Page Cache Management in Virtual Environments

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

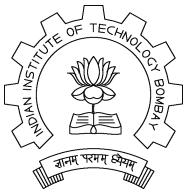
by

Prateek Sharma

Roll No: 09305910

under the guidance of

Prof. Purushottam Kulkarni



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Dissertation Approval Certificate

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay

The dissertation entitled **"Page Cache Management in Virtual Environments"**, submitted by **Prateek Sharma** (Roll No: **09305910**) is approved for the degree of **Master of Technology** in **Computer Science and Engineering** from **Indian Institute of Technology, Bombay**.

Prof. Purushottam Kulkarni
Dept. CSE, IIT Bombay
Supervisor

Prof. Umesh Bellur
Dept. CSE, IIT Bombay
Internal Examiner

Dr. Preetam Patil
NetApp Corp., Bangalore
External Examiner

Prof. Shishir Kumar Jha
SJM SOM, IIT Bombay
Chairperson

Place: IIT Bombay, Mumbai
Date: 28^{th} June, 2012

Declaration

I, Prateek Sharma, declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

_____

Signature

_____

Name of Student

_____

Roll number

_____

Date

We investigate memory-management in hypervisors and propose Singleton, a KVM-based system-wide page deduplication solution to increase memory usage efficiency. We address the problem of double-caching that occurs in KVM—the same disk blocks are cached at both the host(hypervisor) and the guest(VM) page caches. Singleton's main components are identical-page sharing across guest virtual machines and an implementation of an exclusive-cache for the host and guest page cache hierarchy. We use and improve KSM–Kernel SamePage Merging to identify and share pages across guest virtual machines. We utilize guest memory-snapshots to scrub the host page cache and maintain a single copy of a page across the host and the guests. Singleton operates on a completely black-box assumption—we do not modify the guest or assume anything about its behaviour. We show that conventional operating system cache management techniques are sub-optimal for virtual environments, and how Singleton supplements and improves the existing Linux kernel memory-management mechanisms. Singleton is able to improve the utilization of the host cache by reducing its size(by upto an order of magnitude), and increasing the cache-hit ratio(by factor of 2x). This translates into better VM performance(40% faster I/O). Singleton's unified page deduplication and host cache scrubbing is able to reclaim large amounts of memory and facilitates higher levels of memory overcommitment. The optimizations to page deduplication we have implemented keep the overhead down to less than 20% CPU utilization.

we argue that second-level page-cache allocation to Virtual Machines plays an important role in determining the performance, isolation, and Quality-of-Serice of Virtual Machines. The second-level cache is a contended resource, and also competes with the memory allocated to the Virtual Machines themselves. We show that the phenomenon of swapping-while-caching is particularly detrimental to VM performance. Our solution is to utilize cache-partitioning, and we have implemented a per-file page cache in the Linux kernel. Our framework allows fine-grained control of the page cache by applications and virtual machines.

Contents

Chapter 1

Introduction

In virtual environments, physical resources are controlled and managed by multiple agents — the Virtual Machine Monitor(VMM), and the guest operating systems (running inside the virtual machines). Application performance depends on both the guest operating system and hypervisor, as well as the interaction between them. The multiple schedulers (CPU, I/O, Network), caches, and policies can potentially conflict with each other and result in sub-optimal performance for applications running in the guest virtual machines. An example of guest I/O performance being affected by the combination of I/O scheduling policies in the VMM and the guests is presented in [14].

In this thesis we consider the effects of physical memory being managed by both the VMM(Virtual Machine Monitor) and the guest operating systems. Several approaches to memory management and multiplexing in VMMs like ballooning and guest-resizing exist [54]. We focus on techniques which do not require guest support(page-sharing) and consider system-wide memory requirements, including that of the *host* operating system.

The primary focus of our memory-management efforts is on the behaviour of the *page-cache*. The page-cache [26] in modern operating systems like Linux, Solaris, FreeBSD etc is primarily used for caching disk-blocks, and occupies a large fraction of physical memory. The virtualization environment we focus on is KVM(Kernel Virtual Machine) [31], which is a popular hypervisor for Linux, and allows unmodified operating systems to be run with high performance. KVM enables the Linux kernel to run multiple virtual machines, and in-effect turns the operating system(Linux) into a VMM(also called *hypervisors*). We consider the effectiveness of using conventional OS policies in environments where the OS also hosts virtual machines. We show that the existing operating system techniques for page-cache maintenance and page-evictions are inadequate for virtual environments.

In contemporary Virtualized environments, the Operating System plays an additional role of a hypervisor. That is, it enables running multiple Virtual Machines by multiplexing physical resources (such as memory, CPU-time, IO bandwidth) among the virtual machines. Resource management policies are thus enforced at two levels : within the Operating Systems running inside the virtual machines (managing the virtualized resources among processes) , and the hypervisor managing the resources among the virtual machines. We explore this hierarchical control and management of resources for one par-

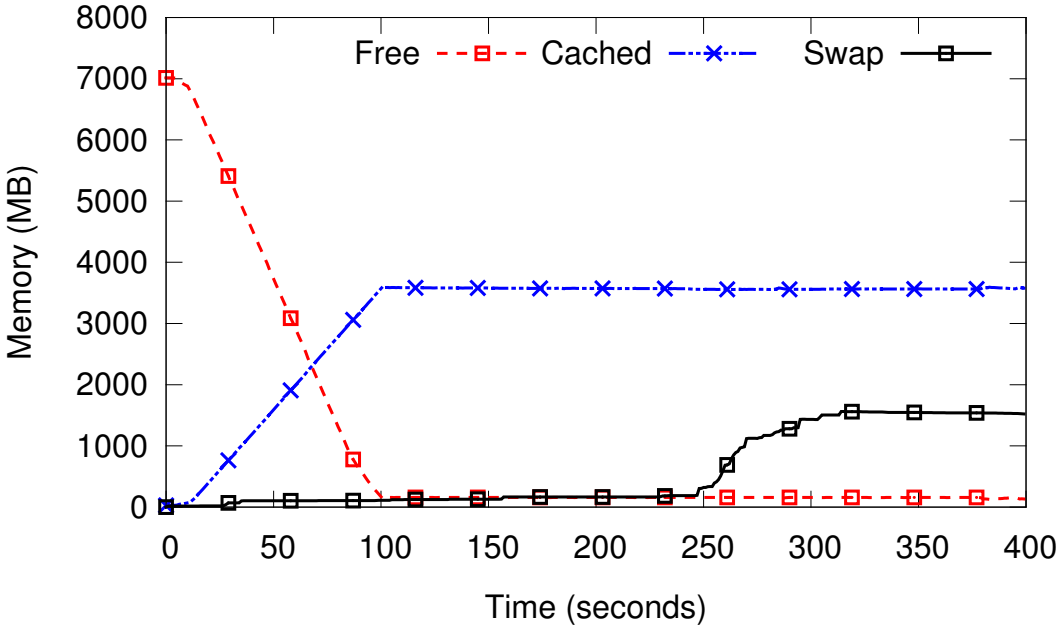


Figure 1.1: Memory usage graph for a sequential disk-read workload. The start of the workload corresponds to a sharp drop in free memory and an increase in swap usage. Due to the heavy I/O, the cache is not dropped inspite of the memory pressure

ticular resource : main-memory. In particular, we analyze the effects of multiple levels of page cache present in virtualized environments.

In most KVM setups, there are two levels of the page-cache—the guests maintain their own cache, and the host maintains a page-cache which is shared by all guests. Guest I/O requests are serviced by their respective caches first, and upon a miss fall-through to the host page-cache. This leads to double-caching: same blocks are present in the guest as well as the host caches. Furthermore, the host-cache sees a low hit-ratio, because pages are serviced from the guest's page-cache first. This double caching wastes precious physical memory and leads to increased memory-pressure, causing swapping and performance loss. In particular, the problem of swapping is detrimental for virtual setups. Figure 1.1 shows the memory-usage graph when guest VMs execute I/O intensive workloads, and illustrates that the host system starts swapping even in the presence of cached pages. Note that the guests maintain their own page-caches, and the host caching leads to swapping of pages belonging to VMs. While this unfortunate situation can be ameliorated with existing techniques like using direct I/O and `fadvise` for the guest VMs etc, we show that they adversely affect VM performance.

This thesis addresses the problem of multiple levels of cache present in virtual environments, and we seek to implement an *exclusive-cache*. Exclusive caching entails not storing multiple copies of the same object in multiple locations in the cache hierarchy. While multi-level caching and exclusive caches are well studied in the context of network-storage systems [28, 20, 56] and CPU architectural caches, our work is the first to focus on exclusive caching in KVM-based environments. Furthermore, we implement a com-

pletely *black-box* approach—requiring no guest modifications or knowledge. We do not rely on graybox techniques like intercepting all guest I/O and page-table updates found in Geiger [30] and XRAY [9]. Another constraint we adhere to is that our solution must not cause performance regressions in non-virtualized environments, since the OS(Linux) serves both as a conventional OS running userspace processes and virtual machines. Thus, we do not change any critical kernel component. This prevents us from implementing specialized techniques for second-level cache-management which are found in [25, 56, 60, 61].

Page deduplication across Virtual Machines [54, 32, 33] is an effective mechanism to reclaim memory allocated to the VMs by the hypervisor in a completely guest-transparent manner. To implement the exclusive page-cache, we utilize content-based page deduplication, which collapses multiple pages with the same content into a single, copy-on-write protected page.

1.1 Contributions

This thesis makes two important contributions.

- An exclusive-cache solution (called "Singleton") is developed and evaluated.
- A new page-cache design (called "Per-File-Cache") is proposed, implemented and evaluated.

1.1.1 Singleton

As part of this work, we design and evaluate Singleton, a Kernel Samepage Merging (KSM) based system-wide page deduplication technique. Specifically, our contributions are the following:

- We optimize existing KSM duplicate-page detection mechanisms which reduce the overhead by a factor of 2 over the default KSM implementation.
- We implement an exclusive host page-cache for KVM using a completely black-box technique. We utilize the page deduplication infrastructure (KSM), and proactively evict redundant pages from the host cache.
- Through a series of workloads and micro-benchmarks, we show that Singleton delivers higher cache-hit ratios at the host, a drastic reduction in the size of the host-cache, and significantly improved I/O performance in the VMs.
- We show that proactive management of host cache provides higher levels of memory overcommitment for VM provisioning.

Our implementation is a non-intrusive addition to the host kernel, and supplements the existing memory-management tasks of the VMM (Linux), improves page-cache utilization and reduces system-load.

1.1.2 Per File Cache

- A new page-cache architecture for the Linux kernel is designed and implemented.
- The page-cache is partitioned by file, and hence called the per-file page-cache. The cache design allows for fine-grained control of a file's cache. For example, the size of the cache and the caching algorithm can be specified for each file.
- We explore the problem of utility based cache partitioning and show that it yields significantly higher cache utilization (by upto an order of magnitude) as compared to the existing unified caches found in all operating systems.

1.2 Outline

A brief overview of existing literature in the areas of page-deduplication, exclusive-caching, page-caches, partitioned-caches is given in 2. Comparisons with related work are made throughout the rest of the report wherever applicable. In chapter 3, we introduce the architecture of KVM, KSM, QEMU, which will help motivate the problems and the solutions that we propose. Chapter 4 describes Singleton — the motivation behind exclusive-caching in KVM, the design and architecture , and the performance improvements. In chapter 5 we turn our attention to the problems with unified caches and propose our per-file cache design. Finally, we conclude in chapter 6 and list the future work. Appendices contain some implementation details of Singleton and Per-file cache.

Chapter 2

Literature Review

2.1 Page Deduplication

Transparent page sharing as a memory saving mechanism was pioneered by the Disco [15] project, although it requires explicit guest support. Inter-VM content based page sharing using scanning was first implemented in VMWare ESX Server [54]. While the probability of two random pages having exactly the same content is very small, the presence of a large number of common applications,libraries etc make the approach very feasible for a large variety of workload combinations [33, 32, 29, 34, 19]. Furthermore, page deduplication can also take advantage of presence of duplicate blocks across files (and file-systems). Storage deduplication for virtual environments is explored in [63, 47]. Page sharing in hypervisors can be broadly classified into two categories—scanning-based and paravirtualized-support. Scanning based approaches periodically scan the memory areas of all VMs and perform comparisons to detect identical pages. Usually, a hash based fingerprint is used to identify likely duplicates, and then the duplicate pages are unmapped from all the page tables they belong to, to be replaced by a single merged page. The VMWare ESX-Server [54] page sharing implementation, Difference Engine [27] (which performs very aggressive duplicate detection and even works at the sub-page level), and KSM [7] all detect duplicates by scanning VM memory regions. An alternative approach to scanning-based page sharing is detecting duplicate pages when they are being read-in from the (virtual) disks. Here, the virtual/emulated disk abstraction is used to implement page sharing at the device level itself. All VM read-requests are intercepted and pages having same content are shared among VMs. Examples of this approach are Satori [40] and Xenshare [32]. This approach is not possible with KVM because it does not primarily use paravirtualized I/O.

2.2 Exclusive Caching

Several algorithms and techniques for implementing exclusive caching in a multi-level cache hierarchy exist. Second-level buffer management algorithms are presented in [61, 60]. Most work on exclusive caching is in the context of network storage systems— [25],

DEMOTE [56], XRAY [9].

An exclusive-cache mechanism for page-caches is presented in Geiger [30], which snoops on guest page-table updates and all disk accesses to build a fairly accurate set of evicted pages. However it uses the paravirtualized drivers and shadow page-tables features of Xen, and its techniques are inapplicable in KVM and hardware-assisted two-dimensional paging like EPT and NPT [1].

2.3 Memory overcommitment

One way to provide memory overcommitment is to use conventional operating systems techniques of paging and swapping. In the context of VMMs, this is called host-swapping [54], where the VMM swaps out pages allocated to VMs to its own swap-area. Another approach is to dynamically change memory allocated to guests via a ballooning method [54, 49], which "steals" memory from the guests via a special driver. Several other strategies for managing memory in virtual environments, like transcendent memory [37], collaborative memory management [50] exist, but they require explicit guest support or heavy hypervisor modifications.

Transcendent memory [37] is a radical solution to dynamic memory management, which mandates memory regions which are not in explicit control of the host kernel. That is, a large area of memory is reserved, and is not directly addressible by the kernel. It is used to store objects (usually pages). Users (VMs and host) can use this object store to store pages. The key feature of transcendent memory is that there are no guarantees made about the availibility of these objects. That is, the objects(pages) which are stored in the transcendent memory area are not persisent. Thus only clean pages can be stored in the transcendent memory. If a lookup for a page in the transcendent memory fails, it must be bought back from the disk. Thus the users of transcendent memory must make changes to several memory access operations. Transcendent memory has been used mainly to implement several kinds of cleancaches (a page cache for clean pages). For example, one application is to provide a compressed pool of pages, as done by `ramzswap`.

2.4 Cache Management

The work closest to ours is by Pei Cao [18, 16, 17], which 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 [18] 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 [8, 24] 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 pro-

posed and prototyped. The Karma-cache system [58] use marginal gains to guide placement of data in a multi-level cache hierarchy — address ranges with a higher marginal gain are placed higher. It implements various heuristics for cache allocation, file-access pattern detection, and replacement.

[5] demonstrate an implementation of a policy controllable buffer-cache in linux. 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, a 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 [36] and HURD [55] micro-kernels. Disk cache partitioning is also explored in [53]. The RACE system [62] performs looping reference detection and partitions the cache for sequential, random and looping files. Similarly, DEAR [22] presents an implementation study of caching using adaptive block replacement based on the access patterns.

Cache partitioning is a very widely studied problem in CPU architectural data caches (L2) which are shared among multiple threads. Work by [41, 6] details several cache partitioning schemes, where the algorithms decide on which application threads get how many cache ways(lines). The goal is almost always to maximize the IPC count via minimizing the number 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. We must emphasize here that all the utility-based L2 cache partitioning work has not found application in real CPUs.

Page-cache management for virtual environments is covered in [51], however it requires changes to the guest OS. Ren et.al., [46] 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 checksumming and eliminated from the cache. LPU does not provide a guest-host exclusive cache, nor does it implement any inter-VM page sharing. Instead, all VM I/O traffic goes through a custom LPU buffer-cache implementation. We believe that having a custom high-traffic page-cache would suffer for scalability and compatibility issues—the page-cache contains millions of pages which need to be tracked and maintained in an ordered list (by access time) for eviction purposes. This is not a trivial task: the Linux kernel has been able to achieve page-cache scalability (with memory sizes approaching 100s of GB and 100s of CPU cores contending for the LRU list lock) only after several years of developers' efforts. Hence our goal with Singleton is to minimize the number of system components that need to be modified, and instead rely on proven Linux and KVM approaches, even though they may be sub-optimal.

Chapter 3

Background: Virtualization with KVM

This chapter presents the relevant background which will help motivate our solutions. We present the relevant KVM architecture, and since KVM uses the rest of the Linux kernel for most of the services, we provide the necessary Linux background as well. In particular, the Kernel Samepage Merging(KSM) page deduplication mechanism is detailed.

