Per-VM Page Cache Partitioning for Cloud Computing Platforms

Prateek Sharma*, Purushottam Kulkarni† and Prashant Shenoy*
*University of Massachusetts Amherst
{prateeks,shenoy}@cs.umass.edu
†Indian Institute of Technology Bombay
puru@cse.iitb.ac.in

***Abstract*—Cloud computing has become popular for shared hosting of third-party applications. A cloud platform may multiplex virtual machines running different customer applications onto a single physical server, raising the potential for performance interference between such applications. In particular, when a hypervisor shares the file system page cache between virtual machines, as is common in Linux environments, it is possible for one VM to impact the performance seen by other co-located VMs. To address this drawback and improve performance isolation, we design a page cache which is partitioned by VMs. Such a design provides the ability to control fine-grained caching parameters such as cache size and eviction policies individually. Furthermore, the deterministic cache allocation and partitioning provides improved performance isolation among VMs. We provide dynamic cache partitioning by using utility derived from the miss-ratio characteristics.**

We implement our page cache architecture in the Linux kernel and demonstrate its efficacy using disk image files of virtual machines and different types of file access patterns by applications. Experimental results show that the utility-based partitioning can reduce the cache size by up to an order of magnitude while increasing cache hit ratios by up to 20%. Among other features, the per-file page cache has fadvise integration, a scan-resistant eviction algorithm (ARC) and reduced lock-contention and overhead during the eviction process.

I. Introduction

Cloud computing has emerged as a popular paradigm for running a variety of third-party applications. In such an environment, customers lease computational and storage resources from the cloud platform and pay for those resources on a pay-as-you-go basis. Cloud platforms provide a number of benefits such as on-demand allocation of server and storage resources. A typical cloud platform runs customer applications inside virtual machines and multiple virtual machines (VMs) from different customers may be mapped onto each physical server.

Since multiple VMs can be co-located on a server, there is a potential for performance interference between co-located VMs, and the underlying hypervisor must provide performance isolation between these VMs. Modern hypervisors provide strong mechanisms to partition resources such as the CPU—for example, by dedicating CPU cores to VMs or partitioning CPU bandwidth across VMs. While resources such as CPU and network can be partitioned, resulting in performance isolation, not all resources are isolated in this manner. In particular, the in-memory file system page cache is a shared resource in hypervisors, particularly in Linux environments. A page cache is a memory buffer that caches frequently accessed data on disk [29]. Achieving a high hit rate for the page cache is critical for achieving good I/O performance in VMs. However, when the page cache is shared across VMs, it is possible for one VM to cause eviction of cached page belonging to another VM, resulting in higher cache misses and lower performance. Such performance interference can be exacerbated in cloud environments where co-located VMs run arbitrary applications belonging to different customers.

The primary cause for such interference is the use of a unified page cache by the underlying hypervisor, and the use of a LRU-based eviction policy that operates on a single LRU list across all VMs. We argue that unified caches result in poor utilization and interference and the lack of fine-grained control over the page cache contents, size, and eviction policy leads to non-deterministic and suboptimal resource allocation. Furthermore, the LRU policy and unified nature implies that the page cache is susceptible to cache-pollution occurring due to large sequential reads/writes from VM applications. This cache pollution leads to enlarged cache sizes, and decreases the free memory available on the physical server. This reduction in free memory results causes increased memory pressure for the running VMs and forces cloud providers to consolidate a smaller number of VMs on their physical servers.

To address these drawbacks, in this paper, we propose a new page cache design, that logically partitions the page cache on a *per-VM* basis. We call our page cache the *per-VM page cache*. Since each VM gets it own page cache, the hypervisor can provide better performance isolation across VMs. Furthermore, each VM's cache can be managed differently in a manner that is best suited to that VM's application, which can *improve* performance. For example, each VM can have different eviction algorithms depending on their access-patterns, usage, priority, etc. The per-VM page cache allows more fine-grained control over a globally shared resource (the page cache) among applications and users and yields better performance isolation and service differentiation.

Our per-VM cache partitioning technique enables setting the cache size, eviction policy, etc. for each VM individually. In addition to manual control of these parameters from userspace, we have also devised a utility-based cache partitioning heuristic which seeks to minimize miss-ratios. Our per-VM page cache is a drop-in replacement for the existing Linux page cache. We also allow users to control cache behaviour using

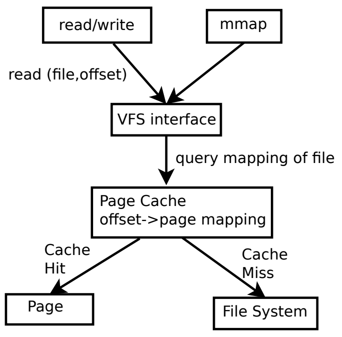

Fig. 1. The logical architecture of the Page Cache interface. File-I/O cache hits get serviced without going through the file-system itself.

the existing `fadvise` POSIX system-call and the `procfs` kernel interface. The VM-based partitioning reduces cache pollution by restricting the cache size of an individual VM. While some of these benefits could be obtained in unified page caches by using containers [16], [23] or control-groups [2], the partitioned-cache yields a much cleaner design and implementation.

