stage1

prateek sharma

19 September 2011

Contents

1 Introduction

Memory is one of the important physical resources in virtualized environments. 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 of hypervisors to overcommit resources. That is, virtual machines are given resources in excess of what is actually available.

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.

2 Motivation - 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).

This works ideally for overcommiting CPU and networking resources. However, the total memory in a system is fixed, and Guest operating systems

expect to be able access all the memory they 'see' at boot-time. [hotswap and pluggable memory will be discussed later]. Overcommiting memory is thus an important tool if we seek to improve server consolidation. While CPU,network usage on idle workloads can be exploited to increase the number of VMs running on a physical machine because they are inherently shared resources and operating systems and processes are aware of their shared nature. Kernels ofcourse do not expect their physical memory (of the machine) to belong to someone else.

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.

2.1 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.

2.2 Other overcommitment approaches

1. Transcendent memory
2. CleanCache

3. zCache

4.

3 Page Sharing

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?

3.1 Scanning vs Disk based sharing

2 competing approaches. (2)

4 KSM

KSM (Kernel Samepage Merging) is a scanning based CBPS tool. KSM is a kernel-thread which runs on the host system and periodically scans guest VM memory regions looking for identical pages. Once two pages are found to be the same, they are replaced by a single KSM page. This new page is COW protected. 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. 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. 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 .

4.1 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 $\text{VM}_{\text{MERGEABLE}}$. 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 in work in any environment and ensure decent average-case performance while avoiding large worst-case penalties.

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

4.2 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)

5 Analysis of shared pages

5.1 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.

5.2 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.

5.3 Anonymous

5.4 Mapped

5.5 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.

5.6 Free

5.7 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)

5.8 Exp 2.1 : KSM with no pagecache pages

Run KSM but skip all guest pagecache pages.

5.9 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)

6 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 $\{\text{Unshared}_{\text{Pages}} + \text{Scanning}_{\text{cost}} + \text{Sharing}_{\text{cost}} + \text{COW}_{\text{cost}}\}$

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.

7 Lookahead optimization

Assume there are 2 VMs with reasonably high sharing. As the previous analysis(TODO) showed, lot of sharing is 'organized' neatly by page flags. In particular, if we discard for shared pages of the cached variety (page cache / slab etc) then most of the sharing is of pages mapped to same files on disk (program text sections etc), or same malloced pages (same applications creating the same 'data').

In such scenarios the following scenario occours frequently: Assume page number X of VM_2 is shared with X' of VM_1, Then with a high probability, X+1 will be shared with X'+1 .

Having given the explanation of the scenarios in which this holds true, we now give an experimental validation.

The main problem with the substring 'peek' optimization is that it only is effective when the pages are shared for the first time. After that, KSM will do a stable tree search before comparing. Thus a lot of overhead is incurred even in low page-write/high sharing environments. Each stable tree search is a log(s) operation, and for S sharing pages, we have Slog(s).

How to reduce this? The same trick can be used! If a page is a KSM page (testing for this is O(1), just check the flags), then we peek ahead anyway. If the peeked page is identical (one memcmp) and the rmap is stable, we're done! So even in this case we have reduced the operation to O(1) comparisons instead of log(s).

This can be combined with the previous page-cache rule. I.E : page cache pages are only compared **once** (lookahead). If they dont match, dont touch them at all. How to implement this?. Need to atleast goto start of the file?

7.1 Implementation

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 : $\text{lookup}_{\text{stable}}$ -> $\text{lookup}_{\text{unstable}}$ -> $\text{insert}_{\text{unstable}}$ Lookahead : lookahead -> $\text{insert}_{\text{stable}}$-> $\text{KSM}_{\text{default}}$. The optimization does not incur any extra overhead in the case where the lookahead fails.

1. Pattern chages with Time
2.

7.2 Tracing

7.3 Implementation and analysis

(log u etc)

7.4 Results

7.5 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**

7.6 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 (3)

8 Problem of double-caching

8.1 Existing approaches

Geiger, that hypervisor memory thing, etc.

8.2 Importance of exclusive caches.

8.3 Exp 6: Memory savings with exclusive caches

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

8.4 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)

8.5 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]

8.6 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.

8.7 KSM solution

9 KSM-implementation of double-caching

We use detect pages in the host pagecache already present in the guests by using KSM. KSM already has a nice, sorted tree of all guest pages (in the stable and unstable trees) at the end of every scan. So 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 .

9.1 Implementation and analysis

9.2 Results

9.3 Exp 7: Number of pages evicted out due to duplicates

9.4 Exp 8: Overhead of cache eviction

(2)

10 Qualitative survey of dynamic memory management for VMs.

10.1 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.**

10.2 Tmem: cleancache, ramszwap, etc.

10.3 Collab2

10.4 Ballooning and Hotplug

(2)

11 Conclusion

(1)

12 Future Work

13 References