XLH: More effective memory deduplication scanners through cross-layer hints

Konrad Miller Fabian Franz
Marc Rittinghaus Marius Hillenbrand Frank Bellosa

Karlsruhe Institute of Technology (KIT)

Abstract

Limited main memory size is the primary bottleneck for consolidating virtual machines (VMs) on hosting servers. Memory deduplication scanners reduce the memory footprint of VMs by eliminating redundancy. Our approach extends main memory deduplication scanners through Cross Layer I/O-based Hints (XLH) to find and exploit sharing opportunities earlier without raising the deduplication overhead.

Prior work on memory scanners has shown great opportunity for memory deduplication. In our analyses, we have confirmed these results; however, we have found memory scanners to work well only for deduplicating fairly static memory pages. Current scanners need a considerable amount of time to detect new sharing opportunities (e.g., 5 min) and therefore do not exploit the full sharing potential. XLH's early detection of sharing opportunities saves more memory by deduplicating otherwise missed short-lived pages and by increasing the time long-lived duplicates remain shared.

Compared to I/O-agnostic scanners such as KSM, our benchmarks show that XLH can merge equal pages that stem from the virtual disk image earlier by minutes and is capable of saving up to four times as much memory; e.g., XLH saves 290 MiB vs. 75 MiB of main memory for two VMs with 512 MiB assigned memory each.

1 Introduction

In cloud computing, virtual machines (VMs) permit the flexible allocation and migration of services as well as the consolidation of systems onto fewer physical machines, while preserving strong service isolation. However, in that scenario the available main memory size limits the number of VMs that can be colocated on a single machine.

There may be plenty of redundant data between VMs (inter-vm sharing), e.g., if similar operating systems (OSes) or applications are used in different VM instances. Moreover, previous studies have shown that the memory footprint of VMs often contains a significant amount of pages with equal content within a single instance (self-sharing) [3]. In both cases, memory can be freed by collapsing redundant pages to a single page and sharing it in a copy-on-write fashion. However, such pages cannot be identified using traditional sharing mechanisms (see § 6.1) as the isolation of VMs leads to the so-called semantic gap [8]; that is lost semantic information between abstraction layers. The host, for example, does not know which ones of the guests' memory pages represent file contents.

Prior work has made deduplication of redundant pages possible and thereby lowered the memory footprint of guests. In the following, we use *host* interchangeably with virtual machine monitor (VMM), hypervisor, or host OS to describe the system layer underneath the guest OS.

Paravirtualization closes the semantic gap through establishing an appropriate interface between host and guest [6, 18] to communicate semantic information. This implies modifying both host and guest.

Such an interface has previously been used to help deduplicating *named* memory pages—memory pages backed by files: Satori [18] successfully merges named pages in guests employing sharing-aware virtual block devices in Xen [2]. Paravirtualization-based approaches have only been used selectively and rudimentarily to make sharing of *anonymous* memory (e.g., heap/stack memory) possible, through hooking calls such as `bcopy` [6].

Applying these modifications to all guests and keeping them compatible with the latest developments at the kernel and hypervisor level is at least a great burden. It might not even be possible at all to modify commercial or legacy guests due to license restrictions or the lack of source code. Moreover, the lack of semantic information that the host has about guest activities is actually one of the key features of virtualization: The host does not know nor needs to know the OS, file system, etc. inside the VM.

Memory scanners mitigate the semantic gap by scanning for duplicate content in guest pages [1, 25]. They index the contents of memory pages at a certain rate, regardless of the pages' usage semantics.

Scanners have their downside when it comes to efficiency. 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 compared to paravirtualization-based systems that merge pages synchronously when they are established.

Memory scanners trade computational overhead and memory bandwidth with deduplication success and latency. Although the scan rate (pages per time interval) is often variable and may be fine-tuned [10, 23], it is generally set to scan very slowly to keep the scanner's CPU and memory bus resource usage low. The default scan rate for Linux/KSM is 1000 pages per second, which results in a scan time of almost 5 minutes per 1 GiB of main memory.

XLH is our contribution that combines the key benefits of both previous approaches. We have observed that:

- All types of memory contents (named and anonymous) contribute to memory redundancy.
- Many shareable pages in the host's main memory originate from accesses to background storage: when multiple VMs create or use the same programs, shared libraries, configuration files, and data from their respective virtual disk images (VDIs).

The main contribution of this paper is to observe guest I/O in the host and to use it as a trigger for memory scanners in order to speed up the identification of new sharing opportunities. For this purpose, XLH generates page hints in the host's virtual file system (VFS) layer, whenever guests access their background store. XLH then indexes these hinted pages soon after their content has been established and thus moves them earlier into the merging stage. In consequence, XLH can find short-lived sharing opportunities and shares redundant data longer than regular, linear memory scanners without raising the overall scan rate.

We have implemented our approach in Linux' Kernel Samepage Merging (KSM) and evaluated its properties. Measurements of kernel build and web server scenarios show that XLH deduplicates equal pages that stem from the VDI earlier by minutes and is capable of merging between 2x and 5x as many sharing opportunities than the baseline system. For the kernel build benchmark, XLH performs constantly better than KSM even if the scan rate is set 5x lower. Our evaluation shows that XLH is able to reach its effectiveness with little to no additional CPU overhead or loss in I/O throughput compared with KSM.

We only modify the *host* in our approach—XLH would not benefit from and thus does not make use of paravirtualization. In fact, due to the generality of our approach, XLH also works for deduplicating native processes when no virtualization is involved. Note that XLH does not solely target disk accesses but issues hints for all I/O that goes through the VFS interface, including network file systems such as NFS. Overall, I/O-advised scanning makes more effective detection of sharing opportunities possible without the need to modify guests.

The remainder of this paper is structured as follows: We analyze semantic and temporal memory duplication properties in the following Section 2 to back up and motivate our approach. We then review Kernel Samepage Merging (KSM)—the memory scanning basis for our implementation—in Section 3 before we describe our approach and the implementation of our prototype thoroughly in Section 4. In Section 5, we present the results of our evaluation. We give an overview of related work on memory deduplication in Section 6. Finally, we conclude and depict future research directions in Section 7.