In addition to improving system performance by increasing cache hit ratios and providing service differentiation, our per-VM cache can also provide additional benefits. For example: when using KVM [17] to run virtual machines, there exist two levels of the page cache [26]. The guest VMs maintain a page cache in their private address-space, while the host OS maintains a second-level cache which is shared by all the VMs. This setup leads to double-caching — duplicate pages are present in the caches. Since a virtual disk for a virtual machine is a file, the per-file cache allows fine-grained control of the second level host cache in virtual machine hosts. The size of the second-level host page cache for each VM can be specified, along with a special eviction algorithm tuned for secondary caching (such as Multi-Queue [33]). This allows improved I/O performance in the guest virtual machines and smaller host page caches, which allows more virtual machines to be hosted. Furthermore, the cache partitioning also increases the isolation among virtual machines.

We have implemented our per-VM page cache as a small, non-intrusive modification (1500 line patch) to the Linux kernel. We conduct an experimental evaluation of our prototype to demonstrate its efficacy using disk image files of virtual machines and different types of file access patterns by applications. Our experimental results show that the utility-based partitioning can reduce the cache size by up to an order of magnitude while increasing cache hit ratios by up to 20%.

The rest of this paper is structured as follows. We present background on the Linux page cache architecture in Section II. The design and implementation of our per-VM page cache is presented in Sections III and IV. Finally, Sections V, VI, and VII present experimental results, related work and our conclusions.

II. Background & Motivation

Cloud computing platforms are designed to run third-party applications inside virtual machines; each physical server may run one or more virtual machines and the hypervisor is responsible for allocating resources to each co-located VM. There has been significant research on reducing performance interference between co-located VM applications [14]. Placement techniques use intelligent placement to avoid co-locating conflicting application on the same server, thereby avoiding such interference. Resource management techniques employed by the hypervisor can also partition various server resources across VMs to provide performance isolation—VMs may be allocated a certain number of CPU cores, memory size, network interfaces, virtual disks, and I/O bandwidth [13], [28]. By enforcing the allocation of physical resources, the hypervisor ensures that the VMs do not interfere with each other and can run in isolation, as if they were the only users of the resources allotted to them.

In this paper, we focus on Linux-based virtualization, which is common in many popular commercial cloud platforms. In this case, the Linux kernel, which normally functions as the OS, also takes on the role of the hypervisor—virtual machines using a virtualization technology such as KVM or Xen then run on top of this Linux-based hypervisor. Interestingly, while Linux allows partitioning of many common resources such as CPU and network, the page cache is a shared resource within Linux. The page cache is a memory buffer that caches recently accessed disk blocks, which allows VMs to improve I/O performance due to this caching (Figure 1 depicts how file I/O operations use the page cache to enhance performance). Since virtual machines use virtual disks, which are typically files located on the hypervisor's native file system, all disk I/O operations of the VM translate to filesystem operations on the hypervisor(Figure 2). This additional level of indirection for disk I/O operations uses the hypervisor page cache as an additional cache level which sits between the physical disk and the VM.

Consequently, a shared page cache in a Linux-based hypervisor has several performance ramifications for co-located VMs. Most importantly, the shared page cache weakens the performance isolation among VMs, since the disk I/O of all VMs has to contend for the shared cache space. A VM having more data in the hypervisor page cache has higher disk I/O rates, since a larger fraction of I/O requests can be fulfilled with the in-memory hypervisor page cache instead of hitting the much slower physical disk. Similarly, a VM starved of hypervisor page cache may suffer in performance even though sufficient physical resources (CPU, memory) are available to it. Thus, page cache sharing among VMs is a threat to performance isolation in cloud environments.

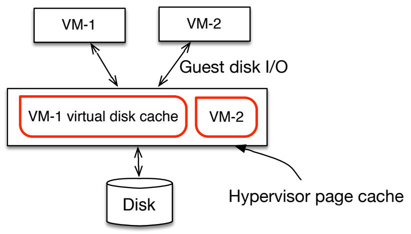

Fig. 2. The hypervisor page cache is shared by VMs.

A. *Linux page cache design*

KVM [17] adds virtualization capabilities to Linux, and the default Linux page cache is used by all the VMs. We shall now see the architectural reasons for the page cache sharing, and also describe some of the other performance issues caused by the existing page cache design.

The Linux page cache implementation is highly sophisticated—it implements a completely lockless read-side page cache [21], and stores the `offset` $\rightarrow$ `page` mapping in a per-inode radix-tree. Multiple files may correspond to a single inode. In Linux, *address-space* refers to a structure referenced by the inode structure. The address-space structure contains the radix-tree mapping. Throughout this paper, we use the terms file, inode, and address-space interchangeably.

Linux has a unified page-eviction mechanism, wherein every page(cached, anonymous, slab) in the system is present in a single logical LRU list. Thus, there is no dedicated LRU list for the file cache pages, and cache pages compete to stay in memory with other pages (belonging to applications, kernel slab caches, etc.). Page eviction is controlled by the swap-daemon (`kswapd`). A variant of the LRU-2 [22] page-eviction algorithm is implemented.

