in a virtual machine environment, it is difficult to detect page cache evictions of the guest OS in a transparent way. When a page in guest's page cache is to be

evicted, the guest OS just simply modifies its internal data structures (like linking the page descriptor to system's free list), leaving the VMM uninformed. Given the above problem, we employ the idea described by Lu and Shen [4], which modifies the guest OS to explicitly notify the hypervisor when a page is evicted from the guest page cache (before reuse). We also maintain an one-to-one mapping between memory pages and disk locations in the VMM, which is queried upon page eviction to obtain the associated block number from an evicting page number.

D. Write Policy

The distribution of the disk cache also raises reliability or fault-tolerance problems. A faulty memory server can cause data losses. Previous work proposes several solutions [8], [13], but they are designed for clusters and too expensive in our environment.

We address the problem by applying *write-through* policy on the disk cache. The VMM commits an 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 disk access latency, since the transfer to the remote cache is fully overlapped with the normal disk write.

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

E. Memory Server Design

The memory server is in charge of storing block contents of the cache, and responding to cache management requests from the client VMM. The memory for storing cache content is called *page pool*. A page in the page pool is indexed by a block number (*bno*) specified by the client VMM at cache admission. A memory server can serve multiple client VMMs in separate sessions. And we assume that a dedicated page pool is allocated for each client.

The architecture of memory server is quite flexible. It can be 1) a dedicated remote machine running the memory service; or 2) a symmetrical remote machine running both the memory service and other virtual machines (Figure 3).

The dedicated model is simple, and can provide best service response time. In the case that large memory requirements are temporary, this model is preferable to live migration because it is more cost-efficient: the memory server is not required to have strong computation power to perform the computation

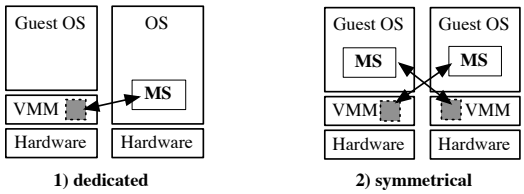

Fig. 3. Different architectures of the memory server. MS stands for memory service. Gray boxes and arrows show the corresponding REMOCA module in the client VMM

of the client node. In Section IV-C, we will investigate how a dedicated memory server can speed up I/O intensive applications and thrashing memory intensive applications.

The symmetrical model is useful to exploit idle memory resources in a virtualization cluster. It can be combined with ballooning to provide a more flexible quality of services (QoS) both within and across physical machines. A drawback of this model is that the memory server's response time will become unpredictable due to the scheduling of other concurrent services.

F. Client-Server Interactions

In this subsection, we will describe the REMOCA protocol between the client VMM and the memory server. We don't give a full protocol description, but illustrate interaction in response to key events.

Cache resizing: to make the protocol work, the client VMM and the memory server should make an agreement on the cache size. This is done by sending a SETSIZE($npages$) message to the server. If the memory server could not allocate sufficient pages for the page pool, it will reply with an error so that the client can either reduce the requested size or contact another server. Client VMMs are also allowed to dynamically expand or shrink the remote cache by sending a SETSIZE message at any time.

Cache admission / update: cache admission is triggered by guest evictions. After setting up the local index structure, the client VMM sends a SENDPAGE(bno, NIL, $data$) message to the server, where bno is the key to the block including the device id and the sector number, and $data$ is the block content (in a page size). Upon receiving the message, the memory server allocates a new page from the client's free page pool, records bno as its key, then copies block data into that page. If the block is already cached in the page pool (this may happen when our cache is not strictly exclusive), its content is updated.

Cache read hit: when a disk read for block bno hits the remote cache, the VMM will allocate a request id ($reqid$) to identify the request, then send a REQUEST(bno, $reqid$) message to the server. The server looks up bno in its page pool, retrieves the page content, and replies to the client with a SENDPAGE(bno, $reqid$, $data$) message. Upon receiving the reply, the client VMM finds the pending request according to $reqid$, verifies bno, copies $data$ to the specified guest buffer and acknowledges the I/O request. After that, the VMM moves

bno's cache index to the head of the queue, making it next to discard.

