Improving memory management in Virtual environments using Page Sharing

Prateek Sharma

19 September 2011

1 Abstract

2 Introduction

Figure 1: In this figure we see..

Virtualization enables us to run multiple Operating Systems (one in each Virtual Machine) on a single physical machine. One of the key drivers of Virtual hosting and cloud computing is the ability to overcommit resources. That is, virtual machines are given resources in excess of what is actually available. Like the Operating System managing hardware resources for user-level processes in a conventional computer system, Virtual Machine Monitors (VMM) control the access to physical hardware for the Virtual Machines. (VMMs are also called hypervisors.) In this report, we consider the problem of memory overcommitment. Unlike CPU and network, memory is non-renewable. Virtual Machines are given a fixed amount of available physical memory when they are created/launched. This makes it imperative to provision memory carefully. Moreover, the optimality of it's allocation is critical because of the non-linear nature of the penalty , in the form of expensive, disk-seek-laden page-faults.

Our contributions are 2 fold:

1. We describe a set of improvements to KSM content-based-page-sharing, and show their impact on performance with microbenchmarks and some real-world workloads. These improvements are particularly effective in the case of desktop VMs running on a single physical server.

2. We show that the page-sharing infrastructure is a powerful tool for improving memory management in virtualized setups. Here, we make 2 important contributions:

3. We use page-sharing infrastructure to implement an exclusive page-cache. That is, the host and guest page caches do not have the same content. We show that double-caching is a serious problem affecting performance and memory resource-utilization, and how our solution impacts the availability of memory.

3 Memory Overcommitment

One of the benefits of virtualization is that it permits overcommitment of physical resources. This works because servers are typically underutilized (they have far more physical resources than required). The ability to virtualize allows us to consolidate multiple machines on a single physical machine by running them as Virtual machines, run by a hypervisor(or Virtual machine monitor).

Just as conventional operating systems multiplex resources among processes and provide concurrency, VMMs also multiplex hardware resources so that multiple Virtual Machines can run concurrently. This is how CPU-time, network and disk bandwidth are shared between Virtual machines. For memory management, this isomorphism between resource management in conventional Operating System kernels and VMMs is not so straight forward. In particular, 'the VM as a process' analogy isnt very convincing. User process' address space is allowed to grow and shrink at will, and the OS is free to swap any user page out to disk. User processes can request for a more memory (for example, by calling malloc/mmap) dynamically. However, when Virtual machines are created, they must be allocated a fixed amount of memory at their boot-time. Once allocated, this memory is 'wired in' and not usable by anyone else, even if the VM is not utilizing it fully. Since memory is allocated and 'committed' at VM boot-time, memory overcommitment is an extremely important feature if we want to increase consolidation ratios. For example, on an physical server with 16G RAM, we can run 8VMs each of 2G memory size each. Thus we have a hard-limit on the consolidation ratio if we do not overcommit memory. Paging and swapping are the fundamental operations which an OS uses to allow several processes to have an illusion of a large, contiguous address-space. How does this idea work for Virtual machines? The VMM can swap out pages belonging to virtual machines, but this leads to a range of problems. For example, there might be double-swapping: the VMM may swap-put a page to disk, only for the VM also to swap the same page out to it's own swap area. This is likely since the page might be at the wrong end of the LRU lists in both the VMM and the virtual machine OS.

To alleviate this problem, modern OS kernels now support memory hotplug support - the total physical memory may grow and shrink dynamically. Thus virtual machines may have some of their allocated memory removed , akin to removing a memory module (DIMM) from physical hardware. Memory hotplug certainly provides some dynamic allocation, but with some caveats. Reducing memory typically involves heavy memory transfers as the kernel tries to move memory-contents out of the removed region. Reducing guest memory dynamically can also be accomplished by using a balloon driver. A special-purpose balloon driver is installed in the VM and it inflates by allocating guest-physical memory for itself. The VMM and the balloon driver co-operate so that the VMM can reclaim the memory pinned by the balloon driver. Ballooning is a more natural operation for conventional OS kernels. Inflating a balloon is akin to a process requesting a some memory. Under memory pressure, the kernel forces some evictions, swaps pages out to disk in accordance to their access patterns and process page-fault-rates etc.

Overcommiting memory is thus an important tool if we seek to improve server consolidation.

Furthermore, memory is seldom 'idle' or free. Operating systems make all

effort to make full use of available physical memory. Even if the system working set isnt large enough, the rest of the memory is going to be used for caching files and other IO data. One approach is implementing the same virtual memory abstraction that processes are forced to use. The only difference is that the OS kernel is the process, and the hypervisor pages out/in guest physical memory. The disadvantages of this approach are several, as highlighted in [waldspurger].