This unified page cache design combined with the LRU-2 eviction results in a large amount of performance problems:

Performance interference: Since every file I/O operation goes through the page cache, every file I/O operation performed by a VM also goes through the hypervisor page cache because the virtual disk of the VM is a file on the hypervisor's filesystem. Thus, the cache occupancy of a VM depends on the rate and access patterns of its disk I/O. Because the hypervisor's page cache is shared between VMs with no constraints, VMs may be starved of cache space and suffer in performance. For example, a VM with a largely sequential disk access pattern (like in the case of media streaming servers) has a larger amount of data going through the page cache when compared to a VM doing random I/O (as is the case for database servers). Thus the performance of the VMs depends on their co-location, something which cloud providers strive to avoid, because they want to provide the same VM performance regardless of the other VMs co-resident on the physical servers.

Lack of QoS knob: Cloud providers and administrators may want to control the allocation of the hypervisor page cache to VMs, and treat it as a quality-of-service knob. The current page cache design prohibits this.

Caching while swapping: The unified LRU lists for page eviction (with all system pages on a single list) presents several problems. The problem of swapping while caching (illustrated in Figure 3) occurs when cache pages are more recent than the swapped anonymous pages. In extreme cases this leads to VMs being killed by the Out-Of-Memory Killer mechanism — even when there exist sufficient memory for the VMs. This problem occurs because of the inability to specify hard limits on the size of the page cache (relative to total memory size). Although the prioritization of cache pages vis-a-vis anonymous pages is dynamically controlled by heuristics in the kernel, it is not always optimum.

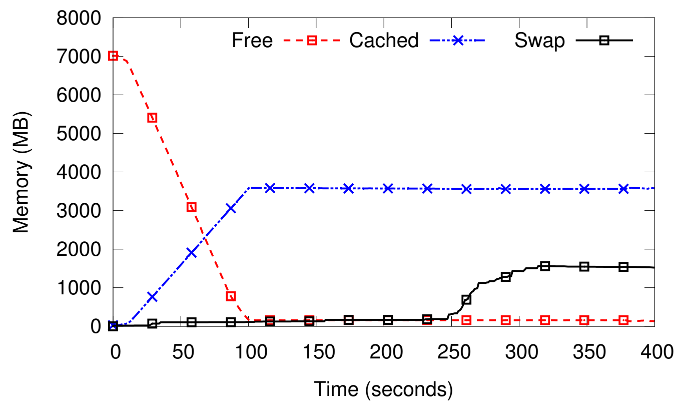

Fig. 3. Memory vs time graph of an I/O intensive workload (from [26]). The system starts swapping pages to disk even in the presence of a large page cache.

Cache eviction algorithm: At the heart of the caching-while-swapping problem is the inadequacy of LRU-2, which is not scan resistant—a sequential read of a large file leads to the cache being "polluted" by the recently read pages. Modern caching algorithms such as the Adaptive Replacement Cache (ARC) [19] or LIRS [15] have been shown to be significantly superior to LRU-k, are scan-resistant, and seem to require fewer magic-parameters.

Second level Caching: There exist a multitude of situations where the page cache is part of a hierarchy of caches. When using Linux to run virtual machines using KVM(which is part of the Linux kernel), the host page cache serves as a second level cache because the guest VMs maintain a cache as well. This leads to double caching [26], with pages present in both host and guest caches. Guest I/O performance can be improved if the host cache implements some algorithm specifically designed for secondary caches such as MQ [33].

Scalability: A single LRU list consisting of millions of pages presents numerous difficulties. On systems with a large amount of memory, the time required to complete the LRU list-scan is prohibitively large. With large scan intervals, the CLOCK heuristic ceases to be effective because it simply divides the pages into active and inactive pools with no ordering within the pools themselves. The virtual-memory system thus has an imprecise view of page access patterns and working-set size. Cold pages thus stay on the LRU list (and thus in memory) much longer because of kswapd's delay in reaching them. This leads to an inflation in the overall cache size without any benefits, since the cold pages are never going to be used in the future. Increasing the page scanning frequency would alleviate the problem — but at the cost of increased overhead due to traversal of the LRU list and especially the costs incurred due to acquiring/releasing the LRU-list spinlock for every page during every scan. Thus, the current page cache design needs an overhaul if it is to keep up with exponentially increasing DRAM capacities (in accordance with Moore's law).

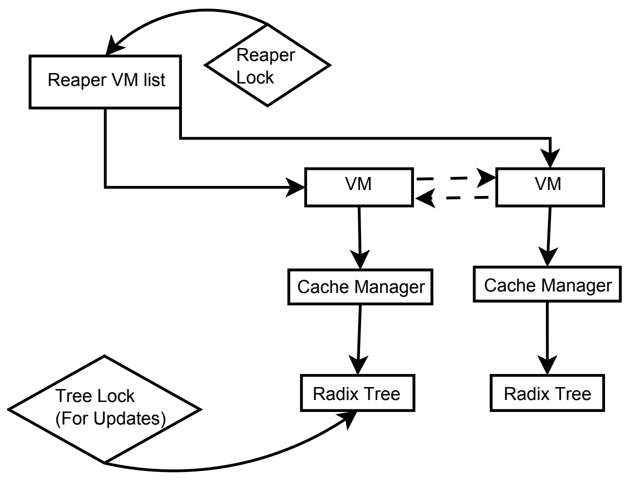

Fig. 4. The architecture of the per-VM-cache. Each VM has its own cache manager, and the reaper maintains a list of all VMs in LRU order.

