Singleton: System-wide Page Deduplication in Virtual Environments

Prateek Sharma   Purushottam Kulkarni

Department of Computer Science & Engineering
Indian Institute of Technology Bombay
{prateeks,puru}@cse.iitb.ac.in

ABSTRACT

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.

Categories and Subject Descriptors

D.4.2 [**OPERATING SYSTEMS**]: Storage Management—*Main memory, Storage hierarchies*; C.4 [**PERFORMANCE OF SYSTEMS**]: Performance attributes

General Terms

Performance

Keywords

Virtualization, Caching, Page-deduplication

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 [8].

In this paper 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 [35]. 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 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) [18], 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 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
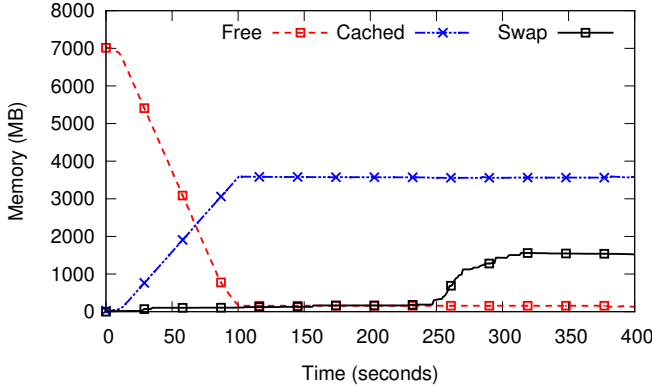
Figure 1: Memory usage graph for a sequential disk-read workload.

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 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 paper 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 [15, 11, 37] and CPU architectural caches, our work is the first to focus on exclusive caching in KVM-based environments. Furthermore, we implement a completely *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 [17] and XRAY [5]. 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 [13, 37, 40, 41].

Page deduplication across Virtual Machines [35, 19, 20] 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

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.

2. BACKGROUND

Singleton presents an exclusive cache solution for KVM, and uses the KSM page deduplication infrastructure. This section presents the relevant background which will help motivate our solution. Some of the optimizations which we have added to KSM to reduce the page-sharing overhead are also presented.

2.1 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 [6] 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.
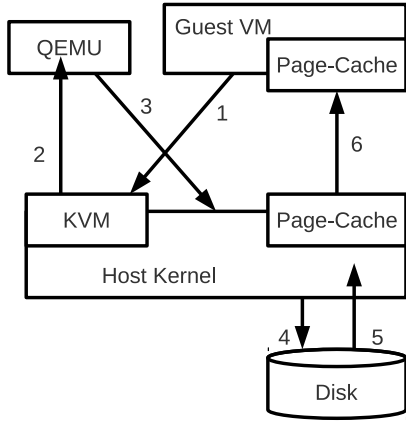
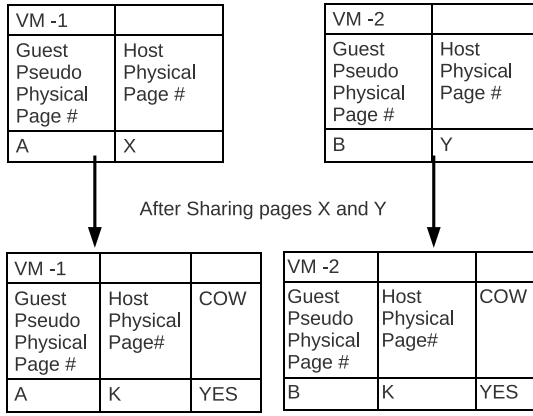**Figure 2: Sequence of messaages to fulfill an I/O operation by a guest VM.**

After Sharing pages X and Y

Figure 3: Copy-on-Write based hypervisor level page sharing.

2.2 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 2 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.