2 Analysis of Memory Duplication

Whether the use of deduplication techniques is effective or not depends mainly on the target workload. Using memory deduplication does improve a system's memory density if the memory footprint of the hosted applications is sufficiently similar. This is generally the case if the same OS, similar programs/libraries and/or data are used.

Duplication quantity An empirical study on memory sharing of VMs for server consolidation performed by Chang et al. found that the amount of redundant pages can be as low as 11% but also as high as 86% depending on the OS and workload [7]. Gupta et al. measured the amount of duplicated memory across three VMs and found that almost 50% of the allocated memory could be saved through memory deduplication [12]. We have performed a study ourselves and found 110 MiB of redundant memory in typical desktop workloads (LibreOffice, Firefox). We moreover measured 400 MiB (39%) of redundant data in one of our benchmarks (§ 5.2).

Duplication sources The sources of duplicated pages and their distribution vary greatly between workloads. Barker et al. measured the number of identical page frames in Ubuntu Linux 10.10 while running a typical set of desktop applications. In their study, over 50% of identical page frames stem from process heaps. They furthermore identified shared library based pages to be the second largest source of duplication (43%) [3]. In a study performed by Kloster et al., between 64% and 94%

of redundant data were located in page caches [14]. In our own aforementioned analysis of desktop workloads 70% of the duplicate pages were part of the page caches while 14% of the duplicates were not backed by files (anonymous). The last 15% were either free or reserved (e.g., driver pages). Although it seems to be favorable to focus on named pages, as many sharing opportunities can be found in page caches, a significant amount of duplicates may stem from anonymous memory regions. In consequence, for a deduplication system to be effective, it needs to exploit sharing opportunities from all sources.

Temporal characteristics of duplicates In our kernel build benchmarks, 80% of all encountered sharing opportunities lived between 30 seconds and 5 minutes. In this scenario, using brute-force scanning to detect short-lived sharing opportunities is not effective. Additionally, it wastes sharing potential by identifying longer-lived sharing opportunities late in the scan process. Figure 1 depicts the significance of the merge latency on how many pages are shared at any given point in time: The later memory is indexed by the scanner, the later a shared page can be established. Indexing sharing opportunities earlier adds to the sharing potential and to longer sharing time-frames.

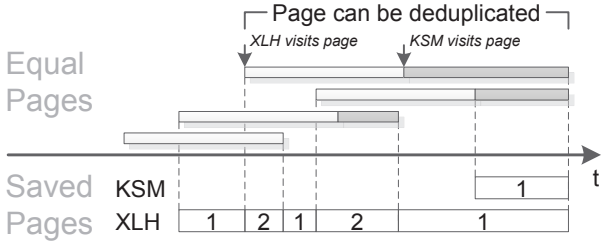

Figure 1: Memory scanners index pages after an expected value of half a scan cycle. XLH visits I/O pages immediately after they are established. If a duplicate is not found until a large proportion of the mean sharing time is over, the deduplication effectiveness is lowered significantly.

3 Memory Scanning with KSM

Our prototype is based on Kernel Samepage Merging (KSM) [1] which is a popular memory deduplication approach in the Linux kernel. KSM single-threadedly scans for and merges equal main memory pages. It is not bound to VMs but works on anonymous memory regions of any process. However, KSM only regards specifically advised pages (`madvise`) as mergeable. QEMU [4] invokes the appropriate call for the memory of each VM.

Page States Every advised page in the host is in one of three states: (1) frequently fluctuating, (2) sharing candidate yet potentially unstable, and (3) shared.

Data Structures KSM allocates a tree-node containing information such as a checksum and sequence number linked to every advised virtual page in the host. Pages that have changed between scan rounds (1) are not recorded or regarded in the scan process until their modification frequency decreases. The tree-nodes of all other pages are linked together into two red-black trees using their pages' full content as the key/index. The *unstable tree* (2) records pages that do not change frequently and are in consequence suitable sharing candidates. They are neither shared yet, nor protected from being written to—their content is thus not stable and may be modified after insertion. The *stable tree* (3), in contrast, stores pages that have already been merged and marked copy-on-write.

Scan Process KSM searches for pages that do not change frequently by gradually calculating a hash value for every page. If the calculated hash differs from the one recorded in the previous scan round, the record is updated but the page is not inserted into either of the trees (1). If the hash value has not changed between scan rounds, the associated page is inserted into the unstable tree (2), employing its content as key. If the unstable tree already contains a page with the same content, the pages are merged, marked read-only, and inserted into the stable tree (3). For any subsequently scanned page, KSM first checks if its content matches a page in the stable tree, in which case the pages are merged immediately.

When all advised pages have been scanned, the unstable tree is dropped and the process is repeated. Only the hash values and the stable tree remain.

4 I/O-Advised Deduplication with XLH

In the following paragraphs we discuss how XLH generates (§ 4.1), stores (§ 4.2), and processes (§ 4.3) deduplication hints interleaved with the periodic memory scan. KSM uses the full page content as the index into its trees. Writing to pages in the unstable tree is not prohibited; such writes, however, may break the reachability in the subtree of that page, thereby lowering the deduplication effectiveness. We present two solutions in § 4.4.

4.1 Generating Deduplication Hints

When a VM reads data from a virtual disk image (VDI), the virtual DMA controller in the host handles the request and reads the physical disk on behalf of the guest (Figure 2). Our assumption is that the target of that DMA transaction is a page in the guest's page cache and thus a good sharing candidate. We assume the same for writes: When a page cache page in the guest is flushed to disk—a new file is created or an old file is written— the host trans-

lates this into a write to the VDI. XLH detects these operations and generates deduplication hints for the source and target guest pages. In contrast to read operations on non-cached files, writes in the guest may be not immediately visible to the host as they can be delayed by the guest's page cache. Generally, the delay is much shorter than the time to the next visit of a traditional memory scanner, however.

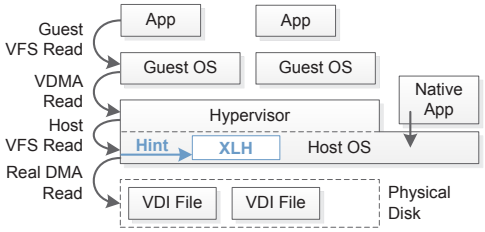

Figure 2: Host-VFS read and write operations are used to trigger hints for the main memory scanner.