Furthermore, the traditional memory-management techniques(paging) and infrastructure(page-caches, LRU lists, etc) do not ideally fit into the virtual setup.

Hypervisor memory management typically uses user-inputs and heuristics to allocate memory to guests. Paging and Ballooning provide for dynamism. However both these approaches are invasive and do not faithfully provide a reasonable approximation to the actual physical system.

Current VMMs have no way of knowing which page to swap out. Bigger performance problems arise when both the host and the guest swap out the page. In this case even the guest swapout will incur 2 IO accesses.

3.1 Why sharing

working set One of the ways of implementing memory overcommit is memory sharing. If 2 VMs could transparently share memory, then the total used machine memory would reduce. There are two major approaches to sharing : COW, CBPS. In this work we focus on CBPS. CBPS is more general and transparent to the VMs.

3.2 Other overcommitment approaches

1. Transcendent memory
2. CleanCache
3. zCache
4.

4 Page Sharing

Content Based Page Sharing (CBPS) methods share multiple pages with the same content and replace them by a single physical page.

On the face of it, the probability of 2 pages having exactly the same content seems pretty small , $2^{-4096*8}$ to be exact. Also, page contents can change at any point in time. However there is enough evidence that inter-VM page sharing does indeed work. Work by these-guys, these-guys, and these-guys indicates that inter-VM sharing is more likely than it seems at first. A natural question to ask is why are there so many shared pages , against all odds? If VMs run the same OS, the kernel text-sections and other data can certainly be shared. Near-univeral libraries like libc and headers are also very likely candidates. If certain applications are common, there is a good chance of having same pages, particularly if their versions are the same (not an unreasonable assumption to make, since most users prefer the same stable, security-patched versions of

common programs like apache, X11 etc). Another route to encounter same pages is if the underlying file-systems used by the VMs have same disk blocks. If the same disk blocks are read into the respective buffer caches, then we can share the corresponding pages. Thus we can also take advantage of the high probability of disk blocks having same content, and the large literature that talks about its feasibility (Data deduplication efforts). Cite loads of disk dedup papers now.

Page sharing in hypervisors can be broadly classified into 2 categories: Scanning based approaches, which 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 PTEs they belong to, being replaced by a single merged page. Examples of this are the vmware ESX server page sharing implementation , Difference Engine, which performs very aggressive duplicate detection and even works at the sub-page level, in addition to introducing compression for 'old pages'. KSM also falls in this category.

The other extreme is page sharing using paravirtualized support. Here , the virtual/emulated disk abstraction is used to implement page sharing at the device level itself. Examples are satori.

Our modifications to KSM bridge the gap between the general-purpose but CPU intensive scanners and the paravirtualized but low-overhead disk-based page sharers.

No VM modifications - can as well read off /proc. General approach No changes to IO stack unlike [satori], [disco], [xenfs], [ventana] reduce KSM overhead sharing remains same

Survey of everything. Some history?

4.1 Scanning vs Disk based sharing

2 competing approaches. (2)

5 KSM

KSM (Kernel Samepage Merging) is a scanning based mechanism to detect and share pages having same content. It is implemented in the linux kernel as a kernel-thread which runs on the host system and periodically scans guest VM memory regions looking for identical pages. The page sharing is implemented by replacing the page-table-entries of the duplicate pages with a common KSM page. There are 3 page-tables in question here (or rather, 3 page-mappings). The guest-vritual to guest-physical is maintained by each guest. However since each guest is just a process, KVM/host kernel maintains a guest-physical to host-virtual mapping also. This mapping is usually same, that is, a guest physical address is same as the host-virtual address of that particular qemu process. And finally we have the host-virtual to host-physical pages , maintained by the host kernel. KSM doesnt care about the first guest-level page-table, since it is opaque to it's operation. The 2nd level mapping is also identical. When 2 pages are found to be the same, KSM changes the page-table entry of the host kernel host-virtual to host physical table (this is a typical kernel page-table for a process. In this case the process is qemu running the VM). WHen 2 pages are

to be merged, KSM destroys the PTEs of both the pages, and allocates a fresh page, and copies the content to this page. THis newly allocated page is a kernel page, with flag PAGE_KSM. Since its a kernel page, it cannot be swapped out. Hence shared pages cannot be swapped out to disk, since kernel memory is not pageable. It should be noted that if the guest changes the PTE of a shared page, then the sharing will be destroyed. If the COW sharing is broken, KSM returns the page to VMs (its no longer PAGE_KSM anymore).

