Increasing memory density by using KSM

Andrea Arcangeli, Izik Eidus, Chris Wright
Red Hat, Inc.
aarcange@redhat.com, ieidus@redhat.com, chrisw@redhat.com

Abstract

With virtualization usage growing, the amount of RAM duplication in the same host across different virtual machines possibly running the same software or handling the same data is growing at a fast pace too. KSM is a Linux Kernel module that allows to share equal anonymous memory across different processes and in turn also across different KVM virtual machines. Thanks to the KVM design and the mmu notifier feature, the KVM virtual machines aren't any different from any other process from the Linux Virtual Memory subsystem POV. And incidentally all Guest physical memory is allocated as regular Linux anonymous memory mappings. But KSM isn't just for virtual machines.

The KSM main task is to find equal pages in the system. To do that it uses two trees, one is the stable tree the other is the unstable tree. The stable tree contains only already shared and not changing KSM generated pages. The unstable tree contains only pages that aren't shared yet but that are tracked by KSM.

The content of the pages inserted into the two trees is the index of the tree, but we don't want to write-protect all the pagetables that points to the pages in the unstable tree. So we allow the content of the pages (so the tree index) to change under KSM and without knowledge of the tree balancing code. Thanks to the property of the red black trees that can keep a tree balanced without checking the node index value, even if the tree becomes unusable, the tree still remains balanced and the worst case insertion/deletion remains O(log(N)), to guarantee the ksm-tree algorithm not to degenerate in corner cases.

To reduce the number of false negative from the unstable tree lookups, a checksum is used to insert into the unstable tree only pages whose checksum didn't change recently, but in the future the checksum can be replaced by checking the dirty bit of the pagetables and shadow pagetables (not with current EPT though). After a full scan of all pages tracked by KSM, the unstable tree is rebuilt from scratch to reset all lookup errors introduced by the pages changing content during the scan.

Whenever KSM finds a match in the stable or unstable tree, it proceeds to write-protecting the pagetables that mapped to the old not shared anonymous page, and it makes them map the new shared KSM page as read-only. If any KVM shadow pagetable was mapping the page, it is updated and write-protected through the mmu notifier mechanism with a newly introduced change_pte method.

1 Nomenclature

The name of this Linux Kernel feature might change. For the scope of this document, the term *KSM* (as in *Kernel Shared Memory* or *Kernel Samepage Merging* if you wish) will be used, even if it may be renamed to *Memory Merging* in the future.

2 KSM objective

The objective of KSM is to increase memory density. KSM is generating shared pages by merging equal pages, and in turn it is making free memory available allowing to run more virtual machines or applications on the same system, than otherwise would be possible without KSM.

3 KSM API

The API to use in KSM has been one of the most discussed parts of the feature on mailing lists, but it's also the least interesting part for the scope of this document and will be only covered briefly here.

While it would be possible for KSM to scan every single anonymous page in the system, it would be wasteful to

scan virtual areas where we don't expect to find any significant amount of equal pages. It would be wasteful not only in CPU terms but in RAM terms too; to keep track of the pages, KSM has to make some slab allocation. The amount of slab allocations increases linearly with the size of the virtual areas registered. Usually Linux applications try to be intelligent in sharing memory either with shared librarians or through fork. Not all applications are generating memory regions with lots of equal anonymous pages in a way that cannot be shared without the KSM feature, so it's worth scanning only the virtual areas that are likely to contain lots of equal pages that cannot be shared by other means.

Processes (through the KSM API) shall simply have the option to register which virtual memory areas should be scanned by the kernel thread that has the task of merging equal physical pages of memory.

The kernel thread that scans the registered virtual ranges can be controlled through sysfs at /sys/kernel/mm/ksm (but if the KSM name changes, supposedly the location is subject to change too). Writing 1 or 0 in *run* respectively starts and stop the kernel thread. *pages_to_scan* and *sleep* control how CPU intensive the scan of the memory will be. The more pages scanned per wakeup and the more frequent the wakeups, the more CPU the kernel thread will take, and the faster the equal virtual memory will be shared. *sleep* is in usec units, pages is in PAGE_SIZE units. *pages_shared* is a read only statistic field showing how many KSM pages are allocated in the system at any given time. *max_kernel_pages* can limit the number of KSM pages, this can be useful especially in the short term because in its first version the KSM pages aren't swappable yet (swapping KSM pages is possible similarly to how tmpfs swaps and it will be addressed shortly).

At time of this writing, it seems likely the final memory merging API that applications can use will be implemented through the madvise syscall with a new MADV_(UN)MERGEABLE *advice* parameter.