2.3 Linux page-cache and page eviction

The Linux page-cache [25] 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.
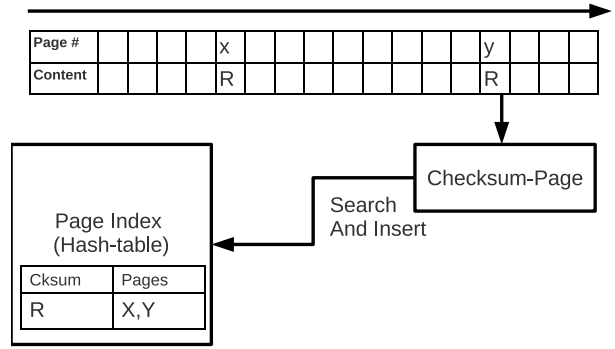

Figure 4: Basic KSM operation. Each page during a scan checksummed and inserted into the hash-table.

Linux uses an LRU variant (specifically, a variant of LRU/2 [27]) 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.

2.4 Page Deduplication using KSM

KSM(Kernel Samepage Merging) [4] 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, 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 4). 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

17

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 finger-print [38] 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 [20, 19, 35, 10]. 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.

3. SYSTEM-WIDE PAGE DEDUPLICATION

3.1 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 (Figure 2). 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 5. 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.
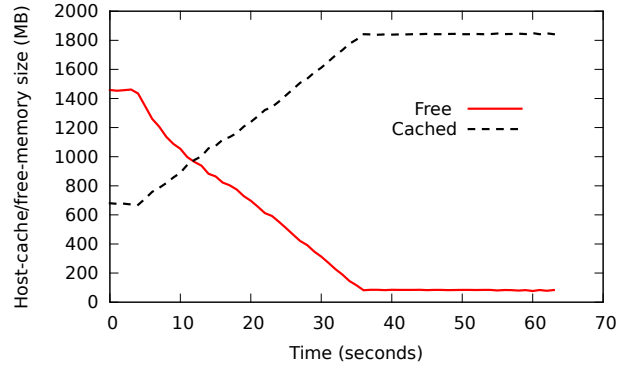

Figure 5: Host free and cached memory on a write-heavy guest workload.

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.

3.2 Potential/Existing approaches

In the context of multi-level exclusive caching in storage systems [13, 37, 5], 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 [37] 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 [5] and Geiger [17] 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 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 [17]. 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 [34]). The mapping from a virtual block number to a physical block

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 1: Bonnie performance with and without caching at the host.

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 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 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.

3.3 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 2.4), 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 [37]). 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 [39] 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 [17], 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 oc-

curred 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.

3.4 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`).

4. 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.