III. Per-VM Page Cache Design

We now describe the design and architecture of the per-VM page cache. The per-VM page cache is a drop-in replacement for the existing Linux page cache and provides the same caching semantics to user-space processes (which includes KVM VMs).

The primary focus of the redesign is to split the page cache from a single large shared cache to multiple, independent caches. Each VM has its own, private, hypervisor page cache, which is unencroachable by other VMs. This separation provides performance isolation and QoS control for the VMs. The per-VM page cache is also configurable by the hypervisor and VM-management layer (such as OpenStack) on a *per-VM basis*. That is, the cache size and other properties such as eviction algorithm can be controlled for every VM individually, and the per-VM cache enforces these policies.

We note that this page cache redesign is completely at the hypervisor level, and VMs do not have to be modified in any way to avail the performance benefits. The VMs will only observe higher disk I/O performance (because of the cache separation) and other performance isolation benefits.

Figure 4 illustrates the high-level architecture. There are three major components :(i) an offset-to-page mapping which is implemented as a radix-tree, (ii) the per-VM cache manager which tracks the pages present in the radix-tree for the purpose of accounting and eviction, and (iii) the reaper thread which maintains the list of VMs for the purpose of eviction of pages. The key point is that the pages belonging to VM are not put on the global LRU lists. Instead, all pages of a VM are handled by its corresponding cache manager. The cache manager handles radix-tree hits, misses, and deletes. This way, we achieve isolation between VM page caches, and can have different cache configurations and algorithms for different VMs. We use locks for every VM's radix tree for synchronizing concurrent updates, and a global lock for the reaper thread for guarding the VM list.

A key feature of our design is that all the caching decisions are local to a VM, and are made by the VM's cache manager. The cache manager is delegated the task of handling the VM's cache pages. The cache manager maintains some sort of a page index (usually some variant of the LRU list) to keep track of page hits, misses, and evictions. Any caching algorithm such as LRU, ARC, LIRS, etc. can be used in the cache-manager module. For our prototype, we have implemented CAR [19] and FIFO (First In First Out) eviction algorithms. The cache manager has the task of ensuring high hit rates for the VM given cache space constraints. It meets this by evicting pages when full and when the reaper shrinks the cache for a VM. It can also ask the reaper for additional cache space.

The cache partitioning is done by the reaper-thread, which balances the size of each VM's cache as well as the global page cache size (total number of pages present in all the page caches). The primary weapon of the reaper is the ability to evict/reap pages. The reaper maintains a list of all active cache-managers (one cache manager per VM), and requests the corresponding cache-manager to evict pages. The VMs in the reaper-list are ordered by least recently used. Like any other LRU list in the kernel, a CLOCK approximation is used. The reaper-list is updated by the reaper-thread whenever evictions have to be performed. Thus, the least recently used VMs are the victims of the reaping, and the VM's cache-manager gets to choose which pages to evict from their cache.

The reaper implements three important functions:

1) Make space for new VM's cache by evicting pages from other VM caches.
2) Make space for a VM's cache by increasing the total cache allocation. This may result in increased swapping.
3) Shrink total page cache if pressure on anonymous memory is high.

The reaper is called by the cache managers (Figure 4) when they request for additional space (either for a new VM, or if the cache manager deems that the VM could benefit from additional cache). All the reaper's external interfaces are non-blocking, and simply update the number of pages to reap. The thread periodically runs and evicts the requested number of pages in an asynchronous manner.

A. Utility based Cache Partitioning

While our per-VM cache implementation can be used for implementing strict limits on the cache-sizes for various VMs (through the `sysfs` kernel interface or the `fadvise` system-call), general situations demand an adaptive solution to determine the cache sizes of various VMs. Page cache partitioning is important to improve the cache hit ratios and overall I/O latency. A good partitioning ensures that every VM gets the "right" amount of page cache at all times, depending on the total space available for caching and the access patterns of the VMs. To determine what the "right" partitioning is, we use *utility* as a metric.

For systemwide page caches, there exist two important dynamically changing values: the number of pages cached for a given VM, and the total number of page cache pages. In systems with a global LRU list (current Linux design), these values are not explicitly manipulated, but change depending on the number of pages evicted and added. One key advantage of system-wide LRU approach is that it naturally adapts to the changes in workload, system-load, and memory pressure.

With a partitioned page cache, manipulating these parameters (cache size of each VM and the global cache size) is an explicit task. While very sophisticated marginal-utility based cache partitioning approaches [24], [30] can be attempted, we have implemented simple and light-weight adaptive heuristics to manage the cache sizes. Part of the reason we have not considered complicated heuristics is that the insertions and deletions from the caches are performed from inside critical sections (thus holding spinlocks) and need to be extremely fast.