In the current implementation, only anonymous pages (like the ones generated by malloc) can be merged with KSM, but perhaps in the future this could be extended to other kind of pages.

There will likely be an option to avoid compiling the KSM code into the kernel to save kernel .text for those embedded systems where the KSM feature won't be required, in which case madvise will fail if passed the relevant memory merging advice parameter.

4 KSM and KVM

One of the primary users of KSM is the Linux *Kernel Virtual Machine*. When the same guest OS and guest applications are running in different virtual machines, lots of equal anonymous memory will be generated on the host/hypervisor system. So it is ideal to always keep KSM enabled with parameters like sleep = 5000 pages_to_scan = 60, so that around 12000 virtual pages are scanned each second, allowing a max memory merging rate of 46.87MB/sec (the max rate would materialize only if all virtual pages scanned during a second of time, are found to be equal to some other page tracked by KSM). With this setting the KVM kernel thread should use around 10% of one CPU core. On very large systems, however, more aggressive settings can be used, up to dedicating a CPU core to the KSM kernel thread.

Although at present KSM is only capable of merging equal anonymous memory on the host system, KVM virtualization allows KSM running on the host to share pagecache, tmpfs, or any other type of memory allocated in the guest, because all guest memory is backed by host anonymous memory.

5 KSM at CERN

We had great feedback from CERN and Lawrence Berkeley National Laboratory related to the computations they're running to crunch the LHC generated data. Their scientific reconstruction jobs generate lots of equal pages while they run. With KSM enabled they achieve memory sharing rates up to 750MB if they run two similar 2GB jobs (without KSM the sharing is limited to 250MB). They conclude they are able to run 3 jobs in parallel on a 4GB machine, instead of only 2 before. This cumulatively saves a very significant amount of memory, given the number of nodes involved in the computations.

They're not yet using virtualization on top of Linux, so to use KSM, their application has to call into the KVM API directly (by using a malloc wrapper). If they were to use KVM as a hypervisor (instead of proprietary hypervisor solutions running underneath of Linux) they wouldn't need to change their application at all, and all memory would be merged transparently at the host kernel level.