For detecting identical pages, KSM uses a red-black tree of pages. Each node is a page, and the nodes are ordered according to the page contents. This guarantees an O(log(n)) search+insert for pages. KSM maintains 2 search-trees. The stable tree and the unstable tree. Each page which belongs to a region registered with KSM can be one of 3 states: Present in stable tree - "shared pages" Present in unstable tree - "unshared pages" Not present in any tree - "volatile pages"

The unstable tree is used for page comparison within a single pass, and is destroyed after the completion of a pass. (A 'pass' is completed once KSM thread has finished trying to find a match for every page of all VMs running on the host.) The reason is that the pages in the unstable tree are not write-protected, and are inserted into the tree based on their content at the time of insertion. Once a page has been inserted, its contents may change. This approach leads to false-negatives. There are no false positives in KSM, because pages are locked and COWed upon sharing. If an attempt to write a KSM page is made, the sharing is immediately broken.

The number of pages in a system can be quite large, and thus even the unstable tree may get prohibitively large and difficult to maintain and search. To alliveate this problem, KSM does not insert frequently changing pages (volatile pages) into the unstable tree. These pages are thus untracked. To determine volatility, a checksum (jhash2) is used to compare the page contents with the previous KSM scan iteration. If the page checksum has changed between scans, it is deemed volatile and we move onto the next page.

This process goes on repeatedly, and thus has a consistant impact on the performance of the system. KSM typically consumes between 10-20% CPU on a single CPU core.

There is a clear CPU-overhead vs sharing trade-off here, and this is true for any scanning-based approach, as mentioned earlier. We can make KSM more aggressive by increasing the number of pages scanned before sleeping, and reducing the amount of time spent sleeping between two passes. This increases sharing opportunites at the cost of increased CPU usage by KSM, but at the same time if the sharing is very short-lived, then the overheads induced by all the page-locking and COW faults tends to increase the scanning overhead. Therefore we will consider host-cpu-% (in particular the CPU usage of the KSM thread) and the amount of pages shared (and hence the memory reclaimed).

A more comprehensive cost-model will be presented later in section .

5.1 Algorithm

for all pages:
if(search page in stable tree) :
insert page in stable tree ;
return

if(checksum(page)!=old_checksum(page)) :
set page as volatile ;
return
if(search page in unstable tree) :
add merged pages in stable tree ;
return
else
insert page in unstable tree ;

5.2 KSM comments

[Good bad points] KSM, although almost exlcusively used by KVM guests is implemented as a general-purpose memory-scanning page-sharing tool. Any process wishing to share its common pages with other such willing processes can mark its anonymous areas as $VM_{MERGEABLE}$. Typically this is done by passing a flag during malloc. In the context of KVM, this flag is set by qemu when it is allocating memory (using malloc) for guests to run in. It is important to note here that in the KVM/QEMU setup, virtual machine physical memory is just a malloc'ed memory area in the guest. [Insert figure here?]

Some design aspects of KSM, like the use of binary trees instead of hash-tables, and the lack of any heuristics help it work in any environment and ensure decent average-case performance while avoiding large worst-case penalties. KSM also strikes a fine balance between agressive scanning and maximizing page-sharing. When the VMs are running heavy workloads , the unstable tree doesnt grow too large because the pages will become volatile (checksum changes during passes) .This leads to a reduction in total number of pages to compare against. Thus during high VM loads, KSM thread holds off, ensuring a low CPU overhead. On the other hand, during idle periods, there are fewer volatile pages, and KSM tends to do more comparisons ; trading off an increased CPU utilization with an increased sharing ratio.

Diagram of operations, history, and some implementation details Focus on generic nature

5.3 Exp 1: KSM effectiveness

Small experiment which shows that KSM shares large % of same pages. Done earlier in fingerprinting project.
Expected result : $>$ 90% sharing of KSM, so good enough.
Reason : establish some ground truths: KSM works .
Setup : Random workload (doesnt matter) and take fingerprint and see KSM shar% . Run with 1,2,3 VMs. (1)

6 Analysis of shared pages

Having looked at the feasibility of inter-VM page sharing earlier, we turn to analyzing in greater detail the kinds of pages that are shared. We start off by analysing the page-sharing ratio of pages by their guest-flags. This gives us an indication of the kind of pages actually shared.

6.1 By flags

The key insight of this document/report is that not all pages are the same. KSM, which in the virtualization context shares identical memory regions between VMs, at present treats all pages as equal.

A breakup of pages being shared by their flags is given below: ....

QEMU allocates a large contiguous memory region (of the host) as the guest RAM.