On rare occasions, the server may reply with an ERROR ($reqid$) message, indicating that the requested block cannot be found. If the client VMM gets this message, it will delete the orphaned cache index and fall back to normal disk I/O. Note that we assume the connection between the VMM and the memory server is reliable and ordered so that messages seldom get lost. The server does not reply to SENDPAGE or DISCARD (describe later) requests. Since we currently use write-through, errors happened to them can always be recovered in the REQUEST stage.

Cache eviction: an eviction on the remote cache happens when: 1) trying to admit a new cache block and the queue is full, so that the block at the queue head has to be evicted; or 2) an access that hits the remote cache, so we have to discard the cached version to keep the cache exclusive. In either cases, the VMM will send a DISCARD(bno) message to the server. The server will locate the page identified by bno and return it to the free pool.

Under this protocol, the cache size, cache placement and replacement policies are completely controlled by the client VMM. Write-back policy can also be implemented by sending an additional REQUEST message before discarding the block.

III. Prototype Implementation

We have implemented a prototype of REMOCA on the Xen hypervisor [14] (version 3.1.0). A virtual machine in Xen is called a *domain* (DomU). There is a special domain that provides I/O services for other domains, namely domain 0 (Dom0). Our prototype consists of two modules: the *local module* and the *memory service module*.

Since every disk access from DomU passes through the back-end block driver in the Dom0 (*block-be* in Figure 4), we extend it to implement the local module. Important data structures maintained in the module include: a cached indices list, which keeps track of all the disk blocks currently cached by the remote machine; a hash map, which supports fast looking up for a block in the cache indices; a tracking table ($p2s$) and a reverse mapping table ($s2p$), which are consulted upon page evictions; and a request buffer, which is used to queue and track pending requests to the memory server.

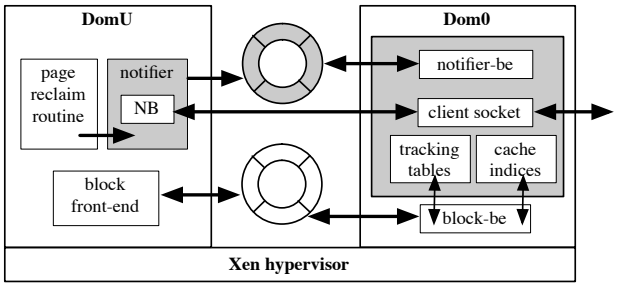

Fig. 4. Structure of the local module in the REMOCA prototype. The shaded areas show our implementation. The arrow indicates the control flow or data flow between each component.

A. Handling Guest Evictions

We modify the page reclaim routine in the DomU to implement the eviction notification. We set up a shared ring buffer between each DomU and the Dom0. An event channel is also established for the notification. This architecture follows the "split device model" [15] of other para-virtualized device drivers in Xen. We also allocate a pool of pages in the DomU, namely the *notification buffer* (NB in Figure 4). The notification buffer is shared with Dom0 through the standard grant table mechanism in Xen and is mapped by the Dom0 once at guest startup.

When the page reclaim routine in the DomU recycles a page in the guest page cache, it allocates a page from NB, and copies the content of the page to be evicted into it. Then, the page's machine frame number is put into an entry in the shared ring buffer, together with the handle of the allocated NB page. The notifier back-end (*notifier-be* in Figure 4) in the Dom0 is notified through the event channel after a batch of pages are processed. The back-end driver processes page evictions in the ring buffer asynchronously in a separate thread. It sends block contents to the memory server using the copies in the shared NB pages and clears tracking table entries as necessary.

B. Cache Management

Each I/O page passed to the block back-end driver is filtered through our local module. For a read, if the requested block hits our cache, we allocate an entry from the request buffer and send the request immediately. We receive server responses in a separate thread. When a response arrives, we look up the request buffer to find the corresponding request. Then the page data is assembled directly into the I/O page provided by the DomU. After all the segments of a read request are processed, either through our remote cache or through real disk I/Os, we acknowledge the block front-end driver using the original mechanism. Since we use the write-through policy, a write request is directly passed to the block layer. Finally, for either read or write, the tracking table is updated for every I/O segment.

