KSM++: Using I/O-based hints to make memory-deduplication scanners more efficient

Konrad Miller ‡ Fabian Franz † Thorsten Groeninger †
Marc Rittinghaus † Marius Hillenbrand ‡ Frank Bellosa ‡
Karlsruhe Institute of Technology (KIT)
‡firstName.lastName@kit.edu †firstName.lastName@student.kit.edu

Abstract

Memory scanning deduplication techniques, as implemented in Linux' Kernel Samepage Merging (KSM), work very well for deduplicating fairly static, anonymous pages with equal content across different virtual machines. However, scanners need very aggressive scan rates when it comes to identifying sharing opportunities with a short life span of up to about 5 min. Otherwise, the scan process is not fast enough to catch those short-lived pages.

Our approach generates I/O-based hints in the host to make the memory scanning process more efficient, thus enabling it to find and exploit short-lived sharing opportunities without raising the scan rate. Experiences with similar techniques for paravirtualized guests have shown that pages in a guest's unified buffer cache are good sharing candidates. We already identify such pages in the host when carrying out I/O-operations on behalf of the guest. The target/source pages in the guest can safely be assumed to be part of the guest's unified buffer cache. That way, we can determine good sharing hints for the memory scanner. A modification of the guest is not required.

We have implemented our approach in Linux. By modifying the KSM scanning mechanism to process these hints preferentially, we move the associated sharing opportunities earlier into the merging stage. Thereby, we deduplicate more pages than the baseline system. In our evaluation, we identify sharing opportunities faster and with less overhead than the traditional linear scanning policy. KSM needs to follow about seven times as many pages as we do, to find a sharing opportunity.

Keywords Memory-deduplication, KSM, Hinting, VMM

1. Introduction

Memory size has become a scarce resource in many computing scenarios, foremost in cloud computing. In this scenario, virtual machines (VMs) make flexible allocation, reallocation and consolidation of multiple operating systems onto fewer physical machines possible, while maintaining strong service isolation.

The number of VMs that can be consolidated is highly dependent on the amount of available memory. However, in such environments, there is plenty of redundant data. Pages with equal content can be merged to a single page and shared in a copy-on-write fashion.

RESoLVE'12 March 3rd, 2012, London

However, the information loss that occurs in the interplay between I/O-devices, host physical-, guest physical-, and guest virtual memory leads to the so-called semantic gap [7]. The lack of semantic information that the host[1] has about guest activities is actually one of the key features of virtualization: The host does not know or need to know what OS, file system, etc. are used inside the VM. This way, however, there is not enough semantic information to share duplicate pages through traditional sharing mechanisms.

Prior work has made deduplication of redundant pages possible and thereby increased the available main memory, either by closing the semantic gap with paravirtualization [5, 20], or by mitigating the semantic gap through scanning for duplicate contents in anonymous pages [1, 25]. Both techniques have drawbacks: *Paravirtualization* implies modifying both the host and guest. To apply these modifications to all guests is at least a great burden: In some cases it cannot be achieved as easily as loading a kernel module, which is a common way to implement paravirtualization for device drivers. It might not even be possible at all to modify commercial or legacy guests due to license restrictions or the lack of source code. *Memory scanning*, in turn, has its downside when it comes to efficiency and effectiveness. Especially the merge latency – the time between establishing certain content in a page and merging it with a duplicate – is higher in systems based on content scanning. Although the scan rate (pages per time interval) is often variable and may be fine tuned [9, 23], it is generally set to scan very slowly. Memory scanners directly trade computational overhead and memory bandwidth with deduplication success.

The main contribution of this paper is the combination of hints, based on read and write operations in the virtual file system of the host, with memory-scanning–based deduplication systems: We propose to use those I/O-based hints to make the memory scanning process more efficient and in consequence enable it to find and exploit short-lived sharing opportunities without raising the scan rate.

Figure 1 gives an overview over the longevity of sharing opportunities. Pure brute-force scanning for duplicate pages in main memory either requires too many resources to justify the benefit, or is too slow to identify the majority of the plenty short-lived sharing opportunities in time.

The source of identical pages in different VMs' memory is most commonly their virtual disk image (VDI). Between 63.77% and 94% of shareable pages are part of the page cache [15], a property which is exploited in approaches that employ sharing-aware (virtual) block devices to find duplicates in main memory [5, 16, 20]. These approaches, however, cannot find duplicate pages in anonymous memory.

[1] We use the term host interchangeably with virtual machine monitor (VMM), hypervisor, or host OS to describe the system layer underneath the guest OS.

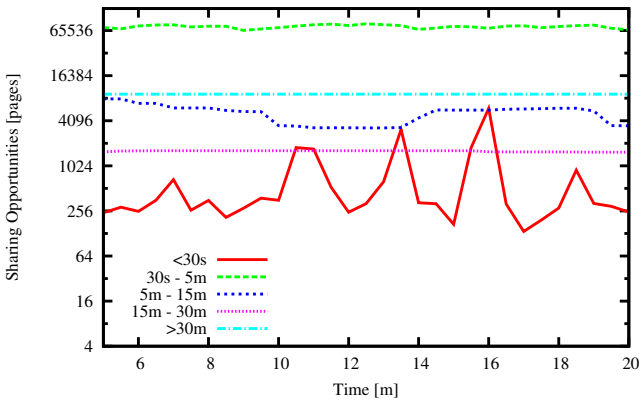

Figure 1. The longevity of sharing opportunities between two VMs that are compiling Linux v3.0. Notice the logarithmic scale. Sharing opportunities that last between 30 sec and 5 min are by far the most common.

