Small is Big: Functionally Partitioned File Caching in Virtualized Environments

Zhe Zhang *Han Chen* *Hui Lei*
{zhezhang,chenhan,hlei}@us.ibm.com
IBM T. J. Watson Research Center

Abstract

File cache management is among the most important factors affecting the performance of a cloud computing system. To achieve higher economies of scale, virtual machines are often overcommitted, which creates high memory pressure. Thus it is essential to eliminate duplicate data in the host and guest caches to boost performance. Existing cache deduplication solutions are based on complex algorithms, or incur high runtime overhead, and therefore are not widely applicable. In this paper we present a simple and lightweight mechanism based on functional partitioning. In our mechanism, the responsibility of each cache becomes *smaller*: the host only caches data in base images and a VM guest only caches its own "private data", which is generated after the VM has started. As a result, the overall effective cache size becomes *bigger*. Our method requires very *small* change to existing software (*15* lines of new/modified code) to achieves *big* performance improvements – more than *40%* performance gains in high memory pressure settings.

1 Introduction

File cache management is among the most significant factors affecting a computing system's performance. To hide or mitigate the ever enlarging gap between CPU and disk I/O speeds, the OS and applications normally use large memory spaces to cache file data. Researchers have also proposed running applications entirely in memory in order to eliminate the bottleneck of file access [14]. Although cache space can be used to serve both read and write requests, read requests are mostly synchronous and thus are more sensitive to the effectiveness of cache management. We focus on improving file read performance in the remaining of this paper. Effective handling of write requests is beyond our current work scope.

Optimizing file cache management is not easy. The problem is made even more challenging by the advent of cloud computing. A server in a cloud datacenter suffers from two types of redundant caching. Along the vertical dimension, virtualization leads to a longer file I/O path consisting of nested software stacks [9]. Depending on the caching policy, a file block may be cached in the OS page caches of both the VM guest and the hypervisor host. Besides cache space wastage, this also causes compounded prefetching, which is likely to be overly aggressive. Along the horizontal dimension, different VM guests often contain identical or similar files, such as the OS, database software, utilities, and so forth. However, such content similarity may not be easily exploited.

Redundant caching significantly reduces the effective cache size. This problem is exacerbated when a large number of VMs are over-provisioned on a physical host, which increases competition for the limited cache space. Unfortunately, over-provisioning is a common practice by cloud providers for better cost saving.

A number of techniques have been proposed to mitigate redundant caching in a virtualized environment, and they mainly fall into one of two categories. Cooperative caching mechanisms [6, 10, 11] leverage knowledge on what is cached in other caches to make better cache replacement decisions. This inevitably involves intricate communication among multiple virtual or physical machines. The resultant complexity in development and maintenance causes resistance from cloud operators. Page sharing mechanisms [13, 16, 17] rely on frequent memory scanning or tracing to merge data blocks with identical content. They achieve memory savings at the cost of high processing overhead. In a general-purpose cloud computing system, where workload patterns cannot be predicted, it is difficult to determine whether and when to enable such mechanisms. Due to their respective limitations, neither cooperative caching nor page sharing mechanisms have seen wide adoption in production systems.

We address the problem from a novel angle, by following the principle of *functional partitioning* and making each cache serve mostly distinct portions of data requests. To the extent that different caches contain different content, redundant caching can be minimized or eliminated. Two extreme methods are to use the host cache to serve all requests and completely bypass guest caches, and vice versa. However, the former method causes frequent context switch for guests to access host memory, while the latter does not facilitate the sharing of common blocks across VM guests. We advocate a partitioned caching scheme where the host cache keeps data with high chance of being shared by multiple guests, and each VM guest keeps data that is likely to be used by itself only.

Based on the observation of high content similarity among VM images [5], and the intuitive assumption that two VM guests are unlikely to generate private data blocks with identical content, we assign base image data to be cached on the host, and VM private data to be cached on each guest. [1] This way, the responsibility of each host or guest cache becomes *smaller*, and the overall effective cache size becomes *bigger*. This simple, lightweight method requires only *small* changes to existing OS and virtualization driver (*15* lines of new/modified code in our implementation) to achieves *big* performance improvements (more than *40%* improvement in high memory pressure settings) over policies used in existing production systems.

2 Current Practice

In a virtualization environment, a read request for a file block goes through multiple software layers. Let's first examine the cache performance characteristics and then analyze the caching policies used in current practices.