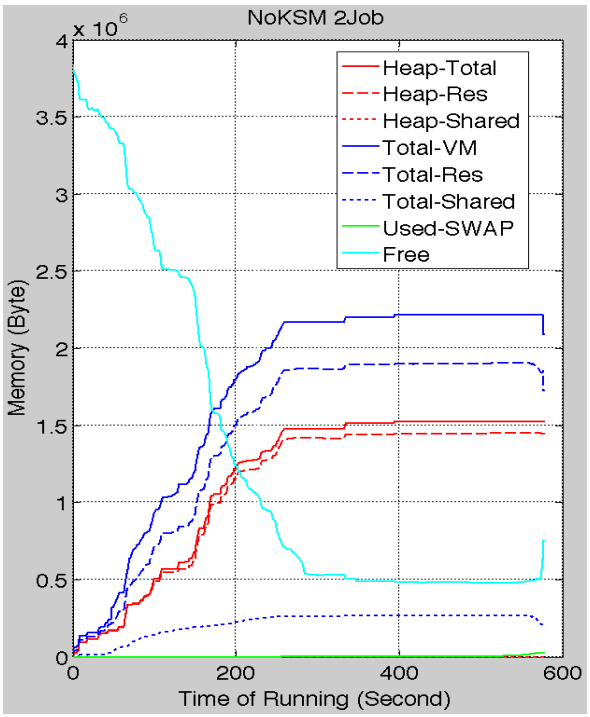

Figure 1: 2 LHC reconstruction jobs without KSM

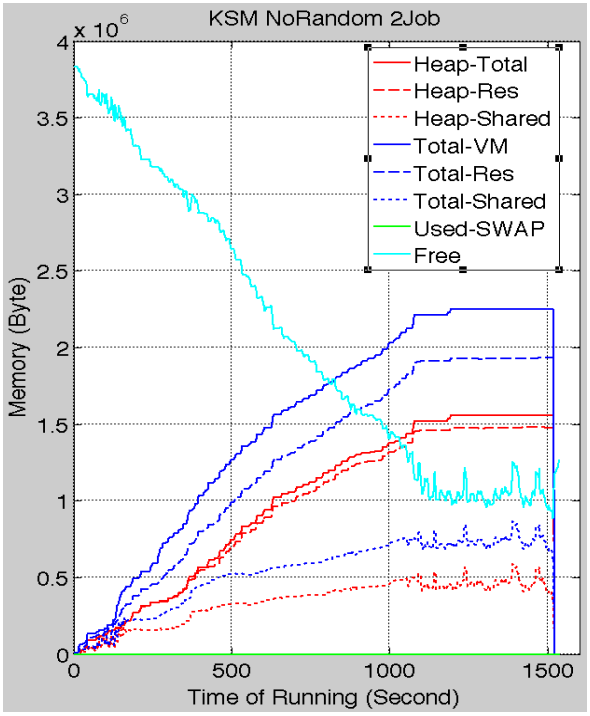

Figure 2: 2 LHC reconstruction jobs with KSM

6 KSM and embedded

KSM is suitable to be run on embedded systems too; the important thing is not to register in KSM regions that won't likely have equal pages. For each virtual page scanned, KSM has to allocate some rmap_item and tree_item, so while these allocations are fairly small, they can be noticeable if lots of virtual areas are scanned for no good.

Furthermore, these KSM internal rmap/tree data structures are not allocated in high memory. To avoid early out of memory conditions, it is especially important to limit the amount of lowmem allocated on highmem 32bit systems that might have more than 4GB of memory, but these shouldn't fit in the embedded category in the first place.

7 KSM and swap size

When KSM merges pages, it frees memory. However, it must be clear that the shared KSM pages remains shared only as the virtual machines using them are only reading from and not writing to those pages. So there is no actual guarantee that the memory freed by KSM as result of creating shared KSM pages will remain free. To obviate this problem administrators must tune the swap size appropriately, to ensure that even if the amount of shared memory would decrease significantly (if the workload of the virtual machines suddenly changes) the host Linux Kernel will not run out of physical memory.

8 KSM tree algorithm

The KSM tree algorithm is built around the concept that to find equal pages we add each page in the registered virtual memory areas to a Red Black Tree. The index of the tree is the content of the page itself. The function that searches the tree to find an equal page, will check the *memcmp()* return value to decide if to go left, right, or if we already found an equal page indexed into the tree.

9 KSM pass

A *KSM pass* for the scope of this document is intended as a entire full scan of all virtual areas marked VM_MERGEABLE by madvise, so registered in KSM.

10 Computational complexity

The usual page size for x86 architectures (and most other architectures) is 4096 bytes. But on average the *memcmp()* function will break out of its inner loop before processing all 4096 bytes. This is because the pages are unlikely to be all equal except for the last bits. The cost of finding an equal page, will be the cost of *memcmp()* multiplied by the number of levels in the tree. Thanks to the rbtree, the computation complexity of all insert/search/delete functions is O(log(N)) (where N is the total number of pages scanned by KSM). So even if we hit the absolute worst case where the first 4092 bytes of all pages scanned by KSM are equal, and only the last 4 bytes differs, the KSM tree algorithm will not degrade too much.

11 Stable and unstable trees

The KSM tree algorithm uses two rbtrees, one called *stable tree* (as in Figure 3) and one called *unstable tree*. Using two trees is an optimization and also increases the probability of quickly sharing the pages that are the most likely to be good candidates for sharing as well as reducing the instability of the *unstable tree*. The algorithm flow chart is visible in Figure 4.

For each anonymous page scanned, the kernel thread proceeds searching a match first in the *stable tree* that only contains already shared pages (shared so in turn write protected, hence their content is stable). If a match is found in the *stable tree*, the anonymous page is merged with the KSM page found in the *stable tree*.

If no match is found in the *stable* tree, KSM checks if the anonymous page has changed content recently using a checksum.

If the checksum changed since the last *KSM pass*, KSM updates the checksum and will defer the search of the *unstable tree* to the next *KSM pass* (assuming that the checksum won't change again). This is to avoid merging or adding to the *unstable tree* pages that changes content frequently.

If instead the checksum didn't change KSM proceeds searching the *unstable tree* that only contains anonymous pages scanned previously but not merged by KSM yet. If a match is found in the *unstable tree* KSM merges the anonymous page under scan, with the anonymous page in the *unstable tree*, and the resulting KSM merged page is added to the stable tree (the anonymous page found in the *unstable tree* is removed from the *unstable tree* and freed). If a match is not found in the *unstable tree* KSM adds the page to the *unstable tree*.

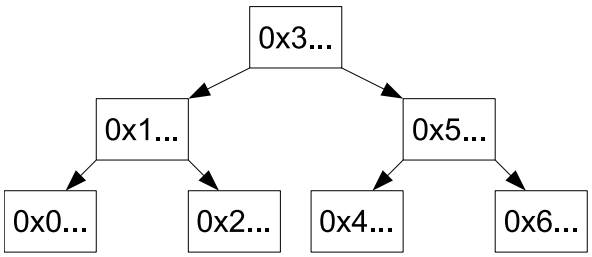

Figure 3: KSM *stable tree*

In the future, instead of the checksum, a dirty bit in the pte (and spte) can signal KSM if a page is worth adding to the *unstable tree* or not, or special instructions can be used if provided by the CPU to compute a checksum faster than jhash2.