Our approach is also based on this property, that most shareable pages stem from the VDI. We show that extending main memory scanning with out-of-order hints can improve the deduplication effectiveness significantly. Hints are issued for all main memory pages that are the target of disk-read or source of disk-write operations (VFS-read, VFS-write, or `mmap` flush) in the *host*. When the guest issues a read request from the VDI, the virtual DMA controller in the host handles the request and performs a read request on the physical disk on behalf of the guest. The assumption is that the target of that DMA transaction is a page in the guest's buffer cache and thus a good sharing candidate. The same is true in the opposite direction – when a buffer cache page is flushed to disk, we record this as a write to the VDI in the host. In both cases the host generates a hint to the memory scanner for the involved memory page to check the page for being a duplicate the next time it runs. Note that we only modify the host in our approach – we do not rely on paravirtualization techniques.

We have implemented our approach in Linux. By modifying the KSM scanning mechanism to process these hints preferentially, we move the associated sharing opportunities earlier into the merging stage and can thereby deduplicate more pages than the baseline system. In our evaluation, we show that we can identify sharing opportunities faster and with less overhead than traditional linear scanning policies.

The remainder of this paper is structured as follows: We review Kernel Samepage Merging (KSM) – the basis for our implementation – in Section 2 before we describe our approach and the implementation of our prototype thoroughly in Section 3. In Section 4, we present the results of our evaluation. We give an overview of related work on memory deduplication in Section 5. Finally, we conclude and depict future research directions in Section 6.

2. Memory Scanning with KSM

Our implementation is based on Kernel Samepage Merging (KSM), also known as Kernel Shared Memory. KSM is a popular memory-deduplication approach built into the Linux mainline kernel. It can be used to scan for and merge equal, anonymous main memory pages. Note that KSM is not bound to VMs – it works on anonymous memory regions of any process. However, KSM only regards specifically advised pages as mergeable. As all virtual memory that stems from VMs is regarded as anonymous memory by the host due to the semantic gap, QEMU [3] invokes the appropriate `madvise` call on all virtual memory areas of all VMs.

KSM uses two red-black-trees as its main data structures. The stable tree stores guest pages that have already been merged, while the unstable tree records pages that do not change frequently, and are thus suitable sharing candidates, without protecting those pages from being written to.

Figure 2 depicts the KSM scan process. KSM incrementally searches for pages that do not change frequently by calculating a hash value (jhash2) for every scanned page. If the calculated hash value differs from the one recorded in the previous scan round, the hash value record is updated and the scan continues with the next page. Otherwise the associated page is inserted into the *unstable tree*. The entire page's content is used as an index for insertion or lookup, not the hash value. In the average case the insertion or lookup code only needs to `memcmp` a small portion of the page to infer if it needs to go left or right on every one of the $O(log\ n)$ levels in the tree.

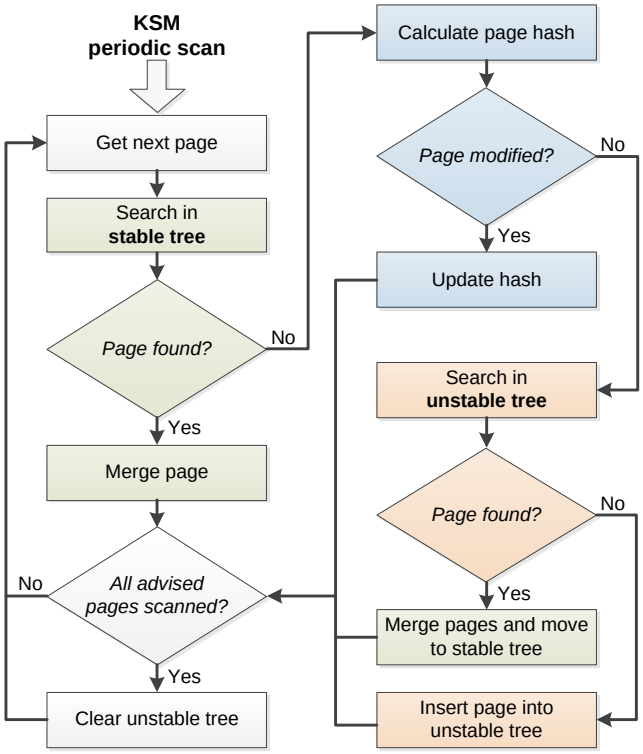

Figure 2. KSM's high-level workflow. KSM allocates a structure containing information such as a checksum and sequence number for every page. *Get next page* refers to getting the next such structure from a circular list that contains an entry for every advised page in the system. Only pages whose checksum has not changed since the last visit are inserted into the unstable tree. For a merge, one of the involved pages must already be in either the stable or the unstable tree. KSM drops the unstable tree and starts over after it has visited all advised pages.

Pages are *not* mapped read-only on insertion into the unstable tree but remain writable for the VM; the location in the tree is thus purely based on the value the page had at the time of its insertion. The tree may break if pages in the unstable tree are written. The described heuristic keeps the memory scanner from inserting frequently written pages into the unstable tree, however it does not guarantee that the pages will not be modified after being inserted. In our benchmarks with an unmodified Linux v3.0 KSM, up to almost 70% of the nodes cannot be reached in the unstable tree after a full scan, due to page modifications after insertion. A very radical approach is chosen to clean up broken branches of the unstable tree:

When a full scan has been performed, the entire unstable tree is dropped and a new one is built from scratch.

Up to this point the algorithm has not merged pages yet. Before calculating the hash value of a page as described above, KSM first checks if the page's content is already in the stable tree – the data structure that contains all merged pages. If that is the case, the pages are merged. If the page is not in the stable tree, and the page also has not changed recently according to its hash value, the page is searched in the unstable tree. If a page with the same content can be found in the unstable tree, the page is copied, remapped read-only, and inserted into the stable tree. Then, the matched page is purged from the unstable tree and the two source pages are freed. If no match was found, the page is only inserted into the unstable tree. This may also happen when there actually is a page with the same content in the unstable tree, which is not reachable, however, because the page itself or a page in the search path has changed since its insertion.