Our implementation of memory service module is a user level process which uses `mmap()` and `mlock()` system calls to allocate pinned memory from the OS. The service module also maintains a hash table to support fast lookup of block data for a block number. We use TCP connection between the memory service and the local module. Requests and responses are batched to improve the service throughput.

C. Memory Overhead

The memory overhead of our implementation is low. On a 64-bit machine, the sizes of each cache index and the tracking table entry (plus a reverse mapping hash table) are 40 bytes and 32 bytes respectively. For a machine with 1GB physical memory and 1GB remote cache, these data structures consume about 18MB of the local machine memory (assume 4KB page size). The notification buffer, the shared ring buffer and the request buffer use additional (but fixed amount of) pages,

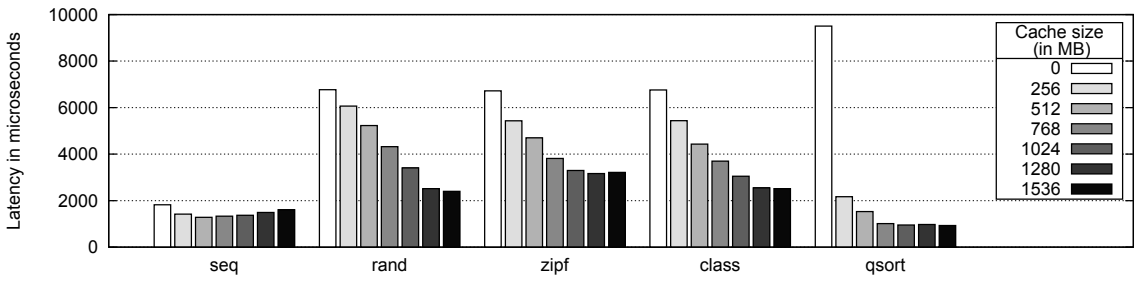

Fig. 5. Mean access latency for each read request of the microbenchmarks.

depending on their capacities. In our implementation, they are about 3MB for each domain.

IV. Evaluation

In this section, we perform experimental evaluations on our implementation of the REMOCA prototype. The goals of our evaluation are 1) to see the potential improvement on the disk access latency using our prototype, 2) to analyze factors that may affect the efficiency of the remote cache, 3) to validate our design of exclusive cache management, and 4) to show how REMOCA can improve the performance for realistic applications.

A. Experimental Platform

Our experimental platform consists of two PCs. One runs the client VMM and the other runs the memory service. Each of the PCs has an Intel Core 2 Duo 6300 1.86 GHz processor, 2 GB of main memory and a Seagate ST3160811AS 160 GBytes SATA hard disk. We install Xen-3.1.0 on the local machine, patched with our local module implementation. The driver domain (Dom0) is configured with 1024 MB of DRAM. The Dom0, the DomU and the remote memory server all run SUSE Linux Enterprise Server 10 Linux distribution, with kernel version 2.6.18. We run the Dom0 and the DomU on different processor cores. The two machines are connected to a D-Link DES-1016D 1 Gbps switch by Realtek RTL8168B network adaptors on both ends. The disk access latency and the network latency of the platform are shown in Figure 1.

B. Access Latency Analysis

In this subsection, we will study how REMOCA can improve the average disk access latency for a virtual machine, and discuss factors that may affect its efficiency. We will also validate our exclusive cache design as described in Section II-C by measuring the read hit ratios of each benchmark. Our evaluation is based on 5 microbenchmarks, which consists of 4 pattern-access tests for guest buffer cache and a *quicksort* program to test the guest swap cache.