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;
}
```

4.1 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 2. The scrubbing interval thresholds are between 100,000 and 200,000 pages scanned by KSM (scrubbing-interval algorithm presented in section 3.4), 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 [30] 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.

4.2 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 use-

Workload	Description
Sequential Read	Iozone [26] 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 [7] 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 2: Details of workloads run in the guest VMs.

fulness, 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 2. The results from four VMs running sequential,random, and zipf I/O are presented in Figure 6. 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 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 6). 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 3). 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.
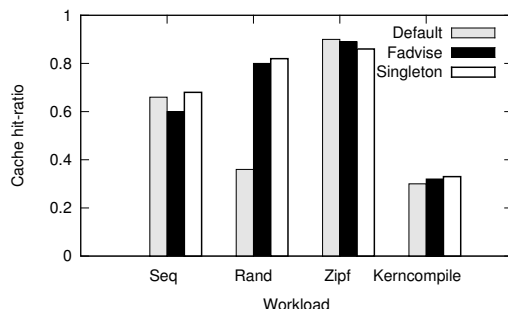

Figure 6: Host page-cache hit ratios.

	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 3: Guest I/O performance for various access patterns.

4.3 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). 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 8a 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
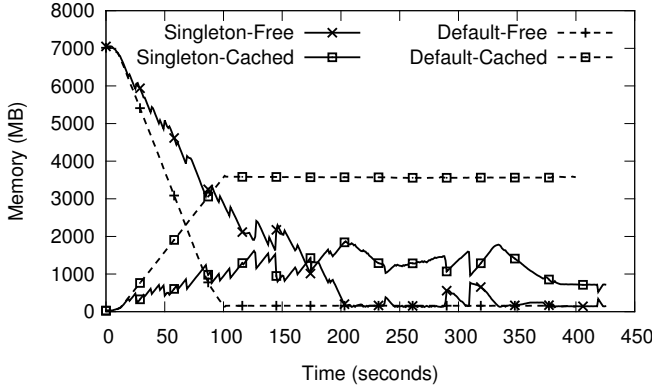
Figure 7: Memory usage graph for a sequential read workload with and without Singleton.

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 3.4) 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 8b. 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 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 7. The periodic scrubbing results in sharp falls in the cache-size, 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 8c). 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.

4.4 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). 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: Number of running VMs till system crashes or runs out of memory.

4.5 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 3. 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 8. 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.

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
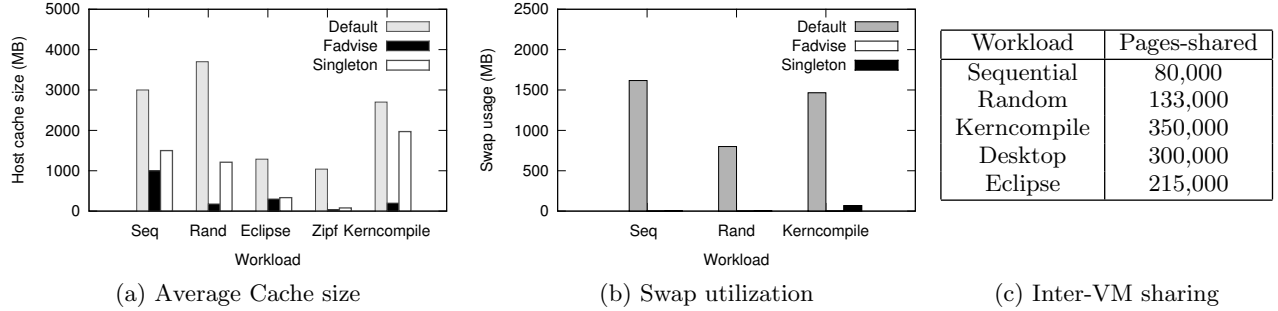
(a) Average Cache size   (b) Swap utilization   (c) Inter-VM sharing

Figure 8: Comparison of memory utilization with various workloads.

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

Table 5: Page eviction statistics with and without Singleton.

	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 6: 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 7: Impact on host performance (Eclipse) due to kernel-compile workload running in VM.

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 5, 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 perfor-

mance(Table 6). 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 improved 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 7).

4.6 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.

- The exclusive cache enables larger guest working sets to be present in memory, resulting in improved I/O

	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 8: Scrubbing overhead and host load averages.

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.

5. RELATED WORK

Page deduplication : Transparent page sharing as a memory saving mechanism was pioneered by the Disco [9] project, although it requires explicit guest support. Inter-VM content based page sharing using scanning was first implemented in VMWare ESX Server [35]. 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 [20, 19, 16, 21, 10]. Furthermore, page deduplication can also take advantage of presence of duplicate blocks across files (and filesystems). Storage deduplication for virtual environments is explored in [42, 29]. 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 [35] page sharing implementation, Difference Engine [14] (which performs very aggressive duplicate detection and even works at the sub-page level), and KSM [4] 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 [24] and Xenshare [19]. This approach is not possible with KVM because it does not primarily use paravirtualized I/O.

Cache Management : Page-cache management for virtual environments is covered in [33], however it requires changes to the guest OS. Ren et.al., [28] 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 conteding 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.

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 [41, 40]. Most work on exclusive caching is in the context of network storage systems— [13], DE-MOTE [37], XRAY [5].

An exclusive-cache mechanism for page-caches is presented in Geiger [17], which snoops on guest pagetable 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].

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 [35], 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 [35, 31], which "steals" memory from the guests via a special driver. Several other strategies for managing memory in virtual environments, like transcendent memory [23], collaborative memory management [32] exist, but they require explicit guest support or heavy hypervisor modifications.

6. FUTURE WORK

To reduce the page deduplication and scrubbing overhead even further, we are in the process of implementing additional optimizations to KSM which we hope will bring down the overhead to negligible levels.