The scanning process is repeated until all advised pages are scanned. Then, the unstable tree is dropped and the process starts from the beginning. Only the hash values and the stable tree remain.

3. Approach and Implementation

To show the effectiveness and efficiency of our approach, we have extended KSM in Linux 3.0. We have chosen KSM because Linux with QEMU is a popular host and KSM already implements many of the mechanisms we need.

In the following paragraphs we will explain our approach in detail and highlight some key issues that we encountered and how we solved them. We discuss how we generate (§ 3.1), store (§ 3.2), and process (§ 3.3) deduplication hints interleaved to KSM's periodic memory scan. We need to add hinted pages to KSM's unstable tree immediately, and bypass KSM's heuristic to filter out frequently changing pages, to be effective. To keep the unstable tree from degenerating due to modified pages, we map hinted pages read-only to detect such changes (§ 3.4).

3.1 Generating Deduplication Hints

We have extended `madvise` so we can call it from inside the kernel and added a new advise-class for our hinting mechanism. We have then added code to the VFS read functions (`read`, `readv`) to advise the read target pages, and the VFS write functions (`write`, `writev`) to advise the write source pages with our new flag and thereby generate a deduplication hint. The code only generates a hint when the target/source is a page that is already advised as mergeable to KSM (e.g., it is part of a VM guest's memory). In this case, `madvise` inserts the hinted memory area into our storage structure to record this area to be processed by the page scanner.

We do not need to modify the guest in any way; the mechanism even works for regular, non-VM processes if the associated page is mergeable by KSM.

3.2 Storing Hints and Coping with Bursts

I/O is generally bursty; as a result, I/O-based hints are also issued in bursts. When obeying to the KSM scan rate (generally set to very low settings), we cannot always keep up with the amount of incoming hints [2].

Storing our hints in an unbounded queue does not work well in this scenario, as we need to actively deploy aging and pruning mechanisms. Otherwise, the queue will keep growing and the processed hints will get older over time. Eventually, we will fall behind to a state in which we do not find any sharing candidates through the hinting mechanism at all, as the hinted pages have already changed their content until we process them.

We have chosen another option to store our hints with low overhead and no required maintenance work: we implemented a bounded circular stack (Figure 3).

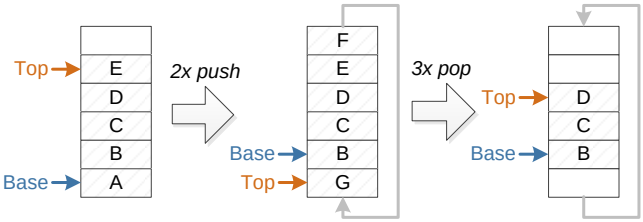

Figure 3. Storage of hints in the bounded circular stack.

Due to the nature of a bounded circular stack, we will always process the newest hints first while old hints are automatically overwritten when the stack is full – an automatic pruning and aging mechanism which proved to be fast and robust.

3.3 Processing Deduplication Hints

We have extended the KSM workflow to incorporate our hinting mechanism. As depicted in Figure 4, we have built an operating cycle for processing our hints similar to that of KSM. Our hinting mechanism runs interleaved with the full system scan-spurts (wake-ups) that KSM already implements for anonymous, advised pages. We obey the rate limits set for KSM and produce roughly the same load as KSM with the same settings. The interleaving ratio is configurable – `hint_runs` hint-processing-spurts interleaved with `scan_runs` scan-spurts. Using this policy, we can guarantee that the linear scan, which can also catch non-I/O sharing opportunities, does not starve due to a flood of hints.

When we are in a scan-spurt, we run the traditional linear scanning policy only. In a hint-processing-spurt however, we process hints as long as there are hints left and we have not exceeded the scan rate. The remaining slots that we can process until exceeding the scan rate are used for the linear scan.

Our mechanism first checks whether the hinted page's content is in the stable tree already. If this is the case, we remap the page to the one in the stable tree and free the hinted page. If the hinted page is not in the stable tree, we calculate the checksum of the page and check the unstable tree. If a sharing partner is found, we merge the pages and add the resulting page into the stable tree. If we cannot find a sharing candidate we add the page to the unstable tree.

3.4 Degeneration of the Unstable Tree

The original KSM implementation has a heuristic that keeps pages that are written frequently from being inserted into the unstable tree. Only pages that keep the same hash value over two scanning rounds are inserted (see Section 2). We want to merge hinted pages as quickly as possible, without waiting for an entire scan cycle to finish. Therefore, we add pages to the unstable tree immediately when processing a hint.

The pages that are associated with virtual DMA read and write requests can generally be assumed to be part of the unified buffer cache in the guest. The unstable tree degenerates predictably when applications running inside the guest fill the page-cache from the inside of the VM. In this situation, the guest replaces buffer cache entries that stem from the VDI without the host noticing. Thus, hinted pages in the unstable tree become modified and thereby unreachable in the tree. Furthermore, when pages in inner tree-nodes change, their siblings and therefore entire sub-trees may also become unreachable. The host can only pick up on such modifications

[2] Even with aggressive settings we would not be able to cope with the hint flood of some bursts. In workloads with very high I/O-throughput some million hints can be generated in a matter of seconds.