We now present a formal treatment of the cache partitioning:

Each VM i, when launched, is soft-allocated C_i number of pages in the hypervisor page cache. This represents the maximum possible size that the cache allocation can grow. If this maximum size is reached, then there are two possible cases:

1) The page-eviction algorithm of VM-i evicts pages to make room for new pages.
2) The VM's cache controller asks for an increase in C_i.

The cache-partitioning problem is thus: Given n VMs, with a total of M physical memory pages present in the system, determine C_i and F, where $F = \sum^n C_i$ and $F + A = M$, where A is the number of 'other' pages which are managed by `kswapd`. There may be user and system-defined constraints on the minimum and maximum limits for each C_i. The objective is to assign values to C_i so as to minimize the expected number of cache-misses, given recent cache access history.

In our implementation, a VM's hypervisor page cache size (C_i) is allocated proportional to the VM's marginal utility of the cache. The cache utility of a VM [24] is the the benefit it gets from an extra cache page allocated to it. The benefit is measured in terms of decrease in the number of misses that the VM encounters. Marginal Utility(MU) is function of cache size, and is the slope of the Miss-Ratio-Curve. Thus,

$$\text{MU}_s = \text{miss}(s+1) - \text{miss}(s) \tag{1}$$

Where miss(s) is the number of misses that occur with a cache size of s. The optimum partition of a cache among k VMs is obtained by solving:

$$\text{Total Utility} = U_1^{x_1}(f_1) + U_1^{x_2}(f_2) + \ldots + U_1^{x_k}(f_k) \tag{2}$$

Where $U(f_i)$ is the utility function of VM f_i. Assuming a cache size of F, an additional constraint is:

$$F = x_1 + x_2 + \ldots + x_k \tag{3}$$

Thus, given accurate and complete miss-ratio curves, a cache can be partitioned optimally. This general cache-partitioning problem is NP-Complete. However, if the utility functions are convex, then a simple greedy algorithm suffices [24]. The miss-ratio based utility function as defined above depends on the VM's access-pattern (sequential, random, cyclic, etc) and is also dynamic in nature. Miss-ratio curves are obtained by running the LRU stack-distance algorithm [6] on an access-trace of a VM.

An important point to note is that we also consider the *read-ahead successes* when determining the utility. If the read-ahead success-rate is very high, then the VM is running a sequential access application, which will most likely not benefit from the extra cache. Therefore the utility is calculated as:

$$\text{Utility} = \frac{\text{Misses} - (\text{Read Ahead Successes})}{\text{Total Accesses}} \tag{4}$$

This allows us to quickly detect sequential accesses and not waste precious cache on them. This approach also nicely integrates with the ARC's 'single-use' list [19], since we can potentially also use the shadow-list success-rate as a guide for sequentiality and utility. That is, a VM with very high hits in the shadow-list should get a larger cache. A shadow-hit implies a cache-hit had the cache been double the size, thus is a perfect input for a utility function.

VM cache shrinking is handled by the reaper thread since it is incharge of the evictions. As mentioned earlier in Section III, the VMs are present in an LRU order on the reaper list. The least recently used VM is chosen as the victim. The number of pages to evict from a victim VM is proportional to the size of the VM. This ensures that any "wrong" decisions by our allocation strategy do not cause catastrophic damage to the performance of VMs with small page cache footprint.

In our current implementation, the total space allocated for all the VMs in the cache F, keeps growing until the system starts to swap. On swapping, it decreases to reduce the memory pressure. We integrate with the existing page-eviction metrics of pages scanned and pages evicted, which are used by kswapd to determine the proportion of VM page cache and anonymous pages to keep/evict.

It must be emphasized here that our design incorporates support for both adaptive and user-specified changes for all parameters, including cache sizes and growth-rate for each VM. Users/processes can control the VM's cache allocation via the kernel `procfs` interface or by using the `fadvise` system-call. The adaptive allocation and partitioning heuristics are the default in case no allocation-policy is specified from userspace.

IV. Implementation

Our per-VM page cache is implemented in the Linux kernel as a drop-in replacement for the existing unified cache. Our implementation is restricted to the virtual-memory subsystem of the kernel, and every effort has been made to minimize the footprint of our changes. The total size of the patch required for implementing the per-VM cache is about 1500 lines.

In order to implement the partitioned cache, we steal pages destined for the unified page cache by removing them from the systemwide LRU list and by putting them on the `Un-evictable LRU` list. This prevents the swap daemon from touching these pages, and results in the LRU lists containing only non-cache pages (anonymous, slab). Our design does not require any changes in the management/eviction of the LRU lists—that task is still performed by `kswapd`.

Page cache pages are managed by putting them in a per-VM cache manager. Each VM's cache manager maintains its own cache state: the size, eviction-algorithm are all independently configurable. The existing Linux LRU-2 implementation is replaced by the CAR [5](CLOCK with Adaptive Replacement)

Value	Symbol	Adaptive change
VM page cache-size	C	Increases when miss-rate exceeds global miss-rate. Decreases when hit-rate is lower than global hit-rate
Total cache size	F-size	Increases when free-space exists. Decreases during swap activity and when swap daemon is frequently scanning anonymous pages for eviction

TABLE I
SUMMARY OF CACHE ALLOCATION PARAMETERS

algorithm. CAR is the CLOCK approximation of the Adaptive Replacement Cache(ARC) [19]. Our CAR implementation is only around 300 lines of code, compared to the ˜2000 lines required for the existing Linux page eviction implementation.

