An analysis of Inter-VM page sharing for KVM

prateek sharma

19 September 2011

Contents

1 Title

Improving memory management in Virtual environments using Page Sharing

2 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, since we believe that memory is almost always the resource with the most contention. Unlike CPU and network, memory is non-renewable. 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 Motivation

One of the benefits of virtualization is that it permits overcommitment of physical resources. This works because servers are typically woefully underutilized. 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.

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 Comparison

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 (which was the first page sharing implementation i know of), 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

5 Page Sharing

Content based page sharing [carl],[],[] is a useful technique for memory overcommit. Total physical memory available is a hard constraint, and increasing the percieved amount of memory in the system is useful . CBPS solutions operate by scanning VM memory regions, looking for pages matching in content. Memory contents are dynamic, and thus are page contents as well. Content of a page can change any time, and any CBPS solution must account for this and prevent illegal sharing. Illegal sharing is akin to mapping the wrong physical page to an address space of a process, something which is a fundamental task of Operating Systems.

6 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 .

6.1 KSM comments

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.

7 Mem Layout

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.

8 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.

9 Patterns

9.1 Substrings

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 assumption holds to a large degree: 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?

9.1.1 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.

10 DATA collected

comparisons (stable | unstable | failed | hits) lookahead (success | failures) pages sharing per VM pages sharing per VM per flag-type

11 Ideas

11.1 Flags as fingerprints.

Question: can flags of a page serve as a fingerprint of its contents? Obviously flags by themselves have no meaning. But if we consider the flags of a physical page **over time**, then is there a case for "identifying" stable/unstable/volatile pages? FOr example, if the flags of a page have changed since the last scan then clearly there is a high probablility of its contents being changed. Ofcourse, a large number of page writes will not affect the flags : writes to anonymous pages by processes, pages mapped by the kernel for all the caches, etc.

One thing that the flags wont tell us is sharing **between** VMs, ala memory buddies. But in the same spirit as the previous argument, can a case be made about observing a history of page flags and predicting (very roughly) the page sharing potential between 2 VMs. Again note that a snapshot of the flags probably wont help at all. But a history of page flag changes etc might?

An optimization that comes to mind is to identify the 'files' or memory regions based on the flags and compute checksums for the first few pages only. This can get messy since the first pages are likely to contain the same content anyway (magic numbers, environment variables etc?) and thus not be an effective fingerprint.

11.2 Flags for adaptive scan-rate

The real irony of scanning based (particularly KSM) approach is that the overhead **increases** when the guest dirty-rate decreases. The reason is that when dirty rate is low, the number of volatile pages drops, which means everything is either in stable or unstable tree. If the sharing is low, then the unstable tree is very large. This implies log(U) comparisons **per page**. Overall the cost is: cost(hash) + cost(search).

Walking down the unstable tree also has bad cache behaviour. <verify this!>

The scan-rate of any periodic scanner must be adaptive. One heuristic which can be used is the rate of change of page-flags over time for a particular VM. If the flags havent changed much, then we can assume that the dirty-rate is under check. Ofcourse this is a big assumption which needs to be justified. Anyway, even if a wrong decision (scan/no-scan) is made, we can scan the memory region in the next round anyway.

12 Implementation Details

12.1 Architecture

13 Experiments

13.1 Lookahead

1. Number of lookahead success.

Do with :Static VMs, httperf, lookahead as a % of $\text{pages}_{\text{shared}}$ Static VMs have high success . httperf todo: kernelbench : low success for lookahead . apt-get install emacs auctex : 160K shared pages, all lookahead!

1. Performance impact of lookahead

Reduced cpu% with lookahead ON vs OFF.

13.2 $\text{Flag}_{\text{filter}}$

KSM overhead breakdown: Required?

Benefits of $\text{only}_{\text{mapped}}$:

1. Kernel compile
2. Web Server?
3. 'Find' ? md5sum?

14 Conclusion

15 References

[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.

[satori] Satori: Enlightened page sharing

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

[xenshare] Ecient 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

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

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

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

[mendley] http://www.mendeley.com/groups/498921/vmm-memory-sharing/papers/

KSM unstable tree question

Hello everyone .

I've been trying to understand how KSM works (i want to make some modifications / implement some optimizations) . One thing that struck me odd was the high number of calls to $\text{remove}_{\text{rmapitemfromtree}}$.

Particularly, this instance in $\text{cmp}_{\text{andmergepage}}$:

/*

- As soon as we merge this page, we want to remove the
- $\text{rmap}_{\text{item}}$ of the page we have merged with from the unstable
- tree, and insert it instead as new node in the stable tree.

*/ if (kpage) { $\text{remove}_{\text{rmapitemfromtree}}(\text{tree}_{\text{rmapitem}})$; $\text{lock}_{\text{page}}$(kpage); $\text{stable}_{\text{node}}$ = $\text{stable}_{\text{treeinsert}}$(kpage); if ($\text{stable}_{\text{node}}$) { $\text{stable}_{\text{treeappend}}(\text{tree}_{\text{rmapitem}}, \text{stable}_{\text{node}})$; $\text{stable}_{\text{treeappend}}(\text{rmap}_{\text{item}}, \text{stable}_{\text{node}})$; }

Here, from i understand, we've found a match in the unstable tree, and are adding a stable node in the stable tree. My question is: why do we need to remove the $\text{rmap}_{\text{item}}$ from unstable tree here ? At the end of a scan we are erasing the unstable tree anyway. Also, all searches first consider the stable tree , and then the unstable tree. What will happen if we find a match in the unstable tree, and simply update $\text{tree}_{\text{rmapitem}}$ to point to a $\text{stable}_{\text{node}}$?

Thanks for reading. I'd love to share the ideas i have for (attempting to) improve KSM, if anyone is intereseted.

Prateek