3.0.1 KVM architecture and operation

KVM(Kernel Virtual Machine) is a hardware-virtualization based hypervisor for the Linux kernel. The KVM kernel module runs virtual machines as processes in the host system, and multiplexes hardware among virtual machines by relying on the existing Linux resource-sharing mechanisms like its schedulers, file-systems, resource-accounting framework, etc. This allows the KVM module to be quite small and efficient.

The virtual machines are not explicitly created and managed by the KVM module, but instead by a userspace hypervisor helper. Usually, QEMU [11] is the userspace hypervisor used with KVM. QEMU performs tasks such as virtual machine creation, management and control. In addition, QEMU can also handle guest I/O and provides several emulated hardware devices for the VMs (such as disks, network-cards, BIOS, etc.). QEMU communicates with the KVM module using a well-defined API using the `ioctl` interface. An important point to note is that the virtual machines created by QEMU are ordinary user-space processes for the host. Similar to memory allocations for processes, QEMU makes a call to `malloc` to allocate and assign physical memory to each guest virtual machine. Thus, for the host kernel, there is no explicit VM, but instead a QEMU process which has allocated some memory for itself. This process can be scheduled, swapped out, or even killed.

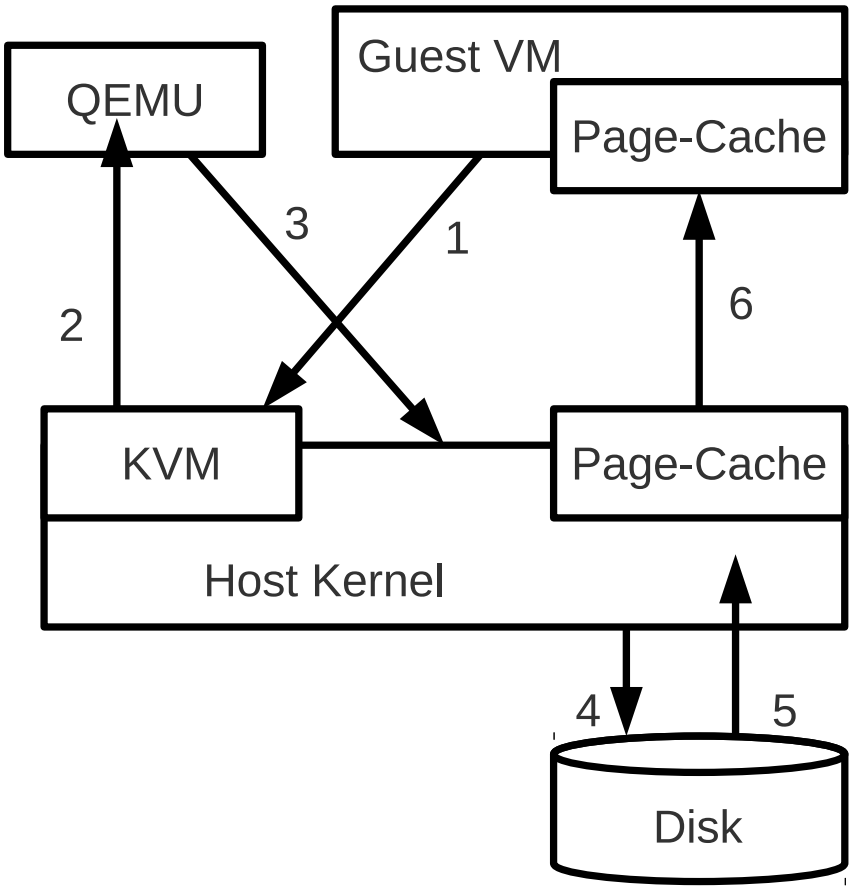


Figure 3.1: Sequence of messages to fulfill an I/O operation by a guest VM.

3.0.2 Disk I/O in KVM/QEMU

The guest VM's "disk" is emulated in the host userspace by QEMU, and is frequently just a file on the physical disk's filesystem. Hence, the emulated disk's read/write are mapped to file-system read/write operations on the virtual-disk file. Figure 3.1 depicts the(simplified) control flow during a guest VM disk I/O operation. A disk I/O request by the guest VM causes a trap, on which KVM calls the QEMU userspace space process for handling. In the emulated disk case, QEMU performs the I/O operation through a disk I/O request to the host kernel. The host reads the disk block(s) from the device, which get cached in host-page-cache and passed on to the guest via KVM. For the guest, this is a conventional disk read, and hence disk blocks are cached at the guest as well.

3.0.3 QEMU caching modes

Writeback: In the writeback mode, the host page cache is used for all IO. But unlike writethrough, the write notification is sent to the guest as soon as the data hits the page cache, not the disk. Thus there is a chance of data loss if the host crashes. The virtual disk corruption problem has also been observed when the qemu process is forcibly killed.

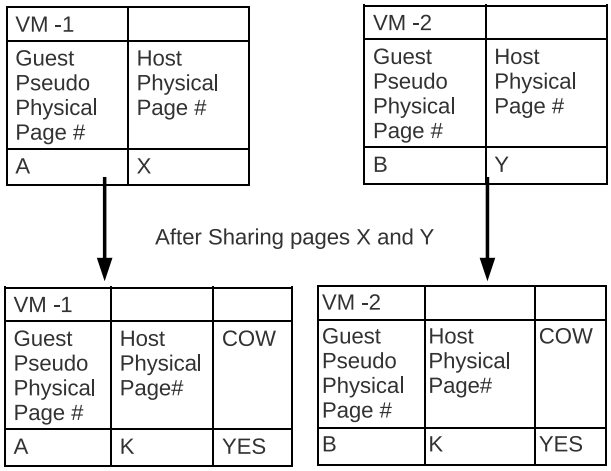


Figure 3.2: Copy-on-Write based hypervisor level page sharing.

Writethrough: This is the default caching mode. All guest IO goes through the host page cache. This mode does badly with some disk formats (qcow, qcow2 for example).

None: Uses direct IO using O_DIrect and hence bypasses the page-cache of the host OS. Supposed to give bad results [4] . Multiple VMs using O_Direct can affect performance since this is equivalent to multi-threaded synchronous IO.

VirtIO: VIRTIO is not strictly a caching mode, but instead a more different IO handling mechanism. Disks mounted as virtio are not emulated by qemu. Instead the host kernel has drivers which directly interact directly with the frontends of the corresponding drivers in the guest VMs. Virtio is esentially a ring-buffer implementation. It requires special drivers in the guests.

3.0.4 Linux page-cache and page eviction

The Linux page-cache [42] is used for storing frequently accessed disk-blocks in memory. It is different from the conventional buffer-cache in that it also stores pages belonging to `mmap`'ed files, whereas traditional buffer-caches restricted themselves to `read/write` I/O on file-system buffers. In a bid to improve I/O performance, a significant amount of physical memory is utilized by the kernel as page-cache.

Linux uses an LRU variant (specifically, a variant of LRU/2 [44]) to evict pages when under memory pressure. All the pages are maintained in a global LRU list. Thus, page-cache pages as well as pages belonging to process' private address spaces are managed for evictions in a unified manner. This can cause the kernel to swap out process pages to disk inspite of storing cache pages. The page-cache grows and shrinks dynamically depending on memory pressure, file-usage patterns, etc.

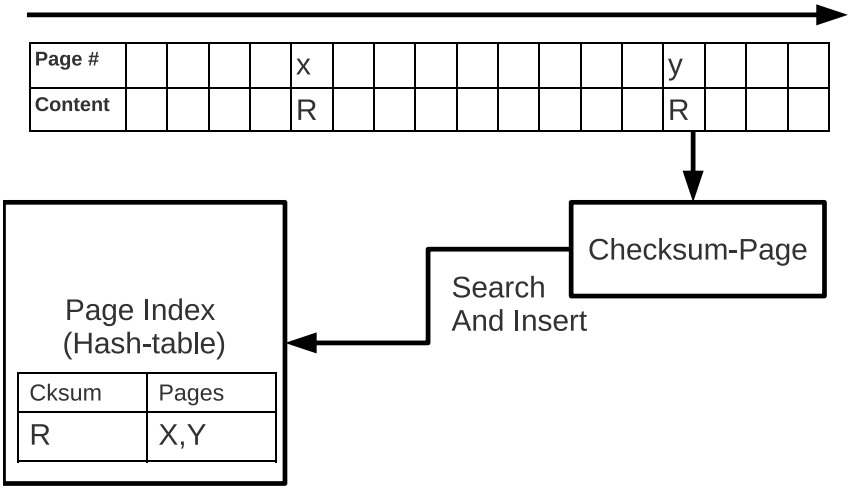


Figure 3.3: Basic KSM operation. Each page during a scan checksummed and inserted into the hash-table.

3.0.5 Page Deduplication using KSM

KSM(Kernel Samepage Merging) [7] is a scanning based mechanism to detect and share pages having the same content. KSM is implemented in the Linux kernel as a kernel-thread which runs on the host system and periodically scans guest virtual machine memory-regions looking for identical pages. Page sharing is implemented by replacing the page-table-entries of the duplicate pages with a common KSM page.

As shown in Figure 3.2, two virtual machines have two copies of a page with the same content. KSM maps the guest-pseudo physical page of both machines A and B to the same merged host physical page K. The shared page is marked copy-on-write(COW) — any modifications to the shared page will generate a trap and the result in the sharing being broken. To detect page similarity, KSM builds a page-index periodically by scanning all pages belonging to all the virtual machines.

KSM originally used red-black binary-search trees as the page-index, and full-page comparisons to detect similarity. As part of Singleton, we have replaced the search-trees with hash-tables, and full-page comparisons with checksum(`jhash2`) comparisons. In each pass, a single checksum-computation is performed, and the page is inserted into a hash-table(Figure 3.3). Collisions are resolved by chaining. To reduce collisions, the number of slots in the hash-table is made equal to the total number of pages.

Due to volatility of the pages (page-contents can change any time) and the lack of a mechanism to detect changes, the page-index is created frequently. Periodically, the page-index(hash-table) is cleared, and fresh page-checksums are computed and inserted. The KSM scanning-based comparison process goes on repeatedly, and thus has a consistent impact on the performance of the system. KSM typically consumes between 10-20% CPU on a single CPU core for the default scanning-rate of 20MB/s. The checksumming and hash-tables implementation in Singleton reduces the overhead by about 50% compared to the original KSM implementation (with search-trees and full-page comparisons).

To see that KSM can really detect and share duplicate pages, the memory fingerprint [57] of a VM is calculated and compared for similarity. The number of pages that KSM shares compared to the actual number of pages which are duplicate (which is obtained by the fingerprint) determines the sharing effectiveness of KSM. The memory fingerprint of a VM is simply a list of the hashes of each of its pages. By comparing fingerprint similarity, we have observed that KSM can share about 90% of the mergeable pages for a variety of workloads. For desktop workloads(KNOPPIX live-CD), KSM shares about 22,000 of the 25,000 mergeable pages. Ideal candidates for inter-VM page sharing are pages belonging to the kernel text-section, common applications, libraries, and files [33, 32, 54, 19]. These pages are often read-only, and thus once shared, the sharing is not broken.

At the end of a scan, KSM has indexed all guest pages by their recent content. The index contains the checksums of all guest pages, including the duplicate and the unique pages. Moreover, this index is created periodically (after every scan), so we are assured that the checksum corresponding to a page is fairly recent and an accurate representation of the page content. Thus, the KSM maintained page-index can be used as a snapshot of the VM memory contents.

Chapter 4

Singleton: System-wide Page Deduplication

In this chapter we present the design, implementation, and performance evaluation of Singleton, a system which has been developed to implement an exclusive page-cache and increase memory overcommitment in virtual environments.

4.1 Motivation: Double caching

A pressing problem in KVM is the issue of double-caching. All I/O operations of guest virtual machines are serviced through the page cache at the host (Figures 3.1 and 4.1). Because all guest I/O is serviced from the guest's own page-cache first, the host cache sees a low hit-ratio, because "hot" pages are already cached by the guest. Since both caches are likely to be managed by the same cache eviction technique (least-recently-used, or some variant thereof), there is a possibility of a large number of common pages in the caches. This double-caching leads to a waste of memory. Further, the memory-pressure created by the inflated host cache might force the host to start swapping out guest pages. Swapping of pages by the host severely impacts the performance of the guest VMs. An illustration of how guest I/O impacts the host page cache is shown in Figure 4.2. A single VM writes to a file continuously, which causes a steady increase in the amount of host-page-cached memory and corresponding decrease in the free memory available at the host.

Double caching can be mitigated if we provide an exclusive-cache setup. In exclusive caching, lower levels of cache(the host page-cache in our case) do not store an object if it is present in the higher levels(the guest page-cache). Any solution to the exclusive caching problem must strive for a balance between size of the host page cache and performance of the guests. A host-cache has the potential to serve as a 'second-chance' cache for guest VMs and can improve I/O performance. At the same time, large host page-caches might force guest VM pages to be swapped out by the host kernel—leading to severely degraded performance. Singleton provides an efficient exclusive cache which improves guest I/O performance, and reduces host-cache size drastically.

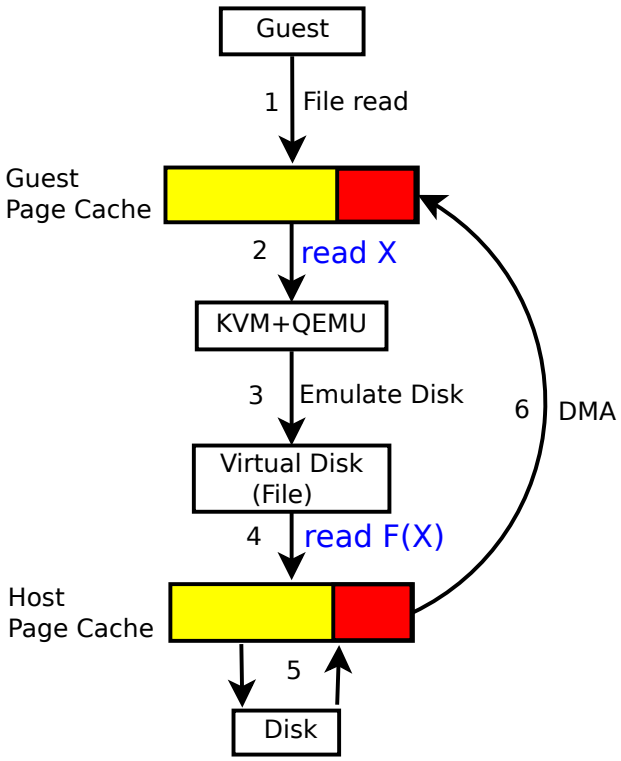


Figure 4.1: A read system-call from the guest application in KVM-QEMU setup. A disk for the VM is simply a file in the host file system

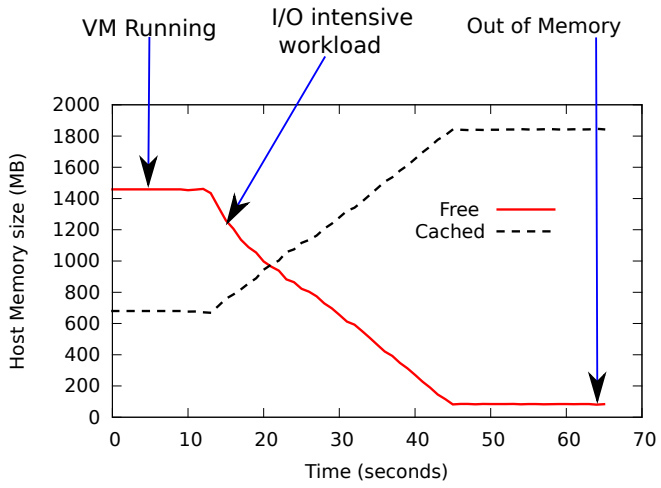


Figure 4.2: Host free and cached memory on a write-heavy guest workload.

The shared nature of the host-cache also makes it important to provide performance isolation among the virtual machines as well as the processes running on the host system. Disk I/O from an I/O intensive guest VM can fill-up the host-cache, to the detriment of the other VMs. Not only do other VMs get a smaller host cache, but also suffer in performance. The memory-pressure induced by one VM can force the host-kernel to put additional effort for page allocations and scanning pages for evictions, leading to increased system load.

4.2 Potential/Existing approaches For Exclusive Caching

In the context of multi-level exclusive caching in storage systems [25, 56, 9], it has been shown that exclusive caches yield better cache utilization and performance. Exclusive caches are usually implemented using explicit co-ordination between various caches in a multi-level cache hierarchy. DEMOTE [56] requires an additional SCSI command for demotion notifications. Gray-box techniques for inferring cache hits at higher levels in the cache hierarchy, like X-RAY [9] and Geiger [30] use file-system information to infer page-use by monitoring inode access-times.

In the host-guest cache setup that virtual systems deal with, notifications can cause a large overhead since the page cache sees high activity. Furthermore, the host/guest page-caches are present on a single system, unlike the distributed client/server storage caches. Solutions to exclusive caching require the lower-level(host) cache to explicitly read-in the items evicted from the higher-level(guest) cache. This is not desirable in our setup: VM performance would be impacted if the host does disk accesses for evicted items, leading to overall system-slowdown.