In the pattern access benchmarks, we read 12288 MB of data from a 1536 MB dataset stored in a disk file. The dataset is divided into many *records* of the same size, which is configurable. We run the guest OS with 256 MB of memory. This configuration is far less than the total data size of each benchmark, leading guest to initiate paging. Four patterns are chosen to access the records within the dataset, according to [4], [9]:

- *sequential (seq)* — The records are scanned sequentially, from the first to the last. If the amount to read exceeds the size of the dataset, we will wrap to the first record after the last one is scanned.
- *random (rand)* — We select a record randomly, and all the records in the dataset have a uniform probability to be chosen.
- *zipf* — We randomly select the record with a Zipf distribution. That is, record i is selected with a probability proportional to $1/i^{\alpha}$, where $\alpha = 1.0$.
- *class* — We select the records randomly. The records in the dataset are divided into two classes: first $1/10$ of them are 10 times more likely to be selected than the others.

Within each record, we scan through every word sequentially. We assume that the entire dataset is stored contiguously on the physical disk layout. By default, we set the record size to 96 KB (24 pages). A smaller record size can result in a wider latency gap (see Figure 6).

The quicksort (*qsort*) benchmark reads 768 MB of prepared random integers from the disk, then sorts them in its heap, using `qsort()` function provided by the standard C library. When the guest memory is lower than 768 MB, it will suffer severe thrashing behavior by accessing the swap partition constantly.

As described in Section II, the principle of REMOCA is to reduce disk accesses by transferring memory pages through the network. Therefore, the ability of which REMOCA can improve the average disk access latency of the guest OS depends on two factors: 1) the latency gap between the disk and the network; 2) the number of disk I/Os that can be diverted to the remote cache. The latter is determined by the read hit ratio of our remote cache. The former, however, is decided by both the underlying hardware and the application behavior.

Figure 5 shows DomU's average read latency observed in the back-end driver. Each latency is a combination of both remote cache accesses and real disk accesses. We vary the size of the remote disk cache from 256 MB to 1536 MB. A zero remote cache size means that REMOCA is disabled, which is our baseline case. And the "1536 MB cache" case can be

considered as infinite cache size, since it can accommodate the entire dataset for every benchmark. Because REMOCA does not improve disk writes, we will focus on the read latency in the following discussion.

1) The latency gap: We study the latency gap of each microbenchmark by examining two extreme cases: the baseline case and the infinite cache case. In Figure 5, the latency of the baseline case (without the remote cache) varies for different microbenchmarks. For *seq*, the latency is 1.8 ms, which is far below the measured 10 ms in Figure 1. The latencies for *rand*, *zipf* and *class* are roughly the same, which are around 6.7 ms. And for *qsort*, the value is 9.5 ms.

The reason for the above results is that the disk controller can optimize sequential accesses by asynchronously prefetching contiguous blocks into its internal buffer. Also, the buffer cache management algorithm in Linux performs read-ahead. The *seq* benchmark walks through the entire dataset sequentially, thus the effect of these optimizations is maximized, leaving little room for further optimization through REMOCA. Actually, in some extreme case like *seq*, a hit on the remote cache even has negative effects on the mean latency due to the cache management overhead. Such slowdowns can be avoided by disabling REMOCA when the sequential access pattern is observed.

The *rand*, *zipf* and *class* benchmarks read individual records sequentially, so that the disk latency is determined by the record size. Also, due to concurrent requests and disk scheduling, the access latencies are lower than the round-trip latency shown in Figure 1. The amount of sequential accesses in the *qsort* benchmark, however, is insignificant. This is because for a swap cache, the guest OS only reloads pages when they are to be accessed, and the *quicksort* algorithm does not access the memory in a sequential manner.

To validate our analysis, we run the pattern access benchmarks in various record sizes and see how the latency speedup can be affected by the amount of sequential portion in a workload. As shown in Figure 6, the speedup drops for all benchmarks when the record size increases. In summary, the latency gap between accessing the remote cache and the disk can affect the efficiency of our remote cache. REMOCA is more useful when the sequential access pattern is not dominant.

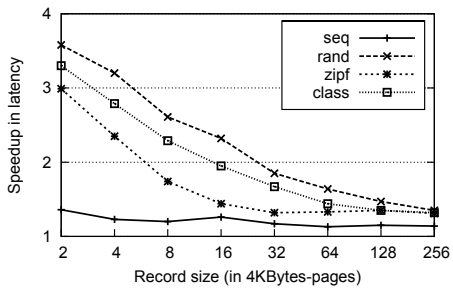

Fig. 6. Latency speedup for various record sizes.

