VSWAPPER: A Memory Swapper for Virtualized Environments

Nadav Amit Dan Tsafrir Assaf Schuster
Technion – Israel Institute of Technology
{namit,dan,assaf}@cs.technion.ac.il

Abstract

The number of guest virtual machines that can be consolidated on one physical host is typically limited by the memory size, motivating memory overcommitment. Guests are given a choice to either install a "balloon" driver to coordinate the overcommitment activity, or to experience degraded performance due to uncooperative swapping. Ballooning, however, is not a complete solution, as hosts must still fall back on uncooperative swapping in various circumstances. Additionally, ballooning takes time to accommodate change, and so guests might experience degraded performance under changing conditions.

Our goal is to improve the performance of hosts when they fall back on uncooperative swapping and/or operate under changing load conditions. We carefully isolate and characterize the causes for the associated poor performance, which include various types of superfluous swap operations, decayed swap file sequentiality, and ineffective prefetch decisions upon page faults. We address these problems by implementing VSWAPPER, a guest-agnostic memory swapper for virtual environments that allows efficient, uncooperative overcommitment. With inactive ballooning, VSWAPPER yields up to an order of magnitude performance improvement. Combined with ballooning, VSWAPPER can achieve up to double the performance under changing load conditions.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management—Virtual memory, Main memory, Allocation/deallocation strategies, Storage hierarchies

General Terms Design, Experimentation, Measurement, Performance

Keywords Virtualization; memory overcommitment; memory balloon; swapping; paging; page faults; emulation; virtual machines; hypervisor

ASPLOS '14, March 1–4, 2014, Salt Lake City, UT, USA.

ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/2541940.2541969

1. Introduction

The main enabling technology for cloud computing is machine virtualization, which abstracts the rigid physical infrastructure and turns it into soft components easily managed and used. Clouds and virtualization are driven by strong economic incentives, notably the ability to consolidate multiple guest servers on one physical host. The number of guests that one host can support is typically limited by the physical memory size [6, 25, 27, 87]. So hosts overcommit their memory to increase their capacity.

Memory of guest virtual machines is commonly overcommitment via a special "balloon" driver installed in the guest [84]. Balloons allocate pinned memory pages at the host's request, thereby ensuring that guests will not use them; the pages can then be used by the host for some other purpose. When a balloon is "inflated," it prompts the guest operating system to reclaim memory on its own, which often results in the guest swapping out some of its less frequently used pages to disk.

Ballooning is a common-case optimization for memory reclamation and overcommitment, but, inherently, it is not a complete solution [28, 68, 84, 88]. Hosts cannot rely on guest cooperation, because: (1) clients may have disabled or opted not to install the balloon [10, 53, 74]; (2) clients may have failed to install the balloon due to technical difficulties [77, 78, 79, 80, 81, 82, 83]; (3) balloons could reach their upper bound, set by the hypervisor (and optionally adjusted by clients) to enhance stability and to accommodate various guest limitations [14, 19, 59, 69, 75, 84]; (4) balloons might be unable to reclaim memory fast enough to accommodate the demand that the host must satisfy, notably since guest memory swapping involves slow disk activity [31, 46, 84]; and (5) balloons could be temporarily unavailable due to inner guest activity such as booting [84] or running high priority processes that starve guest kernel services. In all these cases, the host must resort to uncooperative swapping, which is notorious for its poor performance (and which has motivated ballooning in the first place).

While operational, ballooning is a highly effective optimization. But estimating the memory working set size of guests is a hard problem, especially under changing conditions [27, 35, 45], and the transfer of memory pages between guests is slow when the memory is overcommitted [17, 31,

36, 46]. Thus, upon change, it takes time for the balloon manager to adjust the balloon sizes and to achieve good results. Hosts might therefore rely on uncooperative swapping during this period, and so guests might experience degraded performance until the balloon sizes stabilize.