More pressing is the problem of actually generating the eviction notifications—modifications to both the host and the guest OS memory subsystems will be required. However, in spite of the benefits of exclusive caching, modifications to the operating system are not

Operations	VM using Direct I/O	VM using host cache
putc	34,600	33,265
put_block	48,825	51,952
rewrite	14,737	24,525
getc	20,932	44,208
get_block	36,268	**197,328**

Table 4.1: Bonnie performance with and without caching at the host.

straight-forward. The first challenge is to get notifications of evictions—either by explicit notifications from the guest, or by using I/O-snooping techniques like those developed in Geiger [30]. The fundamental problem is that there is no easy way to map disk blocks in the host and the guest page cache. The hypervisor (QEMU) supports a large number of virtual disk formats (RAW, LVM, QCOW, QCOW2, QED, FVD [52]). The mapping from a virtual block number to a physical block number (which the host file system sees) can be determined fairly easily in case of RAW images, but one would need explicit hypervisor support in other cases. The lack of a common API for these image formats results in a complex co-ordination problem between the host, the guest, and the hypervisor. Clearly, we need a better solution which does not need to contend with this three-way coordination and yet works with all the above mentioned setups and environments.

Direct IO: An existing mechanism to overcome the wastage of memory due double-caching is to bypass the host page-cache. This can be accomplished by mounting the QEMU disks with the `cache=none` option. This opens the disk-image file with the direct-IO mode (`O_DIRECT`). However, direct-I/O has an adverse impact on performance. Table 4.1 compares performance of file operations on two VMs running the Bonnie [2] file system benchmark. In one case, both Virtual Machines mount their respective (virtual) disks with `cache=writeback` option (QEMU default) set and in the other we use the `cache=none` option. Table 4.1 shows the Bonnie performance results of one of the VMs. Bypassing the host cache results in almost all operations with direct I/O to be slower than with caching. With direct I/O, the block read rates are 6x slower, the read-character rate 2x slower. Further, the average seek rate with Direct I/O was 2x slower than with host-page-caching—185 seeks per second with direct I/O and 329 seeks/second with caching. Clearly, the I/O performance penalty is too much to pay for a reduced memory usage at the host —host cache is not used with direct I/O. Additionally, using `O_DIRECT` turns off the clever I/O scheduling and batching at the host, since the I/O requests are immediately processed. Direct-I/O scales poorly with an increase in number of VMs, and we do not consider it to be a feasible solution to the double-caching problem.

Fadvise: Additionally, the hypervisor can instruct the host kernel to discard cached pages for the virtual disk-images. This can be accomplished by using the POSIX `fadvise` system-call and passing the `DONTNEED` flag. Fadvise needs to be invoked periodically by

the hypervisor on the disk-image file for it to have the desired effect. All file data in the cache is indiscriminately dropped. While fadvise mitigates double-caching, it fails to provide any second-level caching for the guests. The `DONTNEED` advise can also potentially be ignored completely by some operating systems, including the previous versions of the Linux kernel.

4.3 The Singleton approach

To implement a guest-exclusive cache at the host, Singleton uses KSM and the page-index it maintains to search for pages present in the guests. As mentioned earlier (Section 3.0.5), KSM maintains a snapshot of contents of all pages in its search indexes (red-black trees in case of default KSM, hash-tables in Singleton).

Singleton's exclusive caching strategy is very simple and presented in Algorithm 1. We look-up all the *host* page-cache pages in the KSM maintained page-index of all the VMs to determine if a host-cache page is already present in the guest. The host page-cache pages are checksummed, and the checksum is searched in KSM's page-index. An occurrence in the guest page-index implies that the page is present in the guest, and we *drop* the page from the host's page-cache.

A page in the host's page cache is said to belong to VM V if an I/O request by V resulted that page being bought into the cache. Pages in the page-cache belong to files on disk, which are represented by inodes. We identify a page as belonging to a VM if it belongs to the file which acts as its virtual-disk. To identify which file corresponds to the virtual machine's disk, we pick the file opened by the QEMU process associated with the VM.

Algorithm 1 Singleton's cache-scrubbing algorithm implemented with ksm.

```
After scanning B pages of VM V:
   For each page in the host-cache belonging to V:
       If (page in KSM-Page-Index)
          drop_page(page);
```

Dropping duplicate pages from the host page-cache is referred to as *cache-scrubbing*. The cache scrubbing is performed periodically by the KSM thread—after KSM has scanned (checksummed and indexed) B guest pages. We refer to B as the *scrubbing-interval*.

After dropping pages from the host-cache during scrubbing, two kinds of pages remain in the host cache : pages not present in the guest, and pages which might be present in the guest but were not checksummed (false negatives due to stale checksums). Pages not present in the guest, but present in the host-cache can be further categorized thus: 1. Pages evicted from the guest. 2. Read-ahead pages which were not requested by the guest. The false-negatives do not affect correctness, and only increase the size of the host-cache. False negatives are reduced by increasing KSM's scanning rate.

Cache-utilization of the host's cache will improve if a large number of evicted pages are present(eviction based placement [56]). Keeping evicted pages in the host-cache increases the effective size of cache for the guests, and reducing the number of duplicates across the caches increases exclusivity. To reduce the multiplicative read-ahead [59] as well as to reduce cache size, read-ahead is disabled on the host. We treat the guest as a black-box and do not explicitly track guest evictions. Instead, we use the maxim that page-evictions are followed by page-replacement, hence a page replacement is a good indicator of eviction. Page replacement is inferred via checksum-changes. A similar technique is used in Geiger [30], which uses changes in disk-block addresses to infer replacement. To differentiate page-mutations(simple writes to a memory-address) from page-replacement, we use a very simple heuristic: a replacement is said to have occurred if the checksum and the first eight bytes of the page content have changed.

Singleton introduces cache-scrubbing functionality in KSM and runs in the KSM thread (`ksmd`) in the host-kernel. We take advantage of KSM's page-index and page-deduplication infrastructure to implement unified inter-VM page deduplication and cache-scrubbing. The cache-scrubbing functionality is implemented as an additional 1000 lines of code in KSM. The `ksmd` kernel thread runs in the background as a low-priority task (`nice` value of 5), consuming minimal CPU resources. Singleton extends the conventional inter-VM page deduplication to the entire system by also including the host's page-cache in the deduplication pool. While the memory reclaimed due to inter-VM page sharing depends on the number of duplicate pages between VMs, Singleton is effective even when the workloads are not amenable to sharing. Since all guest I/O passes through the host's cache, the number of duplicate pages in the host's cache is independent of the inter-VM page sharing. Singleton supplements the existing memory-management and page-replacement functionality of the hypervisor, and does not require intrusive hypervisor changes. While our implementation is restricted to KVM setups and not immediately applicable to other hypervisors, we believe that the ideas are relevant and useful to other hypervisors as well.

4.4 Scrubbing frequency control

The frequency of cache scrubbing dictates the average size of the host cache and the KSM overhead. To utilize system memory fully and keep scrubbing overhead to a minimum, a simple scrubbing frequency control-loop is implemented in Singleton. The basic motivation is to control the scrubbing frequency depending on system memory conditions (free and cached). A high-level algorithm outline is presented in Algorithm 2. The `try_scrub` function is called periodically (after KSM has scanned 1000 pages). We use two basic parameters: maximum amount of memory which can be cached (`th_frac_cached`) and minimum amount of memory which can be free (`th_frac_free`), both of which are fractions of the total memory available. The scrubbing frequency is governed by the time-period `t`, which decreases under memory pressure, and increases otherwise. With host cache getting filled up quickly, Singleton tries to increase scrubbing rate and decreases it

otherwise. The time-period has minimum and maximum values between which it is allowed to vary(not shown in the algorithm). The time-period is also a function of number of pages dropped by the scrubber (`scrub_host_cache`).

Algorithm 2 Singleton's frequency control algorithm.

```
try_scrub (th_frac_cached, th_frac_free) {
  Update_memory_usage_stats(&Cached, &Free, &Memory);
//Case1: Timer expires. t is current scrub interval
  if(cycle_count-- <= 0) {
    Dropped = scrub_host_cache() ;
   //returns num pages dropped
    prev_t = t ;
    t = prev_t*(Cached + Dropped)/Cached;
 }
//Case2: Memory pressure
 else if(Cached > Memory*th_frac_cached ||
             Free < Memory*th_frac_free) {
    Dropped=scrub_host_cache();
    prev_t = t ;
    t = prev_t*(Cached - Dropped)/Cached;
 }
  cycle_count=t;
}
```

4.5 KSM Optimizations implemented

To get a better understanding of KSM sharing, the KSM code was instrumented with static tracepoints using the kernel TRACE_EVENTS feature. Trace events allows ftrace [12] static tracepoints to be placed anywhere in the kernel code and are very light-weight in nature. We primarily use trace-events to generate a lot of `printk` output. Every page that KSM scans is traced, as are all the operations (tree search/insert) it goes through. This generates a significant amount of trace-log output (about 0.5 GB/minute). We have used the information obtained from the detailed trace logs to improve our understanding of KSM operations as well as page-sharing in general.

4.5.1 Exploiting spatial locality using Lookahead

Using the KSM trace logs, it is observed that shared pages are often clustered together, occurring consecutively. Thus shared pages have a high spatial locality : If a page is shared, then the next page is also shareable(with some other page) with a high probability.

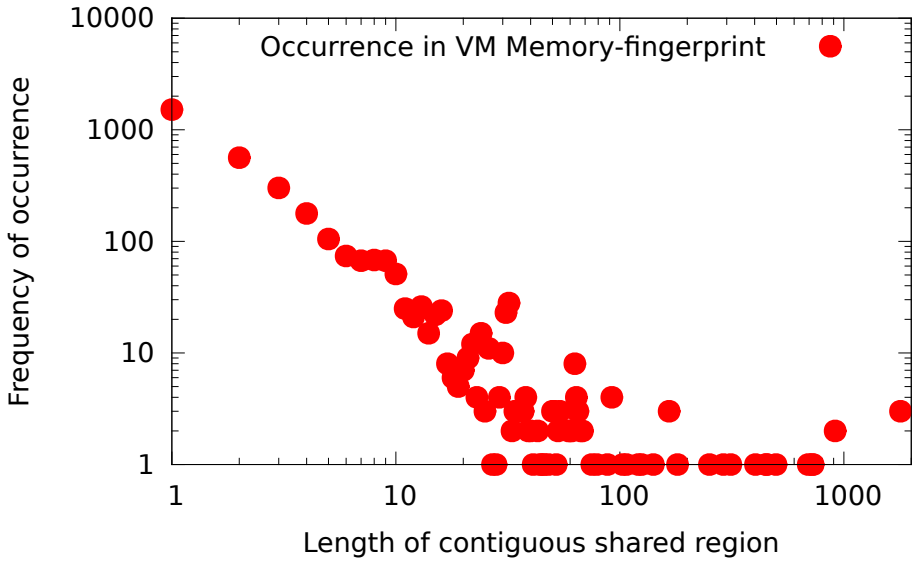


Figure 4.3: Spatial locality among shared pages.KSM trace is collected for 2 similar desktop VMs. The graph plots length of consecutive shared-pages vs their frequency of occurrence in the trace.

This can be seen from Figure 4.3, which shows the frequency of contiguous shared pages in a trace.

A natural way of expressing the page-sharing problem is to represent the page contents as alphabets, and the memory contents of a VM as the strings. The degree of sharing is equal to the number of common characters in two strings. In the context of the previous observation (that the shared pages are clustered), we can say that the VM-strings have a large number of common substrings. The presence of substrings can be explained by the fact that a large amount of common pages are courtesy of common files. These files can be programs, common data, etc. Thus when the files are loaded into memory, if they are the same (have the same contents), then the VM memory regions will have a large number of common pages too. Furthermore, these pages will be consecutive, since files loaded into memory are typically mapped into consecutive pages as far as possible by operating systems.

4.5.2 KSM implementation of Lookahead

The presence of spatial locality leads to a very natural improvement in the KSM algorithm. If a page X is merged with another page Y, then we check if page X+1 can be merged with Y+1, before searching in the stable and unstable trees(see Figures 4.4 , 3.3). In other words, we do a lookahead and see if the pages are common, before doing a full search.

A more detailed algorithm follows:

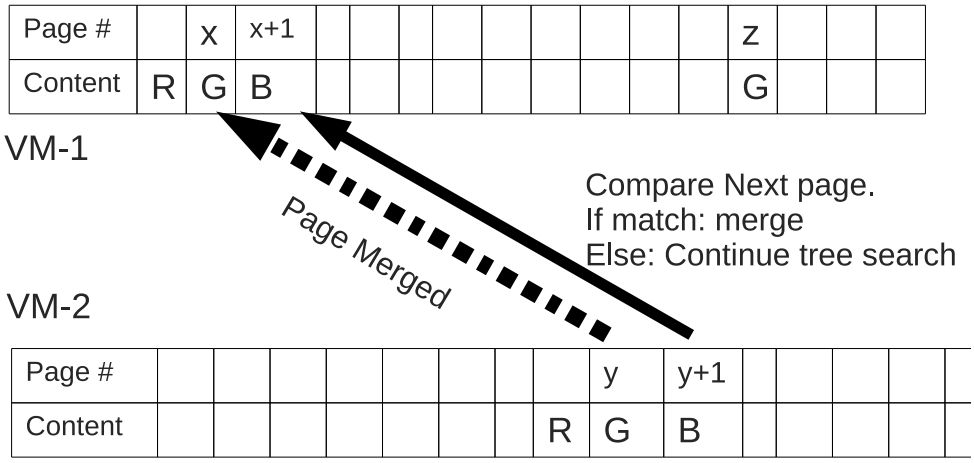


Figure 4.4: Lookahead illustration. The consecutive shared-pages are often file-backed. In this example the Red-blue-green pages probably a common file opened by the two VMs.

The lookahead optimization reduces the average cost of searching for a page. Assuming that the probability of a page being part of a duplicate file is p, the expected cost with the lookahead optimization in place is now :

$$p * Cost(\text{page-merge}) + (1 - p) * Cost(\text{tree search}) \tag{4.1}$$

Although the lookahead optimization is particularly effective for KSM because of its existing design, exploiting spatial locality among shared pages will benefit any page deduplication mechanism.

The lookahead trick is particularly effective when the pages are shared for the first time. Once the pages are merged, there will be no additional overhead in the subsequent passes (assuming they have not changed and the COW is not broken). To detect whether a page is shared/merged, KSM simply checks a flag(PAGE_KSM). Thus the lookahead optimization reduces search costs only when the pages can be potentially shared. Once the sharing is established, it plays no role (and hence can offer no improvements) in the subsequent passes.

The lookahead optimization has negligible benefits if the page-index is a large-enough hash-table, and hence is turned-off for the all the experiments. However this observation of spatial locality among pages for inter-VM page deduplication is the first such work in literature.

4.5.3 Performance with Lookahead optimization

The main advantage of lookahead is the reduced search cost, and hence a reduction in KSM overhead. We compared vanilla-KSM(KSM) with KSM+lookahead(KSM-Look) on the same workloads and recorded the ksm page sharing statistics along with the lookahead successes, and the ksm overhead on the host. Lookahead gives the most improvements for the Desktop workload (Table 4.2). This is because desktop environments load a large number of common-files into memory on bootup (X11 fonts, graphical environment, etc).

Algorithm 1 Algorithm for Lookahead optimization

```
 Z=current_page_under_KSM_scanner
 //X is pfn of latest merged page
 //in the scan.
 //X was merged with Y .
% If(equal(Z, Y+1)) :
      merge(Z, Y+1)
      X = Z
      Y = Y+1
 else :
    Search trees for Z
    If matched with W
         X = Z
         Y = W
```

The surprising result is the increased shared pages due to lookahead. This is surprising because without the lookahead the pages would have been merged anyway, albeit after a tree search. The success of lookahead to increase the shared pages can be explained thus: Because lookahead decreases the average cost of searching for a duplicate page, the scanning rate of KSM increases slightly. The increase in scanning rate results in increased page-sharing, because KSM can detect short-lived sharing opportunities. Correspondingly, the KSM overhead (with lookahead enabled) also *increases* (Also as evidence of increased scanning rate).

The lack of success of lookahead in the case of kernel-compile (there are a large number of shared pages and common files in this workload) also has a subtle explanation. Even though there are a large number of common files, most of them are small (less than a page size). Also the files are not explicitly `mmapped` or read into memory by the compiler. Instead they are present in the page-cache. Since the duplicate pages are present largely in the page-cache, they may not be located in contiguous memory regions. This is the reason that lookahead cannot detect such duplicates. Thus even though the benchmark accesses the same files in the same sequence, because they are scattered differently in their respective page-caches, lookahead is not very successful.

4.5.4 Filtering by Guest Page flags

The cost of KSM scanning is dictated by the number of page-comparisons. To reduce the number of page-comparisons, we filter the pages and thus do not scan (or compare) certain pages in the guests. The guest pages can be classified according to their page-flags, and pages which do not have certain flags set can be ignored by KSM.