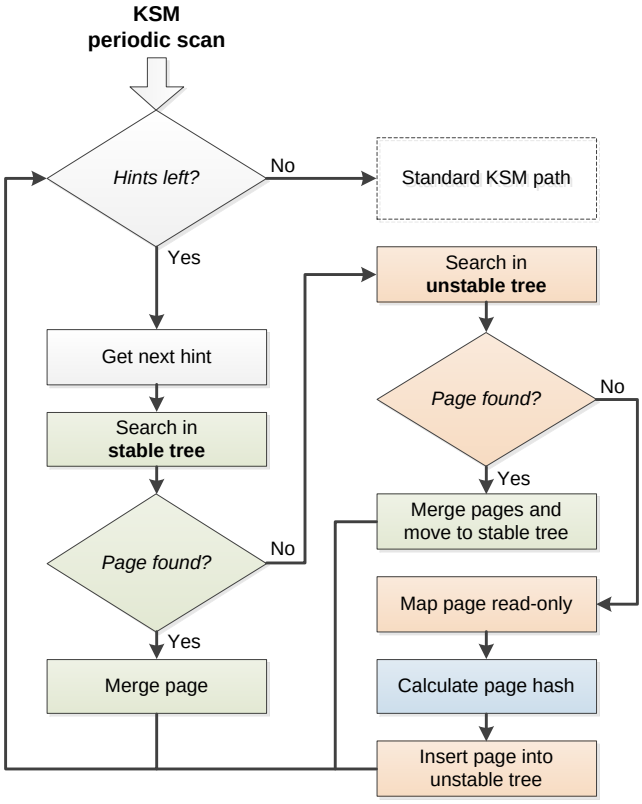

Figure 4. The high-level workflow of our hinting mechanism. For every hint we process, we first check if a merge partner can be found in either the stable or in the unstable tree. If we find a match, we merge the pages in the same way KSM does. Otherwise, we remap the page read-only (if it is not already), calculate its hash-value and insert it into the unstable tree. When all hints have been processed before the scan rate is exceeded, we continue with the regular KSM scan path.

once the guest flushes the written page to the virtual disk. KSM's natural repair mechanism (i.e., dropping the unstable tree after each complete scanning round) is slowed down by our modifications as the number of full memory scan cycles per time decreases: The scan rate stays constant, but multiple hints can and will be issued on the same pages during a scan cycle.

One way to counter the more likely degeneration of the unstable tree is to map hinted pages that are inserted into the unstable tree as read-only. This way we can use write faults on hinted pages as a signal to remove these pages from the unstable tree and thereby prevent the tree from becoming unstable when hinted pages are written. This is not the same mechanism as breaking COW pages, which happens when writing to a page in the stable tree. The page does not need to be copied but only remapped read-write and removed from the tree. A flush operation from the buffer cache to the (virtual) disk has a much higher latency than a page fault does. We can therefore map hinted pages read-only with justifiable overhead.

4. Evaluation

We have conducted several benchmarks to show the effectiveness and efficiency of our approach. We were particularly interested in seeing whether our system can merge more pages with an overhead that is comparable to KSM, our baseline system. To get results that can be related to prior work, we have chosen one of the benchmarks that were used to evaluate Satori [20]: Compiling the Linux kernel with 512 MB of RAM available in the VM. Our benchmarks have been conducted on a PC with an Intel i7-2600K processor, 24 GB of Kingston DDR3 RAM in total, and an Intel X25-E SSD. Ubuntu Linux 11.04 64-bit served as the host and also as the guest.

To be able to measure sharing opportunities and exploited sharing accurately, we have written a kernel module comparable to Exmap [4] that dumps page table information and page content digests [11]. Furthermore, we output internal information and statistics from KSM and our hinting mechanism through the sysfs interface.

We divided our evaluation into the following three parts. First, we compare the impact that different configuration parameter values have on the deduplication performance and use these results to determine good default settings for the following experiments in § 4.1. We also address the stability of the unstable tree and the impact of our approach to mitigate the degeneration of the tree in this paragraph. We demonstrate that our revised algorithm is much more effective than the baseline system through merging short-lived pages in § 4.2. Finally, we show that the efficiency of our approach is superior to KSM in § 4.3 – we need to visit about a seventh of the pages and thus need less CPU cycles and memory bandwidth to find a sharing candidate.

Unless specifically stated otherwise, we use the parameters that can be found in Table 1 for our benchmarks.

Parameter	Value	Description
scan_run	1	Interleave each scan-spurt...
hint_runs	1	... with one hint-spurt
pages_to_scan	100	# of pages to scan on wake-up
sleep_time	100	Sleep-time between spurts $[ms]$
map_hints_ro	1	Hinted pages are mapped read-only
stack_size	1024	Size of the hints buffer

Table 1. Default settings in our various benchmarks.

4.1 Configuration Parameters

We have introduced several new configuration parameters and have conducted experiments to determine good default values for the subsequent benchmarks in this section.

Stack Size We use a bounded circular stack to store our hints. The size can be configured through the procfs interface. We have run a kernel build with different sizes of that hint buffer to find a good configuration. The results are depicted in Figure 5. We have chosen a stack size of 1024 entries as the default.

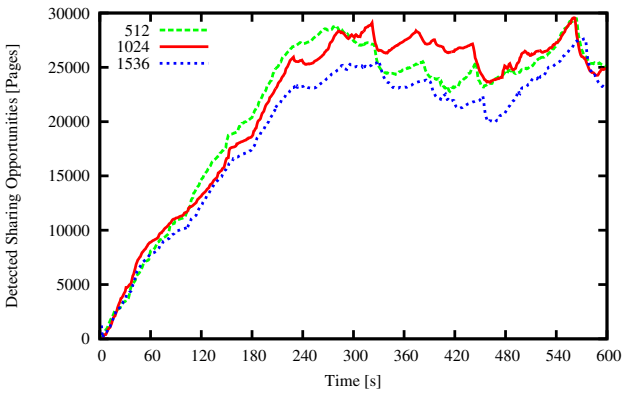

Figure 5. Deduplication effectiveness with varying stack sizes.