This 'checksum' here has really nothing to do with the KSM Tree algorithm itself. The 'checksum' is not used to find equal pages to share; rather, it's only an heuristic to try to keep the *unstable tree* more stable and to avoid wasting time with bad sharing candidates. Even if we eliminate the checksum, the algorithm would still work.

If a page changes content frequently, besides risking the generation of false negatives from the *unstable tree* lookups, we'll likely only waste CPU by sharing it, because a copy-on-write page fault will likely happen soon enough, breaking the sharing.

12 When the *unstable tree* becomes unstable

We must avoid write protecting pages that aren't shared yet, or the whole virtual memory scanned by KSM would be write protected most of the time, in turn leading to a flood of copy-on-write page faults. The *stable tree* only contains shared KSM pages, and we know all pages inside it aren't going to change content because they have to be write protected in the pagetables and shadow pagetables in the first place in order to be shared. So a lookup in the *stable tree* is fully reliable and can't return false negatives. It's just like a lookup of any other regular rbtree in the kernel, where the index doesn't change under the tree after the node is indexed

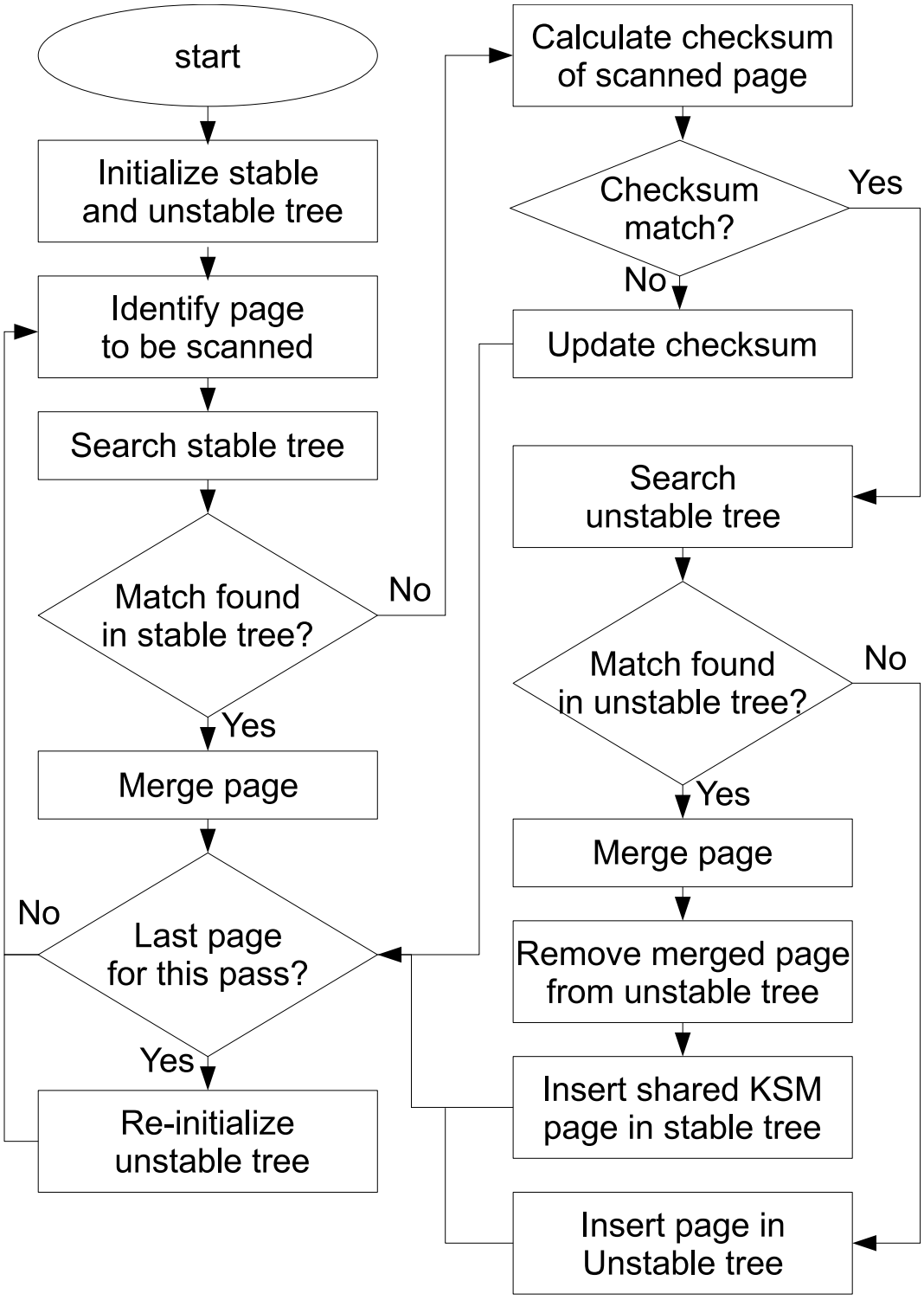

Figure 4: KSM Tree algorithm flowchart

into the tree. The problem is the lookup of the *unstable tree* because the *unstable tree* only contains regular anonymous pages not shared yet, that can be still written to by applications.

Because the rb_insert()/rb_erase() functions that balance the rbtree while inserting and deleting an element from the tree are unaware of the index value, we're guaranteed the rbtree will remain well-balanced regardless of where we insert any new node in the tree. We are also guaranteed that all insert, search, and delete operations will not degrade in terms of computational complexity, even after the *unstable tree* becomes really unstable.

An example of the *unstable tree* while it's still stable can be seen in Figure 5. If an application writes to a page indexed in the *unstable tree* that had the first byte set to 0x03 when it was inserted in the stable tree, and it changes it to 0x07 afterwards, the *unstable tree* might become unstable as in Figure 6 and lookups might start to generate false negatives.