Having looked at the feasibility of inter-VM page sharing, in this section we analyze

Workload (on Two VMs)	Avg. Shared Pages (Vanilla)	Avg. Shared Pages (with Lookahead)	CPU util. (Vanilla)	CPU util. (with Lookahead)	Total Lookahead successes
Boot up	8,000	11,000	12%	12%	4,000
Kernel Compile	26,000	30,000	19%	22%	16,000
Desktop VM use	31,000	62,000	14.6%	16.8%	50,000

Table 4.2: Lookahead performance on 3 workloads. Using lookahead increases the number of shared pages significantly with a small increase in CPU utilization.

what kinds of pages contribute to sharing. We start off by analysing the page-sharing ratio of pages by their guest-flags. The guest OS keeps track of all the pages in its physical memory in the mem_map array. Each page has with it information associated with it like its page-flags, reference-count (map-count), etc. This page information is not available at the host, because the guest physical memory is just a process address space (since QEMU guests are simple user processes).

Obtaining guest pageflags

Since the guest page-flags and other memory-management data maintained by the guest OS is transparent to the host and not accessible, co-operation with the guest is required. For our experiments, a simple guest daemon runs in the guest periodically. This deamon reads the page flags from `/proc/kpageflags`, and writes to a pre-determined memory location. The host kernel accesses this memory location by simply reading the corresponding page. To ensure that the daemon can write to a fixed physical memory location, a memory-hole is created at guest's boot-time. This prohibits the kernel from mapping any pages at the hole location, and we are guaranteed that the daemon has an exclusive access to that location. Since the boot-hole is not mapped, we need to use the ioremap mechanism (which is typically used for mapping DMA regions).

It turns out that filtering pages by their flags is not a consistent way of obtaining pages with a high probability of being shareable. On different workloads, different kinds (by flags) pages were found to be shared. Hence no uniform filter can be used to reduce scanning and comparison cost.

Assume that the boot-hole is created at 100MB, and the size is 1MB. The host kernel then simply reads the contents of the page at Guest starting virtual address + 100MB. In this way, the host kernel can directly read the guest page flags without any userspace interaction.

Scanning only dirtied pages: A fundamental limitation of KSM (and all other scanning-based page-deduplication mechanisms) is that page-dirty rates can be much higher than the scanning rate. Without incurring a large scanning overhead, it is not possible for a brute-force scanner to detect identical pages efficiently.

We are interested in reducing the scanning overhead by only checksumming dirtied pages—similar to VM Live Migration [23], where only dirtied pages are sent to the destination. Conventional techniques rely on write-protecting guest pages, and incur expensive faults on a guest access to that page. Instead, we intend to use a combination of techniques based on hardware-assisted page-dirty logging and random sampling. In some cases, like AMD's Nested Page Tables (NPT) implementation [1], it is possible to obtain a list of dirtied pages without the expensive write-protect-trap approach seen in VM Live-migration. AMD's NPT implementation exposes dirty page information of the guests (pages in the guest virtual address space),which can be exploited to perform dirty-logging based scanning. Further, dirty logging overhead or scanning overhead can be reduced by sampling and subset of pages and by eliminating "hot" pages from the working set in the scan process.

4.6 Experimental analysis

Cache scrubbing works by proactively evicting pages from the host's page-cache. In this section we explore why additional cache management is required for the host's page cache, and why the existing Linux page eviction and reclaiming mechanisms are sub-optimal for virtual environments. We show how Singleton improves memory utilization and guest performance with a series of benchmarks. Our results indicate that significant reductions in the size of the host page-cache, an *increase* in the host page-cache hit-ratio, and improvement in guest performance can all be obtained with minimal overhead.

Workload	**Description**
Sequential Read	Iozone [43] is used to test the sequential read performance.
Random Read	Iozone is used to test random-read performance.
Zipf Read	Disk blocks are accessed in a Zipf distribution, mimicking many commonly occurring access patterns.
Kernel Compile	Linux kernel (3.0) is compiled with make allyesconfig with 3 threads.
Eclipse	The Eclipse workload in the Dacapo [13] suite is a memory-intensive benchmark, which simulates the Eclipse IDE [3].
Desktop	A desktop-session is run, with Gnome GUI, web-browsing, word-processor.

Table 4.3: Details of workloads run in the guest VMs.

4.6.1 Setup

Since scrubbing is a periodic activity and can have drastic impact on system performance when the scrubbing operation is in progress, all experiments conducted are of a sufficiently long duration (atleast 20 minutes). The workloads are described in Table 5.1. The scrubbing interval thresholds are between 100,000 and 200,000 pages scanned by KSM (scrubbing-interval algorithm presented in section 4.4), and is of the order of once every 30-60 seconds. The cache-threshold is set as 50% of the total memory and the free-threshold is 10%. For read-intensive benchmarks, the data is composed of blocks with random content, to prevent page deduplication from sharing the pages. For guest I/O, virtIO [48] is used as the I/O transport to provide faster disk accesses.The experiments have been conducted on an IBM x3250 blade server with 8GB memory, 2GB swap-space and one 150GB SAS hard-disk(ext4 file-system). In all the experiments otherwise stated, we run 4 VMs with 1 GB memory size each. The hosts and the guest VMs run the same kernel (Linux 3.0) and OS(Ubuntu 10.04 x86-64 server). To measure the performance on each of the metrics, a comparison is made for three configurations:

Default: The default KVM configuration is used with no KSM thread running.

Fadvise: This runs the page deduplication thread and calls `fadvise(DONTNEED)` periodically.

Singleton: Page deduplication and eviction based cache placement is used.

4.6.2 Host-cache utilization

The host cache sees a low hit-ratio, because "hot" pages are cached by the guest. Because of double-caching, if the host's cache is not large enough to accommodate the guest working set, it will see a low number of hits. Our primary strategy is to not keep pages which are present in the guest, and preserve pages which are *not* in the guest. This increases the effective cache size, since guest cache misses have a higher chance of being serviced from the host's page-cache. Presence of pages being present in the guest provides additional knowledge to Singleton about a cached page's usefulness, which is not available to the access-frequency based page-eviction mechanism present in the host OS(Linux) kernel. We exploit this knowledge, and remove the duplicate page from the host cache.

Singleton's scrubbing strategy results in more effective caching. We run I/O intensive workloads in the guest VMs and measure the system-wide host cache hit-ratio. The hit-ratio also includes the hits/misses of files accessed by the host processes. Details of the workloads are in Table 5.1. The results from four VMs running sequential,random, and Zipf I/O are presented in Figure 4.5. For four VMs running sequential read benchmark (Iozone) the cache-hit ratio is 65%, an improvement of about 4% compared to default case (vanilla KVM). A significant reduction in cache-hits is observed when using `fadvise(DONTNEED)` (16% less than Singleton). Calling `fadvise(DONTNEED)` simply drops all the file pages, in contrast to Singleton which keeps pages in the cache if they

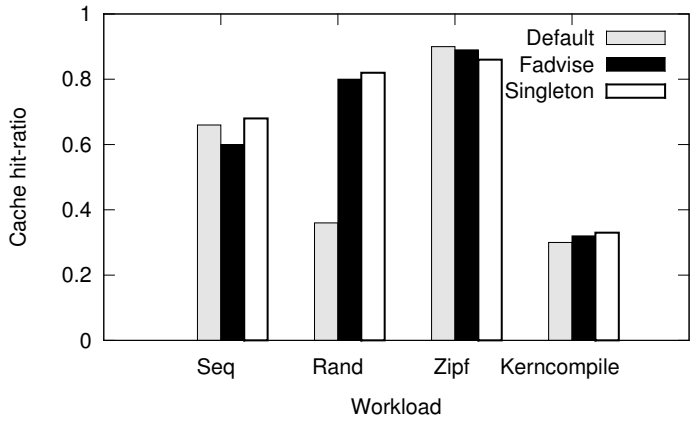


Figure 4.5: Host page-cache hit ratios.

are not present in the guest. Thus, Singleton's eviction based placement strategy is more effective, and keeps pages to accommodate a larger guest working set.

Scrubbing impacts random-reads more, since the absence of locality hurts the default Linux page-eviction implementation. By contrast, keeping only evicted pages leads to a much better utilization of cache. The cache-hit ratio with Singleton is almost *2x* the default-case (Figure 4.5). For this experiment, the working set size of the Iozone random-read was kept at 2GB, and the VMs were allocated only 1 GB. Thus, the host-cache serves as the second-chance cache for the guests, and the entire working set can be accommodated even though it does not fit in the guest memory. For workloads whose working-sets aren't large enough, the host-cache sees a poor hit ratio : about 35% in case of the kernel-compile workload. In such cases, the scrubbing strategy only has a negligible impact on host cache utilization. We have observed similar results for other non I/O intensive workloads as well.

The increased cache utilization translates to a corresponding increase in the performance of the guest VMs. For the same setup mentioned above(four VMs executing the same workloads), sequential-reads show a small improvement of 2% (Table 4.4). In accordance with the higher cache-hit ratios, random-reads show an improvement of about 40% with Singleton over the default KVM setup. Similar gains are observed when compared to `fadvise(DONTNEED)`—indicating that by utilizing the host-cache more effectively, we can improve the I/O performance of guests. We believe this is important, since disk-I/O for virtual machine is significantly slower than bare-metal I/O performance, and one of the key bottlenecks in virtual machine performance.

	Sequential reads (KB/s)	Random reads (KB/s)	Zipf Reads (KB/s)
Default	4,920	**240**	265,000
Fadvise	4,800	280	260,000
Singleton	5,000	**360**	270,000

Table 4.4: Guest I/O performance for various access patterns.

4.6.3 Memory utilization

The Linux kernel keeps a unified LRU list containing both cache and anonymous(not backed by any file, belonging to process' address space) pages. Thus, under memory pressure, anonymous pages are swapped out to disk even in the presence of cached pages (Figure 1.1). Without the proactive cache-scrubbing, we see an increased swap traffic, as the host swaps pages belonging to the guest's physical memory. This swapping can be avoided with scrubbing. The preemptive evictions enforced by Singleton also reduce the number of pages in the global LRU page-eviction list in Linux. This leads to reduction in the kernel overhead of maintaining and processing the list of pages, which can be quite large (millions of pages on systems with 10s of gigabytes of memory). Scrubbing supplements the existing Linux memory-management by improving the efficiency of the page-eviction mechanism.

The periodic page evictions induced by scrubbing reduces the size of the cache in the host significantly. We ran I/O intensive benchmarks, which quickly fill-up the page-cache to observe Singleton's ability to reduce cache size. Figure 4.7 shows the average cache size over the workload-runs, when the workloads are running on four virtual machines. The host cache size with Singleton is **2-10x** smaller than the default KVM. Compared to the `fadvise(DONTNEED)` approach which drops all cache pages, Singleton has a larger cache size. The cache-size can be further reduced if needed by increasing the frequency of the cache-scrubbing. However, our scrubbing-frequency algorithm (presented in Section 4.4) enables us to make a more judicious use of available memory, and increases the scrubbing frequency only when under memory-pressure.

A lower average cache size increases the amount of free memory available, prevents swapping, and reduces memory pressure. In addition to scrubbing, Singleton also employs inter-VM page deduplication, which further decreases the memory usage. A reduction in the amount of swapping when different workloads are run in the guests can be seen in Figure 4.8. Without scrubbing, the swap is utilized whenever the guests execute intensive workloads which fill-up the host page-cache. In contrast, using `fadvise(DONTNEED)` and Singleton results in no/minimal swap-space utilization.

As Figure 1.1 shows, pages are swapped to disk even though a significant amount of memory is being used by the page-cache. Scrubbing prevents this kind of behaviour, as illustrated in Figure 4.6. The periodic scrubbing results in sharp falls in the cache-size,

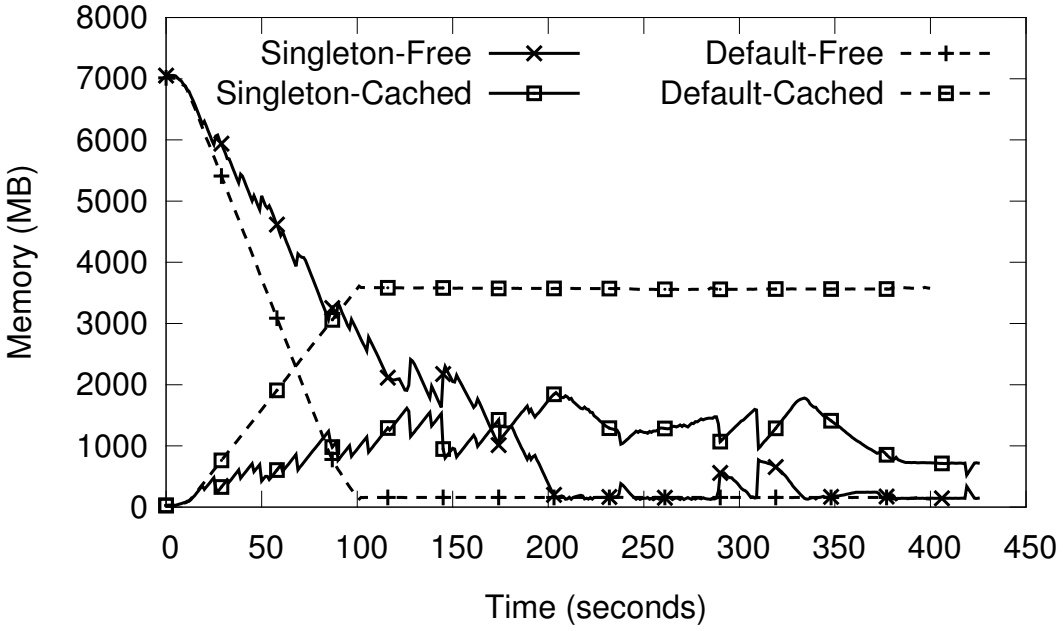


Figure 4.6: Memory usage graph for a sequential read workload with and without Singleton.

Workload	Pages-shared
Sequential	80,000
Random	133,000
Kerncompile	350,000
Desktop	300,000
Eclipse	215,000

and increases the amount of free-memory. This reduction in memory pressure and the reduced swapping reduces the system load and paging activity. In the kernel-compile and eclipse workloads, a further reduction in memory-usage is observed because of the inter-VM page-deduplication component of Singleton. When identical workloads are running in four guest VMs, we can see significant amount of pages being shared (seen in Table 4.6.3). Out of a total 1 million pages (1 GB allocated to each of the 4 VMs with 4KB pages), the percentage of pages shared varied from 8% in the case of sequential read workload to 35% with the kernel-compile workload. The page-sharing is dependent on the workload—same files are used in the case of kernel compile, whereas only the guest kernel pages are shared with the sequential read workload.

An additional benefit of Singleton is that it helps provide a more accurate estimate of free memory, since unused cache pages are dropped. This can be used to make more informed decisions about virtual-machine provisioning and placement.

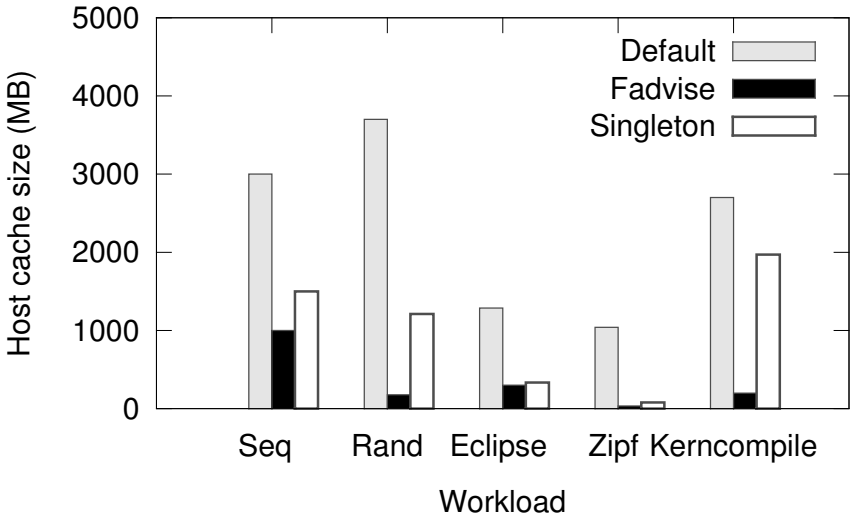


Figure 4.7: Average Cache size

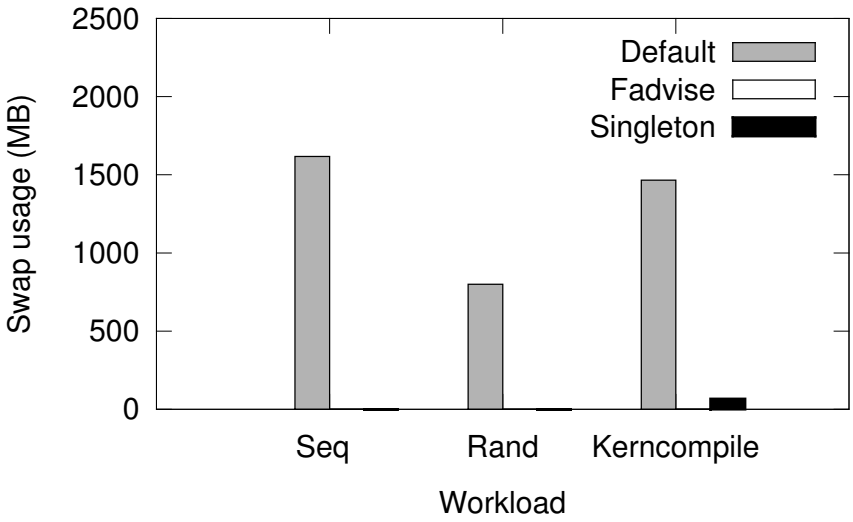


Figure 4.8: Swap occupancy

4.6.4 Memory overcommitment

The increased free memory provided by Singleton can be used to provide memory overcommitment. To measure the degree of overcommitment, the total amount of memory allocated to virtual machines is increased until the breaking-point. The breaking-point is the point at which the performance degradation is unacceptable(cannot SSH into the machine, kernel complains of a lock-up, etc) or the Linux Out-Of-Memory killer(OOM) kills one of the VMs. On a system with total 10GB virtual memory(8GB RAM + 2GB swap), 8 virtual machines(1 GB allocated to each) are able to run without crashing or being killed. Three kinds of VMs running different workloads(sequential-reads, kernel-compile, and desktop). The desktop VMs run the same OS (Ubuntu 10.04 Desktop), and benefit from the inter-VM page-deduplication, since the GUI libraries, application-binaries etc are shared across all the VMs. The number of desktop VMs were increased until the system crashed, and with Singleton we were able to run 7 desktop VMs in addition to 2 kernel-compile VMs and 2 sequential-I/O VMs (Table 4.5). A total of 11GB of memory was allocated to the VMs, with 1.5GB used by the host processes. Without Singleton, the number of VMs able to run is 8, after which the kernel initiates the Out-of-memory killing procedure, and kills one of the running VMs to reduce the memory pressure. Thus, the page deduplication and the cache-scrubbing provides a good combination for implementing memory overcommitment for virtual machines.