As opposed to previous I/O-based approaches, we did not modify the guests in any way. Our mechanism even works for regular, non-VM processes, thereby enabling XLH to also deduplicate main memory efficiently in native environments (e.g., when using Zero-Install applications). Note that our approach is generic and may be applied to other environments as well, e.g., the Win32 file API layer. Moreover, the hint generation is fully decoupled from the deduplication process; there might as well be more than one hint source and other triggers for hints, e.g., from a page fault handler.

4.2 Storing Hints and Coping with Bursts

Hints need to be stored until they are asynchronously processed by the memory scanner. Memory scanners adhere to a certain scan rate, which is generally set to a low value (e.g., 1000 pages per second) to keep the overall impact of the scanner on the system performance within reasonable bounds. This way, however, memory scanners cannot always keep up with processing the number of incoming hints. The hint rate may be constantly higher than the scan rate, leading to an ever-growing buffer and suggesting the use of a pruning mechanism. Moreover, I/O is bursty; in consequence, I/O-based hints are also issued in bursts. Some million hints can be generated in a matter of seconds leading to a long backlog of hints and thus to outdated hints by the time the scanner gets to them. This effect calls for an aging mechanism.

We have at first stored our hints in an unbounded queue. When running our benchmarks, the system eventually fell behind to a state in which it could not find *any* sharing candidates through the hinting mechanism at all, as the hinted pages had already changed their content before they were processed.

A *bounded circular hint-stack* (Figure 3), however, proved to be an appropriate data structure to store hints with low overhead. The hint-stack keeps the history of the last unprocessed `stack_size` disk accesses.

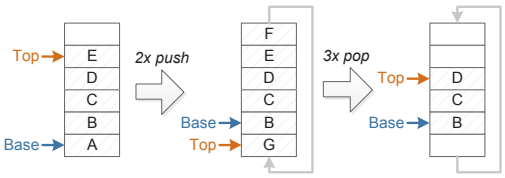

Figure 3: Storage of hints in the bounded circular stack.

Due to the nature of a bounded circular stack, XLH always processes the newest hints first while old hints are overwritten when the stack is full—an automatic pruning and aging mechanism which turned out to be fast and robust. Periodic maintenance is not required.

The stack size is configurable through procfs. In our benchmarks we found that XLH shares most pages if a full stack can be processed by the memory scanner within about 15 to 30 seconds. At KSM's default scan rate this results in a stack size of about 8k to 16k entries.

4.3 Processing Deduplication Hints

Our hint processing loop, depicted in Figure 4, runs interleaved with the full system scan spurts (wake-ups) that KSM already implements. XLH shares the global rate limit set for KSM and produces roughly the same CPU-load as an unmodified KSM with the same settings.

The interleaving ratio is configurable; `hint_runs` hint-processing spurts are interleaved with `scan_runs` scan spurts. A ratio of 0:1 corresponds to the original KSM implementation. Our default ratio is 1:1. Using this policy, XLH can guarantee that the linear scan, which can catch non-I/O sharing opportunities, does not starve due to a flood of hints.

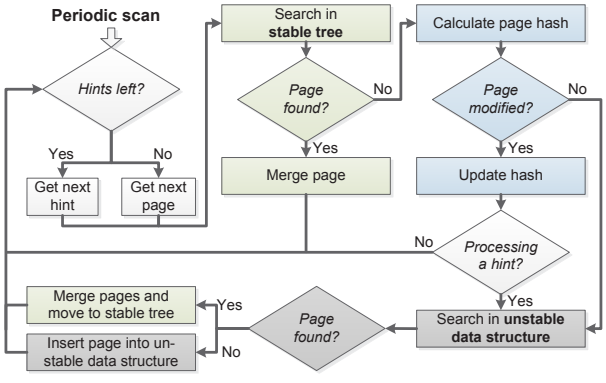

Figure 4: The high-level workflow of our new hint processing loop. When all hints have been processed before the scan rate is exceeded, XLH continues with the linear KSM scan to keep the rate constant.

When XLH is in a scan spurt, it runs the traditional linear scanning policy, only. In a hint-processing spurt, however, XLH processes hints as long as it has hints left and it has not exceeded its scan allowance. The remaining scan slots are then used for the linear scan.

Our mechanism first checks whether the hinted page's content is already in the stable tree. In this case, XLH remaps the page to the one in the stable tree and frees the hinted page. If the hinted page is not in the stable tree, XLH calculates the checksum of the page's content and checks the unstable data structure. If a sharing partner is found, XLH merges the pages and moves the resulting page into the stable tree. If XLH cannot find a sharing candidate it adds the page to the unstable data structure.

4.4 Degeneration of the Unstable Tree

The original KSM implementation has a heuristic that keeps frequently written pages from being inserted into the unstable tree. Only pages that keep the same hash value between consecutive scan rounds are considered (see Section 3). XLH however adds pages to the unstable data structure when processing hints that do not have a sharing partner at that point in time.

As pages are *not* marked read-only on insertion into the unstable tree but remain writable for the VM, the location in the tree is purely based on the content the page had at the time of its insertion. If pages in the unstable tree are subsequently modified, the tree may degenerate and entire branches may become unreachable (see Figure 5).

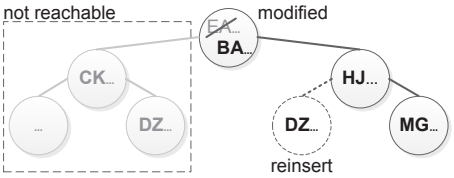

Figure 5: Nodes in the unstable tree may become unreachable due to modification of the contained pages. In this example, the first byte of the content of a page has changed from 'EA...' to 'BA...' leading to a second page with the content starting with 'DZ...' to be inserted a second time. The page duplication is not detected.

Although, the pages associated with virtual DMA operations are generally part of the guest's page cache, and are thus modified infrequently (see Section 2), the effect of the degeneration is not negligible and reduces the effectiveness of the merging stage. Even running Linux with an *unmodified* KSM reveals that almost 70% of the nodes cannot be reached in the unstable tree after a full scan, due to page modifications after insertion (kernel benchmark).