For our discussion, we will classify pages into the following categories: Kernel-mapped Anonymous Mapped/FileBacked PageCache+other caches (inode etc) Free Pages

We will now look at each category, and give reasons/justifications for sharing/not sharing pages in a particular category. In the future sections, we seek to give experimental justifications of our claims.

6.2 KernelMapped: This includes kernel text+data+stack, as well as any

other memory region not available to guest user processes. This includes memory reserved by BIOS, DMA, other kernel buffers etc. These pages are sharable to a large extent, because typically stable/longterm kernels tend to be used, specially in enterprise workloads. However, sharing pages belonging to guest kernel may be a security issue[ksmsecurity]. Also since the amount of kernel memory is small, the tradeoff between increasing memory sharing and the associated security risks seems to tilt in favour of not sharing kernel memory regions.

6.3 Anonymous

6.4 Mapped

6.5 PageCache

We claim that pagecache pages should not be shared between VMs. By their very nature, contents of pagecache pages are ephemeral and just a cache for files on disk. Pagecache pages are also dropped first under memory pressure, and thus form a 'true' cache. Under the current, default KSM+KVM settings, a very large amount of pages shared are infact belong to the guest pagecache. This inflates the sharing ratio, and we argue that this sharing is 'meaningless' and brings very limited performance gains. The inflation of sharing ratio also witholds the real status of memory consumption in the system (host+guests).

Consider a scenario where a large number of pages are reported to be shared, thus increasing the amount of free memory available in the system. A guest VM, under memory pressure, will simply drop the clean pagecache pages, and replace them with mapped/anonymous pages belonging to some process address space. This incurs a COW cost for all the guests involved (whoever's pagecache pages were being shared with each other). A provisioning/placement tool, or an administrator might be led into making a wrong provisioning decision based on the inflated free memory it sees. This problem is compounded by the fact that typicaly, the linux kenel tends to use almost all available memory for it's pagecache, so the amount of cached pages are often quite large (about 40% of total memory on most workloads).

We have thus far argued that sharing cached pages does not help increase overcommit, and instead skews memory numbers. The disadvantages of sharing cache pages becomes even more apparent when we consider their sharing cost. Since KSM (or any CBPS for that matter) essentially runs an O(nlogn) algorithm for detecting and implementing sharing, any reduction in n is significant. And since cache pages are a large fraction of n, the saving in KSM overhead will be significant. The KSM overhead typically ranges between 5-15% (CPU) , which is not insignificant. With our modifications, we bring it down, while maintaining effective page sharing.

6.6 Free

6.7 Exp 2: Pages shared by flag type

Run some benchmarks (static VMs just booted up ; Kernbench ; HTTP-perf) and see what kinds of pages are shared by flag type. Reason: establish some ground truths : sharing is feasible . Also answer : what kinds of pages are shared? Setup: (1,2,3,5) VMs with different OS running same benchmarks. (diff VMs ; same kernel ; same /var/www)

6.8 Exp 2.1 : KSM with no pagecache pages

Run KSM but skip all guest pagecache pages.

6.9 Exp 3: Page sharing over time

Run some benchmarks and record pages shared over the duration of benchmark. Also record **KSM overhead**. Reason: Show that KSM overhead is significant enough, thus implying the need for some optimizations. Setup: (2) VMs running benchmarks. KSM being profiled using perf.

(1)

7 Sharing Model

In this section we describe a general model for evaluating pagesharing performance. The goal of any CBPS mechanism is to share as many number of pages as possible, with the least possible overhead.

Minimize $\{Unshared_{Pages} + Scanning_{cost} + Sharing_{cost} + COW_{cost}$

A suitably small granularity 't' is required.

Below we describe what an ideal sharing mechanism should be like:

1. Shareable pages at time t should be merged.

2. Pages sharable at t but unshareable at t+epsilon should not be merged. In this case the COW cost + merge cost outweighs the benefit of any traniently free memory.

3. Number of comparisons is minimal. Obviously, under standard assumptions, it is not possible to know exactly when/which page has been modified . This is because the VMs run on real hardware, and give no notifications to the VMM about page writes etc. THe page tables can be marked

as read-only, but this case is prohibitively expensive and thus not considered , because we have set out to model a general,non-intrusive sharing mechanism.

Thus, the expected number of comparisons is minimized, while keeping the expected number of shared pages maximum.

8 Lookahead optimization

8.1 Exp 5: Substrings in shared-map.

Record consecutive pages being shared in some benchmarks. Reason : justify why lookahead works. Setup: tracedump analysis simple python script

8.2 Tracing

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 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 pritnk output. Every page scanned 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.