	Sequential	Kerncompile	Desktop	Total
Default	2	2	4	8
Fadvise	2	2	4	8
Singleton	2	2	7	**11**

Table 4.5: Number of running VMs till system crashes or runs out of memory.

4.6.5 Impact on host and guest performance

The improved cache utilization provides better performance for guest workloads. Performance for I/O intensive workloads running concurrently in four guest VMs is presented in Table 4.4. The overhead of building and maintaining a page-index periodically (done by the KSM thread) does not interfere with guest execution because of the minimal CPU resources it requires. The CPU utilization of Singleton and the system load-average during various workloads shown in Table 4.9. The CPU utilization stays below 20% on average for most scenarios. Due to the lower memory-pressure, the system load-average is significantly reduced. Most of the resource-utilization of Singleton is due to the cache-scrubbing, which needs to checksum and compare a large number of cache pages periodically. With the scrubbing turned off (only inter-VM page deduplication), our optimizations to KSM result in an average CPU utilization of just 6%, compared to 20% for the unmodified KSM.

	Avg. pages scanned/s	Cache pages dropped/s	Scan efficiency
Default	1,839,267	1459	0.07 %
Singleton	7	109	99.87 %

Table 4.6: Page eviction statistics with and without Singleton.

Another important improvement is the reduction in the number of pages that the kernel page-eviction process has to scan to evict/drop a page from memory. As mentioned earlier, the kernel maintains a global LRU list for all the pages in memory, and this list can contain millions of entries(pages). Without any proactive cache scrubbing, the cache fills up this LRU list, and the kernel needs to evict some pages in order to meet page allocation demands. The overhead of scanning a large number of pages is significant, and is one of the causes of the system load. We show the average number of pages that the kernel scans (`pgscand` of `sar` tool) during VM workload execution, and also the *scan efficiency*. The scan efficiency is defined as the ratio of the number of pages dropped to the number of pages scanned, and a higher efficiency indicates lower overhead of the page eviction process. The results are presented in Table 4.6, which shows the average number of pages scanned and the scanning efficiency for the host system during an I/O intensive workload. Because singleton drops pages which are not going to be used (since they are present in the guests), the efficiency is very high (99%). This means that 99% of all the cache pages scanned by the kernel for dropping were actually dropped, and thus the overhead of scanning paid off. In contrast, we see very low efficiency (less than 1%) in the default case. The average number of pages scanned during the eviction process is also very high (1.8 million), which also explains the low efficiency. With cache-scrubbing, there are negligible number of pages which are scanned by the swap daemon (`kswapd`), partly because of the lower memory pressure, and also because of the guest cache-content aware eviction process which ensures that only pages which might be used in the future are kept in the cache.

Guest performance isolation: The host-cache is a shared resource among guests, and it can potentially benefit the VMs. However, the host-cache is not equally or proportionally distributed amongst the VMs. VMs doing heavy I/O will have more pages in the host-cache, and can potentially interfere with the operation of the other VMs. The memory pressure induced at the host can trigger swapping of guest pages and increased page-eviction activity, resulting in decreased guest performance. By scrubbing the host-cache, Singleton is able to provide increased performance isolation among guests. With two VMs doing heavy I/O and the other two running kernel-compile workload, the I/O activity floods the host page-cache, and reduces the kernel-compile performance(Table 4.7). Scrubbing prevents this from happening, and the result is improved kernel-compile performance(6%).

In addition to providing isolation among guests, cache-scrubbing can also provide im-

	Sequential Read speed	Kernel compile time
Default	7,734 KB/s	3165 s
Fadvise	7,221 KB/s	3180 s
Singleton	7,432 KB/s	**2981 s**

Table 4.7: Impact of I/O interference on the kernel-compile workload.

	Eclipse benchmark time	Kernel compile time
Default	65 s	3300 s
Fadvise	62 s	3500 s
Singleton	**60 s**	3200 s

Table 4.8: Impact on host performance (Eclipse) due to kernel-compile workload running in VM.

proved performance for applications running in the host system. Processes running on the host (along with the virtual machines) also share the page-cache with the VMs. Without scrubbing, the cache-size can increase, and the memory pressure can adversely affect the performance of the other host-processes/VMs. A common use-case of virtualization is running desktop operating systems in virtual machines. These VMs run along with existing host processes. On a desktop-class system (3GB memory), we run one VM(1 GB memory) running the kernel-compile workload, and run the Eclipse workload on the host. This mimics a common usage pattern. The workload executing in the VM results in performance degradation on the host. With Singleton, a 10% improvement in the workload running in the host(Eclipse) is observed(Table 4.8).

4.6.6 Summary of results

The important results based on our experimental evaluation are as follows:

- Singleton provides increased host-cache utilization due to system-wide page deduplication. In our case, upto **2x** increase in host cache hit-ratios was observed with random-read workload.

	Singleton CPU %	Singleton load average	Default load average
Sequential	17.74	5.6	**12.3**
Random	19.74	4.8	**10.3**
Kerncompile	11.7	5.3	6.0
Zipf	10.2	4.9	4.9

Table 4.9: Scrubbing overhead and host load averages.

- The exclusive cache enables larger guest working sets to be present in memory, resulting in improved I/O performance in the guests, especially for random I/O, where we have observed a 40% improvement.
- Memory utilization with Singleton is significantly improved. Host cache sizes show a **2-10x** decrease. The lower memory-pressure results in much lesser swapping—with 4 VMs and over different workloads, we observed close to no swap usage.
- Singleton's page deduplication and exclusive cache enable increased levels of memory overcommitment. In our setup, we were able to run 11 VMs instead of 8 VMs without Singleton.

Chapter 5

Per-file Page-cache in Linux

5.1 Introduction

Page caches are used to cache frequently accessed data on disk, and are present in all modern operating systems. They are an important component of the memory hierarchy, and are certainly the largest component of the same. Conventionally, page-caches have been unified — there exist a single page-cache for the entire system, which consists of pages belonging to various files. The page-eviction mechanism is typically LRU, and there exists a single LRU list which tracks all the pages in the page-cache.

We argue that unified caches are suboptimal for modern workloads and usage patterns. Instead, we propose a new page-cache design, which splits the page-cache (logically) by *file*. We call our page-cache the *per-file page-cache*. Thus, each file gets its own page-cache, and can be managed differently. For example, we may have different eviction algorithms for different files depending on their access-patterns, usage, priority, etc. While these benefits may be possible to obtain in unified page-cache, the partitioned-cache design yields a much cleaner design and implementation.

Our per-file page-cache is an attempt to improve the cache utilization and memory management in general purpose operating systems, and its utility is not restricted to virtualization. In the context of KVM, the per-file page-cache allows for fine-grained control of secondary cache associated with each VMs. The virtual machines can ofcourse use the per-file page-cache if they are running the linux kernel.

5.1.1 Contributions

The following page-cache improvements have been made:

- We study the need for paritioned page-caches and disk caches.
- A per-file page-cache is implemented in the Linux kernel. Our changes significantly alter the architecture of the memory-management subsystem of the kernel.
- Preliminary experimental results suggest that the per-file cache is able to keep the size of the cache down without sacrificing the cache hit-rates.

5.2 Need for fine-grained Page-cache control

Historically, the page-cache presents a black-box optimization — its contents,policies are not exposed to the applications. We argue that allowing more fine-grained control of the page-cache should increase the effectiveness of the cache.

The reason why some control of the page-cache should be exposed to applications are :

- The page-cache occupies a very significant amount of physical memory, and thus it is important to be able to control the use of that resource.
- Because of ever increasing memory sizes, the cached memory is growing at a faster rate. Thus the need to split the giant LRU list, so as to make it more manageable, and make better eviction decisions. Because of giant CLOCK-managed lists, it is difficult to accurately estimate the relative-recency of pages — a long scanning interval implies that almost all pages will be marked as 'active'.
- While the page-cache is ubiquitous and has been for more than 3 decades in all UNIX variants, read system-calls are still expected to hit the disk. Due to lack of control over the cache, several applications (such as HTTP-caching services like Squid and Varnish) manage their own cache (in user-space). This can lead to double-caching, and increased work for application developers. If control of the cache is exposed, the caching applications can use the existing kernel infrastructure to maintain pages in LRU order etc.
- The cache hierarchy of modern computing systems are getting deeper and more complex. Virtual environments have an additional level of page-cache at the host. Solid State Devices with low read-latencies are being frequently used to augment in-memory page-caches. With more complex cache hierarchies, the need may arise to manage the kernel-maintained page-cache in a more fine-grained way. For example, on a virtual-machine host with a flash-based SSD, the hypervisor may need to decide on the cache composition of the data belonging to the various virtual machines it hosts to guarantee certain quality of service and latency requirements. A VM with a high working-set might be given a large cache on the SSD, while another might be allocated a smaller amount of in-memory page-cache at the host. Such multi-layered, multi-object, multi-agent cache optimization needs caches which allow and enable such optimizations.

While the page-cache can be reorganized in several ways, we have chosen to do so by partitioning it by file. Some of the benefits of our partitioned per-file page-cache are listed below:

Eliminating Double caching With multiple levels of page-cache present, the phenomenon of double-caching is observed. Since the disk IO operations of the guests go through the hypervisor(also called the host), the block is cached in both the host and the

guest. This wastes memory without improving the overall cache-hit rates. Furthermore, due to the unified page-cache and memory-management found in Linux(and other OSes like FreeBSD,Solaris, etc), the cache pages compete with the other pages(belonging to user processes) for memory, and can cause pages to swapped out to disk.

Eliminating Cache Interference Since the host page cache is shared between multiple VMs, the performance of Virtual Machine guests is liable to be affected by the activity of the other VMs. For example, a VM doing heavy IO can hog all the host page-cache, and even cause the pages allocated to some other virtual machine to be swapped out, severely affecting its performance. To combat this, we propose a page-cache partitioning scheme wherein we split the page-cache by file.

Controlling Cache size and eviction policy With the cache-partitioning, we can control the size of the cache allocated to the file. Furthermore, depending on the access patterns and other user-given parameters, different files can choose to run their own cache-eviction algorithm/policy.

Improved Scalability An additional benefit of page-cache partitioning is that it presents a very natural solution to the page-cache scalability and LRU-list lock contention problems which are being encountered for large-memory systems with multiple SMPs. We show that our implementation shows good SMP scalability and reduced lock contention. We have also replaced the default linux page-eviction algorithm with CAR (Clock with Adaptive Replacement, which is CLOCK approximation of the Adaptive Replacement Cache). Using CAR improves the overall cache hit-ratios in most cases.

Fadvise integration In certain situations, it is not beneficial to cache file data. For example, files which are used only once do not benefit from keeping their data in the cache after the file has been closed, etc. This pattern is extremely prevalent — files opened by common UNIX utilities like cat, cp, rsync etc all pollute the cache. While any LRU scheme will evict these pages eventually, they can potentially increase the memory pressure, and can hang-on to the LRU lists for too long. Furthermore, it may be prudent to not use memory for cache which does not contribute to performance improvement at all. The POSIX fadvise system-call allows applications to control the cache behaviour by allowing them to specify how the file is going to be used via certain flags (such as `DONT_NEED, NO_REUSE, WILL_NEED, SEQUENTIAL, RANDOM`). However, the only action of these flags is on the cache contents already present and not the future accesses. For example, DONT_NEED on linux simply drops the cache when called — future reads on the file still pollute the cache. A policy controllable cache (such as ours) allows a much more effective implementation of fadvise. We have modified the fadvise implementation so that it calls the per-file-cache, and the flags have the desired effect and are "sticky" and persistant rather than single-shot.

Energy savings With memory banks taking a significant portion of the energy budget, turning-off DRAM is a viable solution. Our per-file-cache can thus enable saving energy as well.

5.3 Background : The Page Cache

The page cache is a key piece of the memory hierarchy, and provides reduced latency for reads and writes by caching data in memory and batching writes. Essentially, the page cache maps:

$$(file, offset) \rightarrow page \tag{5.1}$$

It provides the page-frame where the data corresponding to the (file,offset) pair resides. If the data is not present in memory, it performs a disk read to fetch the data. All file I/O operations such as read,write,mmap system calls are serviced by the page cache.

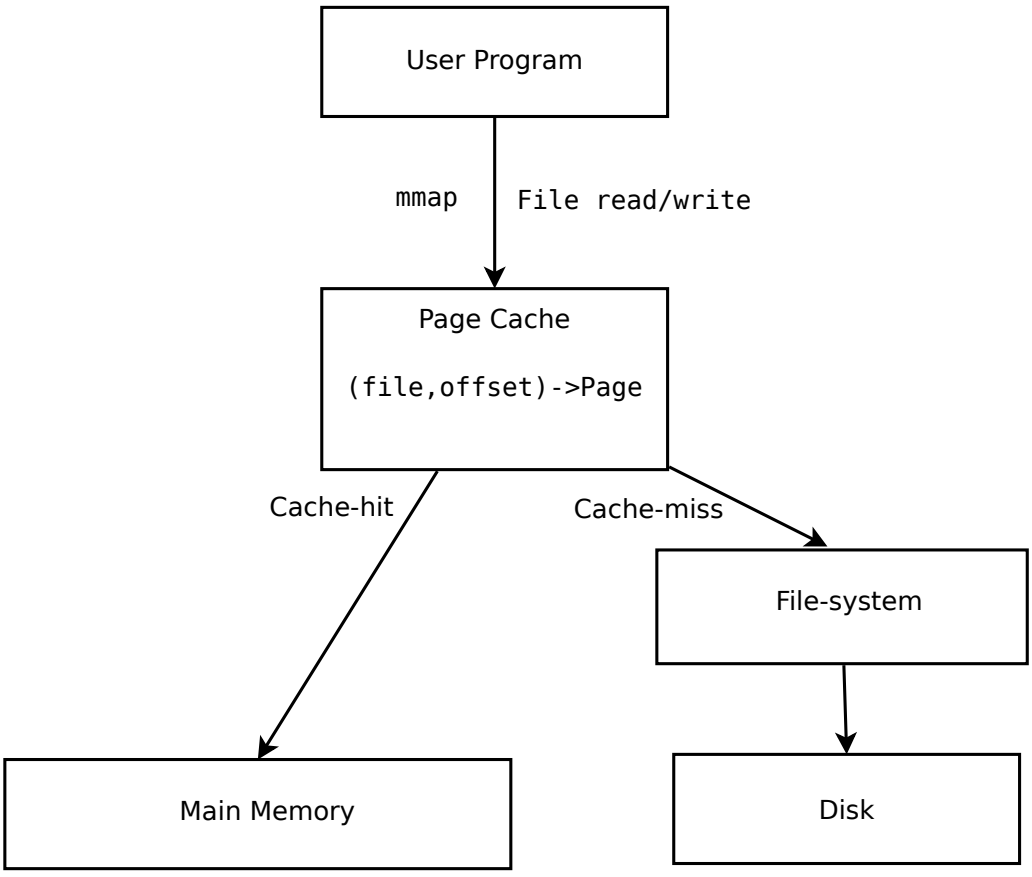


Figure 5.1: The logical architecture of the Page Cache found in almost all Operating Systems. The file I/O requests get serviced without going through the file-system itself. Mmap calls are handled the same way.