Interleaving Ratios Another parameter that we have introduced is the ratio between hint-spurts (`hint_runs`) and scan-spurts (`scan_runs`). The deduplication performance with different configurations is depicted in Figure 6.

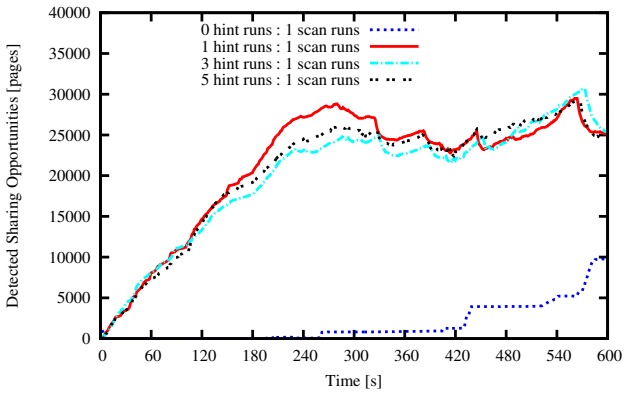

Figure 6. Deduplication performance with different hint-spurt vs. scan-spurt ratios. The 0:1 configuration corresponds to KSM.

The 0:1 benchmark corresponds to the original KSM implementation, as no hint-spurt is interleaved with one scan-spurt. We have chosen the 1:1 configuration as the default for all subsequent experiments.

The Unstable Tree's Stability Our hinting mechanism inserts pages into the unstable tree without obeying to KSM's heuristic: KSM only inserts pages into the unstable tree if their hash value has not changed since the last pass, while we insert hinted pages immediately. We have examined how much this affects the stability of the unstable tree and the overall memory-deduplication performance.

A good metric for the stability of the tree is the ratio between the number of nodes in the tree and the number of nodes that can be found when searching for them in the tree. If a page cannot be found in the tree, it cannot be merged. Instead, it is inserted again – this time in another place. Table 2 shows the percentage of reachable pages in the unstable tree in different configurations in the kernel build benchmark after a full scan cycle.

Vanilla KSM	Hints (all rw)	Hints (hints ro)
33.6% - 53.0%	10.0%	98.9%

Table 2. Percentage of the unstable tree that is reachable at the end of a scan cycle. Adding hints without KSM's hash value heuristic degenerates the tree. Mapping the hinted pages read-only but leaving the scanned pages read-write improves the tree's stability significantly. The I/O-pages are responsible for degenerating the tree.

We have found that pages which are part of the buffer cache are the ones that are most likely to change among all pages in the unstable tree. This happens when buffer cache pages are written back, evicted, and replaced by another file's pages in the guest. Buffer cache pages have the characteristic that they do not change frequently, which is why they make it into the unstable tree in the first place, but definitely change at the time they are replaced.

Using I/O-based hints, pages from the buffer cache are inserted into the unstable tree earlier than other pages. This gives them more time to degenerate the unstable tree – an unwelcome side effect. To mitigate this effect, we map all pages that are inserted into the unstable tree through hints to be read-only, as described in § 3.4.

Note, that we map only a fraction of the unstable tree nodes to be read-only in our approach. To show that we are mapping the "right" pages read-only and leave the ones that do not degenerate the unstable tree read-write, we have run a benchmark where our hinting mechanism is active and *all* pages are unconditionally mapped read-only when they are inserted into the unstable tree. Furthermore, we have added a modified KSM version without hinting, that also maps all pages that are inserted into the unstable tree read-only. This effectively keeps the tree from degenerating altogether – all nodes are always reachable.

The resulting merge-performances are depicted in Figure 7.

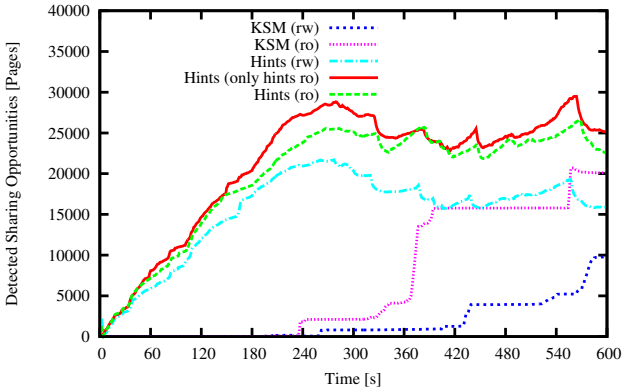

Figure 7. Merge performances depend heavily on the stability of the unstable tree and the temporal locality of unstable tree accesses.

In the long run, deduplication ratios depend on the quality of the unstable tree. If it degenerates, the effectiveness of KSM drops drastically. KSM's hash heuristic is performing poorly in our benchmark. We get much higher deduplication ratios when we map all items in the unstable tree read-only. In the hinting mechanism we always map hinted entries that are inserted into the unstable tree read-only. There is not much benefit in also trapping updates of pages that were added to the unstable tree by scanning. Not mapping pages read-only at all does damage the tree. However, as the accesses are close together in this benchmark, it does not hurt the performance as much as we would have anticipated after having seen Table 2. In a less fortunate access order, hinting without mapping unstable tree pages read-only would find much less sharing opportunities, most likely even less than the default KSM process. If we map the hinted pages read-only we keep an almost perfectly intact unstable tree and thereby avoid this negative effect.

Concluding Remarks Concerning the Unstable Tree We map hinted pages read-only solely to work around KSM's design decisions yet stay comparable to the original KSM performance. The bottom line is that red-black trees are a poor choice even when adding the checksum heuristic; the unstable tree degrades even without our extensions. We just happen to catch the pages that degrade with the highest probability to break the tree. Hash tables, as used in ESX [25], are a suitable choice to deal with unstable pages without the need for mapping pages read-only or pruning degenerated data. When using a hash table to store deduplication candidates, there is also no need to flush the contents after a scan-round.