To avoid the instability and the resulting false negatives to be permanent, KSM re-initializes the *unstable tree* root node to an empty tree, at every *KSM pass* (i.e. after completing a full scan of all virtual areas registered in KSM). This way, a new *unstable tree* is rebuilt from scratch at every *KSM pass* and the false negatives won't be sticky.

To further decrease the probability of false negatives from the *unstable tree* lookups, we could also remove pages from the *unstable tree* if we find a dirty bit set or the checksum being not uptodate anymore during the tree walk, even though we're not doing it in the current implementation as it'd make the search in the *unstable tree* slower than just a *memcmp()* for each level of the tree.

Because all long-term important sharable pages are going in the stable tree over time, the *stable tree* guarantees us that the important sharable pages are going to be merged without any risk of false negatives, regardless of any temporary instability of the *unstable tree*.

If all goes well and there are no false negatives, while inserting an anonymous page in the *unstable tree*, KSM will find a page with equal content already indexed in the *unstable tree*, so KSM will merge them together, it will create a new KSM page with equal content added to the *stable tree* and remove the indexed anonymous page from the *unstable tree*, and finally will free both anonymous pages.

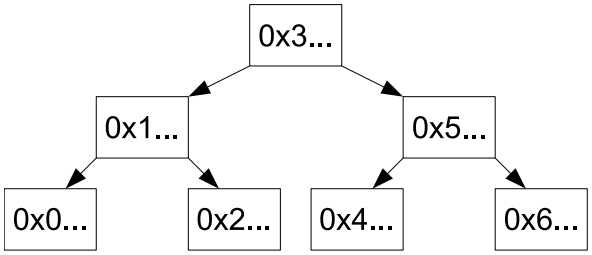

Figure 5: KSM unstable tree while still stable

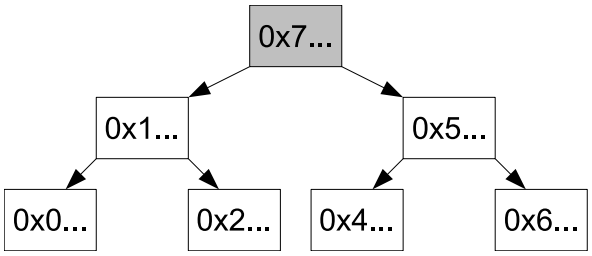

Figure 6: KSM unstable tree gone unstable after application write

13 Page merging

The procedure used for page merging involves two functions: *page_wrprotect()* and *replace_page()*. The former write protects all pagetables mapping the page passed as parameter (and sptes too through change_pte() mmu notifier discussed below) method; the latter merges two pages by updating the pagetables accordingly (and sptes too through change_pte() mmu notifier), and then by freeing the merged anonymous page that no pagetable (or spte) maps anymore.

One final *memcmp()* is required after *page_wrprotect()* returns to be sure both pages being compared cannot change while *memcmp()* runs. Only if the final *memcmp()* succeeds (returning zero) *replace_page()* is called to merge the two pages.

If there is a match in the *stable tree*, the KSM page already in the *stable tree* is merged with the anonymous page under scan.

If there is a match in the *unstable tree*, a new KSM page is allocated and the content of one of the anonymous pages is copied to it, and both anonymous pages are merged with it.