In KSM, 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. KSM's repair mechanism 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 leading to multiple visits of pages within a scan round. As XLH worsens the unstable tree's degeneration we provide two possible solutions in our implementation and compare their characteristics in § 5.4:

Read-Only Unstable Tree Nodes One way to counter the more likely degeneration of the unstable tree is to mark hinted pages that are inserted into the unstable tree as read-only. This way XLH can use write faults on hinted pages as a signal to remove these pages from the unstable tree and thereby prevent the tree from degenerating when hinted pages are modified. That 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 is only marked read-write and removed from the unstable tree.

Unstable Hash Table An alternative option is to replace the unstable tree with a hash table. When a page in the table is modified, the reachability of other pages in the hash table is not affected as there are no inner nodes that can be broken. However, we have to pay attention to the runtime effects of a hash table. Traditional hash tables work well for a fixed working set size.

5 Evaluation

We were particularly keen to see whether XLH can merge more pages with an overhead that is comparable to KSM, our baseline system. Consequently, we chose the amount of main memory that is saved when deduplicating different workloads with fixed computational and memory overhead settings as our prime metric in the evaluation. After describing our benchmark setup in § 5.1, we explore the deduplication effectiveness for several different workloads in § 5.2. As we wanted to get results that are relatable to prior publications, we have chosen two of the benchmarks that were used to evaluate Satori [18]: Compiling the Linux kernel and the Apache web server performance when serving static files to httperf [19]. We have also mixed both benchmarks. Additionally, we have measured how long it takes for the baseline system as well as XLH to deduplicate the almost static memory footprint when solely booting many VMs.

We have confirmed that the overhead stays in the area of the baseline system using three metrics: Time spent in the deduplication stage, total time the benchmarks require from start to completion, and the CPU usage (via

top) during the execution. To push our hint generation and storage implementation to its limit, we have complemented our kernel build and Apache benchmarks with the file system benchmark bonnie++ [9]. Details can be found in § 5.3.

Finally, we have explored the runtime impact of our two solutions to the degenerating unstable tree problem. In § 5.4, we show that both solutions lead to comparable deduplication performances.

5.1 Benchmark Setup

We integrated XLH in Linux 3.4 and use QEMU [4] with KVM—a popular virtualization environment—in our benchmarks. KSM already provides data structures, mechanisms and the linear scanning policy for memory deduplication. We extended the Linux kernel by only around 600 SLOC.

All benchmarks have been conducted on a PC with an Intel i7 quad-core processor, 24 GiB RAM, and an SSD. Ubuntu 11.04 served as the host and also as the guest OS. Guests were assigned one VCPU each.

Unless specifically stated otherwise, we use the parameters in Table 1 for our benchmarks. The mapping of the sleep-time between spurts and the number of slots in the hint buffer are listed in Table 2. Intuitively, one would need a larger hint buffer for longer wake-up intervals as more hints aggregate between runs. Recall that XLH uses the fixed size of the hint buffer for pruning outdated hints.

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
hash_table_size	256 K	# of unstable hash slots
RAM size	512 MiB	Size of virtual main memory

Table 1: Default settings in our various benchmarks.

sleep_time	20 ms	100 ms	200 ms
stack_size	40960	8192	4096
full scan time	44 s	220 s	440 s

Table 2: The mapping of sleep time and hint buffer slots in our benchmarks. The time of a full scan cycle for two VMs with 512 MiB each is also shown.

In our experiments, we first determine the maximum available sharing opportunities without merging pages to show how far from the optimum the different approaches are. That is achieved with a kernel module comparable to Exmap [5], which once per second dumps page table information and page content digests. Then we re-run the experiments with different configurations of KSM and XLH. Internal information and statistics such as the number of exploited sharing opportunities are directly dumped from the deduplication code through sysfs.

5.2 Deduplication Effectiveness

The goal of XLH is to increase the memory density of virtualized environments by identifying sharing opportunities more quickly. When equal pages are identified earlier, the time that those pages are shared is extended. Furthermore, new sharing opportunities can be detected and shared which were previously not exploited due to slow scan cycles.

The metric we use to compare XLH with the baseline system KSM is the merge effectiveness at equal scan rates and thus at equal load settings. We define the merge effectiveness as the number of merged pages at a certain point in time after starting the benchmark. A steeper rise in those graphs indicates that more pages are merged in a given time interval, which is a consequence of fewer pages being checked before merging a page. A higher level in those graphs indicates a greater amount of saved memory and thus a better approach in terms of effectiveness.

In the following benchmarks, we compare XLH in both implementation flavors, read-only tree (**XLH RO**) and hash table (**XLH HT**), which have been described in § 4.4, with the KSM scanner in its vanilla implementation (**KSM**) as well as with an improved KSM version that marks the unstable tree read-only to mitigate unstable tree degeneration (**KSM RO**).

Booting many VMs One of the main reasons to do memory deduplication in a virtualization scenario is to be able to quickly consolidate many VMs on a single physical machine. TravisCI [22] spawns a new VM for every compute job they run for customers. Jobs often run shorter than 5 minutes, however. XLH performs particularly well when it comes to booting VMs as most of the boot process consists of loading secondary storage contents (programs, libraries) into main memory.

We have booted 25 VMs in parallel, starting 10 seconds apart. When using *XLH* all VMs were fully booted after 530 seconds using approximately 5 GiB of physical memory. With KSM, the total boot time has been almost exactly the same. However, up to this point KSM had only merged 53% of the sharing opportunities that XLH had had merged.

Kernel Build We have compiled the Linux kernel in two VMs on the same host. Before performing the benchmark, we have fully booted the VMs and waited until the static sharing opportunities were shared. The resulting deduplication effectiveness for the kernel build at different scan rates is depicted in Figure 6.