5.3.1 Buffer cache

Traditionally, UNIX has had a Buffer cache for caching disk data. The buffer cache layer was directly above the disk subsystem, and provided caching for all disk data, and deferred writes. However, the introduction of the `mmap` system call necessitated the introduction of the unified Page cache in the seminal 4.0 release of SunOS [26]. SunOS 4.0 had the first complete implementation of mmap, and that necessiated a divergence from the contemporary UNIX memory-management design. The SunOS virtual-memory architecture [26] still acts as a blueprint for the modern unix derivates, and the current linux architecture closely matches the one described in the 1987 paper.

With `mmap`, the files are accessed in a page granularity, which is typically larger than the block/buffer size. Moreover, a page cache allows a unified cache to be used for read/write and mmap system calls. Without a page cache, a separate cache would have to be maintained for `mmap`'ed files.

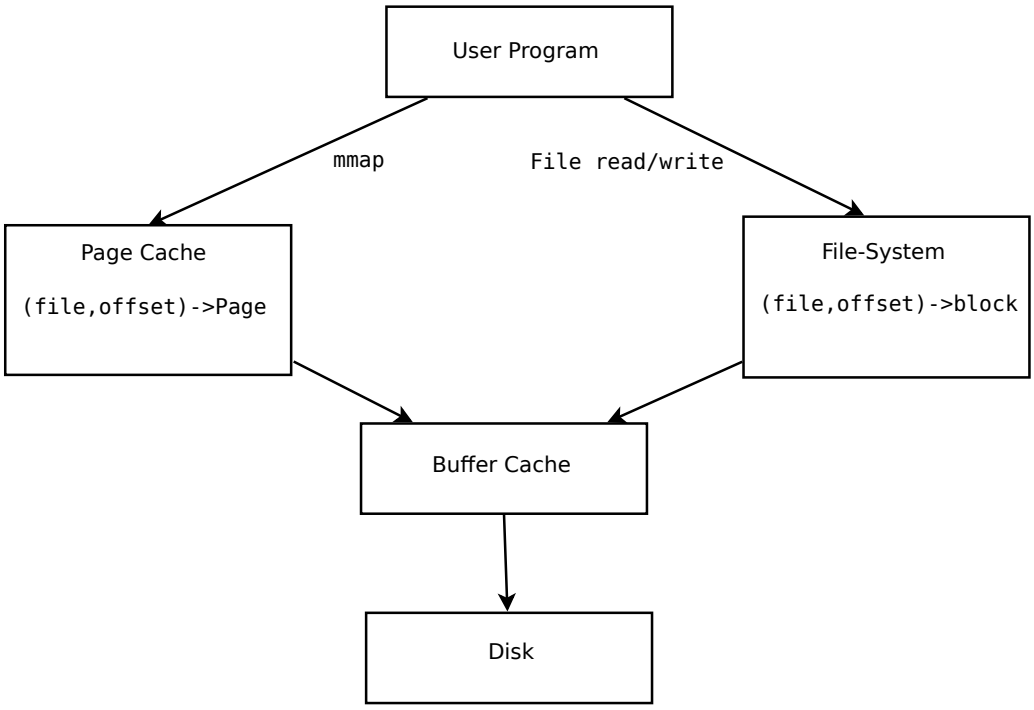


Figure 5.2: The traditional Buffer Cache architecture, as found in early versions of UNIX. Since buffer and page sizes differ, mmap handling is messy.

5.3.2 Linux Page Cache

The linux page cache implementation is very sophisticated — it implements a completely lockless read-side page cache [42], and stores the offset → page mapping in a radix-tree. By lockless, we mean that requests for pages present in the cache do not acquire a spin-lock in the absence of any writers. Writers need to take the radix-tree spin-lock to prevent concurrent writers. Readers need only take the light weight RCU read lock. RCU (Read Copy Update) [38] is a modern synchronization primitive which replaces

conventional reader-writer locks. RCU disables preemption and avoids the expensive spin-lock acquiring cost (which is equivalent to a inter-CPU cache-line invalidation). The fast 'read-lock' on RCU comes at a price — writes are slower than reader-writer locks. Hence, RCU is primarily meant for situations and scenarios where reads heavily outnumber the writes. The kernel has an RCU API for linked-lists, allowing kernel subsystems to use RCU synchronization in a safe,easy manner.

Note that the lockless design implies that readers *can* proceed in the presence of writers. The mapping returned by the radix-tree lookup might be stale or not accurate. Thus, readers only do the read on the mapping speculatively, and then lock the page itself (which is the output of the mapping), and then verify whether the page is the correct object requested. This is possible since each page contains pointers to the address-space object that it belongs to, and also the offset of the page within the file/address-space(Figure 5.3). Overall, the lockless radix-tree design is very clever, sophisticated, elegant, and cleanly implemented.

5.3.3 Linux Page eviction

Linux has a unified page-eviction mechanism, wherein every page in the system is present in a single logical LRU list(Figure 5.4). Thus, every cached page, anonymous page belonging to the processes, pages belonging to the slab-caches — all compete for staying in main-memory. This eviction process is co-ordinated by the swap-daemon (`kswapd`). If a page is a cached page and is clean (its contents match those on disk), the page is unmapped from the radix-tree, and put into the free-list. Anonymous pages are written to a swap-area and then freed.

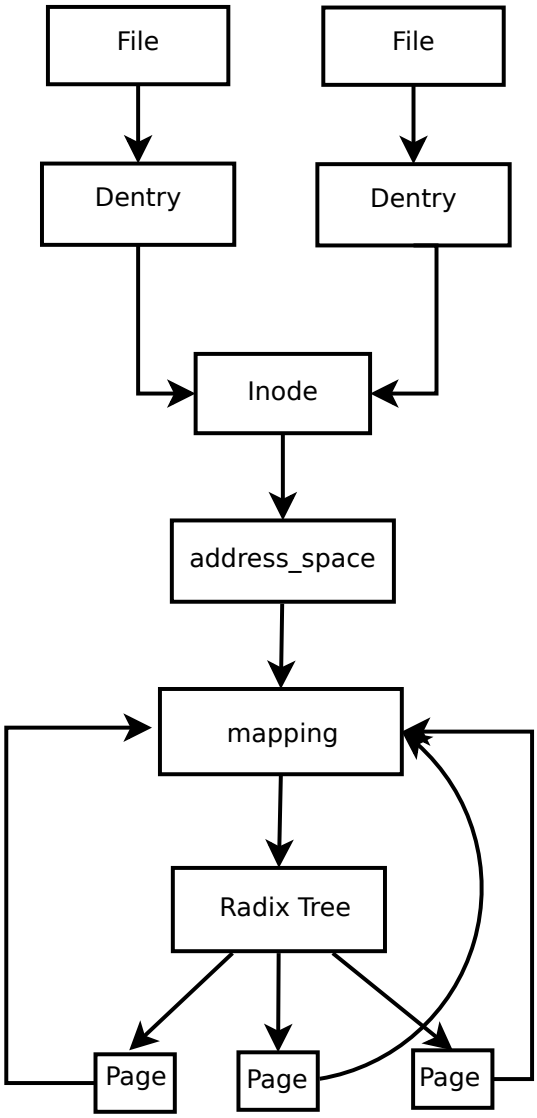


Figure 5.3: Important structures for memory management in Linux. The mapping contains the radix-tree. Multiple files can have the same inode structure. Each inode has an address-space associated with it, which contains the mapping.

The kernel implements a variant of the LRU-2 [44] page-eviction algorithm. The LRU lists are approximated by using the CLOCK heuristic — a flag(`PageActive`) is set if a page is accessed, and the CLOCK hand resets the flag during the page eviction scan of the list. Maintaining a strict LRU order is prohibitively expensive since it requires a list update on every cache-hit. On SMP systems, this would imply a spin-lock on every cache-hit. All Operating Systems adopt the CLOCK strategy and shun strict LRU ordering.

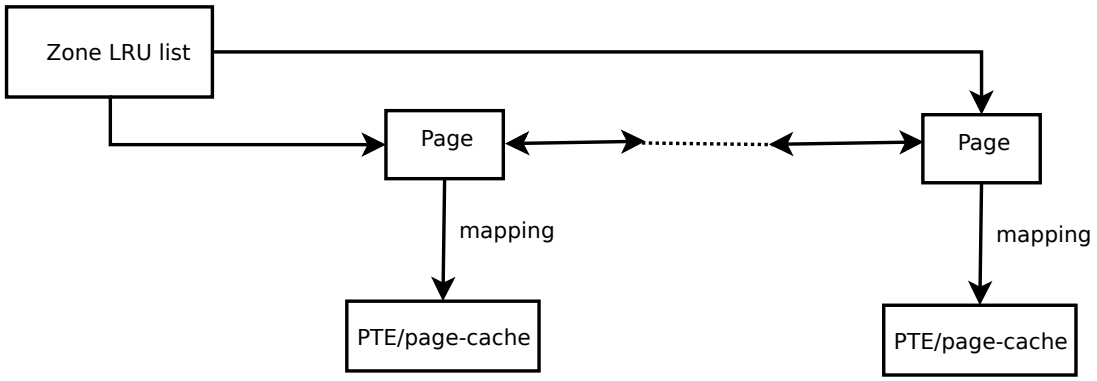


Figure 5.4: Linux global(Zone) LRU list architecture. Each page stores the object that maps it (reverse mapping). The reverse mapping is either the radix-tree mapping in case of file-backed pages, or the vm-area in case of anonymous pages. The vm-area locates the Page Table Entry (PTE) of the page.

The LRU list is divided into the Active and the Inactive list. Pages start on the inactive list, and upon being referenced are promoted to the active list. All list updates/movement happens during the scan, and not synchronously on page accesses. For eviction, the inactive list is scanned since it consists of 'cold' pages which have been accessed only once recently, and the pages are evicted/freed. If a page is marked active, it is moved to the active list and its active-flag is cleared. The active list is also scanned — pages which are active are given one more chance on the active list, while the others are moved back to the inactive list.

The LRU lists are periodically scanned, and the scanning frequency increases under memory pressure. Pages which are to be evicted are handled appropriately. The clean pages (pages which contain data which is also found on disk) are simply evicted . Anonymous pages need to be written to a swap-device. In any case, the evicted pages are then put into the free-lists, and are free for reuse. The situation is complicated by the presence of pages with the 'swap-token'. The swap-token is given to the anonymous pages which are swapped to disk. Instead of putting the pages on the free-list immediately, the pages are given the swap-token to 'hold'. If the page is requested by a process in a short time-frame after granting the token, then the swap-token is reclaimed, and the page is ready for use. This avoids bringing the page from the swap-space(disk) to memory. The swap-token is shown to increase the page-reclaim efficiency. The primary advantage of the swap token is that it reduces the blocking time of the reclaim operation.

The LRU list are protected by a spinlock (zone-lock), which needs to be taken for every operation (scan, evict, insert). The zone-lock is heavily contended, and several attempts have been made by the kernel developers to reduce the contention. The primary problem is that the lock overhead is very high. That is, the cost of acquiring and releasing the zonelock is much higher than the cost of the critical section itself. Here, the critical section either tests-and-sets some page-flag (PageActive), or evicts the page and removes it from the list, or inserts the page into some other location on the list (or some other LRU list entirely — Active → Inactive and vice versa). To counter the lock overhead, several

optimizations have been implemented. The most effective is the 'gang-locking'. Instead of taking the lock for every page, several pages are removed (isolated) from the LRU list and moved to a temporary list. Pages on this temporary list are then scanned/evicted.

5.4 Utility based cache partitioning

One of the key benefits of a partitioned our per-file cache is the cache allocation is based on the utility of the cache, and not just purely on the basis of the recency or frequency of usage, as is common in most other simple cache management techniques. A 'one size fits all' caching policy cannot handle the myriad of file sizes, access patterns, user-requirements, etc.

5.4.1 File Access Patterns

The access-pattern of a file is the trace of all the block accesses for that file over time.File access patterns have been shown to have a lot of variation. For example, some files are accessed sequentially, others show a random behaviour, while some others are cyclic. Ofcourse, the access pattern of a file is dynamic and changes depending on the application. One can observe that if the access-pattern of a file is known apriori, the effectiveness of the caching will increase. For example, if a file is known to display cyclic reference-patter, and if the cycle length is known, then the file can be allocated blocks equal to the length of the cycle. Several studies have documented file access patterns for various workloads [21]. Work by [22] also shows that the automatic classification of file access patterns (into sequential, random, cyclic, etc) can be exploited to increase the cache hit-ratio, since the caching algorithms can be tuned for a particular access-pattern. If the files shows consistent access-patterns, then the this knowledge can be exploited to move the caching strategy closer to the optimal, since the competitive-ratio of online-caching algorithms is bad only because of lack of oracle-access to the access-patterns. If the access pattern is know, an optimal algorithm can be easily implemented. While such access-pattern based caching is not implemented in any production operating system, a prefetch variant of this is found in Apple's Mac OS X Operating Systems. Instead of doing a sequential readahead for file prefetch operations, OS-X records the file's access patterns and stores it permanently on disk. When the file is opened the next time, the blocks which are present in the prefetch history are fetched. This optimization is deployed specifically for launching executables to reduce the launch-time of frequently used applications.

5.4.2 Utility vs Demand-based caching

A important advantage of partitioned caches is that they allow for utility based cache partitioning. In unified caches with a single LRU list,the cache allocation to various agents(files in our case) is demand based. That is, the files doing the most I/O get the most cache, since the pages are more likely to be recently used. The cache Utility of a

file [45], on the other hand, 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 file encountered. Marginal Utility(MU) is function of cache size, and is in fact the slope of the Miss-Ratio-Curve. Thus,

$$MU_s = miss(s+1) - miss(s) \tag{5.2}$$

A generalization is to arbitrary changes in cache sizes (the Utility U) is:

$$U_b^a = miss(a) - miss(b) \tag{5.3}$$

Where miss(s) is the number of misses that occur with a cache size of s.

The optimum partition of a cache among k files is obtained by solving this:

$$TotalUtility = U_1^{x_1}(f_1) + U_1^{x_2}(f_2) + \ldots + U_1^{x_k}(f_k) \tag{5.4}$$

Where $U(f_i)$ is the utility function of file f_i.

Assuming a cache size of F, an additional constraint is:

$$F = x_1 + x_2 + \ldots + x_k \tag{5.5}$$

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 [45] suffices for the case when there are only two competing files.

Algorithm 3 A simple greedy algorithm to find the optimum cache partitioning for two files, assuming that the miss-ratio curves are convex

```
while(free_space) {
   foreach(file) {
     size = file.size ;
     file.util = get_util_value(file) ;
     if(file.util > MAX_UTIL) {
         file.size++ ;
         free_space-- ;
      }
   }
}

int get_util_value(file f)
{
   return file.misses(file.size) - file.misses(file.size-1);
}
```

The miss-ratio graphs can be of a variety of shapes. Most workloads and applications exhibit a 'flat' graph, where the miss ratio drops to a constant value and remains constant even with an increase in the cache size. In such cases, it would not make sense to increase the cache size, since there would be no decrease in the miss-rate and hence no utility for the cache.

The LRU algorithm obeys the stack property, meaning that a larger LRU-managed cache always has more hits than a smaller one, given the same access pattern. This allows the calculation of the number of misses for all the sizes smaller than the current one.

Sampling techniques can be used. To exploit the stack property, a the page must be identified in the LRU list, and its access-counter updated. This technique may not be applicable in the case of LRU approximations like CLOCK.

5.4.3 Challenges in MRC generation

In this section we enumerate some of the challenges in generating Miss-Ratio Curves (MRC) for every file. An ideal cache-partitioning scheme minimizes the total misses, given the a fixed cache-eviction algorithm. This is done by finding the minimum value of the sum of all the miss-ratio-curves. Thus the first challenge is to obtain the miss-ratio curve. If the eviction algorithm is LRU or its variant, then the stack-distance [] property can be used. The stack-distance property allows the calculation of the number of misses that would have occurred had the cache been smaller than its current size. Thus, we can obtain the number of misses for all cache sizes smaller than the current size. The stack-distance algorithm has a time-complexity of O(n), where n is the number of pages in the cache. This must be repeated for all cache sizes, which brings the total time complexity of generating a full Miss-Ratio Curve to be O(n*n).

While MRCs have been shown to be generated with low costs for a variety of cases such as hardware caches and storage caches, the cost is simply too prohibitive if done for each file inside the kernel. Some of the other reasons why generating MRCs inside the kernel is challenging are:

- Most stack-distance based MRC construction techniques need an access-trace. That is, every access of the cache must be recorded. In an operating system page cache, cache hits must be fast. In the default configuration, the kernel simply performs the index to page mapping and returns the page, with no additional computation. We do not intend to make this fast-path slow due to recording every access and storing it. Standard techniques like hash-tables [] which are used for stack-distance calculation cannot be easily used inside the kernel, because dynamically allocated memory is scarce, and there is no bound on the number of accesses. Also, there is very limited apriori information about the number of accesses and the eventual size of the file-cache. Thus, even allocating hash-table buckets might prove to be tremendously wasteful, since the in-kernel memory footprint of small-files (due to the page-cache book-keeping) might be much larger than the file itself.