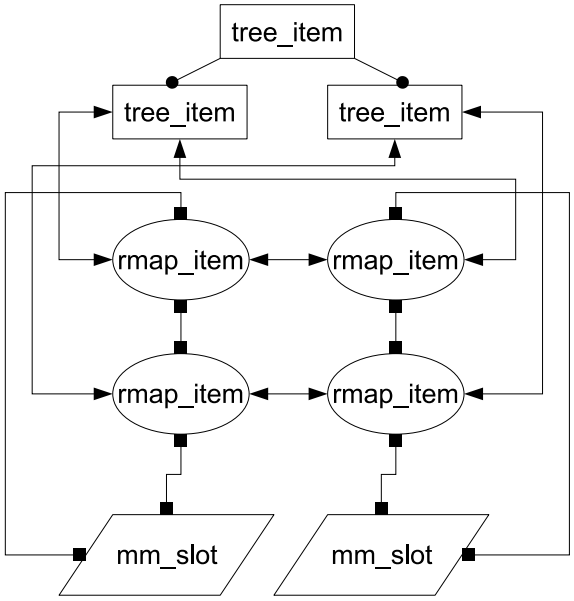

Figure 7: rmap_item and tree_item relation

The merge path is a fairly slow path: if it would run all the time, it would mean that all virtual addresses scanned by KSM are sharable all the time which certainly isn't the case most of the time. In the future we could however optimize the *unstable tree* merge path to transform an anonymous page into a KSM page in place to avoid one page copy and to optimize away some minor pte/spte mangling for one of the two anonymous pages being merged together.

14 KSM rmap_item and tree_item

A physical page is represented in the the *stable* and *unstable* trees by the tree_item structure. The tree_item is a rmap structure that contains the head of a list that links all virtual addresses that map each physical page. The virtual addresses (the list elements) themself are represented by a rmap_item structure. Their relation is shown in Figure 7.

So during the *stable* and *unstable* tree lookups KSM, walks the tree_item list to find the virtual address (in the rmap_item) to call get_user_pages and to obtain the physical page address to run *memcmp()* against.

15 KSM rmap_item and tree_item out of sync with the Linux VM

The reason get_user_pages() is called during the tree walk is that we're tracking virtual addresses instead of page frame numbers in the tree_item. This is because the tree_item and the rmap_item are maintained out of sync with the Linux VM. This means that if an anonymous page is swapped out or unmapped, we'll find out only during the tree lookups or during the KSM scan on the virtual areas registered. Whenever we find a virtual address not mapped in the pagetables, we drop the respective rmap_item and if that was the last rmap_item in the tree_item linked list, we also drop the tree_item.

16 KSM rmap_item and tree_item in sync if KSM would register its mmu notifier methods

We considered to change KSM to avoid the get_user_pages() call during the tree walk by storing a pointer to the physical page directly in the tree_item by using mmu notifiers that would notify us whenever a rmap_item should be dropped. However, in addition to making the code more complicated, that would require global spinlocks that would serialize the rbtrees lookups against mmu_notifier invalidate methods, and it might lead to applications to scale worse because every time a virtual area is unmapped that global lock would be taken. We want KSM to run in the background without affecting the regular runtime of applications as much as possible. Furthermore, replace_page() used by KSM to merge the pages would then re-enter KSM again through a mmu notifier invocation in replace_page() after it mangles the pte, a case that would require some special handling and perhaps rmap_item refcounting. Because keeping the rmap_item and tree_item fully synchronized isn't required to efficiently find equal pages, we think it's simpler to maintain them out of sync, with the main disadvantage of the tree walk requiring get_user_pages calls, but we prefer KSM itself to be a bit slower in merging memory and not to risk slowing down the actual applications with global locks.

Strictly speaking for the unstable tree a per-mm *unstable tree* protected by a per-mm lock would be feasible but the *stable tree* spinlock would need to be global if we want to share memory system-wide.

There are various implementations but likely the way KSM will implement the rmap_item scan over the vmas with the madvise API is to keep an list ordered by address of rmap_item for each mm with vmas with VM_MERGEABLE set, and to resync the rmap_item

list in a inner loop, with the outer loop being the vma->vm_next loop. Any rmap_item instantiated in a previous *KSM pass* but found not anymore in the range of any VM_MERGEABLE vma will be dropped, and new rmap_item will be created for each new virtual address that has a pagetable pointing to an anonymous page in a VM_MERGEABLE vma. This way the madvise syscall will not have to call KSM, and it will only have to split vmas if needed and set or clear the VM_MERGEABLE flag in the virtual areas passed as parameter to madvise MADV_MERGEABLE.