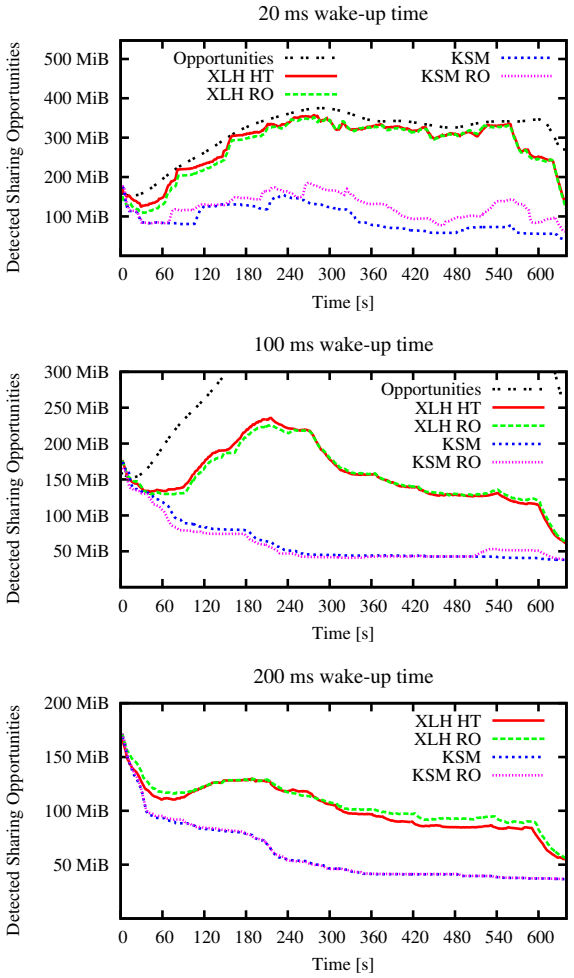

Figure 6: Kernel build merge performance with varying wake-up times. The first two graphs also show the maximum sharing opportunities.

In this benchmark, our extension always performs significantly better than KSM. For a short wake-up interval of 20 ms XLH shares almost all available sharing opportunities. Even with those very aggressive settings, where KSM occupies about 70% of a CPU core for its scan process, XLH can merge 2x to 5x as many pages as KSM. Both KSM and XLH are currently not multi-threaded and thus limited by the speed of a single CPU core. For longer wake-up intervals XLH also deduplicates 2x to 3x more effectively. In this benchmark, XLH even deduplicates more effectively than KSM if it scans 5 times slower (Figure 7).

The following numbers are taken from the 20 ms benchmark: XLH detects and shares almost 10 times as many new sharing opportunities in total (172000 vs. 17500). When considering the sharing opportunities that both systems detect, XLH detects those opportunities 243 seconds earlier (median) than KSM. The histogram of the time

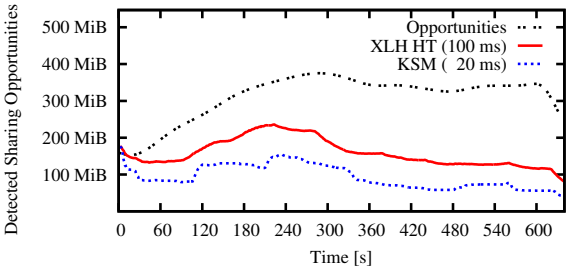

Figure 7: XLH performs constantly better than KSM in the kernel build even if the scan rate is 5x lower.

that pages remain shared while running the benchmark (Figure 8) shows that we can find many additional, short-lived sharing opportunities that KSM is not capable of detecting.

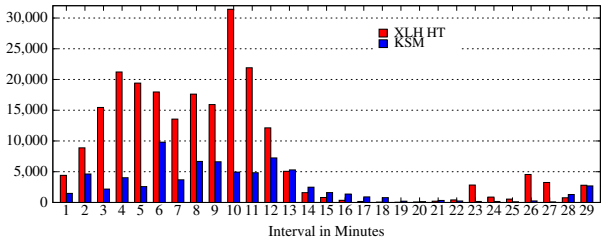

Figure 8: Histogram depicting the time. Sharing opportunities stay shared throughout the kernel-build benchmark.

Apache We have set up an Apache web server to serve random files in each one of two VMs. One httperf instance per server requests files in a predefined order.

The request order cannot be randomized directly in httperf. To make this benchmark less deterministic than the previous kernel build, we emulate random access patterns by statically shuffling the *file names* of the generated, served files in the servers. This way, when httperf accesses the same file name on both instances, different files will be returned; files with the same content will in consequence be returned at different times in the benchmark.

The total size of the served files exceeds the size of the page cache in each VM. Yet, parts of the guests' page caches overlap and therefore sharing opportunities exist even though file accesses are random.

We have configured httperf to establish 24000 connections per VM and to request 20 objects per second through each one of the connections from the Apache web servers. The merge performance of different scan rate configurations can be found in Figure 9.

When XLH cannot keep up with processing the stream of requests and constantly drops hints, our effectiveness is lowered significantly. XLH needs to process matching hints from both virtual machines in order to merge the

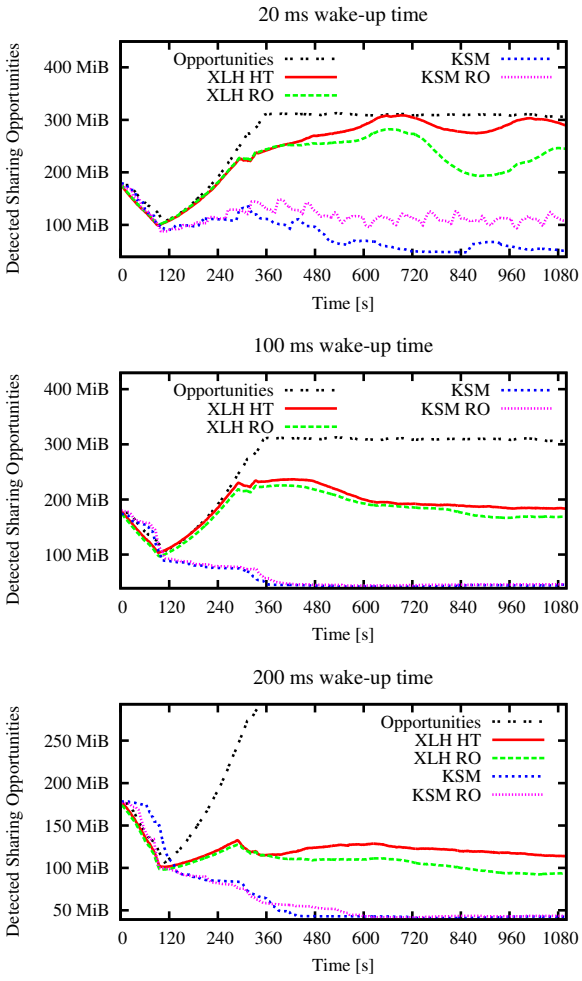

Figure 9: Apache/httperf merge performance.