One could argue that, compared to the tree-approach, there is a performance drawback in using a hash value of the entire page as the index. Note that we do not need to freshly calculate the index hash value but re-use the checksum calculated in the previous stage of the scanning algorithm.

4.2 Deduplication Effectiveness

The key point of this work is to enable memory scanning deduplication systems such as KSM to find short-lived sharing opportunities

quickly enough to exploit them and thereby further increase the memory density of virtualized environments. We have run our I/O-hinting extended mechanism, the original KSM mechanism, and KSM with a read-only unstable tree in different scan rate settings in the kernel-build benchmark. The results can be seen in Figure 8.

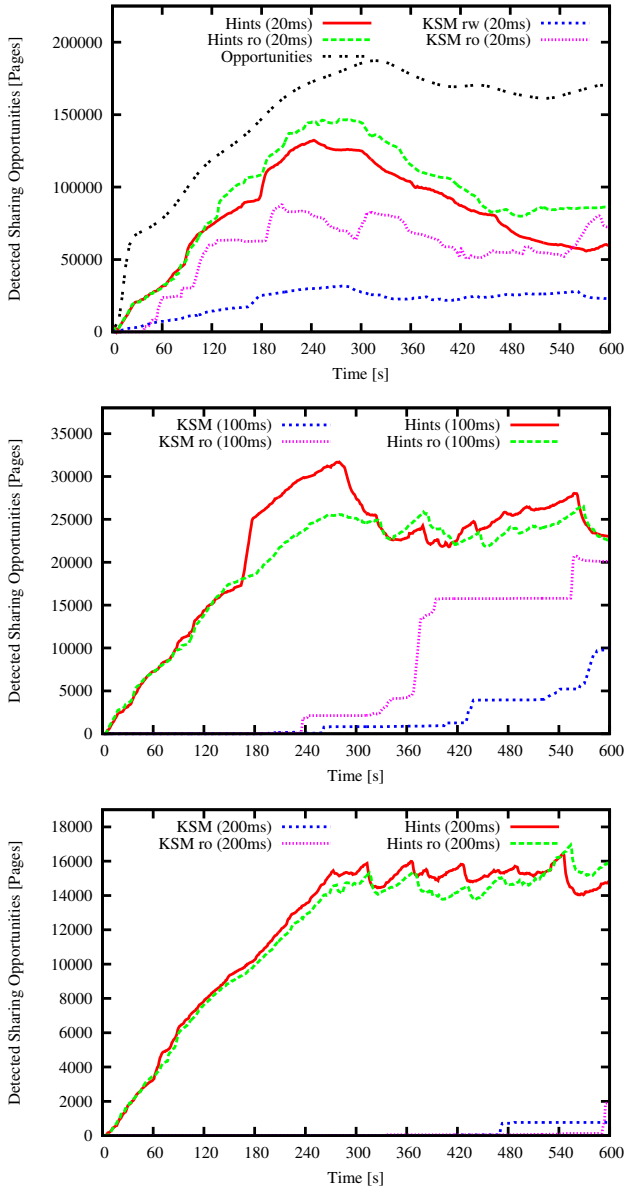

Figure 8. Kernel build merge performance at 100 pages per wake-up with varying wake-up times. Top to bottom: 20 ms, 100 ms, 200 ms. The first graph also shows the theoretical maximum of sharing opportunities that could be exploited.

Our extension always performs better than the original KSM and even compared to KSM with an unstable tree that is fully mapped read-only. As expected, the improvement is at its peak when given long time intervals between wake-ups. This is the time when efficiency is most important and this efficiency directly affects the deduplication effectiveness.

The merge-latencies are reduced significantly. We start merging pages from the start. Although we let KSM run for two scan cycles to warm up before we start the workload and measurements, KSM needs a long time before it starts deduplicating pages at all. It first needs to calculate the hash-values for the new pages in the workload and does not start deduplicating those pages until the next round.

4.3 Deduplication Efficiency

We have already demonstrated how effective the hinting mechanism is. What remains to be done is an evaluation of the efficiency. Our work would be of no use if we traded memory bandwidth and CPU cycles for the gained effectiveness; KSM can already do this within limits – just set it to scan very aggressively.

The runtime variation between our approach and the default KSM was below 1% in our experiments. We do not raise the effectiveness by doing more work, but by making smarter choices when and where to invest our duty cycles. This can be clearly seen when we compare the number of pages that we need to check until we find a sharing candidate. Figure 9 depicts the number of pages that the scanner needs to visit before it finds a sharing opportunity over the time it takes for the kernel to build. How many pages need to be visited to find a merge candidate depends highly on the state of the unstable tree. In the case of KSM without hinting, it also depends on where in the scan process KSM is. The further it is in the scan cycle the less pages it needs to visit until it finds a sharing partner for a page it has already indexed. Keep in mind that KSM drops its unstable tree after a full scan. This does not set the efficiency back all the way as all merged pages remain stored in the stable tree across scan cycles. When using our hints extension, the scan efficiency is not only much higher but also much less varying.

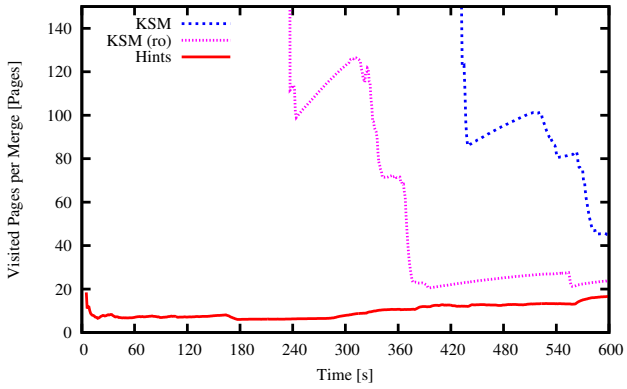

Figure 9. The number of pages that the different approaches need to visit before they find and merge a sharing opportunity.