17 MMU notifier change_pte() method

When KSM merge pages, we don't want to teardown all secondary pagetables (e.g. the VT shadow pagetables instantiated by KVM). To avoid that a new change_pte() method is used by *replace_page* that will update all sptes that pointed to the old anonymous page to point to the location of the new KSM page.

If we used the invalidate_page() method instead of introducing a new change_pte() mmu notifier method, KSM would have destroyed the sptes in turn requiring KSM to take minor faults to recreate them later as the guest returns to access those guest virtual addresses, by rereading the kernel pagetables.

Side note: due to some short term limitation right now KVM will always trigger write faults as far as the Linux VM is concerned even if the guest issued a read memory operation, so lack of change_pte() method would have prevented the shared KSM pages to be mapped by any shadow pagetable at all. This limitation in the KVM page fault will however be addressed in the future, but change_pte() will still remain an useful optimization even then, by preventing KVM to vmexit to rebuild invalidate sptes (even if the sptes in the future could be rebuilt by KVM with readonly permissions without triggering copy-on-write faults in the Linux VM if the guest issued a read access on a KSM page).

change_pte() takes the Linux pte as parameter and it makes sure the sptes are marked readonly if the pte passed as parameter is readonly.

change_pte() is also used in the Linux VM write protect page faults triggering on KSM pages as an optimization to avoid tearing down sptes (do_wp_page()).

Not all MMU notifier users are required to implement the new change_pte() method; if not implemented, it will simply fallback to the invalidate_page() backwards compatible behavior, which is safe but less efficient for users like KVM. For the MMU notifier users that don't manage real secondary pagetables, but only a secondary tlb (like GRU), implementing the change_pte() method is unnecessary.

18 KSM not working on pages under GUP

It is interesting to note that page_wrprotect() has to fail for any page that is temporarily pinned by get_user_pages() users (to avoid generating I/O corruption on the drivers that accesses the pinned pages directly) and it will only function on drivers that uses MMU notifier and that can unpin the pages immediately after get_user_pages() returns. So to allow KSM to work on KVM guest physical memory, we had to remove the page pinning on the shadow pagetable mappings (in short that means calling put_page() immediately after get_user_pages() returns, and entirely relaying on mmu notifiers methods to teardown shadow pagetables before the corresponding virtual address is teardown by the Linux VM, either because of VM pressure or userland action).

All get_user_pages() pins shall be temporary; if not, the pinned pages cannot be paged out by the VM in case of memory pressure. So if the pins are temporary as they've to be, KSM will simply be able to write protect those pages (and then possibly to merge them) in one the next *KVM passes*. Drivers that use the pages returned by get_user_pages() in a persistent way like KVM must use MMU notifiers and release the page pins to be transparent to the Linux VM and in turn to allow KSM to merge pages on those memory regions too.

19 KSM multi threading

In the future it will be possible to add more than one KSM kernel thread by adding a read-write mutex or spinlock that protects each tree. Starting more than one KSM kernel thread will be helpful if somebody wants to dedicate more than 100% of one CPU core at merging pages.

20 KSM swapping

KSM pages cannot be swapped at this time; KSM pages are effectively nonlinear entities mapped in the middle

of linear anonymous vmas and the Linux VM swap logic cannot cope with them at this point in time.

Because KSM to function requires its own internal rmap logic and because we surely don't want to hurt the Linux Kernel VM memory footprint when KSM is not enabled, likely an external rmap functionality shall be implemented to allow the Linux VM to call into KSM to unmap all pagetables mapping the shared KSM pages. The swapin path will also require some change because the anonymous fault won't be suitable for swapping-in KSM pages if they've been swapped-out, similarly to how tmpfs swaps out the tmpfs shared pages.

21 Reduce *memcmp()* length in tree lookup maintaining rbtree cumulative info

It should be possible to add to the tree_item some rbtree related metadata information on the status of the left and right nodes. This metadata information can tell the tree lookup function the offset of the first byte that differs between the current node physical page, and the two physical pages in the right and left nodes. That will require adding a callback to rb_insert() and rb_erase() called with proper information during each tree balancing rotation of the nodes, so that this metadata can be recalculated at every rebalance of the tree. With this information, we should be able to significantly reduce the cumulative amount of memory compared by the *memcmp()* function during a worst case of tree lookup.

If rb_insert() and rb_erase() will be extended like above, the rebalancing callback could also be used by get_unmapped_area() to allow it to work in O(log(N)) instead of the current O(N) (where N is the number of vmas in the mm_struct).