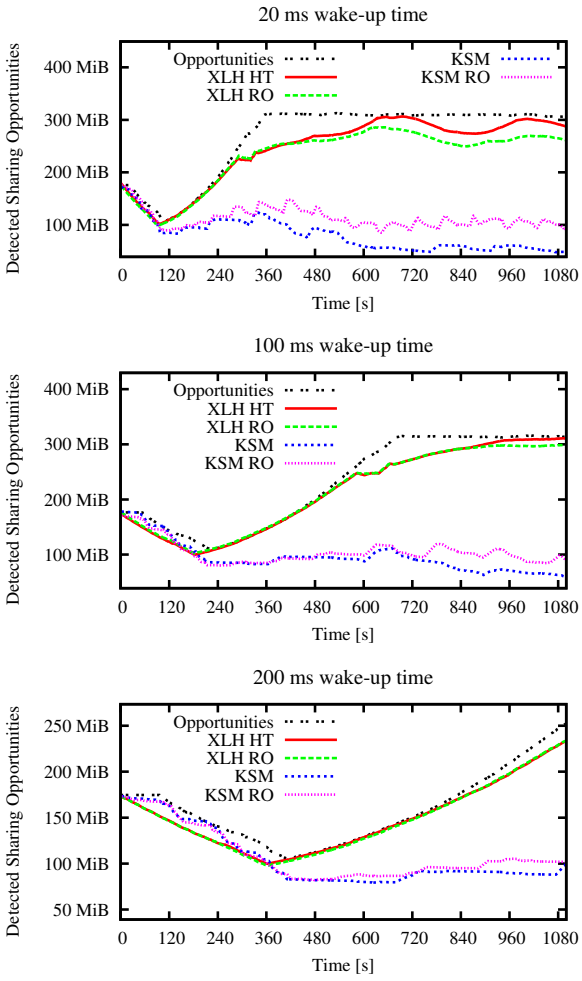

Figure 10: Apache/httperf merge performance with scaled request rates: No hints are dropped in these benchmarks.

two pages. Dropping one of the potential merge partners is enough to make the deduplication dependent on the linear scanner to find the other page in time. The effect of dropped hints can be easily seen when reducing the request rate. If XLH can process most of the hints, we get almost perfect results for all three scan rates (Figure 10). In this benchmark we have scaled the number of requests per second down to 1/4 of the original setting for 100 ms wakeup time and to 1/8 for 200 ms wakeup time.

Just like in the kernel benchmark, almost 10 times as many new sharing opportunities are detected and shared in the full-speed 20 ms benchmark (242233 vs. 22012). In this scenario, XLH detects those sharing opportunities 215 seconds earlier than KSM in the median. The histogram of the time that pages remain shared while running the benchmark is depicted in Figure 11.

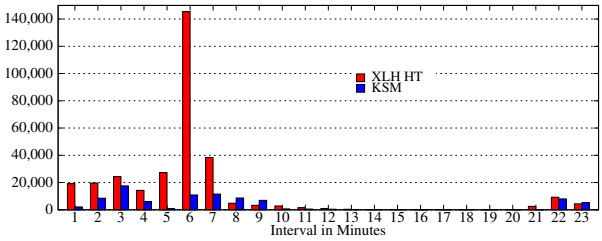

Figure 11: Histogram depicting the time, sharing opportunities stay shared throughout the apache benchmark.

Mixing Scenarios We have also run a benchmark in which both previously described benchmarks were executed at the same time. One VM was compiling the kernel while the other one was serving files with Apache.

In our benchmarks we can see a draw between vanilla KSM and XLH when mixing benchmarks (Figure 12).

That also reflects in the total number of sharing opportunities detected. XLH merges only 11% more sharing opportunities than KSM in the 20 ms benchmark (19483 vs. 17573). Theoretically however, KSM may be as much as twice as good as XLH with an interleaving ratio of 1:1 in case of completely useless hints. We have not encountered such results, though. In such a case, the user of the system may fine tune XLH through the interleaving ratio to mitigate this effect. Moreover, in cloud computing environments such as Amazon EC2, the provider may colocate VMs with similar memory footprints. Although the total number of shared pages is comparable, XLH merges pages 110 seconds earlier in the median. The histograms of XLH's and KSM's sharing time look similar in this benchmark (Figure 13).

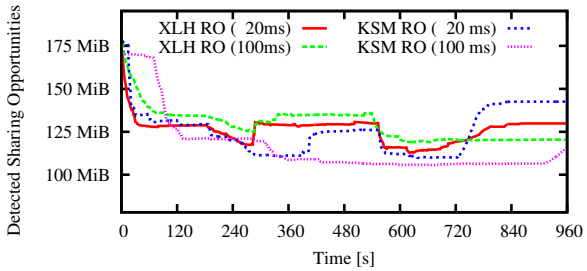

Figure 12: Merge performance with mixed workloads.

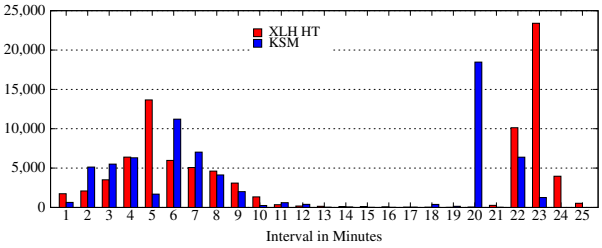

Figure 13: Histogram depicting the time, sharing opportunities stay shared throughout the mixed benchmark.

5.3 Deduplication Efficiency

XLH does not trade memory bandwidth and CPU cycles for the gained effectiveness; KSM can already do this within limits if set to scan aggressively.

Total Runtime Overhead The runtime variation between XLH and the default KSM was below 1% in our kernel build and Apache experiments. That is also true for the throughput that we have measured with httperf. To support the claim that XLH does not increase the system's load further than KSM with equal scan rates, we have measured the CPU consumption of the scanner thread when building the Linux kernel (Table 3).

Approach	20 ms	100 ms	200 ms
XLH HT	67.05%	33.61%	16.94%
XLH RO	66.19%	34.13%	16.17%
KSM	68.75%	27.47%	16.32%
KSM RO	68.92%	28.12%	17.52%
Average	67.72%	30.83%	16.74%

Table 3: CPU consumption: mean calculated from `top` measurements taken every second.