We note in passing that the use of ballooning constitutes a tradeoff that embodies both a benefit and a price. The benefit is the improved performance achieved through curbing the uncooperative swapping activity. Conversely, the price for clients is that they need to modify their guest operating systems by installing host-specific software, which has various undesirable consequences such as burdening the clients, being nonportable across different hypervisors, and entailing a small risk of causing undesirable interactions between new and existing software [39]. The price for vendors is that they need to put in the effort to support different drivers for every guest operating system kernel and version. (We speculate that, due to this effort, for example, there is no balloon driver available for OS X under KVM and VirtualBox, and the latter supports ballooning for only 64-bit guests [16].) Therefore, arguably, reducing the overheads of uncooperative swapping could sway the decision of whether to employ ballooning or not.

Our goal in this paper is twofold. To provide a superior alternative to baseline uncooperative host swapping, to be used by hosts as a performant fall back for when balloons cannot be used. And to enhance guests' performance while ballooning is utilized under changing load conditions. We motivate this goal in detail in Section 2.

We investigate why uncooperative swapping degrades performance in practice and find that it is largely because of: (1) "silent swap writes" that copy unchanged blocks of data from the guest disk image to the host swap area; (2) "stale swap reads" triggered when guests perform explicit disk reads whose destination buffers are pages swapped out by the host; (3) "false swap reads" triggered when guests overwrite whole pages previously swapped out by the host while disregarding their old content (e.g., when copying-on-write); (4) "decayed swap sequentiality" that causes unchanged guest file blocks to gradually lose their contiguity while being kept in the host swap area and thereby hindering swap prefetching; and (5) "false page anonymity" that occurs when mislabeling guest pages backed by files as anonymous and thereby confusing the page reclamation algorithm. We characterize and exemplify these problems in Section 3.

To address the problems, we design VSWAPPER, a guest-agnostic memory swapper to be used by hypervisors. VSWAPPER is implemented as a KVM extension and is comprised of two components. The first is the Swap Mapper, which monitors the disk I/O performed by a guest while maintaining a mapping between its unmodified memory pages and their corresponding origin disk blocks. When such mapped memory pages are reclaimed, they need not be written to the host swap file; instead the Mapper records their location in the guest's virtual disk for future reference and discards them, thereby eliminating the root cause of silent writes, stale reads, decayed sequentiality, and false page anonymity. The second component is the False Reads Preventer, which eliminates false reads by emulating faulting write instructions directed at swapped out pages. Instead of immediately faulting-in the latter, the Preventer saves the written data in a memory buffer for a short while, in the hope that the entire buffer would fill up soon, obviating the need to read. We describe VSWAPPER in detail in Section 4.

We evaluate VSWAPPER in Section 5 and find that when memory is tight, VSWAPPER is typically much better than baseline swapping and is oftentimes competitive with ballooning. At its worst, VSWAPPER is respectively 1.035x and 2.1x slower than baseline and ballooning. At its best, VSWAPPER is respectively 10x and 2x faster than baseline and ballooning, under changing load conditions. In all cases, combining VSWAPPER and ballooning yields performance comparable to the optimum.

We discuss the related work in Section 6, outline possible future work in Section 7, and conclude in Section 8.

2. Motivation

2.1 The Benefit of Ballooning

Current architectural support for machine virtualization allows the host operating system (OS) to manage the memory of its guest virtual machines (VMs) as if they were processes, and it additionally allows the guest OSes to do their own memory management for their internal processes, without host involvement.

Hardware provides this capability by supporting a two-level address translation mechanism (Figure 1). The upper level is controlled by the guest and is comprised of page tables translating "guest virtual addresses" (GVAs) to "guest physical addresses" (GPAs). The lower level is controlled by the host and is comprised of tables translating guest physical addresses to "host physical addresses" (HPAs). A guest physical address is of course not real in any sense. The host can (1) map it to some real memory page or (2) mark it as non-present, which ensures the host will get a page fault if/when the guest attempts to access the non-present page. Consequently, when memory is tight, the host can temporarily store page content on disk and read it back into memory only when handling the corresponding page faults [62]. We denote the latter activity as *uncooperative swapping*, because the host can conduct it without guest awareness or participation .