8.3 Lookahead

Using the KSM trace logs, it can be observed that shared pages are often clustered together, occouring consecutively. Thus shared pages have a high spatial locality : If a page is shared, then the next page is also shareable with a high probability. This can be seen from this figure (SOME FIGURE) where the pages which are shared with some other page are shown, for 2 VMs.

We can represent the page contents as alphabets, and the memory contents of a VM are the strings. This string representation is a natural way of expressing the page-sharing problem in general. The degree of sharing is equal to the number of common characters in two strings. In this analogy, the characters in a string can change, since the contents of a page may change at any time. However as far as quantifying page sharing is concerned, we consider the pages being shared at a particular instant of time.

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 mapped into memory, if they are the same (have the same contents), then they 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).

8.4 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. In other words, we do a lookahead and see if the pages are common, before doing a general search. A more detailed algorithm follows:

If(equal(page Z, previous_matched_page+1)) :
merge
else :
Search trees for Z
If matched with W
previous$_m atched_p age = W$

If(equal(Page X , Page Y)) :
previous$_m atched_p age = Y$

Analysis: The lookahead optimization reduces the average cost of searching for a page. Assuming that the probability of a page being duplicate file is p, the expected cost with the lookahead optimization in place is now $p * Cost(merging) + (1 - p) * Cost(treesearch)$

The lookahead tricky 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 (provide 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), and moves on to the next page if they are. 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.

8.5 Exp 4: Lookahead success

Run benchmarks on VMs (1,2,3) to on and record lookahead successes. Also record **KSM overhead Compare vanilla KSM overhead with lookahead-optimization**

8.6 Performance of lookahead

The main advantage of lookahead of is reduced search cost, and hence a reduction in KSM overhead is to be expected. The KSM overhead is measured by the CPU-% of the ksmd thread. We compared vanilla-KSM with KSM with lookahead on the same workloads and recorded the ksm page sharing statistics along with the lookahead successes, and the ksm overhead on the host. The workloads were run on 2 VMs and were: STATIC: The 2 VMs are booted up. The VMs in question were ubuntu-10.04-server with 1G of ram allocated. KCOMPILE: The standard kernel-compile benchmark was run in the 2VMs. DESKTOP: The 2VMs were desktop VMs (Ubuntu-10.04-desktop) and a typical user session was simulated.

Results: Lookahead gives the most improvements for the DESKTOP workload. This is because desktop environments load a large number of common-files into memory on bootup (X11 fonts, graphical environment, etc). The surprising result is the increase in 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, as discussed in PREVIOUS DISCUSSION ON RATE VS SHARING. Correspondigly, 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 are not contiguous. This is the reason that lookahead cannot detect 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.

Experiment conclusion: Lookahead does not add any significant overhead to KSM, while increasing the total page sharing.

8.7 junk

This lookahead heuristic was implemented in KSM. <See below for results>. We found that upto one third of comparisons can be eliminated.

Lookahead value.

Here is how the lookahead optimization works in the context of KSM. KSM default case : $lookup_{stable}$ -> $lookup_{unstable}$ -> $insert_{unstable}$ Lookahead : lookahead -> $insert_{stable}$-> KSM_default. The optimization does not incur any extra overhead in the case where the lookahead fails.

9 Problem of double-caching

In this section we consider the behaviour and performance of the page-caches of the host and the guests in typical KVM setups. The discussion in this section is relevant only to KVM setups, and not immediately applicable to XEN, for example. There are 2 levels of page-cache in typical KVM setups : one is maintained by the guest OSes in their own pseudo-physical address-space. The other is the maintained by the kernel at the host. All IO in the Virtual Machines goes through the page cache at the host (hereafter referred as host-page-cache). A typical IO request generated by the VM causes a trap, which calls the QEMU userspace process, which handles the IO. In the emulated-disk case, QEMU performs the IO operation, and notifies the guest (via KVM).

A glaring problem in this setup is that the same disk block is cached in 2 places: the host page cache, and the guest's !.

We can see the amount of memory utilized by the page-cache in the host when various workloads in the VMs are run in FIGURE X.

9.1 QEMU caching modes

Virtual disks can be mounted in 4 caching modes in QEMU. None: uses direct IO using O_DIrect and hence bypasses the page-cache of the host OS. Supposed to give bad results [http://www.mail-archive.com/kvm@vger.kernel.org/msg30649.html] . But no evidence has been found of that so far??

writeback writethrough: VirtIO. Although 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.