We do not raise the effectiveness by doing more work, but by making smarter choices for when and where to invest duty cycles. That can also be clearly seen when we compare the number of pages that XLH needs to check until it finds a sharing candidate. In the kernel build scenario XLH needs to visit 2-5 pages until it finds a sharing candidate while the linear scan needs to visit 18-260 pages. In the Apache scenario XLH visits between 4-8 pages to find a sharing opportunity while KSM visits 16-30 pages.

Memory-Scanning Overhead XLH needs additional memory for the hint buffer, which contains an 8-byte pointer for each slot and locks to serialize accesses. Most of XLH's work is amortized by the fact that it does it in the place of an equally costly operation of KSM. 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 and for marking hinted pages read-only. Storing and retrieving hints is very cheap ($O(1)$).

We have confirmed that neither the VFS-based hint trigger nor the hint buffer is a bottleneck by stress-testing this particular subsystem via the bonnie++ [9] file system benchmark. Figure 14 shows that the disk throughput of our enterprise class SSD does not vary significantly when choosing XLH over KSM.

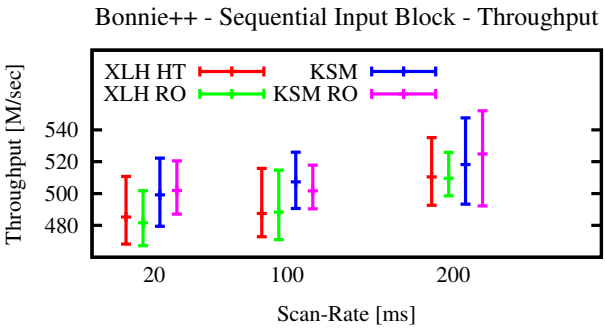

Figure 14: Disk throughput in 30 bonnie++ runs. The error-bars show the .05 and .95 percentiles.

5.4 The Unstable Tree's Stability

KSM only inserts pages into the unstable tree if their hash value has not changed since the last pass, whereas XLH inserts *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 each node in the tree. If a page cannot be found, it cannot be merged. Instead, it is inserted again—this time in another place (Figure 5). The percentage of reachable pages in the unstable tree for the kernel build benchmark after a full scan cycle is shown in Table 4.

Vanilla KSM	Hints (all RW)	Hints (hints RO)
33.6% - 53.0%	10.0%	98.9%

Table 4: Percentage of the unstable tree that is reachable at the end of a scan cycle.

We have found that pages that are part of the page cache are the ones that are most likely to change among all pages in the unstable tree. That happens when a page cache pages is written back, evicted, and replaced by another file's page in the guest.

Using I/O-based hints, pages from the page cache are inserted into the unstable tree earlier than other pages. That way, they have more time to degenerate the unstable tree, an unwelcome side effect. To mitigate this effect, we implemented two strategies:

Marking the Unstable Tree Read-Only One possible strategy to keep the unstable tree from degenerating is to mark pages that are inserted through a hint to be read-only. To show that XLH marks the "right" pages read-only and leaves 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 marked read-only when they are inserted into the unstable tree. Furthermore, we have added a modified KSM version without hinting that also marks 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 deduplication performances are depicted in Figure 15.

In the long run, deduplication ratios depend on the quality of the unstable tree. If it degenerates, the effectiveness of KSM drops drastically. We get much higher deduplication ratios when XLH marks all items in the unstable tree read-only. However, there is not much benefit in also trapping updates of pages that were added to the unstable tree by scanning. Not marking pages read-only at all does damage the tree.

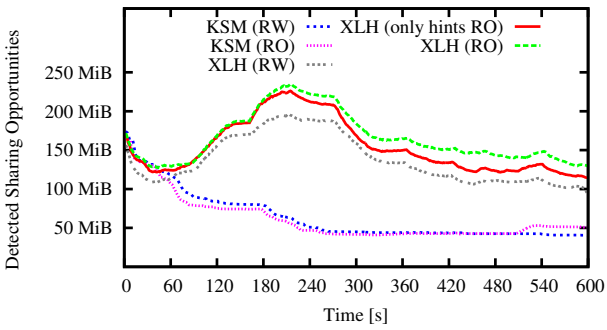

Figure 15: The merge performance depends heavily on the stability of the unstable tree and the temporal locality of unstable tree accesses.

Replacing Unstable Tree with a Hash Table Hash tables, as used in ESX [25], are a suitable choice to deal with unstable pages without the need for marking pages read-only or pruning degenerated data.

As KSM already calculates page hashes when scanning for duplicates, using a page hash as index into the hash table does not incur an extra cost. However, in contrast to the unstable tree, the hash table cannot easily be resized and thus does not scale well when the number of pages to monitor for redundant data changes frequently and to a large extent.

The performance depends highly on tuning the parameters to the workload at hand: If the hash table has many fewer entries than the number of pages in the system, then lookups become expensive due to chaining. That is also the case in workloads that generate many pages with colliding hash values. Yet, we observed in our benchmarks that the hash table approach performs as well or even better than using a read-only unstable tree when tuned to good values. We have used a hash table size of 256K entries in our various benchmarks throughout the paper.

5.5 Concluding Remarks

We have shown that XLH is able to quickly deduplicate the memory of newly booted VMs, which is especially beneficial when sandboxing short-running jobs or migrating many VMs at once. We have further demonstrated XLH's superior merging effectiveness compared to conventional linear memory scanners. XLH is capable of freeing up to 5x more memory than KSM by exploiting short-lived sharing opportunities, thereby finding 10x as many pages with equal contents. Moreover, XHL merges sharing opportunities 2-4 minutes earlier and thus leverages existing sharing potential. We have also evaluated XHL in an unfavorable scenario and found that it did not worsen the sharing performance compared to KSM. We also found XLH's influence on workload run-time and I/O throughput to be negligible.

6 Related Work

Virtual memory allows mapping different address space regions to the same region in physical memory. The underlying mapping mechanism can be used to establish communication and to allow coordination. Mapping can also reduce the memory footprint of processes and VMs by sharing memory regions of identical content. XLH is a novel approach to identify pages of same content.