The file-cache eviction and free-space management is performed by a *reaper* thread, which is called when the total number of pages present in caches of all the files exceeds a threshold or when the system is low on free memory. The reaper thread maintains a list of all inodes ordered by LRU order of file accesses. To approximate the LRU, we use a 2-chance CLOCK — a file is declared 'cold' after being given two scans during which it has the chance to get accessed again. The reaper thread is implemented as a kernel-thread.

Our implementation is SMP-ready — the primary advantage of splitting the LRU list by file is the reduction in the contention of the LRU zone-lock. The spinlocks used in the implementation and their usage is detailed below:

reaper-lock: The reaper thread protects its list using the reaper-lock. The lock is acquired during inode additions/deletions, and the reaping itself, when the reaper walks down the reaper-list and updates it, or evicts pages from the inodes on the list. Since the number of inodes which need to be scanned during the reaping may be very large, reaping may take a large amount of time. The reaper-lock overhead is reduced by releasing and reacquiring the lock after scanning every inode, so that file open/close operations are not affected for a long period of time. The reaper-lock is not heavily contended because it is only acquired by the reaper and when files are opened/discarded. This is in contrast to the zone lock which is acquired for every cache miss.

inode-lock: The inode-lock is an important part of existing Linux inode synchronization. The inode-lock is acquired to prevent concurrent deletes of the inode via the reaper.

CAR-lock: The cache manager (CAR in our case) needs to protect its page index against concurrent cache-misses on the same file. We must emphasize that the lock does not destroy the lockless property of the page cache implementation. The lock is only acquired under two conditions:

1) Page additions (which corresponds to cache misses). On a cache-miss, the radix-tree lock has to be taken anyway. This is required because the reaper can also request an eviction concurrently on the same file.
2) Page evictions. If the eviction has been forced by the reaper thread. Since CAR uses CLOCKS, there is no lock acquired for a cache hit.

Improved scalability by reducing lock contention for the page eviction process was an important design goal for the per-file cache, and the primary advantage of the partitioning is the reduction in the lock contention for the LRU list lock.

V. RESULTS

The per-VM page cache is a general purpose page cache and can be used by VMs and *any* other user-space process. Due to the double caching phenomenon [26] which occurs with VMs, we use user-space processes doing file-IO to mimic VMs.

To test the effectiveness of our cache, we run multiple I/O intensive workloads. The workloads are described in Table II. All I/O workloads are generated by using fio (flexible I/O tester) [1], and the working set size of each workload is atleast two times larger than the total memory available.

Workload	Description
rand-seq	Mix of random and sequential read workloads
kerncompile	Kernel compile(Linux 3.0) with 3 threads

TABLE II
WORKLOAD DESCRIPTION.

A. Cache utilization

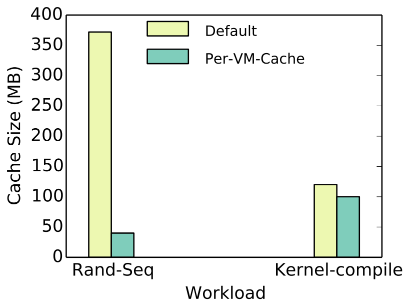

Fig. 5. Average page cache sizes. Per-VM caching results in a large reduction in cache size required. The per-file cache shows an order of magnitude reduction in cache size for this workload, illustrating the effectiveness of the utility based cache partitioning.

Figure 5 compares the average size of the page cache for the random-sequential and kernel-compile workloads (Table II). With the per-VM cache, we use only 40 MB of cache, while the default uses almost all the memory available and occupies 400 MB. This is an order of magnitude difference in the cache sizes. This reduction in cache footprint is due to both a different eviction algorithm (ARC vs LRU-2), as well as an effective demonstration of the utility-based cache partitioning.

Case	Hit-ratio
Default	0.622
Per-VM cache	0.714

TABLE III
SYSTEM-WIDE PAGE CACHE HIT RATIOS FOR THE RAND-SEQ WORKLOAD MIX. WE SEE A 15% IMPROVEMENT.