To record the double caching, we record the cache size in the host. To see the impact of the caching mode. Since the caching mode primarily impacts IO performance, the workloads used are Bonnie++ and Iozone. The relationship between cache size and guest IO performance is shown HERE. The first set of experiments is with a single VM to measure the performance impact of caching modes. The other is with 2VMs with different caching modes to determine interference effects. With one VM, we observe that the caching mode does not have a drastic impact on IO performance in these benchmarks. For raw speeds, the hdaparm -tT test showed this result:

Some claims[that io flamewar] suggest that cache=none performs badly in case of multiple VMs. In FIGURE SOMETHING we run bonnie on 2 VMs with a variety of caching modes. DO I NEED NINE EXPERIMENTS HERE?

Repeat with 4 VMs.

9.2 Multi level Exclusive caching

Multiple levels of cache occour in several components of computer systems. The CPU caches have an L1/L2/L3 cache heirarchy, with the smaller caches occouring closer to the CPU and the larger ones closer to the programs. Multi-level caches are also found in network storage systems. Both the client and the server maintain a cache. In the context of storage systems, the problem of exclusive caching is well studied [MYCACHE, GILL, ETC]. Caching in storage systems fits the host-cache/guest-cache model most closely. In both cases the caches are software managed and are maintained in an LRU order (or some variant thereof).

If both the caches are maintained in an LRU fashion, then the client caches see a good spatial and temporal locality. However it has been found that [REFERENNCE HERE] host caches typically see significantly bad locality of access. Several solutions to this have been proposed - the caching algorithms can be changed to get better cumulative hit ratios, as developed in [REFERENCE -MQ].

Maintaining an exclusive cache needs co-operation and co-ordination between varios caches in the hierarchy. Items can be promoted [GILL] or demoted [MYCACHE]. This notifies the other cache that an item is bought-in/evicted from the cache and the other cache can then take action. This needs explicit

notification of any modifications to the cache. In the host-guest cache setup that virtual systems deal with, the notifications can cause a large overhead since the page cache sees high activity.

More pressing is the problem of actually generating these notifications - they need modifications to the host and the guest OS.

Since maintaining exclusive caches increases the effective cache size, it is can pay off even if modifications to the OS is required. However this is not straight-forward. The first challenge is to get notifications of evictions. This can be done by explicit notifications from the guest, or the by using snooping techniques developed in GEIGER.

The real 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 amount of virtual disk formats (RAW, LVM, QCOW, QCOW2, QED,FVD..). The mapping from virtual block number to physical block number (what the host file system sees) can be determined fairly easily in case of RAW images, but one would need explicit hypervisor support in every other case.

But then we have a complex co-ordination challenge between the host, guest, hypervisor, and all the disk emulation mode it supports (IDE, VIRTIO, VHOST ...). Clearly we need a better solution which does not need to contend with this 3way handshake and yet works with all the above mentioned variances and environments.

9.3 KSM-implementation

It turns out there is a way to obtain an exclusive host and guest page cache. Our solution involves KSM, and uses it's search-trees to search for pages in the host page cache. Recall that at the end of every scan, KSM has built a search tree of all the pages in the guests (which have not changed since the previous scan i.e nonvolatile). The host page-cache pages are compared for duplicates using the stable and unstable tree, and if a match is found, we drop the page from the host cache.

Not having volatile pages in the search tree represents an interesting challenge. On one hand, if a page is freshly bought into the guest cache, then it is volatile. However since the route passes through the host cache , it is also present in the host cache. Since it is volatile, it was bought into both these caches 'recently'. Thus our cache isnt totally exclusive but only partially so. On the other hand, the increased number of comparisons between host cache pages and ALL the VM pages causes intolerably large overhead in terms of ksm cpu usage. We have observed sharp peaks in KSM cpu usage when it flushes the duplicate cache pages, with the CPU usage reaching 90 percent and above, even with only the stable and unstable tree search. Searching all the volatile pages induces too many false-positives and increased overhead to be of much use.

The algorithm is:

At the end of a KSM scan:
For all unmapped pages in the host:
If (page $\in$ KSM stable or Unstable tree):
drop_page(page) ;

This adds a small overhead , BUT it is equivalent to another VM being scanned. The benefits are that we get a properly exclusive cache with strong guarantees about exclusivity. Also the KSM heuristics and nuances actually **help** because we dont scan 'hot' pages (volatile), and the scanning approach ensures that there is no correctness violation .

9.4 Analysis

Any solution to the exclusive caching problem must strive for a balance between size of the host page cache, size of the guest page cache, and performance of the guests. We can trivially get an exclusive cache by not having either of the host and guest caches, by dropping the caches at regular intervals. Another alternative is to emply a balloon module. Ballooning inside a VM will cause it drop some pages, and is a way to reduce the size of the guest-page-cache.