6.1 Sharing of Cloned Content

In traditional 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 process forking or memory-mapped files. In the latter case, memory pages are shared that point to the same file control block (i.e., an inode). When a file is copied, a new control block is created, which points to a copy of the same content. Due to referencing a different control block, accessing this copy via memory-mapped files will lead to redundant data in main memory even if the duplicated disk blocks are later merged via block-layer deduplication. Thus, using memory-mapped files to deduplicate memory among different VMs is not possible, as VMs generally do not share the same file system, but run from separate virtual disk image (VDI) files. Our approach, in contrast, is capable of deduplicating equal memory pages originating from different files and even different memory sources.

When a process is duplicated via forking, the parent's and child's entire address spaces are shared using COW. If either process writes to a page afterwards, the sharing of the target page is broken up. Android's Cygote uses this property to share the Dalvik VM and the core libraries among all processes [20]. This initial cloning has also been used to share whole guest operating systems [15, 24]. The COW semantic only allows sharing pages that already existed *before* a process or VM has been forked. Our approach exploits the full sharing potential because it also deduplicates equal pages that are created at run time, *after* forking.

6.2 Paravirtualization

An established approach to find duplicate main memory pages that stem from background storage is the instrumentation of guest operating systems with the goal 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 [6, 11].

The Xen [2] based Satori [18] seizes this suggestion and uses paravirtualized smart virtual disks to infer the sharing opportunities that stem from background storage.

Collaborative memory management (CMM) [21] uses paravirtualized Linux guests to share usage semantics of the guests' virtual memory system with the hypervisor. Its focus lies on determining the working set size of the guests, especially by telling the hypervisor which guest pages are unused and can thus be dropped. CMM was implemented for the IBM System zSeries.

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 Transcendent Memory [16, 17] and XHive [13]. Transcendent Memory provides a key-value store that can be used by guests 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 that 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.

6.3 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]. ESX is dedicated to running VMs and thus may use memory scanning on all memory pages while KSM only scans pages that have been advised to be good sharing candidates through the `madvise` system call.

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 (see Section 3).

The general trade-off that is involved when using memory scanners is CPU utilization and memory bandwidth versus the time 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 are 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 [7].

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. XLH is such a suggestion.

Pages with similar content can be shared to a great extend through storing compressed patches which are applied on access page faults. Such approaches like Difference Engine [12] could be combined with XLH to identify good candidates for sub-page sharing.

7 Conclusion

When it comes to consolidating many virtual machines on a single physical machine, the primary bottleneck is the main memory capacity. Previous work has shown that the memory footprint of virtual machines can be reduced significantly by merging equal pages. Identifying those pages can be achieved through scanning for equal contents in the host.

We have demonstrated that memory deduplication scanners can be improved significantly when informing the scanner of recently modified memory pages. XLH implements this idea by telling KSM about I/O operations. KSM then processes these pages preferably to deliver superior performance compared to linear scanning.

We have discussed various challenges, such as I/O bursts and degenerating data structures in KSM, and described design alternatives. Our evaluation shows that I/O-based hints can increase the effectiveness of memory scanners significantly without raising the overhead imposed by the scanner. XLH finds more sharing opportunities than KSM and detects them earlier by minutes. Thereby XLH exploits sharing opportunities within and across virtual machines that were not detectable by linear scanners before.

We believe that XLH is already beneficial for a variety of use cases as it is. Therefore, we intend to release our Linux kernel extension soon. We plan to closely analyze memory duplication properties of NUMA architectures to identify good deduplication policies for such systems in the future.

References

[1] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using KSM. In *Linux Symposium 2009*.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., ET AL. Xen and the art of virtualization. SOSP 2003.

[3] BARKER, S., ET AL. An empirical study of memory sharing in virtual machines. In *USENIX ATC 2012*.

[4] BELLARD, F. Qemu, a fast and portable dynamic translator. ATEC 2005.

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

[6] BUGNION, E., DEVINE, S., GOVIL, K., AND ROSENBLUM, M. Disco: running commodity operating systems on scalable multiprocessors. *Transactions on Computer Systems 1997*.

[7] CHANG, C.-R., ET AL. An empirical study on memory sharing of virtual machines for server consolidation. ISPA 2011.

[8] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real. In *HotOS 2001*.

[9] COKER, R. The bonnie++ benchmark. `http://www.coker.com.au/bonnie++/`, 1999.

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

[11] GOVIL, K., TEODOSIU, D., HUANG, Y., AND ROSENBLUM, M. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. SOSP 1999.

[12] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., ET AL. Difference engine: harnessing memory redundancy in virtual machines. *Communications of the ACM 2010 Volume 53*.

[13] KIM, H., JO, H., AND LEE, J. Xhive: Efficient cooperative caching for virtual machines. *Trans. on Computer Science 2011*.

[14] KLOSTER, J. F., KRISTENSEN, J., AND MEJLHOLM, A. Determining the use of Interdomain Shareable Pages using Kernel Introspection. Tech. rep., Aalborg University, 2007.

[15] LAGAR-CAVILLA, H. A., ET AL. Snowflock: rapid virtual machine cloning for cloud computing. EuroSys 2009.

[16] MAGENHEIMER, D., MASON, C., ET AL. Transcendent Memory and Linux. In *Linux Symposium 2009*.

[17] MAGENHEIMER, D., MASON, C., MCCRACKEN, D., AND HACKEL, K. Paravirtualized paging. WIOV 2008.

[18] MIŁÓS, G., MURRAY, D. G., HAND, S., AND FETTERMAN, M. A. Satori: Enlightened page sharing. In *USENIX ATC 2009*.

[19] MOSBERGER, D., AND JIN, T. httperf - a tool for measuring web server performance. *SIGMETRICS Perf. Eval. Review 1998*.

[20] PATRICK BRADY. Anatomy & Physiology of an Android. In *Google I/O Developer Conference 2008*.

[21] SCWIDEFSKY, M., ET AL. Collaborative memory management in hosted linux environments. In *Linux Symposium 2006*.

[22] TRAVISCI COMMUNITY. TravisCI: continuous integration service. `https://travis-ci.org/`, 2012.

[23] VMWARE, INC. ESX Server 3.0.1 Resource Management Guide, 2011.

[24] VRABLE, M., MA, J., CHEN, J., ET AL. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. SOSP 2005.

[25] WALDSPURGER, C. A. Memory resource management in VMware ESX server. *SIGOPS Operating System Review 2002*.

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