2.1 I/O Performance Analysis

A read request is first checked against the guest page cache. If it is a hit, the access latency is 1 memory copy from the guest cache to the application space. If there is a miss in the guest page cache, the request is forwarded to the guest virtual disk, and a new cache page in the guest is allocated. This virtual disk read request is translated by the I/O virtualization layer (e.g., qemu virtio) to a read request to the image file at the host.

The host read request is then checked against the host page cache. If it is a hit, the I/O virtualization layer transfers the block from the host page cache to the guest page cache, thus resulting in 2 memory copies in the entire access path. If there is a miss at the host, disk (or network) I/O is initiated to fetch the missing block from the actual file system. In this case, the total latency is comprised of 2 memory copies and 1 physical device transfer.

Additional factors further influence the I/O performance. Current Linux kernels adopt an aggressive prefetching policy, which appends a number of ensuing blocks to the original request. The prefetching action is compounded and amplified with virtualization. Additionally, each guest cache miss generates a system call to the host OS, causing an additional context switch. Our experiments on a typical x86 server show that accessing guest memory is 2 times faster than host memory, and accessing host memory is 10 times faster than host disk.

2.2 Cache configurations

The host cache and the guest cache can be independently configured to be *ON* or *OFF* in the storage hierarchy, resulting in the following caching policies:

Host cache = *ON*, guest cache = *ON* This policy stores duplicate blocks in two levels. When memory is abundant, it may increase the overall cache hit ratio. When memory resource is scarce, the duplicates cause reduced cache utilization and thus overall performance degradation. Moreover, guest level prefetching makes the file access pattern appear sequential even when it is purely random, this cause inaccurate and unnecessary prefetching at the host level. This is the default policy of the commonly used *libvirt* Linux virtualization toolkit.

Host cache = *OFF*, guest cache = *ON* Intuitively, the guest OS has the most accurate knowledge about its I/O pattern, and is therefore the best place to keep page caches. But it fails to take advantage of any free host memory and possibility for content-aware sharing. Experiments suggest that a popular IaaS cloud uses this policy.

Host cache = *ON*, guest cache = *OFF* When the host is fully responsible for I/O caching, guest memory requirement is reduced. In theory, this policy can achieve significant memory saving when cache blocks are deduplicated with content-aware techniques. But with real-world workloads, any benefit may be negated by high computation and context switch overheads.

Host cache = *OFF*, guest cache = *OFF* With this policy, all file requests go directly to the disk. The system suffers from heavy I/O load. No production system would conceivably use this policy.

To the best of our knowledge, current production clouds use one of the two polices with **guest cache = *ON***. One may be advantageous over the other depending on the resource availability of the host machine. But none of them effectively leverages the content similarity among the VM guests to compress their cache space usage.

3 Functionally Partitioned File Caching

We aim to develop a new cache management mechanism that fully utilizes the cache space on each VM guest and the host, while avoiding duplicate caching of identical data blocks. The mechanism is aimed for general-purpose virtualization environments. Therefore, it should be as simple as possible and avoid workload-specific optimizations. It should also be lightweight, because in a resource-overcommitted system, any optimization effort should be extremely cautious about the overhead it incurs.

3.1 High Level Design

Our solution is based on the principle of *functional partitioning*, which avoids expensive bookkeeping by setting up simple rules for each component to easily understand its own portion of tasks or data. One example is the mitigation of OS jitter by aggregating system tasks to dedicated cores in multicore systems [12]. In the context of VM host-guest caching, we aim to design partitioning rules for the host cache and each guest cache.

On the host side, we believe that it is more important to cache data of VM base images. Our previous work [5] has studied the VM image repository of a production cloud

[1] We assume copy-on-write technologies are used to store VM private data separate from base image files.

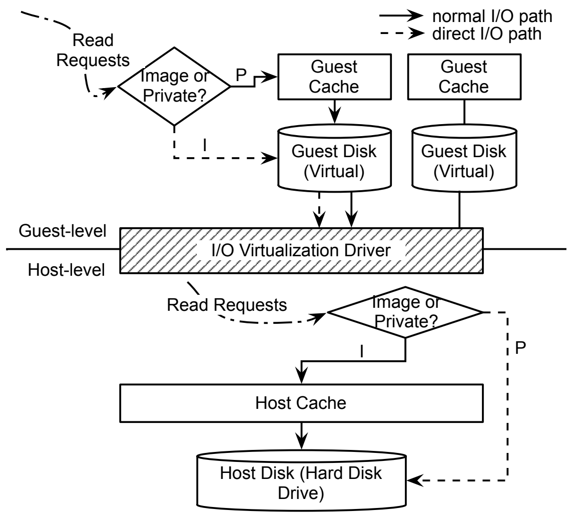


Figure 1: I/O flow with our mechanism

data center, and revealed high content similarity among VM images. If each VM image is broken into 4KB chunks, and all chunks in the repository are deduplicated based on their hash keys, then up to 70% of space can be saved. Even with a 16KB chunk size, this compression ratio can be around 50%. VM image deduplication is done offline when images are captured, and the overhead is hidden from user applications. With this partitioning rule, the base image data has a higher chance to stay in the host page cache and be shared among all guests.