The actual I/O performance is shown in Figure 6. The I/O performance in this case is not perturbed much. This

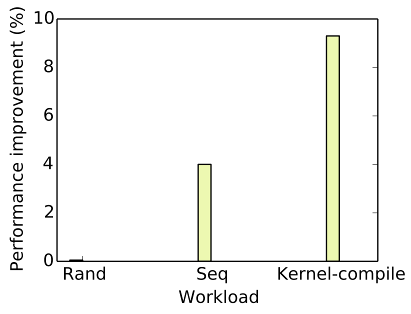

Fig. 6. Application performance improvement with per-VM-cache compared to default Linux page cache.

increase in cache effectiveness (we are able to use a much smaller cache for the same performance) is primarily because of the utility based cache sizing. In both the cases - Sequential and Random workloads, the marginal utility is very low. We identify the sequential workload on the basis of the high read-ahead-success to cache-hit ratio, and thus penalize that file when it asks for more memory. For the random-read case, the working set is larger than the total memory, thus the hit-rate is again quite low. Since margin utility is hit-rate based, and since utility guides the allocation, the file having random accesses is also prevented from growing at a very fast rate. The systemwide cache hit-ratios for the same workload(random-sequential) are presented in Table III. The overall hit-ratios increase by about 15%. Thus, the per-file-cache is able to provide an increase in hit-ratios with a **10x** smaller cache occupancy.

Small-file performance is measured by performing the kernel-compile workload. Due to a large number of open files, it is a good test of the worst-case behaviour of the partitioning scheme. Figure 6 shows an improvement of 10% in compile times of compiling the Linux kernel with the per-VM-cache. We also see a decrease of 20% in the cache size required (Figure 5).

B. Impact on system performance

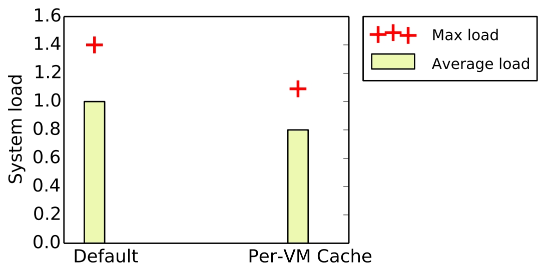

Fig. 7. Load average during the random-sequential workload. The load is 20% less with the per-VM-cache.

Since several key components of the memory-management subsystem have been substantially modified, the performance of our unoptimized implementation was expected to be inferior. However, as Figure 7 shows, the load averages with the per-file cache are *lower*. We hypothesize that this is because of not maintaining and scanning a global LRU list of pages. The CPU utilization of the reaper thread was too close to 0% to measure accurately. System-wide profiling has shown that the CAR shadow-list linear search function(which is called on every cache miss) is the most expensive component of the entire setup. Replacing the linked-list by a suitable data structure to reduce search cost is part of future work.

VI. Related Work

Our work builds on a large volume of work in the areas of utility-based cache partitioning and page cache design.

Work by Pei Cao et al. [8]–[10], describes techniques for application controlled caching — wherein applications can control the contents of the page cache explicitly by specifying which blocks to evict in case the cache overflows. The LRU-SP [10] algorithm which they have devised allows applications to over-rule the kernel's eviction decision. In contrast to our work, the kernel still maintains a unified LRU list, and thus there is no explicit control on the size of each file's cache. Early work in OS disk-caches by [12] models the hit-ratios in a cache hierarchy when each cache in the hierarchy implements a different demand-paging algorithm (such as LRU,FIFO,RANDOM). Several optimizations for OS-level disk-caches have been proposed and prototyped. The Karma-cache system [32] use marginal gains to guide placement of data in a multi-level cache hierarchy — address ranges with a higher marginal gain are placed higher(closer to the application). It implements various heuristics for cache allocation, file-access pattern detection, and replacement. Disk cache partitioning is also explored in [30]. The RACE system [34] performs looping reference detection and partitions the cache for sequential, random and looping files. Similarly, DEAR [11] presents an implementation study of caching using adaptive block replacement based on the access patterns.

An implementation of a policy controllable buffer-cache in Linux is presented in [3]. Policy controllable caches are a natural fit for micro-kernel architectures, where the policy is implemented by servers which need not run in the kernel-mode. Hence, the cache-manager can be abstracted away into a separate server, and it interacts both with the buffer-cache server itself as well as other userspace servers to determine and control the policy. An example of such a scheme has been shown for the Mach [18] and HURD [31] micro-kernels.

Cache partitioning is a very widely studied problem in CPU architectural data caches (L2) which are shared among multiple threads. Work by [4], [20] details several cache partitioning schemes, where the algorithms decide on which application threads get how many cache ways(lines). The goal is to minimize the number of cache-misses. The key insight of the cpu cache partitioning research is that different applications have vastly different utilities. That is, the miss-ratio vs. cache-size (Miss-ratio Curve) of each application is different, and it is beneficial to allocate cache space by choosing a size for each application which minimizes the miss-rate derivative.

Singleton [26] implements a black-box exclusive caching solution for KVM environments by reducing the host page cache size. Page cache management for virtual environments is also covered in [27], however it requires changes to the guest OS. Ren et.al., [25] present a new buffer cache design for KVM hosts. Their 'Least Popularly Used' algorithm tracks disk blocks by recency of access and their contents. Duplicate blocks are detected by checksums and eliminated from the cache. Several approaches to effectively use a multi-tiered cache hierarchies exist(such as Multi-Queue [33]).

Performance isolation can also be provided by using isolation features of operating systems. Several solutions exist to provide complete isolation to process-groups using OS-level virtualization — Jails [16] in FreeBSD, Zones [23] in Solaris, and control-groups (cgroups) in Linux [2]. Our per-VM cache can trivially provide page cache virtualization to process-groups. The I/O-lanes project [7] aims to provide end-to-end I/O isolation in virtualized environments by completely partitioning the I/O stack.