The problem with uncooperative swapping is that it might lead to substantially degraded performance due to unintended interactions with the memory management subsystem of the guest OS. A canonical example used to highlight the problematic nature of uncooperative swapping is that of *double paging* [22, 24, 84], whereby the guest kernel attempts to reclaim a page that has already been swapped out

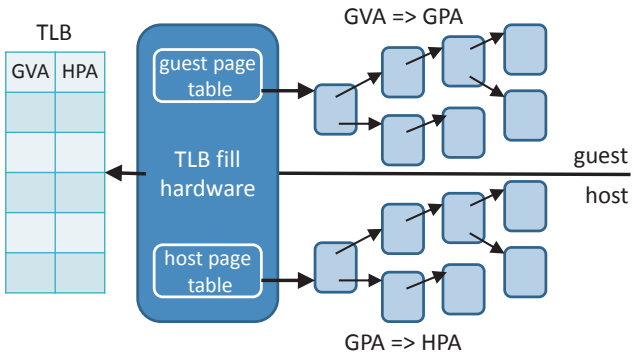

Figure 1. *The Translation Lookaside Buffer (TLB) translates guest virtual addresses (GVAs) to host physical addresses (HPAs), disregarding guest physical addresses (GPAs). Upon a miss, the TLB fill hardware adds the missing entry by walking the guest and host page tables to translate GVAs to GPAs and GPAs to HPAs, respectively. The hardware delivers a page fault to the host if it encounters a non-present GPA⇒HPA, allowing the host to fault-in the missing guest page, on demand. (Figure reproduced from [18].)*

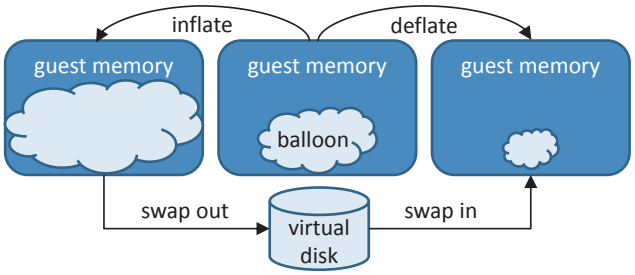

Figure 2. *Inflating the balloon increases memory pressure and prompts the guest to reclaim memory, typically by swapping out some of its pages to its virtual disk. Deflating relieves the memory pressure. (Figure reproduced from [84].)*

by the host, a fact unknown to the guest since it is uncooperative. When such an event occurs, it causes the page contents to be faulted-in from the host swap area, only to be immediately written to the guest swap area, generating wasteful I/O activity that host/guest cooperation would have obviated.

To circumvent difficulties of this sort, Waldspurger proposed to delegate to the guest the decision of which pages to reclaim, by utilizing *memory ballooning* [84]. A memory balloon is a paravirtual pseudo-driver installed in the guest. The balloon communicates with, and gets instructions from, the host through a private channel. It is capable of performing two operations: *inflating* and *deflating*, that is, allocating and freeing pages pinned to the guest memory. Inflating increases memory demand, thereby prompting the guest to run its page reclamation procedure and swap memory on its own (Figure 2). The pinned pages can then be used by the host for other purposes. Ballooning is the prevailing mechanism for managing memory of guest virtual machines.