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 [12], 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.

Scrub pages in LRU order: Our current host cache scrubbing algorithm checksums and compares pages in the host's page cache in an arbitrary order which is determined by the kernel-maintained inode list. The overhead of scrubbing is equal to the number of pages in the page-cache which are checksummed but *not* dropped. To improve on this, we can look at the page-cache pages in the least-recently-used (LRU) order. We are exploring means to drop and scrub pages from the host page-cache by exploiting the LRU list and aggressively dropping the 'recent' pages, which have higher likelihood of being present in the guest VMs. Further, a heuristic to stop or reduce rate of scrubbing can be formulated based on the fraction of dropped host pages.

Estimating Working-Set-Size: As the cache scrubbing implementation shows, the page search index which KSM builds and maintains is a valuable resource. Below we describe a few other potential uses of the index for hypervisor memory management and other tasks. KSM's search index can be used to estimate the working set size (WSS) of a VM. Several approaches to estimating the WSS of VMs exist [17, 36, 35, 22], but we can exploit KSM's index to estimate the WSS with low overhead and implementation effort. Since page contents are recorded (checksummed), page evictions can be easily tracked, allowing us to use Geiger's [17] approach to estimate WSS by measuring page eviction rate. If the first few bytes of a page have changed since the last KSM scan, we can say that the page has been evicted, with a high probability. Strictly, a changed checksum and initial few bytes implies page *reuse*, but as Geiger [17] shows, it almost always implies page eviction.

7. CONCLUSION

By combining inter-VM page deduplication and host cache scrubbing, Singleton achieves unified redundancy elimination in KVM, and can reclaim massive amounts of memory. Through a series of workloads under varying degrees of memory-pressure, we have shown that host-cache scrubbing is a low-overhead way of implementing an host/guest exclusive-cache in KVM. Our exclusive cache implementation results in tiny host page-caches (of the order of a few megabytes, as compared to several gigabytes without the scrubbing), along with improved guest performance because of better cache utilization.

Singleton does not require any intrusive modification to either the hypervisor or the guest, and works in a wide variety of environments. We achieve significant guest performance gains (upto 40%) along with memory savings(2-4x reduction in cache sizes), despite being conservative about the components of the system we modify. By utilizing the existing page-sharing infrastructure, we have shown how to implement several memory management tasks like eviction-based placement, with minimal modifications and overhead. Further, our modifications to KSM have demonstrated that inter-VM page deduplication can save significant amount of memory with low overhead,

As remarked earlier, page-sharing is a guest-transparent technique to reclaim memory and allow for memory over-commitment. By demonstrating that page-sharing along with its associated benefits (like exclusive caching etc) increases guest-performance, we believe that it is a useful and viable memory overcommitment approach.

8. REFERENCES
[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] A. Arcangeli, I. Eidus, and C. Wright. Increasing Memory Density by using KSM. In *Proceedings of the Linux Symposium*, pages 19–28, 2009.
[5] 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.
[6] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, pages 41–49, 2005.
[7] 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.
[8] D. Boutcher and A. Chandra. Does Virtualization Make Disk Scheduling Passé? *ACM SIGOPS Operating Systems Review*, 44(1):20–24, 2010.
[9] 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.
[10] 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.
[11] 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.
[12] 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.
[13] 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.
[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] P. Lu and K. Shen. Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache. In *Proceedings of the USENIX Annual Technical Conference*, 2007.

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

[24] 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.

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

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

[27] 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.

[28] 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.

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

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

[31] 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.

[32] 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.

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

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

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

[36] R. West, P. Zaroo, C.A. Waldspurger, and X. Zhang. Online Cache Modeling for Commodity Multicore Processors. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 563–564, 2010.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] K. ÂŕJin 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.