- There is a high variability in the number and sizes of the files. This rules out any algorithm which is linear in the size of the files or the number of files.
- Lastly, the access patterns for a file access are dynamic, hence the MRC construction needs to be done dynamically.

These are the primary reasons why we have chosen to shun the standard MRC based cache partitioning approach.

Miss-ratio curves can also be obtained by assuming a Zipf-like access pattern and using modelling techniques to obtain fairly accurate curves [35].

5.5 Per-file Cache Design

Throughout the rest of this document, we shall use the terms file, inode, address-space interchangeably. While the relation between files and address-spaces is not one-to-one, and multiple files may have the same address-space structure associated with them, we are only interested in the address-space.

We have re-designed the linux page cache with the following major changes:

1. Each address-space i contains its own fixed-size cache C_i. There is a many-to-one relation between files and the address-spaces.
2. Exclusion from system(zone) LRU lists. The pages belonging to the file cache are removed from the zone LRU lists by putting them on the Un-evictable LRU list. This prevents the swap daemon from touching these pages.
3. Each address-space maintains its own cache state: the size, eviction-algorithm, etc are all configurable via the sysfs interface. The existing linux LRU/2 implementation is replaced by CLOCK approximation of the Adaptive Replacement Cache(ARC).
4. 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 exceed a fixed limit (F-Size). The reaper thread maintains a list of all inodes ordered by LRU order of file accesses. To approximate the LRU, we use a 4-chance CLOCK — a file is declared 'cold' after being given four scans during which it has the chance to get accessed again.

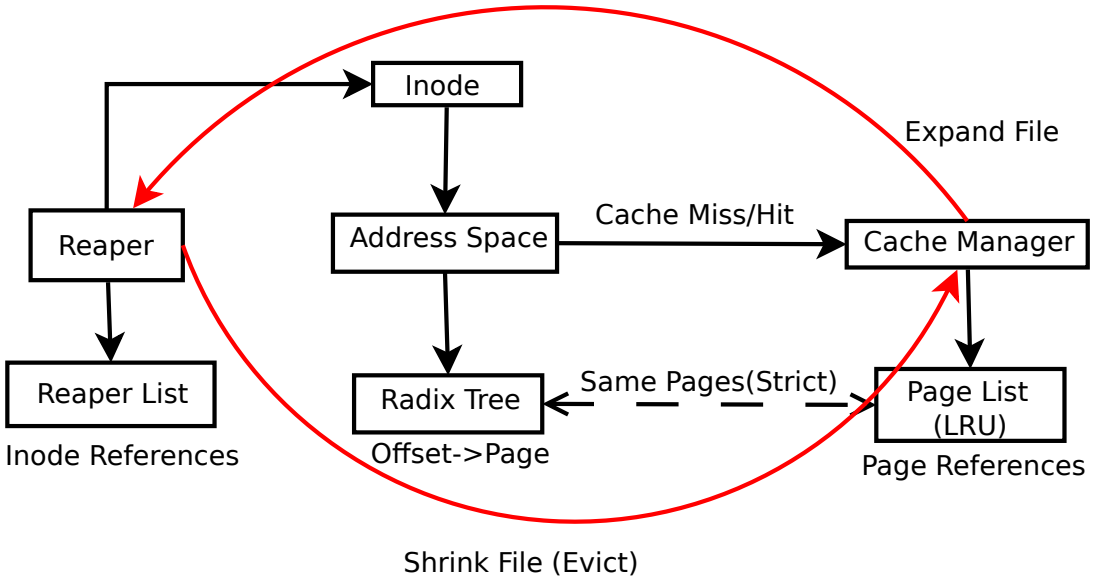


Figure 5.5: The architecture of the per-file-cache. Arrows indicate flow of control or data-dependence.

We now present in detail our design (Figure 5.5). Each file open corresponds to an in-kernel inode creation, which is added to the super-block inode-list of the corresponding file-system's super-block. In addition to this list, we also add the inode to a 'reaper-list'. The reaper-list contains all the inodes which are referenced, and may contain inodes from different superblocks. The reaper-list is protected by a reaper-lock spinlock, which protects against concurrent additions and removals. In Linux, inodes are lazily deleted if there is no reference to them. Typically, this happens after all the files which correspond to the inode have closed, and there are no dentry-cache entries pointing to the inode either. Thus, inode reclaim is either an effect of dentry objects being freed, or a reduction in the inode-cache slab-cache. The slab-cache reclaim is either manual (via sysfs) or performed by vmscan page-eviction to keep the sizes of slab-caches in check.

In the per-file cache, the pages belonging to file address-spaces are not put on the global file-LRU lists. Instead, all pages of a file's address-space are handled by its corresponding cache algorithm. The cache algorithm handles radix-tree hits, misses, and deletes. This way, we achieve isolation between files, and allows us to have different cache configurations and algorithms for different files. The interface for dealing with files and address-spaces remains the same (via `find-get-page`). Since most cache algorithms do not update the LRU list on every access and instead just toggle a bit, we have provided a `PageActive` bit for every page, which can be used by any page-eviction algorithm.

Cache-manager. A key feature of our design is that all the caching decisions are local, and are made by the file's cache manager. The cache manager is delegated the task of handling the file's cache pages for the purpose of eviction. 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 ARC

and FIFO.

The cache manager is required to implement three interfaces :

1. Update cache state on cache-hit
2. Update cache state on cache-miss
3. handle eviction request

CAR: The CAR(Clock with Adaptive Replacement) [10] algorithm is the default cache-manager algorithm for page-evictions. CAR is a CLOCK approximation of the ARC(Adaptive Replacement Cache) [39], and it eschews the strict LRU lists for CLOCKs. CAR has the same features of ARC : split LRU lists and shadow-lists, and has been shown to have performance very close to that of pure ARC. Our implementation of CAR very closely mirrors the canonical one [10]. CAR was chosen because of the benefits of ARC (scan-resistance, adaptive-nature, absence of tunable magic-parameters). Since CAR uses CLOCKs, the cache-hits do not need to update any LRU list. Updating LRU lists is expensive since it requires acquiring a spin-lock on every page-cache hit. Note that the linux page-cache mapping (via radix-tree) is itself lockless for reads, hence it would be prohibitively wasteful to block on list-updates. Figure 5.6 describes the high-level flow-chart of our CAR implementation.

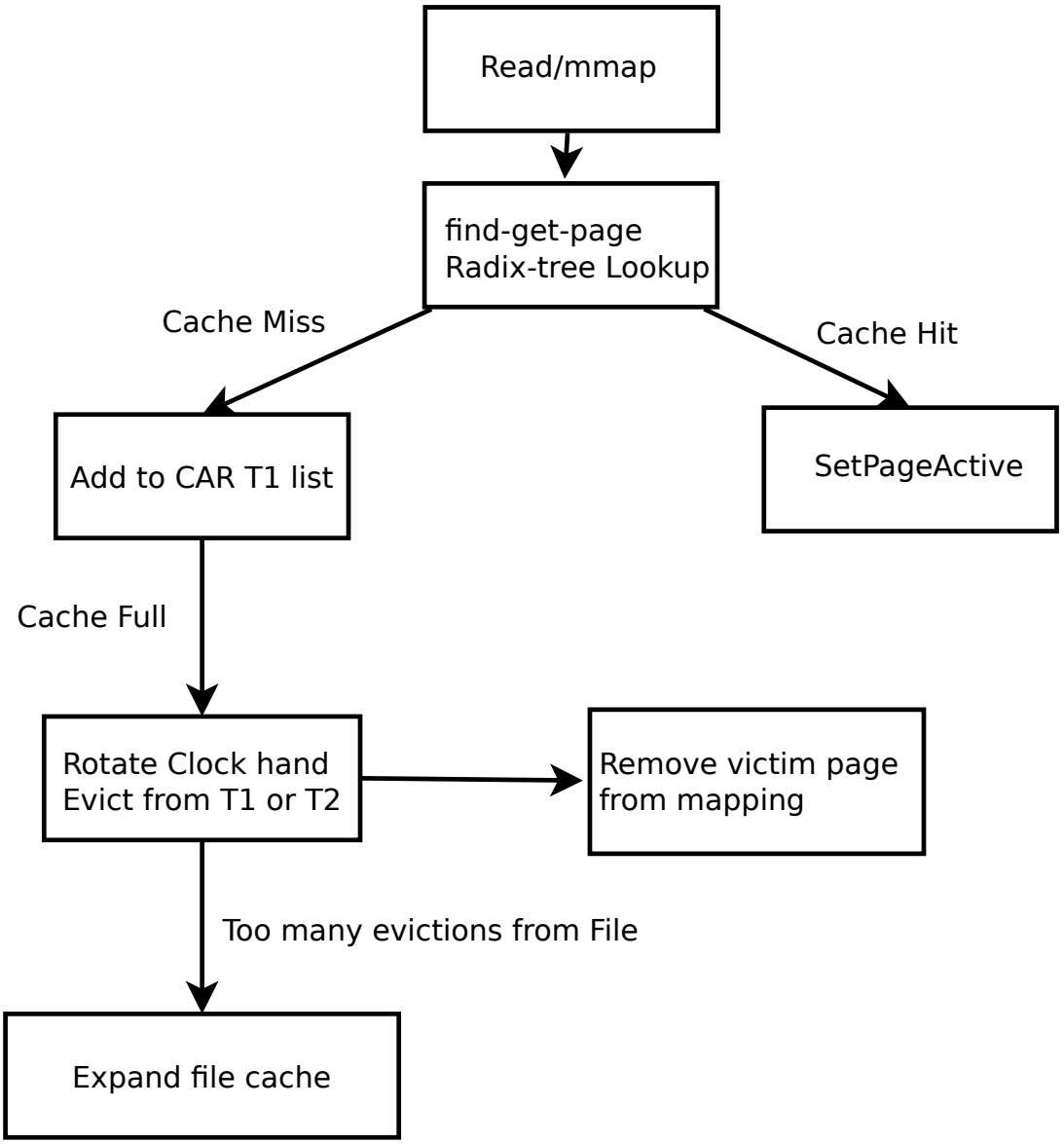


Figure 5.6: Flow-chart of the CAR algorithm implemented. CAR, which acts as the cache-manager for a file, requests the reaper for additional space if it is forced to perform too many evictions

Reaper thread The reaper-thread(Figure 5.7 is a kernel thread which balances the size of each file's cache as well as the global file-cache size (total number of pages present in all file caches).

The primary weapon of the reaper is the ability to evict/reap pages. The reaper maintains a list of all the file inodes, and requests the corresponding cache-manager to evict pages. The files 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 files are the victims of the reaping, and the file's cache-manager gets to choose which pages to evict from their cache.

The reaper implements three important functions:

1. Make space for new file

2. Make space for a file asking for more cache

3. Shrink total file cache if pressure on anonymous memory is high.

The reaper is called by the cache managers(Figure 5.5) when they request for additional space(either for a new file, or if the cache manager deems that the file 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. The eviction flow of control is described in 5.8.

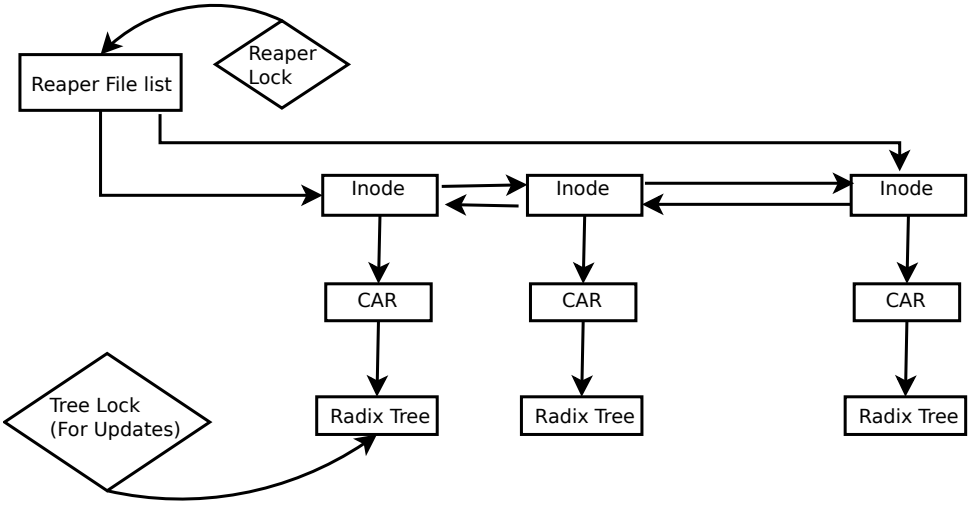


Figure 5.7: The reaper list. Each entry is an inode. The list is protected by the reaper-lock, and the nodes are protected by the tree-lock and inode-lock

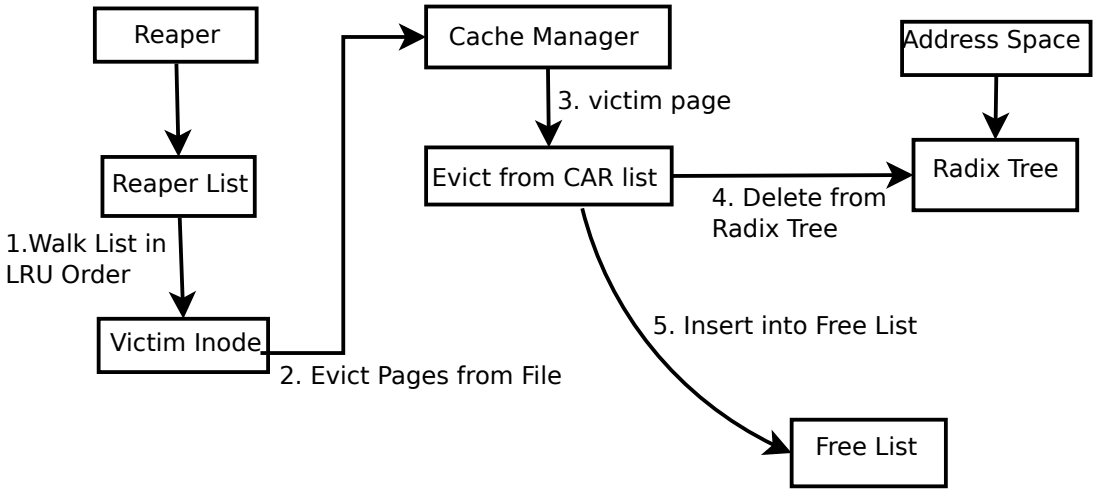


Figure 5.8: Flow of control during a page eviction. The cache manager(CAR) is always responsible for evicting pages. On successfully deleting from the radix-tree, the page is put into the free-list and then reused.

5.5.1 Synchronization Issues

Our implementation is SMP-ready, and a few challenges were faced in resolving deadlocks and page-cache safety.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 evic 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 long 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.

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 and the dentry mechanism.

CAR-lock. The cache manager (CAR in our case) needs to protect its page index against concurrent reads to 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.

2. Page evictions

In the CAR algorithm, the transitions between shadow lists occurs on cache misses only. Cache hits only affect the T_1/T_2 ratio (sizes of the two CAR lists respectively).

5.6 Utility based Adaptive Partitioning

In this section we describe the adaptive nature of the per-file cache. While our per-file cache implementation can be used for implementing strict limits on the cache-sizes for various files etc(through a `sysfs` interface), general situations demand an adaptive solution.

For systemwide page caches, there exist two important dynamically changing values: the number of pages cached for a given file, and the total number of file-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 partitioned page cache, manipulating these parameters (cache size of each file and the global cache size) is an explicit task. While very sophisticated marginal-utility based approaches can be attempted, we have implemented very simple, proof-of-concept

adaptive algorithms 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 definition of our problem:

Each address-space i , when instantiated, is soft-allocated C_i number of pages. This represents the maximum possible size that it can grow. If this maximum size is reached, then there are two possible cases:

1. The page-eviction algorithm of i evicts pages to make room for new pages.
2. The address-space asks for an increase in C_i.

The cache-partitioning problem is thus: Given n files, with a total of M physical memory 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's vmscan. There may be user and system-defined constraints on the minimum and maximum limits for each C_i. The objective function is to assign values to C_i so as to minimize the expected number of cache-misses, given recent cache access history. As mentioned earlier, an optimal solution to the cache partitioning problem is not feasible to calculate inside the kernel's hot cache-hit path.

5.6.1 File Cache size

If there is a lack of free space ($F = M$), and if a file needs grow, then some other files must evict pages. The magnitude of growth is a function of its utility. If the file's utility is greater than the global average utility (of all the files so far) then the file has high utility, and will presumably benefit from the extra cache. If free space exists, then the file's soft-limit C_i is increased in proportion to the free-space and the number of files and their utilities.