On each VM guest, its private data should have higher priority to stay in the cache. The chance that two individual VM guests generate identical private data blocks is relatively low. Moreover, to detect these identical blocks the system needs to scan VM memory spaces during their runtime. Therefore, private data blocks should be kept in the guest memory for faster access. Keeping private data within individual VM boundaries also enhances performance isolation by preventing one guest from generating a large amount of data and polluting the shared host cache.

In summary, besides turning the cache *ON* and *OFF*, the host cache should have another configuration option *I* indicating that it only caches base *I*mage data. Similarly, we should be able to configure each guest cache as *P* where it only caches *P*rivate data. Each cache should switch between *ON*, *I/P*, and *OFF* options depending on its memory availability and I/O workload pattern. At this stage, we have implemented the *I* and *P* policies for host and guest caches respectively. The adaptive mechanism to switch between them is our future work. Figure 1 illustrates the I/O flow when both *I* and *P* policies are applied.

3.2 Implementation Details

Caching Private Data on the Guest: To enforce a guest to cache only its private data, the first challenge is on determining whether a block belongs to private data or base image. The approach we take is to check the modification time (*m_time*) of the file containing the block of data. If the file has been modified after the guest system launch time (*boot_time*), then we consider the block as private data and access it through the guest page cache. This approximation will cause the guest cache to store part of base image data. We consider it a reasonable compromise to avoid tracking the modification time of each block.

For each guest-level file belonging to the base image, we try to follow the data access path of the *O_DIRECT* system flag and make all data requests to this file bypass the page cache. In particular, a read request is served by sending a direct I/O request to the virtual disk, with the application memory buffer as the DMA destination. The requested data will be transferred from the host via the I/O virtualization driver. Depending on the caching policy on the host, the data will be transferred through memory copy from the host page cache, or through DMA from the host hard disk.

Unfortunately, in the current Linux kernel, in order to use direct I/O, both the size and the offset of the read request have to align with the filesystem block size (512 and 4K bytes are common settings). In the future we plan to investigate how to mitigate these alignment restrictions.

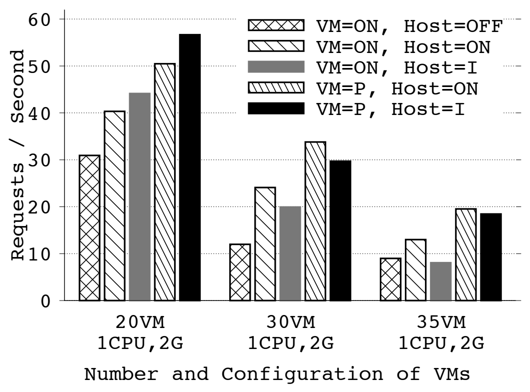
Caching Base Images on the Host: It is relatively easy to enforce the host to cache only base image data. This can be done by opening the copy-on-write file with the *O_DIRECT* flag and the base image file in normal mode. Our implementation is based on the widely used QEMU virtual I/O driver [15]. Currently, when a QEMU copy-on-write (qcow2) file is opened, the base image file that serves as its "backing file" is opened with the same caching flag. We modify the QEMU driver so that the copy-on-write file does not pass the *O_DIRECT* flag to its base file.

Our method assumes that VM images are stored with content-aware storage deduplication techniques, such as [7]. Once the storage layer has compressed the images, the host cache, which resides in the VFS layer above all concrete storage systems, can naturally cache deduplicated image data. Since the deduplication of image data can be done at the repository when new images are checked in, it imposes very small runtime overhead.

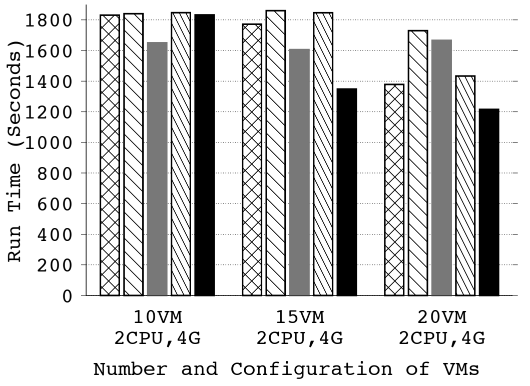
4 Evaluation

We have conducted experiments on an IBM BladeCenter HS22 Server with 24 logical cores (two 6-core Hyper-Threading Xeon E5649) with 64 GB memory, and a 7200 RPM 1TB hard disk. The server runs Linux kernel 2.6.32 and uses KVM as hyperivsor.