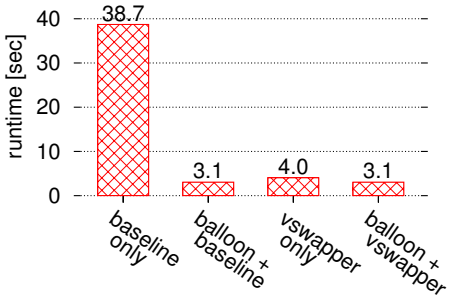

Figure 3. *Time it takes a guest to sequentially read a 200MB file, believing it has 512MB of physical memory whereas in fact it only has 100MB. (This setup is analyzed in detail later on.) The results shown are the best we have observed in favor of ballooning.*

2.2 Ballooning is Not a Complete Solution

Memory ballooning typically provides substantial performance improvements over the baseline uncooperative swapping. Ballooning likewise often outperforms the VSWAPPER system that we propose in this paper. An extreme example is given in Figure 3. The baseline is 12.5x slower than ballooning. And while VSWAPPER improves the baseline by 9.7x, it is still 1.3x slower than the ballooning configurations. One might therefore perceive swapping as irrelevant in virtualized setups that employ balloons. But that would be misguided, because ballooning and swapping are complementary activities.

Since its inception, ballooning has been positioned as a common-case optimization for memory reclamation, *not* as a complete solution [84]. Ballooning cannot be a complete solution, because, inherently, hypervisors cannot exclusively rely on any mechanism that requires guest cooperation, which cannot be ensured in any way. Thus, for correctness, host-level swapping must be available to forcibly reclaim guest memory when necessary. Indeed, when introducing ballooning, Waldspurger noted that "[d]espite its advantages, ballooning does have limitations. The balloon driver may be uninstalled, disabled explicitly, unavailable while a guest OS is booting, or temporarily unable to reclaim memory quickly enough to satisfy current system demands. Also, upper bounds on reasonable balloon sizes may be imposed by various guest OS limitations" [84]. The balloon size is limited, for example, to 65% of the guest memory in the case of VMware ESX [14, 85].

Guests might also lack a balloon due to installation and configuration problems [77, 78, 79, 80, 81, 82, 83], as installing hypervisor tools and making them work appropriately is not always easy. For example, Googling the quoted string "problem with vmware tools" returns 118,000 hits, describing many related difficulties that users experience. Balloon configuration becomes more complex if/when clients need to experiment with their software so as to configure memory reservations for their VMs [19, 59, 69, 75].

Virtualization professionals attest to repeatedly encountering clients who disable ballooning or do not install hypervisor tools for misguided reasons. Brambley reports that "[i]t happens more frequently than I would ever imagine, but from time to time I find clients [that] have not installed the VMware tools in their virtual machine [...] Some times the tools install is overlooked or forgotten, but every once in a while I am told something like: Does Linux need VMware tools? or What do the VMware tools do for me anyways?" [10]. Ozar reports that "There's plenty of bad advice out on the web saying things like: just disable the balloon driver" [53]. van Zanten concludes that the "Misconceptions on memory overcommit [amongst clients include believing that] overcommit is always a performance hit; real world workloads don't benefit; the gain by overcommitment is negligible; [and] overcommitment is dangerous" [74].

Regardless of the reason, balloons are sometimes unavailable or unusable. In such cases, the hypervisor falls back on uncooperative swapping for memory reclamation and overcommitment. We submit that it is far more preferable to fall back on VSWAPPER than on baseline swapping.

2.3 Ballooning Takes Time

So far, we have considered VSWAPPER as a more performant alternative to baseline swapping, to only be used as a fallback for when a balloon is not available or cannot be utilized due to, e.g., reaching its size limit. We have noted that VSWAPPER yields better performance than the baseline, but we have seen that this performance is still inferior relative to when ballooning is employed (Figure 3). Ballooning, however, is superior to VSWAPPER under steady-state conditions only. Steady-state occurs when (1) the balloon manager has had enough time to reasonably approximate the memory needs of the VMs and to inflate/deflate their balloons accordingly, and (2) the VMs have had enough time to react to decisions of the balloon manager by swapping data in or out as depicted in Figure 2.