2) Cache hit ratio: Another factor that has an effect on the disk access latency is the cache hit ratio. The hit ratio, in turn, is determined by the cache size, the cache management policy and the application behavior.

The read hit ratios of our microbenchmarks are shown in Figure 7. In general, the hit ratio increases with the cache size. By comparing it with Figure 5, we can see that the improvement of the hit ratio is directly mapped to the reduction in the combined access latency, except for the *seq* benchmark. Note that for clarity, hits on guest OSes' system partition are not included in Figure 7. This is why we see improvements in latency even when the hit ratio is zero.

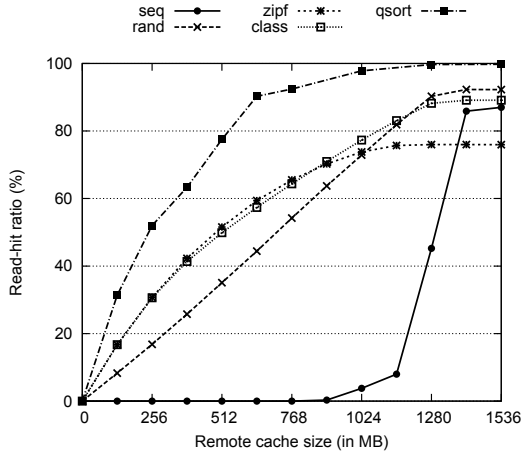

Fig. 7. The corresponding read hit ratio on the remote cache. Note that the result shows only cache hits on the testing partition (i.e. hits on unrelated partitions are not counted).

To see the impact of application behavior on the read hit ratio, we examine the curve of each benchmark individually. The *seq* exhibits a zero hit ratio when the cache size is smaller than 896 MB because a cached block has always been discarded from the LRU queue before it is accessed again, unless the cache is large enough to hold the entire dataset. In the figure, the hit ratio starts increasing rapidly at 1096 MB (before our cache can hold the entire dataset, which is at about 1280 MB) because the LRU algorithm used by the guest page cache is not perfect.

The curve of *rand* is linear since its access pattern is evenly distributed. The curves for *zipf* and *class* benchmarks are alike. They both encounter diminishing returns when the cache size goes larger. This is because the frequently accessed blocks are typically cached in the guest page cache. A larger remote cache only caches the blocks that are less likely to be accessed.

For all the four pattern access benchmarks, the hit ratio gets saturated when the cache size is slightly larger than 1280 MB. As mentioned in Section II-C, the guest page cache and our exclusive remote cache form a unified LRU cache. Ideally, a remote cache of 1280 MB, along with the 256 MB guest memory, is sufficient to hold the entire dataset of 1536 MB for these workloads. But some service processes and shared libraries in the guest OS also occupy some space

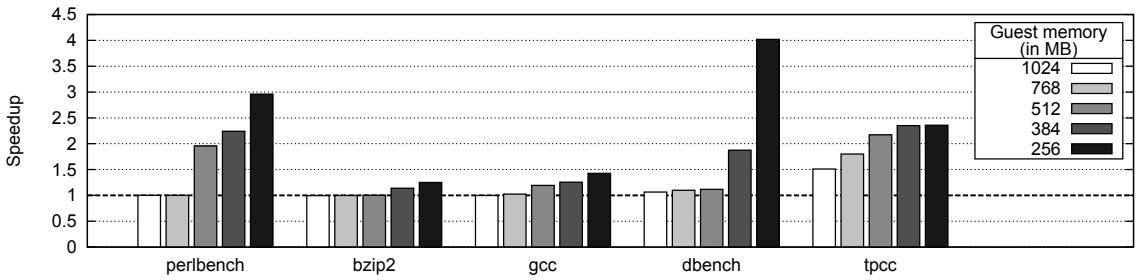

Fig. 8. Speedup for realistic applications. The histogram shows the relative performance gain with REMOCA, under various memory pressures.

in the composed cache, making the saturation point a little larger. Note that the hit ratios at the saturate point are not simply 100% because of cold misses.