An important 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 file is a sequential file, which will most likely not benefit from the cache. Therefore the utility is calculated like:

$$Utility = \frac{Misses - ReadAheadSuccesses}{Accesses} \tag{5.6}$$

Also, files which have been advised via fadvise are handled appropriately in the cache-growth algorithm. Small or use-once files do not get additional cache even if they ask.

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, since we can potentially also use the shadow-list success-rate as a guide for sequentialness and utility. That is, a file 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.

5.6.2 Total File-cache size

In our current implementation, the total space allocated for all the files in the cache F, keeps growing until the system starts to swap. On swapping, it decreases to reduce the memory pressure.

A more sophisticated approach would be to integrate with the existing page-eviction metrics of pages scanned and pages evicted, which are used by kswapd to determine the proportion of file and anonymous pages to keep/evict.

5.7 Performance Evaluation of Per-file Cache

This section presents preliminary experimental results to test the effectiveness of the per-file cache.

5.7.1 Setup and Workloads

To test the effectiveness of our cache, we run multiple I/O intensive workloads. The workloads are described in Table 5.1. All I/O workloads are generated by using fio (flexible I/O tester) , and the working set size of each workload is atleast two times larger than the total memory available.

Workload	Description
rand-seq	2 processes doing sequential and random file reads.
kerncompile	Kernel compile(Linux 3.0). 3 threads are used.

Table 5.1: Workload description.

5.7.2 Cache size

Case	Average Cache Size	Max size
Default	372	400
PFC	40	44

Table 5.2: Cache sizes with the rand-seq workload

Table 5.2 compares the average and maximum size of the page-cache for the random-sequential workload 5.1. With the per-file 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.

5.7.3 I/O Performance

Case	Random	Seq
Default	390	4488
PFC	388	4668

Table 5.3: IO performance (IOPS) for the random-sequential workload. Per-file ca che's IO performance is very close to that of the default.

Case	Hit-ratio
Default	0.622
PFC	0.714

Table 5.4: Cache hit ratios. We see a 15% improvement.

The I/O performance is shown in Table 5.3. The I/O performance in this case is not perturbed much. However, as Table 5.2 shows, the per-file cache uses an order of magnitude less cache size to achieve the same performance. This 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 a 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 5.4. 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.

Case	Compile time	Average Cache size
Default	293 s	120 MB
PFC	268 s	100 MB

Table 5.5: Kernel compile times. Per-file cache does about 10% better than default.Cache hit ratios are 0.984792 for pfc and 0.989283 for default

5.7.4 Overhead

Case	Average Load	Max Load
Default	1.0	1.40
PFC	0.8	1.09

Table 5.6: Load average during the random-sequential workload. The load is about 40% less with the per-file-cache.

Since several key components of the memory-management subsystem have been substantially modified, the performance of our unoptimized system was expected to be inferior. However, as Table 5.6 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. In profiling runs conducted, it has been found that the shadow-list linear search function is the most expensive component of the entire setup. Replacing the linear search by a hash table is part of future work, but several challenges reamin to be tackled. For example, the size of the hash-table. If the number of buckets is too big, then the memory-footprint for small files will be large. However we do need large hash-tables for large files, and we do not know apriori if a file will occupy a large cache footprint.

Algorithm 4 Algorithm for expanding a file's cache

```
static int expand_file(struct CAR_state *CAR_state)
{
int dAccess = CAR_state->dAccess ;
int dMisses = CAR_state->dMisses ;
int dRA = CAR_state->dRA_success ;
int MU_i =  dAccess/(dMisses+dRA+1) ;
int global_MU_i = (rstat->dHits+rstat->dMisses)/  \
                  (rstat->dMisses+rstat->dRA) ;
enum CAR_filetype type = CAR_state->CAR_filetype;
if (type == PFT_small || type == PFT_dontneed) {
return 0; /* No pages for you. */
}
int MU_inv = dAccess/(dMisses+1) ;
to_grow = (CAR_MISS_DELAY/2)*(dAccess+dMisses)/  \
                              (dMisses+1) ;
/* This function is called only after a certain number
 of misses from a file. Say,10. Thus worst case slowdown is 10.
 If empty space exists, grab it? */
if (file_cache_full()) {
pages_freed = do_reaping(CAR_state, to_grow);
CAR_state->C += pages_freed;
} /* Else, we have enough space. Allocate the request. */
else {
rstat->soft_alloc += to_grow;
CAR_state->C += to_grow;
}
return pages_freed;
}
```

Chapter 6

Conclusion & Future Work

In this thesis, we have looked at the problems of page-cache management in virtualized environments. An exclusive-cache solution using page deduplication (Singleton) has been developed. Singleton allows risk-less memory overcommitment, allowing more virtual machines to run on given hardware. Singleton represents the first such exclusive-cache solution for KVM.

A closer examination of the linux page-cache has been done, and a new design (the per-file cache) proposed and implemented. We believe that managed page-caches are the future of memory-management because of ever increasing memory sizes and deepening memory hierarchies. Our per-file page-cache is the first such modern step in that direction, and has several axillary benefits like fadvise integration, utility-based partitioning, per-file caching algorithm selection, etc. We have implemented a highly efficient modern caching algorithm (ARC : Adaptive Replacement Cache) to obtain higher hit ratios with an order of magnitude smaller cache sizes.

Lastly, the underlying theme of this work is the idiosyncrasy of resource management by operating systems in virtual environments. Resources are now managed by two agents completely oblivious of each other (the guest and host OS). This leads to several situations where the two agents enact redundant optimizations and services. The real impact on performance is when the optimizations are complementary, and reduce the performance, as happens in disk-scheduling. While we have focused primarily on page-caches and disk I/O in this work, the idea of resolving the dual-agent conflict is almost immediately applicable to the problem of OS scheduling, paging, security in virtual environments. It is hoped that operating systems of the future have a greater awareness of their place in the virtual machine hierarchy (whether they are hosts or guests), and the work presented herein represents a tiny step in that direction.

6.1 Future Work

Page deduplication

- Extend the dirty-bit guided scanning for AMD NPT to Intel's EPT. This is challenging since EPT does not expose the dirty-bit to the hypervisor.
- Page deduplication in the real world : A comprehensive study about the prevalence and effectiveness of page-deduplication would give a good idea about its benefits.
- KSM for host pages. Currently, KSM only scans anonymous memory areas. However if VM self-sharing is high, then it could be used for host pages as well, and eliminate duplicate libraries etc loaded.

Exclusive Caching

- Eviction based exclusive caching for KVM, by tracking QEMU disk read requests and mapping them to guest reads. Tracking evictions(black-box) is still an open problem.
- Some more performance measurements for different guest operating systems.

Per-File cache

- The performance profiling done is preliminary and needs to be extended and improved. Workloads with combinations of several benchmarks, memory-sizes, cache-parameters (size, policy) etc all need to be varied, and the performance impact of per-file cache measured.
- The CAR shadow list hit-rate and T1-T2 ratio can be used to guide the utility function.
- Measurement of how the new cache design impacts second-level caching — by running multiple VMs.
- Implementation of various caching algorithms like LIRS,MQ, etc to supplement CAR.
- Replace CAR shadow lists by a different data structure.

Bibliography

[1] AMD-V Nested Paging. http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf.

[2] Bonnie++ File System Benchmark. www.coker.com.au/bonnie++/.

[3] Eclipse IDE. http://eclipse.org/.

[4] Linux Kernel Mailing List. http://www.mail-archive.com/kvm@vger.kernelorg/msg30649.html.

[5] C.J. Ahn, S.U. Choi, M.S. Park, and J.Y. Choi. The design and evaluation of policy-controllable buffer cache. In *Parallel and Distributed Systems, 1997. Proceedings., 1997 International Conference on*, pages 764–771. IEEE, 1997.

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

[7] A. Arcangeli, I. Eidus, and C. Wright. Increasing Memory Density by using KSM. In *Proceedings of the Linux Symposium*, pages 19–28, 2009.

[8] Ö. Babaolu. *Hierarchical replacement decisions in hierarchical stores*, volume 11. ACM, 1982.

[9] L.N. Bairavasundaram, M. Sivathanu, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. X-ray: A Non-invasive Exclusive Caching Mechanism for Raids. In *31st Annual International Symposium on Computer Architecture.*, pages 176–187, 2004.

[10] 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.

[11] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, pages 41–49, 2005.

[12] T. Bird. Measuring Function Duration with Ftrace. In *Proceedings of the Japan Linux Symposium*, 2009.

[13] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM SIGPLAN Notices*, volume 41, pages 169–190, 2006.

[14] D. Boutcher and A. Chandra. Does Virtualization Make Disk Scheduling Passé? *ACM SIGOPS Operating Systems Review*, 44(1):20–24, 2010.

[15] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM SIGOPS Operating Systems Review*, 31(5):143–156, 1997.

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

[17] 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.

[18] 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.

[19] C.R. Chang, J.J. Wu, and P. Liu. An Empirical Study on Memory Sharing of Virtual Machines for Server Consolidation. In *IEEE Symposium Parallel and Distributed Processing with Applications (ISPA)*, pages 244–249, 2011.

[20] Z. Chen, Y. Zhou, and K. Li. Eviction-based Cache Placement for Storage Caches. In *Proceedings of USENIX Annual Technical Conference*, pages 269–282, 2003.

[21] J. Choi, S.H. Noh, S.L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. *ACM SIGMETRICS Performance Evaluation Review*, 28(1):286–295, 2000.

[22] 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. *Computers, IEEE Transactions on*, 51(7):793–800, 2002.

[23] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, 2005.

[24] C.D. Cranor and G.M. Parulkar. The uvm virtual memory system. *Proceedings of the 1999 USENIX Annual Technical*, 168:117–130, 1999.

[25] B.S. Gill. On Multi-level Exclusive Caching: Offline Optimality and why Promotions are Better than Demotions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–17, 2008.

[26] R.A. Gingell, J.P. Moran, and W.A. Shannon. Virtual memory architecture in SunOS. 1987.

[27] D. Gupta, S. Lee, M. Vrable, S. Savage, A.C. Snoeren, G. Varghese, G.M. Voelker, and A. Vahdat. Difference engine: Harnessing Memory Redundancy in Virtual Machines. *Communications of the ACM*, pages 85–93, 2010.

[28] X. He, M.J. Kosa, S.L. Scott, and C. Engelmann. A Unified Multiple-level Cache for High Performance Storage Systems. *International Journal of High Performance Computing and Networking*, 5(1):97–109, 2007.

[29] M. Jeon, E. Seo, J. Kim, and J. Lee. Domain level Page Sharing in Xen Virtual Machine Systems. *Advanced Parallel Processing Technologies*, pages 590–599, 2007.

[30] S.T. Jones, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. In *Proceedings of ASPLOS*, pages 14–24, 2006.

[31] 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.

[32] J.F. Kloster, J. Kristensen, and A. Mejlholm. Efficient Memory Sharing in the Xen Virtual Machine Monitor. Technical report, Aalborg University, 2006.

[33] J.F. Kloster, J. Kristensen, and A. Mejlholm. On the Feasibility of Memory Sharing: Content-based Page Sharing in the Xen Virtual Machine Monitor. Technical report, Aalborg University, 2006.

[34] J.F. Kloster, J. Kristensen, and A. Mejlholm. Determining the use of Interdomain Shareable Pages using Kernel Introspection. Technical report, Aalborg University, 2007.

[35] I. Kotera, R. Egawa, H. Takizawa, and H. Kobayashi. Modeling of cache access behavior based on zipf's law. In *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*, pages 9–15. ACM, 2008.

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

[37] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Transcendent Memory and Linux. In *Proceedings of the Linux Symposium*, pages 191–200, 2009.

[38] P.E. McKenney and J.D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[39] 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.

[40] G. Miłos, D.G. Murray, S. Hand, and M.A. Fetterman. Satori: Enlightened Page Sharing. In *Proceedings of USENIX Annual technical conference*, pages 1–14, 2009.

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

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

[43] W.D. Norcott and D. Capps. Iozone Filesystem Benchmark. www.iozone.org.

[44] 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.

[45] 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.

[46] 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.

[47] S. Rhea, R. Cox, and A. Pesterev. Fast, Inexpensive Content-Addressed Storage in Foundation. In *USENIX Annual Technical Conference*, 2008.

[48] R. Russell. VirtIO: Towards a de-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[49] J.H. Schopp, K. Fraser, and M.J. Silbermann. Resizing Memory with Balloons and Hotplug. In *Proceedings of the Linux Symposium*, pages 313–319, 2006.

[50] M. Schwidefsky, H. Franke, R. Mansell, H. Raj, D. Osisek, and J.H. Choi. Collaborative Memory Management in Hosted Linux Environments. In *Proceedings of the Linux Symposium*, 2006.

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

[52] C. Tang. FVD: a High-Performance Virtual Machine Image Format for Cloud. In *Proceedings of the USENIX Annual Technical Conference*, pages 229–234, 2011.

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

[54] C.A. Waldspurger. Memory Resource Management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, pages 181–194, 2002.

[55] 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.

[56] T.M. Wong and J. Wilkes. My cache or Yours? Making Storage More Exclusive. In *Proceedings of USENIX Annual Technical Conference*, pages 161–175, 2002.

[57] 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. *ACM SIGOPS Operating Systems Review*, pages 31–40, 2009.

[58] 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.

[59] Z. Zhang, A. Kulkarni, X. Ma, and Y. Zhou. Memory Resource Allocation for File System Prefetching: from a Supply Chain Management Perspective. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 75–88, 2009.

[60] Y. Zhou, Z. Chen, and K. Li. Second-level Buffer Cache Management. *IEEE Transactions on Parallel and Distributed Systems*, pages 505–519, 2004.

[61] 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*, pages 91–104, 2001.

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

[63] K. Jin and E.L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, 2009.

Appendix A

Kernel changes for Singleton

```
Documentation/vm/page-types.c                        |    3 +-
 arch/x86/include/asm/kvm_host.h                     |    4 +
 arch/x86/include/asm/pgtable_32.h                   |    1 +
 arch/x86/kvm/x86.c                                  |   83 +
 arch/x86/mm/gup.c                                   |    1 +
 drivers/block/virtio_blk.c                          |    9 +-
 drivers/net/usb/dm9601.c                            |    1 +
 fs/drop_caches.c                                    |   42 +
 fs/proc/page.c                                      |   90 +-
 fs/proc/task_mmu.c                                  |    2 +-
 include/linux/ksm.h                                 |    4 +-
 include/linux/mm.h                                  |    3 +
 include/trace/events/ksm.h                          |  144 +
 mm/filemap.c                                        |   27 +-
 mm/ksm.c                                            | 4999 ++++++++++++++++------
 mm/rmap.c                                           |    5 +-
 mm/truncate.c                                       |   50 +-
 mm/vmscan.c                                         |  109 +-
 samples/trace_events/trace-events-sample.h          |    2 +
 .../perf/scripts/python/bin/syscall-counts-record   |    2 +-
 virt/kvm/kvm_main.c                                 |  215 +
 21 files changed, 4362 insertions(+), 1434 deletions(-)
```

Appendix B

Kernel changes for Per-File Cache

```
Makefile                          |    2 +-
fs/drop_caches.c                  |    2 +-
fs/inode.c                        |   15 +-
fs/proc/base.c                    |   25 +-
fs/super.c                        |    1 +
include/linux/fs.h                |   54 ++-
include/linux/per-file-cache.h    |  185 +++++++
mm/Kconfig                        |    1 +
mm/Makefile                       |    2 +-
mm/fadvise.c                      |    6 +-
mm/filemap.c                      |   20 +-
mm/page_alloc.c                   |    5 +-
mm/per-file-cache.c               |  494 ++++++++++++++++++
mm/pf_cart.c                      |  711 ++++++++++++++++++++++++++
mm/pfifo.c                        |   87 +++
mm/reaper.c                       | 1135 ++++++++++++++++++++++++++++++++++++++++
mm/truncate.c                     |    1 +
mm/vmscan.c                       |   19 +-
18 files changed, 2722 insertions(+), 43 deletions(-)
```

Appendix C

Kernel changes For AMD NPT Dirty-bit KSM scanning

```
arch/x86/include/asm/kvm_host.h |    1 +
arch/x86/kvm/mmu.c              |   36 +
arch/x86/kvm/mmu.h              |    3 +-
arch/x86/kvm/vmx.c              |    3 +
arch/x86/mm/hugetlbpage.c       |    2 +
fs/inode.c                      |    3 +
fs/read_write.c                 |    2 +
include/linux/fs.h              |   22 +
include/linux/mm.h              |    1 +
include/linux/mm_types.h        |    2 +
include/linux/mmu_notifier.h    |   48 +
include/linux/pagemap.h         |    1 +
include/linux/rmap.h            |   26 +-
mm/Makefile                     |    1 +
mm/filemap.c                    |   43 +-
mm/filemap_xip.c                |    6 +-
mm/ksm.c                        |  189 +-
mm/mmap.c                       |    3 +
mm/mmu_notifier.c               |   33 +
mm/rmap.c                       |   61 +-
mm/swap.c                       |    1 +
mm/vmscan.c                     |    1 +
virt/kvm/kvm_main.c             |   27 +
24 files changed, 6344 insertions(+), 66 deletions(-)
```