Alas, the process of transferring memory pages from one VM to another is slow [31], and estimating the size of guests' working sets is hard, especially under changing conditions [27, 35, 45]. Ballooning performance is hence suboptimal under changing load conditions, during which the balloon manager is approximating and adjusting the balloon sizes and prompting the VMs to engage in swapping activity. Ballooning is consequently recognized as "useful for shaping memory over time, but inadequately responsive enough to ensure that, for example, the rapidly growing working set of one or more VMs can be instantly satisfied" [17]. Kim et al. observe that "ballooning is useful to effectively reclaim idle memory, but there may be latency, especially when inflating a large balloon; more importantly, when an idle domain that donates its memory becomes active, reclaimed memory must be reallocated to it via balloon deflating [and] this process could be inefficient when an idle domain has a varying working set, since prediction of the active working set size is difficult" [36]. Likewise, Magenheimer et al. observe that "if the light load is transient and the memory requirements of the workload on the VM suddenly exceed the reduced RAM available, ballooning is insufficiently responsive to instantaneously increase RAM to the needed level" [46].

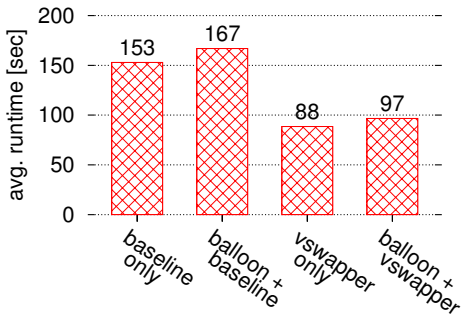

Figure 4. *Average completion time of ten guests running map-reduce workloads in a dynamic setup that starts them 10 seconds apart. (This setup is described in detail later on.)* VSWAPPER *configuration are up to twice as fast as baseline ballooning.*

VSWAPPER proves to be a highly effective optimization that can greatly enhance the performance under dynamic, changing memory load conditions. Its effectiveness is exemplified in Figure 4, which shows the average completion time of ten VMs running map-reduce workloads that are started 10 seconds apart. (The exact details of this experiment are provided in Section 5.2.) In this dynamic scenario, non-VSWAPPER ballooning worsens performance over baseline swapping by nearly 10%, and it yields an average runtime that is up to 2x slower than the VSWAPPER configurations. Ballooning is about 10% worse in the VSWAPPER configurations as well. It is counterproductive in this setup, because the balloon sizes are inadequate, and there is not enough time for the balloon manager to adjust them.

We thus conclude that not only is VSWAPPER an attractive fallback alternative for when ballooning is nonoperational, it is also an effective optimization on top of ballooning that significantly enhances the performance under dynamic conditions.

2.4 The Case for Unmodified Guests

Earlier, we provided evidence that clients sometimes have trouble installing and correctly configuring hypervisor tools, and that there are those who refrain from installing the tools because they wrongfully believe the tools degrade or do not affect the performance. Arguably, such problems would become irrelevant if hypervisors were implemented in a way that provides fully-virtualized (unmodified) guests with performance comparable to that of modified guests. We do not argue that such a goal is attainable, but VSWAPPER takes a step in this direction by improving the performance of unmodified guests and being agnostic to the specific kernel/version that the guest is running.

A guest OS is *paravirtual* if it is modified in a manner that

makes it aware that it is being virtualized, e.g., by installing hypervisor tools. Paravirtualization has well-known merits, but also well-known drawbacks, notably the effort to continuously provide per-OS support for different kernels and versions. In particular, it is the responsibility of the hypervisor vendor to make sure that a balloon driver is available for every guest OS. Thus, from the vendor's perspective, it could be easier to maintain only one mechanism (the likes of VSWAPPER), as it works the same for all OSes.