Finally, we inspect the effect of REMOCA on the guest swap cache by examining the hit ratio curve of the *qsort* benchmark. The curve of *qsort* is higher than the others for two reasons. First, the dataset is relatively small (768 MB). Second, the guest OS seldom reads a swap block before it is written, so that there is no cold misses on the remote cache. Also, a small remote cache of 256 MB can significantly improve the average read latency for *qsort* (see Figure 5). This is because the access pattern of *qsort* has a better locality on the remote cache. The locality of *seq* and *class* is lower since their sets of "hot" blocks are fixed and are constantly cached by the guest OS.

C. Speedup of Real Applications

In this subsection, we evaluate how REMOCA can speedup some realistic data-intensive applications. Our benchmarks include both memory intensive and I/O intensive applications:

- *SPEC CPU2006* — We select 3 benchmarks from the SPEC CPU2006 benchmark suite [16], which are *perlbench*, *bzip2* and *gcc*. We choose the three benchmarks because they are both CPU intensive and memory intensive. Their maximum working set sizes are 591 MB, 873 MB and 955 MB respectively [17]. When the guest physical memory is low, they will begin thrashing and paging to the swap partition.
- *dbench* — *dbench* is a file system benchmark simulating loads on a network file server [18]. It runs each client with the workload traced from the NetBench suite. In our experiments, we run *dbench* with 80 clients. The total size of the workload is about 3.6 GB.
- *TPC-C* — We choose an open source implementation of the TPC-C online transaction processing benchmark (*tpcc-uva* [19]). TPC-C simulates a population of terminal operators executing Order-Entry transactions against a database. In our experiments, we run the benchmark with 32 warehouses, and the size of the database is 3.2 GB.

In this experiment, we fix the size of the remote cache to 1024 MB but vary the size of the guest memory from 256 MB to 1024 MB. Our intention is to see how these applications behave under different memory pressures and how REMOCA can improve their performance in different circumstances.

The corresponding relative speedups are shown in Figure 8. Because the principle of REMOCA is to reduce the disk access latency, its potential performance improvement will depend on how much time the application is stalled by disk I/Os. In the three SPEC CPU benchmarks, there is little notable I/O overhead unless the guest memory is insufficient to accommodate the entire dataset. When the guest does not thrash (column "1024"), the performance ratios for the baseline case and REMOCA are very close, and there is no speedup.

When the available memory of the guest OS falls below the workload size, all of the applications suffer performance degradation. In these cases, REMOCA delivers performance improvements. The amount of improvements depends on the amount of I/O that imposed by thrashing, which in turn depends on the application's behavior.

Memory-bounded perlbench: the largest workload in *perlbench* is a "diffmail" workload. It compares items in a 591 MB dataset pair by pair. The algorithm scans through the entire dataset again and again, thus the working set size is very large and the working set itself changes rapidly. Actually, this application is memory-bounded since it does little computation. When the memory is tight, this behavior results in considerable page swaps. REMOCA can efficiently alleviate the thrashing behavior by caching the entire dataset in the remote memory. Figure 8 shows that REMOCA provides a 3x speedup when the guest memory is as low as 256 MB.

CPU-bounded bzip2 and gcc: in contrast, the locality of *bzip2* and *gcc* is much better. They work on a small portion of the input at a time, which can be efficiently cached by the guest page cache. For these two CPU-bounded applications, REMOCA still improves their performance by more than 25% at 256 MB.

I/O-bounded dbench and tpcc: the *dbench* and the *tpcc* benchmark are I/O bounded, so that their speedups will mainly be determined by the amount of disk reads that are missed in the guest page cache but cached by REMOCA. This portion will increase when the guest memory is smaller. Although the datasets of *dbench* and *tpcc* are too large to be entirely cached by REMOCA, the performance improvements are significant.

In the *tpcc* benchmark, for example, REMOCA increases the service throughput by a factor of 1.5 for a guest OS with 1 GB memory, and doubles the throughput when the guest memory is lower than 768 MB.