VII. Conclusion

With increasing memory sizes, it is imperative to have fine grained control of the page cache. The per-VM page cache is an attempt to design and implement a general-purpose high-performance page cache solution which is designed to be scalable. Some of the limitations of the Linux's virtual memory subsystem, such as a single LRU list, non-scan resistant eviction algorithm, lack of support to specify page cache occupancy have been addressed by our design. By partitioning by using miss-ratio based utility function, we have shown that it is possible to increase cache hit ratios while decreasing the memory required by upto 10x. Among several other features, the per-VM page cache allows each VM to have different cache sizes and replacement algorithms. The per-VM page cache improves performance isolation among VMs, increases VM disk I/O performance, and reduces hypervisor page cache size which in turn allows cloud providers to run more VMs on their physical machines.

Acknowledgements. We thank all the reviewers for their insightful comments, which improved the quality of this paper. This work is supported in part by NSF grants #1422245 and #1229059.

References

[1] fio: Flexible IO tester. http://linux.die.net/man/1/fio.

[2] Linux Control Groups. www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[3] C.J. Ahn, S.U. Choi, M.S. Park, and J.Y. Choi. The design and evaluation of policy-controllable buffer cache. In *IEEE Conference on Parallel and Distributed Systems*, 1997.

[4] G. Almási, C. Caşcaval, and D.A. Padua. Calculating stack distances efficiently. In *ACM SIGPLAN Notices*, volume 38, pages 37–43, 2002.

[5] S. Bansal and D.S. Modha. Car: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 187–200. USENIX Association, 2004.

[6] BT Bennett and V.J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

[7] A. Bilas. Scaling I/O in virtualized multicore servers: How much I/O in 10 years and how to get there. In *International workshop on Virtualization Technologies in Distributed Computing Date*, 2012.

[8] P. Cao, E.W. Felten, A.R. Karlin, and K. Li. *A study of integrated prefetching and caching strategies*, volume 23. ACM, 1995.

[9] P. Cao, E.W. Felten, A.R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems (TOCS)*, 14(4):311–343, 1996.

[10] P. Cao, E.W. Felten, and K. Li. Application-controlled file caching policies. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 171–182, 1994.

[11] J. Choi, S.H. Noh, S.L. Min, E.Y. Ha, and Y. Cho. Design, implementation, and performance evaluation of a detection-based adaptive block replacement scheme. *IEEE Transactions on Computers*, 51(7):793–800, 2002.

[12] C.D. Cranor and G.M. Parulkar. The UVM virtual memory system. *USENIX Annual Technical Conference*, pages 117–130, 1999.

[13] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware*, pages 342–362. 2006.

[14] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Virtual Execution Environments*, pages 41–50, 2009.

[15] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS Performance Evaluation Review*, volume 30, pages 31–42. ACM, 2002.

[16] P.H. Kamp and R.N.M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, 2000.

[17] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[18] C. Maeda. A metaobject protocol for controlling file cache management. *Object Technologies for Advanced Software*, pages 275–286, 1996.

[19] N. Megiddo and D.S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, 2003.

[20] M. Moreto, FJ Cazorla, A. Ramirez, and M. Valero. Online prediction of applications cache utility. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007.*, pages 169–177. IEEE, 2007.

[21] Piggin Nick. A Lockless Page Cache in Linux. In *Proceedings of the Linux Symposium*, pages 241–250, 2006.

[22] E.J. O'neil, P.E. O'neil, and G. Weikum. The LRU-K page Replacement Algorithm for Database Disk Buffering. In *ACM SIGMOD Record*, volume 22, pages 297–306, 1993.

[23] D. Price and A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th USENIX conference on System administration*, pages 241–254, 2004.

[24] M.K. Qureshi and Y.N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006.

[25] J. Ren and Q. Yang. A New Buffer Cache Design Exploiting Both Temporal and Content Localities. In *2010 International Conference on Distributed Computing Systems*, 2010.

[26] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 15–26. ACM, 2012.

[27] B. Singh. Page/slab Cache Control in a Virtualized Environment. In *Proceedings of the Linux Symposium*, pages 252–262, 2010.

[28] Gaurav Somani and Sanjay Chaudhary. Application performance isolation in virtualization. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 41–48. IEEE, 2009.

[29] A.S. Tanenbaum. *Modern Operating Systems*, volume 2. Prentice Hall New Jersey, 1992.

[30] D. Thiébaut, H.S. Stone, and J.L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6):665–676, 1992.

[31] N.H. Walfield and M. Brinkmann. A critique of the GNU Hurd multi-server operating system. *ACM SIGOPS Operating Systems Review*, 41(4):30–39, 2007.

[32] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-all replacement for a multilevel cache. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, 2007.

[33] Y. Zhou, J.F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of USENIX Annual Technical Conference*, 2002.

[34] Y. Zhu and H. Jiang. Race: A robust adaptive caching strategy for buffer cache. *IEEE Transactions on Computers*, 57(1):25–40, 2008.