An important parameter is the frequency of cache scrubbing. Currently, we drop duplicate pages after having completed one pass(all VMs have been scanned). Increasing the frequency of cache scrubbing implies a smaller effective host cache size, and is particularly effective for heavy workloads in the VMs.

FIGURE SHOWS BONNIE WITH 2 VMS WITH the TWO FREQUENCIES.

9.5 Results

We evaulate the effectiveness of the KSM-exclusive-cache solution. The primary metric here is the amount of free and cached memory on the host(as observed by free/vmstat). The IO benchmarks Iozone and bonnie++ were run on VMs with and without exclusive caches. A point to note is that the IO benchmarks are short-running, heavy workloads, precisely the kinds of tests which KSM doesnt do very well on. Inspite of the handicap, substatial performance improvements were found with the host cache scrubbing implementation.

The improvements in the Guest benchmarks can be attributed to less memory pressure on the host in general and a higher cache-hit rate in the host. Our implementation of the exclusive cache guarantees that the cache-hit ratio of the host is not affected. Any page dropped was present in one of the guests. If multiple guests request it, then KSM has already shared it.

Since the ksm based cache scrubber is scanning based, the cache size is dictated by the frequency of the scrubbing. Therefore we cannot rely on it to effectively constrain the cache size during heavy workloads.

With ever increasing memory sizes, any scanning based page merging solution must conted with increasing scanning frequency , or missing a large amount of sharing opportunities as the distance between pages with same contents increases. Depending on the requirement, the administrator can opt for sacrificing one core for detecting page-sharing, or keeping it at a low throttle and share pages which remain so for very long periods of time.

In almost all cases, the average amount of cached memory has been reduced with the scrubbing, even though the scrubbing is performed very few times. (can be seen from the sudden drops in FIGURE).

9.6 Existing approaches

Geiger, that hypervisor memory thing, etc.

9.7 Importance of exclusive caches.

9.8 Exp 6: Memory savings with exclusive caches

How many pages are there in both places? Setup : Benchmarks on VMs.

9.9 Exp 7: Overhead of ksm-exclusive-cache

Run benchmarks on VMs to record KSM overhead (with ex cache) Reason : scanning vast host page cache could be significant overhead.Also savings might help.

Some caching theory references. (1)

9.10 Exp 8: Memory mountains

Look at this problem like L1/2 cache , and build mem-mountains in these cases: (normal ; no guest cache ; no host cache ; swap as ramdisk) Setup : could use IOZone or randal bryant's simple program. Reason : Demonstrate the latencies/throughput of various caches. **This depends on lots of factors like IO schedulers, FS, virtual disk layout etc. Do for any one, for now.**

9.11 Motivation

Virtual Disks are seldom mounted as O_{DIRECT}, which means that the host pagecache also stores disk blocks. The reason for this is performance, since using O_{DIRECT} turns off all the clever IO scheduling and batching(This is apparently heavily debated). Obviously double caching wastes precious memory. There are other solutions which can mitigate this, but those are not considered right now [resizing guests using balloons, keeping a bound on the pagecache sizes ,etc]

9.12 Related work

Exclusive caches are proven to have better performance than general inclusive ones. There are a few ways one can maintain exclusive caches. Geiger snoops on guest pagetable updates and all disk accesses to build a fairly accurate set of evicted pages. So if we cache evicted pages, we're done. However the problem with geiger is that it does not quite work with host page caches. There is no way of knowing that a page is already present in the guest pagecache.

Another issue with Geiger is that **it cannot work with hardware virtualization** since it depends heavily on shadow tables.

10 Qualitative survey of dynamic memory management for VMs.

There are several solutions to provide dynamic memory management for Virtual Machines. This section contains a brief description of some of the them.

10.1 Tmem: cleancache, ramszwap, etc.

Transcendent memory 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 . THis is done by ramzswap. [TMEM]

10.2 Collab2

Collab2 is a framework for providing fine-grained dynamism for memory management for hypervisors. It uses hardware assisted page-transitions to determine guest free/used pages. [COLLAB2]

10.3 Ballooning and Hotplug

[HOTPLUG] paper summary here.

11 Improving memory management with KSM

Just as the exclusive cache problem can be solved using KSM, so can a range of other memory management issues. The key to each solution is that we make use of all the work KSM does during a scan (ie building the search trees) for our purposes. In this section we demonstrate how KSM can be used. The actual implementation and evaluation of the solution is work in progress - we give rationale for why the proposed solution should work.