We evaluate the aforesaid caching policies using three representative cloud application workloads: *1)* the IBM DayTrader benchmark [4], which represents the traditional transaction processing scenario, where the memory work set is relatively stable and the I/O pattern roughly



(a) DayTrader throughput (higher is better)



(b) Hadoop run time (lower is better)

Figure 2: Performance of DayTrader and Hadoop

follows the Zipf's law [3]; *2)* Hadoop [1] *sort* with 6GB randomly generated data, which represents the data intensive analytics scenario, where a large volume of data is processed in a sequential pattern; and *3)* Linux kernel compilation, which represents a light-loaded development and testing scenario where available memory is much larger than the work set.

For succinctness, we use *X+Y* to denote the configuration in which VMs use policy *X* and the host uses policy *Y*, where *X* and *Y* can be *ON*, *I*, *P*, or *OFF*.

DayTrader: We use mid-size (1 vCPU, 2 GB memory) VMs as DayTrader servers. A set of client VMs hosted on a separate server generates the test workload. The generated workload causes the sever VMs to have working sets larger than the available 2 GB memory. We vary the number of server VMs to create different levels of resource consumption: *1)* with 20 VMs, the host does not overcommit CPU or memory; *2)* with 30 VMs, the host overcommits CPU but not memory; *3)* with 35 VMs, the host overcommits both CPU and memory. Figure 2(a) shows the average throughput of a single server VM.

Among the two policies used by production clouds *ON+ON* leads to much better performance than *ON+OFF*. This is because the VM guest memory is not enough to cover the work set. Therefore, additional caching on the host will increase the overall memory cache hit ratio.

The two policies with $VM = P$ further outperforms *ON+ON* significantly. With $20-30-35$ VMs, the throughput is improved by $25\% - 40\% - 50\%$ with *P+ON*, and $40\% - 23\% - 42\%$ with *P+I*. This is because the base image data is not duplicated in host and guest caches, and each guest can use more memory to cache its own private data. This reduces the number of disk accesses when the system's memory pressure is high ($30-35$ VMs), and reduces the number of host memory accesses otherwise (20 VMs). The largest improvement is achieved at 35 VMs, where the system overcommits both CPU and memory resources.

The matchup between host side policies *I* and *ON* is the outcome of several conflicting factors, including compounded prefetching, the relative importance of image versus private data, and so forth. Neither policy is a clean winner under this workload. In particular, it's not always advantageous for the host to cache all data even when it has abundant memory (20 VM).

Hadoop sort: Due to the heavy workload of Hadoop, we use large size (2 vCPU, 4 GB memory) VMs as Hadoop workers. We use $10, 15, 20$ Hadoop VMs to get similar resource consumption scenarios as in DayTrader experiments. Figure 2(b) shows the sort runtime.

Unlike in DayTrader results, here *ON+OFF* outperforms *ON+ON*, especially with heavy resource consumption (20 VMs). This is because Hadoop generates a large amount of private data with poor temporal locality. With $Host = ON$, many private data blocks are fetched to the host cache and later evicted without being reused. Some prefetched blocks could even be evicted without being used at all. This "cache pollution" reduces usable cache space for data with good temporal locality. Moreover, it wastes the disk bandwidth on unuseful prefetching and causes additional memory copies from the host cache to the guest cache or application. For the same reason, *P+I* has better performance than *P+ON*.

When the system has medium to high memory pressure (15 or 20 VMs), partitioning improves performance over *ON+ON* by reducing duplicate caching. *P+I* applies both policies and performs the best. It also outperforms *ON+OFF* by using a relatively small amount of memory to cache the base image data.

As pointed out in [2], the LRU replacement algorithm in Linux kernel is not ideal for data intensive workloads like Hadoop sort. When better replacement mechanisms are developed and adopted, we expect that our techniques can lead to even greater performance gains, because the saved memory space can be more efficiently used.

VM policy	*ON*	*ON*	*ON*	*P*	*P*
Host policy	*ON*	*OFF*	*I*	*ON*	*I*
Time (sec)	417	433	345	436	440

Table 1: Linux kernel *ext*4 module compile time.