When it comes to memory overhead, the only additional memory space we need is used by the hinting queue (configurable, a good choice is 1024 slots containing an 8-byte pointer each) and some locks to serialize access to this shared data structure. Most of our work is amortized by the fact that we do it in the place of an equally costly operation that would have happened in the regular scanning process. Also, we do not change the scan rate. A lookup in the stable or unstable tree costs the same whether it was triggered by a hint or by a periodic scan.

Additional CPU cycles are needed by our hinting mechanism for storing and retrieving hints, for mapping hinted pages read-only, and for mapping hinted pages read-write on a write fault. Storing and retrieving hints is very cheap ($O(1)$), yet these operations can currently not be parallelized, as they need locking at the stack's head index and length variable. Although remapping pages read-only and read-write is considered costly, our benchmarks ran roughly equally long.

The rate in which we map pages read-only due to insertion into the unstable tree and the rate in which we map pages read-write due

to a write operation on a hinted page that resides in the unstable tree correlate directly. After having inserted all pages that make up the guests buffer cache into the unstable tree, every future buffer cache replacement operation on a page is preceded by remapping that page read-write and succeeded by mapping that page read-only again. Note that we will always generate a hint for a page in the buffer cache – either from a read from the VDI into the buffer cache or from the flush operation which writes the buffer cache back to the VDI after a write operation in the guest.

The number of pages that we need to map read-write and read-only in a time-interval depends on the scan rate and the hint-rate – the amount of I/O-operations that are performed on the VDI. In our basic experiment (Table 1) the remap-rate is at roughly 260 pages per second. To put this number in perspective: Linux switches processes between 100 and 1000 times per second. Also keep in mind that we are almost exclusively mapping pages read-only that stem from the buffer cache.

5. Related Work

In this section, we give an overview of related work in the field of memory-deduplication and comment on how the related approaches differentiate from ours.

5.1 Traditional Page-Sharing Approaches

In non-virtualized systems, memory is shared between processes on two occasions: first, when the user explicitly requests shared memory through system calls and second, implicitly through copy-on-write (COW) semantics, when using system calls such as `mmap`, or `fork`. `Mmap` shares the memory content of files with the same inode. When a file is copied, a new inode is created which points to a copy of the same content. Because of the different inode, `mmap`ing this copy will lead to duplicate content in main memory even if the duplicated blocks are later merged via block-layer deduplication. Thus, using `mmap` to share files among different VMs is not possible, as VMs do generally not share the same file system, but run from separate virtual disk images (VDI), which are large files with their own file system inside.

When a process is duplicated via the `fork` system call in today's OSes, the processes' entire address space is shared using COW. Android's Cygote uses this property to share the Dalvik VM and the core libraries among all processes [21]. This "trick" has also been used to share whole guest operating systems [17, 24]. Virtualized environments that use COW semantics only share pages that already existed *before* the process or VM was `fork`ed. Our approach in contrast aims also to deduplicate those equal pages that are created at run time, *after* `fork`ing the VM.

5.2 Memory Scanning

The technique of periodically scanning main memory pages for equal content and then transparently merging those pages to share them in a COW manner was first introduced in VMware's ESX Server [25]. Linux also uses this technique under the name Kernel Samepage Merging (KSM) to increase the memory density of VMs [1]. Both implementations can only merge *anonymous* pages. Guests manage their (virtual) block layer themselves and the host does not know the semantics of the guest's main memory content. ESX is dedicated to running VMs and thus may use memory scanning on all processes' anonymous memory while KSM only scans pages that have been advised to be good sharing candidates and mergeable at all (e.g., not file backed but anonymous) through the `madvise` system call. Only being able to operate on anonymous pages is not a limiting factor for VMs, as the host regards the guests memory as anonymous memory due to the semantic gap.

The KSM and ESX content-based page sharing approaches differ mainly in the way they catalog scanned pages: ESX calculates a hash value for every page when scanning and stores these values in a hint table. When a match is found in the hint table, ESX first re-calculates the hash value of the previously inserted page to check whether the content has changed since the last calculation. If not, the pages are compared bit-by-bit to rule out a hash-collision. Then, equal pages are merged, and their hash value is inserted into another table, the shared table.

KSM also calculates hash values, but only to check whether a page has changed between scan-rounds. It does not use those hash values to infer equality between pages. All pages that have not changed between rounds are inserted into a tree (the full page, not the hash value); duplicates are found on insertion. More details on how KSM works can be found in Section 2.

The general trade-off that is involved when using memory scanners is CPU utilization and memory bandwidth versus the speed in which deduplication targets are identified. KSM and ESX both have a variable scan rate which is configured through setting sleep times and a number of pages that is scanned on every wake-up. Both KSM and ESX suggest scan rates that are fast enough to merge long-lived sharing opportunities with little overhead. However, the current implementations are not well suited to find short-lived sharing opportunities [6].

ESX scans pages in random order, while KSM scans linearly in rounds. Although the original ESX paper [25] states that it could be beneficial to define a heuristic for the scan order, neither KSM nor ESX propose a well suited policy to find sharing candidates more quickly.

Although prior work has been published on sharing similar pages [12] through storing compressed patches which are applied on access page faults, we focus on pages with equal content in this paper. However, I/O-based hints could as well provide good candidates for sub-page sharing.