TABLE I
SPEEDUPS IN COMPARISON WITH THE NATIVE RESULTS

Guest Memory (MB)	dbench (MB/s)		tpcc (tpmC)	
	Native	Speedup	Native	Speedup
1024	224.13	1.00	274.37	1.48
768	194.57	1.05	225.12	1.79
512	148.42	1.12	189.90	2.08
384	58.73	1.76	161.10	2.32
256	14.57	3.33	123.13	2.36

Some previous work reports that in Xen, the disk I/O performance for a guest domain basically does not drop to less than 90% of the platform's native performance [14], [15]. This slowdown is far less significant than the speedup provided by REMOCA in tight memory. We also run the *dbench* and the *tpcc* benchmarks in a native environment (without Xen and REMOCA). As shown in Table I, REMOCA can also achieve significant amount of speedup in comparison with the native results for real-world I/O intensive applications.

D. Management overhead

We measured the CPU overhead for cache management in our REMOCA prototype with Xenoprof [20] by counting the "CPU-clock-unhalted" events. Table II shows the percentage of instruction cycles spent in the local module while running the *dbench* benchmark. We can see that the page copying at guest evictions and the cache lookup operations contribute to the majority of the overhead. The copying overhead can be reduced by applying the page exchanging technique available in Xen, and the cache lookup overhead can be alleviated if we increase the size of the hash table. Even without optimizations, the total CPU overhead (about 15.2 %) is acceptable because the applications are I/O bounded rather than CPU bounded.

TABLE II
CPU OVERHEAD OF THE REMOCA PROTOTYPE

Operations	Percentage
Page copying (in guest OS)	5.9481 %
Cache lookup	8.4781 %
Cache append	0.1044 %
Cache delete	0.1590 %
Cache move	0.1287 %
Tracking table operations	0.1116 %
Notification ring processing	0.2777 %

Besides, we observe that the TCP protocol handling and the network adaptor driver in the Dom0 also impose considerable CPU overheads (33.67% and 13.63% respectively). However, these overheads are independent of our implementation. We believe that the efficiency of REMOCA can be further improved by using a more intelligent network device and adopting a lightweight transmission layer protocol.

V. COMPATIBILITY ISSUES

In this section, we analyze the compatibility of REMOCA with two existing virtual machine techniques: ballooning and the ghost buffer. We also discuss how to combine REMOCA with them to provide a more flexible resource management policy.

Ballooning: ballooning [1] provides an efficient way to dynamically adjust memory allocation among multiple VMs. REMOCA is fully compatible with ballooning since it only changes the path of guest disk accesses, leaving the guest memory layout untouched.

Obviously, if there is adequate memory on the same physical machine, ballooning is the top choice to ease memory pressure for a VM. REMOCA is useful when ballooning is inapplicable, for example, when the installation of balloon driver in the guest OS is not allowed, or when the memory allocation has reached the VM's upper limit (the configured physical memory size at VM startup). Additionally, ballooning can help REMOCA. For example, in the symmetrical model presented in Section II-E, ballooning can be used to gather idle memory resources from multiple VMs to form a larger page pool to serve client VMMs running on other physical machines.

Ghost buffer: ghost buffer [4], [11], [12] is a simulated buffer with index data structure but no actual page content. It can be used to predict the VM page miss ratio for memory sizes beyond its current allocation. Actually, REMOCA is equivalent to a guest buffer as long as we do not transfer page contents to the remote server, and always proceed with disk I/O even on a cache hit.

As we have observed in Section IV, not every application can take the advantage of the remote cache. Enabling REMOCA for a non-beneficial virtual machine can unnecessarily impose CPU, memory and network overheads to transfer or store evicted pages. Ghost buffer provides the flexibility to predict the cache behavior before we actually allocate memory pages on the memory server. The client VMM can also use the predicted cache hit ratio to determine an optimal size of the remote cache.

VI. RELATED WORK

Our work is inspired by *remote paging*, which are originally designed for multicomputers [7], [8], [21]. Remote paging model can take advantage of the fast interconnection in multicomputer systems to improve the paging performance on a node, by using memory on other nodes as a fast backing storage. Some studies show that remote paging over low bandwidth interconnections (like ethernet) can speed up the execution time for real applications as well [7], [13]. With the ever widening performance gap between switch networks and magnetic disks, the remote paging model remains a cost effective way to improve system performance.