Kernel compilation: We measure the time to compile the *ext*4 module of the Linux 2.6.32 kernel with 5 mid-size (1 vCPU, 2 GB memory) VMs. The Linux source code is in the base image and the output belongs to private data of individual VMs. Table 1 shows the runtime of different policies. Compared to state-of-the-practice policies *ON+OFF* and *ON+ON*, our new policies *P+ON* and *P+I* incur around 5% degradation. Interestingly, the *ON+I* policy has the best performance. This may be due to reduced prefetching of VM private data.

5 Related Work

Cache and memory management in cloud environments has attracted intense research interest in recent years. In this section we survey existing techniques and discuss how our method is related to and distinct from them.

Cooperative Caching: Cooperative caching enables a cache to make better replacement decisions based on the knowledge of the content in one or multiple other caches. A number of cooperative caching techniques have been proposed for virtualize execution environments. [10] proposes memory access tracing through host-level exclusive caching. But it is for predicting guest level memory page miss ratio. In [11], the authors propose a hypervisor cache (*hcache*) that is accessible by VM guests and can catch evicted pages from guest caches. In [6], the XHive system is proposed, which achieves VM cooperative caching.

The communication algorithms used in cooperative caching are relatively complex. Moreover, none of them achieves content-aware sharing of base images across VMs.

Page Sharing: Page sharing techniques [16] scan the memory space and build a hash value for each page. Based on the hash values, pages with identical contents are merged to save physical resource usage. Recent page sharing techniques [13] [17] focus on seizing deduplication opportunities more precisely, in order to maximize the benefit of page sharing.

In principle, page sharing can achieve the largest possible memory saving. However, it does so at the cost of CPU and memory bandwidth overhead on scanning.

6 Conclusion

This paper presents a simple yet effective mechanism to improve application performance in a virtualized environment by reducing file cache duplicates. Under memory stress, the proposed *P+I* policy improves performance by more than 40% over the default *ON+ON* policy, while the widely adopted alternative *ON+OFF* policy delivers inconsistent results. We believe that this is a first step towards many interesting further investigations.

I/O workload profiling To guide the design of the functionally partitioned caching mechanism, it is important to understand the data access patterns of both image and private data. We plan to conduct an extensive study on the I/O workload of common cloud applications.

Fine-grained partitioning: As mentioned in section 3, our current implementation of the *P* policy accepts a read request only if the entire destination file is not modified after system boot, and the offset and size align with 4 KB boundaries. This limits the achievable memory saving. We plan to empirically evaluate mechanisms to track the modification time of each page, as well as methods to mitigate the alignment restrictions.

Adaptive policy switching: Despite the memory saving benefit, the *I* and *P* policies should not always be used. For example, when a VM has enough memory to cover the work set, it should turn its cache *ON* to have minimum access latency. We envision our mechanism to make adaptive decisions to switch between caching policies.

Policy customization: When multiple VMs with different workload patterns are mixed on a host, an interesting problem is how to customize host-side caching policies for each one of them. One potential solution is to create an I/O signature for each image to record the typical I/O pattern of VMs using the image. There should also an upper limit on the amount of cache space that can be used by a single VM's private data, in order to prevent cache pollutions as seen in Hadoop sort experiments.

References

[1] Apache hadoop project. `http://hadoop.apache.org/common/`.

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica. Disk-locality in datacenter computing considered irrelevant In *HotOS '11*.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99*.

[4] IBM DayTrader benchmark application. `https://cwiki.apache.org/GMOxDOC20/daytrader.html`.

[5] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei. An empirical analysis of similarity in virtual machine images. In *ACM Middleware '11*.

[6] H. Kim, H. Jo, and J. Lee. Xhive: Efficient cooperative caching for virtual machines. *IEEE Trans. Comput.*, 60:106–119, January 2011.

[7] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Trans. on Storage*.

[8] Kvm best practices for over-committing processor and memory resources. `http://ibm.co/x3nXlz`

[9] D. Le, H. Huang, and H. Wang. Understanding Performance Implications of Nested File Systems in a Virtualized Environment. USENIX FAST '12.

[10] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *USENIX ATC '07*.

[11] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Paravirtualized paging. In *USENIX WIOV '08*.

[12] P. D. V. Mann and U. Mittaly. Handling os jitter on multicore multithreaded systems. In *IEEE IPDPS '09*

[13] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: enlightened page sharing. In *USENIX ATC '09*

[14] J. Ousterhout et. al. The case for RamClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43:92–105, January 2010.

[15] Qemu virtio driver. `http://www.linux-kvm.org/page/Virtio`.

[16] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, Dec. 2002.

[17] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *ACM VEE '09*.