Geiger [13] traces page faults, page table updates, copy-, and disk operations to infer information about page liveliness (allocation, eviction), swapping, page replacement, and the use of the unified buffer caches in the guest. The main focus of Geiger lies in making good resource decisions for single VMs, such as the estimation of the working set size for each VM or secondary caching. However, Geiger does not try to find sharing opportunities between VMs. Geiger is based on previous work on buffer cache placement [8, 27] that uses similar techniques.

5.3 Paravirtualization

An approach to find duplicate main memory pages that stem from background storage more quickly than linear scanning, is to explicitly track changes. (Cellular) Disco's transparent page sharing uses a deduplicating COW-disk to identify file-blocks that can be mapped to the same page in main memory due to equal content. It also hooks calls such as `bcopy` to keep track of shared content [5, 10].

The Xen [2] based Satori [20] seizes this suggestion and uses paravirtualized smart virtual disks to infer the sharing opportunities that stem from background storage.

Collaborative memory management (CMM) [22] uses paravirtualized Linux guests to share usage semantics of the guest's virtual memory system with the hypervisor. Its focus lies in determining the working set size of the guests, especially by telling the hypervisor which pages are unused in the guest and can thus be dropped. CMM was implemented only for the IBM system Z architecture.

XenFS [26] is a prototype for a file system that is shared between VMs and makes it possible to share caches and COW named page mappings across VMs. Two different approaches to shared page caches are Trancendent Memory [18, 19] and XHive [14]. Trancendent Memory provides a key-value store that can be used by guest VMs to cache I/O requests in the hypervisor. XHive practically

implements swapping to the hypervisor (i.e., move pages from the guest to the host). It gives pages which are used by multiple VMs a better chance to reside in memory, but outside of the VM's quota.

All techniques in this paragraph use paravirtualization techniques. They need to modify the guest to work. Our approach in turn works without such modifications and even works with non-VM processes.

6. Conclusion

When it comes to consolidating many virtual machines (VMs) on a single physical machine, the primary bottleneck is the main memory capacity. Previous work has shown that the memory footprint of VMs can be reduced significantly through merging equal pages. Identifying those pages can be achieved through scanning for equal contents and with the help of the guest. We have demonstrated that a very simple heuristic to identify good sharing candidates – as I/O target pages are good sharing candidates – can be implemented without the need of paravirtualization techniques yet deliver superior performance compared to linear scanning. We discuss various implementation challenges, such as the unsuitable, degenerating data structures in the KSM implementation, and describe how we resolve them in the implementation of our approach.

Extending KSM with I/O-hints increases both the effectiveness and the efficiency of memory scanning. KSM now only needs to visit about one seventh of the pages it needed to visit originally to find a sharing candidate. Many sharing opportunities that were not detected previously can now be exploited without introducing measurable overhead.

In the future, we will conduct a larger variety of benchmarks to research the performance characteristics of our approach in different scenarios. Also, we plan to extend Linux' virtual memory system to enable Linux to merge file-backed (named) pages with other file-backed or anonymous pages. This will make it possible to share the host's buffer cache with the guest's buffer caches.

References

[1] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using KSM. In *OLS '09: Proceedings of the Linux Symposium*, July 2009.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, New York, NY, USA, 2003. ACM.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX ATC*, ATEC '05, Berkeley, CA, USA, 2005. USENIX Association.

[4] J. Berthels. Exmap memory analysis tool. `http://www.berthels.co.uk/exmap/`, 2006.

[5] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15, November 1997.

[6] C.-R. Chang, J.-J. Wu, and P. Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, ISPA '11, Washington, DC, USA, 2011. IEEE Computer Society.

[7] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, Washington, DC, USA, 2001. IEEE Computer Society.

[8] Z. Chen, Y. Zhou, and K. Li. Eviction based cache placement for storage caches. In *Proceedings of the USENIX ATC*, Berkeley, CA, USA, 2003. USENIX Association.

[9] I. Eidus. How to use the kernel samepage merging feature, 2009. Documentation/vm/ksm.txt in Linux Kernel v3.0.

[10] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, New York, NY, USA, 1999. ACM.

[11] T. Groeninger, K. Miller, and F. Bellosa. Analyzing Shared Memory Opportunities in Different Workloads. Study Thesis, 2011. System Architecture Group, KIT, Germany.

[12] D. Gupta, S. Lee, M. Vrable, S. Savage, and et al. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM*, 53, October 2010.

[13] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, New York, NY, USA, 2006. ACM.

[14] H. Kim, H. Jo, and J. Lee. Xhive: Efficient cooperative caching for virtual machines. *IEEE Trans. Comput.*, 60, January 2011.

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

[16] R. Koller and R. Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage*, 6, September 2010.

[17] H. A. Lagar-Cavilla, J. A. Whitney, and et al. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, New York, NY, USA, 2009. ACM.

[18] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Paravirtualized paging. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, Berkeley, CA, USA, 2008. USENIX Association.

[19] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Transcendent Memory and Linux. In *OLS '09: Proceedings of the Linux Symposium*, July 2009.

[20] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: enlightened page sharing. In *Proceedings of the USENIX ATC*, Berkeley, CA, USA, 2009. USENIX Association.

[21] Patrick Brady. Anatomy & Physiology of an Android. In *Google I/O Developer Conference*, 2008.

[22] M. Scwidefsky, H. Franke, R. Mansell, H. Raj, D. Osisek, and J. Choi. Collaborative memory management in hosted linux environments. In *Proceedings of the Linux Symposium, Volume 2*, 2006.

[23] VMware, Inc. ESX Server 3.0.1 Resource Management Guide, 2011.

[24] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, New York, NY, USA, 2005. ACM.

[25] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36, December 2002.

[26] Williamson, Mark. Xen Wiki: XenFS. `http://wiki.xensource.com/xenwiki/XenFS`, 2007.

[27] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the USENIX ATC*, Berkeley, CA, USA, 2002. USENIX Association.