As far as we know, MemX [22] is the only work that supports remote memory utilization in virtual machine systems. Their idea is to virtualize a volatile block device from the remote memory and assign it to a VM as its swap partition (similar to [21]). A significant limitation of this mechanism is

that it is inapplicable for non-volatile partitions (i.e. buffer caches), so it is awkward to improve performance for I/O intensive applications such as OLTP. Also, their approach lacks of the flexibility to switch on/off or resize when virtual machines are running.

VII. Conclusion and Future Work

In this paper, we proposed *hypervisor remote disk cache* (REMOCA), which allows a virtual machine to use the memory resources on other physical machines as a cache between its virtual memory and virtual disk devices. REMOCA supports both normal disk partitions and swap partitions in the guest OS. We employ exclusive cache management policy to improve the cache efficiency. The write-through policy is used to enforce data persistency and provide fault-tolerance. We also investigate possible design options of the memory server, and illustrate how the client VMM and the memory server interact through the REMOCA protocol.

We have implemented the REMOCA prototype in Xen hypervisor, and performed experimental evaluations on it. We illustrate how the application behavior, cache size and cache management policy can affect the efficiency of our remote disk cache. Our experiments on real-world applications show that REMOCA can efficiently alleviate the impact of thrashing behavior for out-of-core memory intensive workloads. REMOCA can also deliver significant performance improvement to real-world I/O intensive applications even in comparison with the native results.

REMOCA is compatible with existing techniques like ballooning and the ghost buffer. Our future direction is to investigate how the combination of these techniques can provide a more flexible resource management policy for a virtualized cluster.

Acknowledgment

This work was supported in part by the National Grand Fundamental Research 973 Program of China under Grant No. 2007CB310900, National Science Foundation of China under Grant No. 90718028 and No. 60873052, 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. Zhenlin Wang is also supported by NSF Career CCF0643664. Corresponding author, Xiaolin Wang: 86-10-62763373; fax: 86-10-62767883; e-mail: wxl@pku.edu.cn

References

[1] C. Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. si, p. 181, 2002.

[2] P. Apparao, R. Iyer, X. Zhang, D. Newell, and T. Adelmeyer, "Characterization & analysis of a server consolidation benchmark," in *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA: ACM, 2008, pp. 21–30.

[3] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: running commodity operating systems on scalable multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 412–447, 1997.

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

[5] M. Crovella and A. Bestavros, "Self-similarity in World Wide Web traffic: evidence and possiblecauses," *IEEE/ACM Trans. on Networking*, vol. 5, no. 6, pp. 835–846, 1997.

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

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

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

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

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

[11] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pp. 79–95, 1995.

[12] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Geiger: monitoring the buffer cache in a virtual machine environment," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5, pp. 14–24, 2006.

[13] 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*, pp. 15–15, 1996.

[14] 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*, pp. 164–177, 2003.

[15] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the Xen virtual machine monitor," *OASIS ASPLOS 2004 workshop*, 2004.

[16] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[17] D. Gove, "CPU2006 Working Set Size," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 90–96, 2007.

[18] A. Tridgell, "Dbench filesystem benchmark," http://samba.org/ftp/tridge/dbench/.

[19] D. R. Llanos, "TPCC-UVa: an open-source TPC-C implementation for global performance measurement of computer systems," *SIGMOD Rec.*, vol. 35, no. 4, pp. 6–15, 2006.

[20] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the xen virtual machine environment," in *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. New York, NY, USA: ACM, 2005, pp. 13–23.

[21] S. Liang, R. Noronha, and D. Panda, "Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device," *Proceedings of International Conference on Cluster Computing (CLUSTER '05)*, 2005.

[22] M. R. Hines and K. Gopalan, "MemX: Supporting Large Memory Applications in Xen Virtual Machines," *Proceedings of 2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC'07), a workshop in conjunction with Super Computing 2007*, 2007.