11.1 Estimating WSS

There are several ways of estimating the Working set size of processes and VMs. Determining the WSS is important for estimating the memory requirement of VMs. If the WSS is known, the dynamic memory management techniques (ballooning and hotplug) can be applied.

11.1.1 related work

Several ways of estimating WSS exist: [WALDS] uses hardware perf counters to estimate Miss-rate-curves. [GEIGER] , which can snoop on evictions, can determine if the WSS is greater than the allocated memory size. [ESX] uses a sampling approach to determine WSS if it is less than the allocated memory, so that ballooning can be employed.

The challenge in determining the WSS is determining memory accesses without the overhead. Trapping all accesss is prohibitive for performance as well actually simulating the LRU histogram and calculating WSS. Most approaches use a sampling approach.

11.1.2 KSM Solution

We use KSM to estimate WSS of guests. We already know which pages are volatile. Hence it is easy to determine the writeable working set and the page dirty rate of each VM.

Estimating page-reads is trickier. If a VM reads from it's own address space then some form of trapping+sampling is required. However there is another way to determine page accesses - via the host page cache. We determine the number of page-ins for every VM, this gives the number of page-reads.

We can also track page evictions ala Geiger. Here a heuristic is used: if the first 8 bytes of a page have changed then we claim an eviction. This is reasonable because evictions imply page resuse, and page mutations by a process is typically append or some write to a random location in the page. Evictions are much more important than plain volatile information , because evictions convey much more information about the working set.

Thus: WSS = Volatile pages + page-ins + evictions.

11.1.3 Disk buddies

Memory Fingerprinting allows VM placement to maximize page sharing , as demonstrated in [MEMORY BUDDIES]. A memory fingerprinting mechanism can be trivially implemented using KSM. In addition to memory fingerprinting, we can use the page sharing statistics and the lookahead success rate to determine the probability of the virtual disk images to have a large number of common disk blocks. Thus we can determine if 2 VMs colocated should have their virtual disks stored on the same network server. This is useful if the disks are stored on a content-addressible-storage backed volume. Virtual Disks are frequently created from a small set of tempalates, and there is a large scope for common blocks. This disk-image similarity hint can potentially speed up CAS deduplication and serve as hint to determine duplicate blocks efficiently.

11.1.4 Host swapping

The host memory management subsystem can use KSM information for swapping out guest pages to disk. By their nature, stable tree pages cannot be swapped out since they are kernel pages, so using KSM prevents us from swapping out shared pages. To avoid double-swapping, free-pages in the guests should not be swapped out by the host. To prevent swapping out guest free pages, the guest needs to notify the host. A solution called FreePageHinting

[http://www.linux-kvm.org/wiki/images/f/ff/2011-forum-memory-overcommit.pdf] uses a per-guest bitmap of free/used pages.

KSM can be used to guess free pages. Zero pages are obviously free - they are also shared by KSM, so they cannot be swapped out. Further work on designing a free-page-guessing heuristic using KSM needs to be done.

11.1.5 someting else

12 Conclusion

13 Future Work

14 References

14.1 Caching

[Geiger] Buffer cache monitoring

[MyCache]

[gill] Promotions are better than demotions

[zhao] Dynamic Memory Balancing for Virtual Machines

[lru1] MRC using hypervisor exclusive cache

[lru2] (mailed by puru)

[walds2] MRC using hardware performance registers

14.2 General

[kvm]

[xen]

[virtio]

14.3 page sharing

[esx] Memory management in the Vmware ESX hypervisor

[ksm] Increasing memory density using KSM - Anthony Liguiri

[diffengine] Difference engine: Harnessing memory redundancy in virtual machines

[satori] Satori: Enlightened page sharing

[pv] A Paravirtualized Approach to Content-Based Page Sharing //Uses content-addressed page tables instead of SPTEs. Weird!

[introspect] Determining the use of Interdomain Shareable Pages using Kernel Introspection //Perfect analysis of PG_{flag} and sharing . goldmine of data.

[feasib] On the Feasibility of Memory Sharing in Virtualized Systems //same as paravirt

[xenshare] Efficient Memory Sharing in the Xen Virtual Machine Monitor //Nice discussion on hashing, hash-tries, sharing potential,workingset, a funny vulnerability (timing based attack.process writes random pages to guess which page contents of other process/kernel!)

[cblock] Content-Based Block Caching

[balancing] Dynamic memory balancing for virtual machines

[xencow] Memory CoW in Xen
[domain] Domain Level Page Sharing in Xen Virtual Machine Systems

14.4 Memory

[transcendent] Transcendent memory: Re-inventing physical memory management in a virtualized environment