22 KSM benchmark

We run all test cases using a Linux 2.6.30-rc6 kernel, with a fairly recent KVM external module and the KSM patchset posted on the Linux Kernel Mailing List on 20 April 2009 with Message-ID: 1240191366-10029-1-git-send-email-ieidus@redhat.com (which still uses the old ioctl API and not madvise yet). To merge memory at the fastest possible peace, KSM clearly has been tuned so that the single threaded kksmd kernel thread runs at 100% CPU load (sleep = 0, pages_to_scan = 1000000). The hardware used is on a common and cheap Intel Q9300 Core 2 Quad at 2.50GHz with 4 GigaByte of 800mhz DDR2 memory.

We intend to measure here the max speed of KSM in merging pages under best, worst and real life cases. The number of MegaBytes per sec of memory merged by KSM when KSM runs at full CPU utilization is a relevant parameter to measure, because it shows how fast KSM is at merging pages. In real life environments it's unlikely KSM will be tuned to run at full CPU utilization (with the exception of very large servers with many CPUs and several dozen GigaBytes of RAM), but the fastest KSM is at merging pages at full CPU utilization, the lower CPU KSM will take when tuned for real life environments. We could have statistically measured the average CPU utilization instead, but measuring the amount of RAM merged per second and maxing out the CPU utilization allows for a much more reliable measurement of the efficiency of the algorithm under different workloads. The forth column of the output from *'vmstat 1'* will be used to monitor the progress KSM does in merging memory.

The worst case for KSM that should practically never materialize in practice (unless of course malicious users can run their own malicious applications) can be exercised with an application that allocates one gigabyte of memory and that makes all pages equal except for the last 4 bytes of each page. The first copy of this application called *ksmpages* will fully populate the *unstable tree*. Running a second copy will merge all pages in the *unstable tree* and it will create equal amounts of KSM shared pages in the *stable tree* and free one gigabyte of memory in the process. The *stable* and *unstable* trees generated will have many levels and the *memcmp()* will not break before at least 4092 bytes have been read for each level of the tree.

The best case for KSM can be exercised with an application that allocates one gigabyte of memory and initializes all pages to the same value. KSM when started will quickly free one gigabyte of memory minus one KSM page that will be the only one indexed in the *stable tree*. The unstable tree will not be empty only before the very first merge. A single *memcmp()* and a single level of the *stable tree* has to be walked in order to merge the pages.

The real life case for KSM can be measured by running two copies of a popular proprietary guest OS in KVM with 1G of memory each, wait both of the to finish booting, and finally start KSM and see how fast the memory

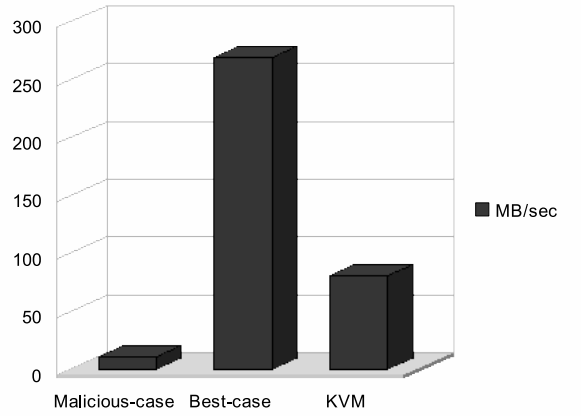

Figure 8: KSM *benchmark*

is merged (i.e. freed). Then we stop the kksmd thread, we start a third VM of the same OS, and we start kksmd again.

23 Conclusions

Considering that even the worst possible malicious case on one of the cheapest workstation hardware configurations with very cheap motherboard and northbridge, definitely makes progress at 10.62 MegaBytes merged per second (note that the only side effect of malicious behavior is an higher CPU utilization), that the fixed cost of the virtual address scanning and page merging is CPU bounded at 269.05 MegaBytes per second, and that the real life KVM case merges pages at 80.70 MegaBytes per second, we're comfortable this algorithm (even without the future possible further optimizations) in the background will be able to merge pages efficiently in virtualization and scientific environments and in embedded systems as well.

24 References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Chapter 13—Red-Black Trees* Introduction to Algorithms, Second Edition. The MIT Press, September, 2001.

Vincenzo Innocente, Summary of the evaluation of KSM for sharing memory in a multiprocess environment, `https://twiki.cern.ch/twiki/bin/view/LCG/EvaluationKSM0409`, 20 May 2009.

Izik Eidus, KSM version used for benchmark, `http://kerneltrap.org/mailarchive/linux-kvm/2009/4/20/5521504`, 20 April 2009.

Andrea Arcangeli, ksmpages.c source and some benchmark, `http://kerneltrap.org/mailarchive/linux-kvm/2009/3/31/5349904`, 31 March 2009.

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*