Avoiding paravirtualization could similarly be advantageous for clients in terms of portability. Note that the balloon drivers of KVM, vSphere, XenServer, and Hyper-V, for example, are incompatible, such that the per-guest driver of one will not work with another. In the era of IaaS clouds, it is in the interest of clients to be able to move their VMs from one cloud provider to another without much difficulty, on the basis of the technical and economical merits of the cloud systems, optimally in a transparent manner [11, 44, 48]. Having paravirtualization interfaces negates this interest, as they are hypervisor specific. For example, installing the tools of the hypervisor used by Amazon EC2 will not serve VMs in Microsoft Azure and vice versa. Also, every additional installation and removal of hypervisor tools risks triggering problems, compatibility issues, and undesirable interactions between new and existing software [39]. Anecdotal evidence based on interaction with enterprise cloud clients indeed suggests that they tend to prefer not to install hypervisor tools as long as their workloads performance remains reasonable [20].

A final benefit of refraining from installing a balloon in a guest is that it prevents *over-ballooning*, whereby the guest OS experiences a sudden spike in memory demand that it cannot satisfy, causing it to terminate some of its running applications before the balloon manager deflates its balloon. We have conducted some limited experiments with the VMware hypervisor, vSphere 5.1, and learned that in this environment over-ballooning seems to be a rare corner case.[1] Conversely, in the KVM/QEMU-based experimental setup we utilize in this paper, over-ballooning was more frequent, prompting our Ubuntu guests to terminate running applications with their out-of-memory (OOM) or low-memory killers under memory pressure. Using VSWAPPER without ballooning eliminated this problem, as depicted in Figure 5.

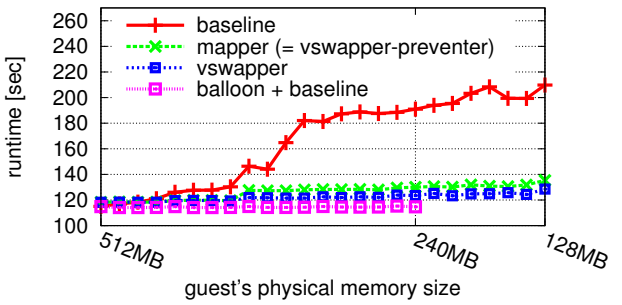

Figure 5. *Over-ballooning in our KVM/QEMU experimental setup, when compressing the Linux kernel code with pbzip2 from within a 512MB guest whose actual physical memory size is displayed along the X axis. Ballooning delivers better performance, but the guest kills bzip2 when its memory drops below 240MB.*

3. Problems in Baseline Swapping

If we are to improve the performance of virtual systems that employ uncooperative swapping, we need to have a thorough understanding of why it really hinders performance. We have characterized the root causes of the degraded performance through careful experimentation. The aforementioned double paging problem did not turn out to have a dominant effect or notable volume in our experiments, probably because bare metal (non-virtualized) swapping activity is typically curbed so long as the system is not thrashing [6], and because the uncooperative guest believes it operates in an environment where memory is sufficient.[2] The problems that did turn out to have a meaningful effect and that we were able to address are listed next.

Silent Swap Writes: So long as memory is plentiful, much of the memory of general purpose OSes is dedicated to caching file content long after the content is used, in the hope that it will get re-used in the future [6]. When memory gets tight, unused content is discarded and the corresponding memory frames are freed by the OS.

In a virtual setup with uncooperative swapping, it is the host that decides which pages to swap, whereas the guest OS remains unaware. The host can nonetheless make an informed decision, as it too maintains per-frame usage statistics, allowing it to victimize unused pages. If a victim page is dirty, the host writes it to its swap area so as to later be able to recover the correct data.

The question is what to do if the page being reclaimed is clean. One alternative is to just discard it. But then the host would need: (1) to track and maintain correspondence between guest memory pages and the original file blocks from which they were read; (2) to handle subtle consistency issues (to be addressed later on); and (3) to treat clean and dirty reclaimed pages differently, mapping the former to the original file and the latter to its swap area. The easier alternative—that hypervisors like KVM and VMware's vSphere favor [76, p. 20]—is to keep all reclaimed guest pages in the host swap area, saving them there even if they are clean and identical to their origin file blocks. Worse, current x86 server hardware does not yet support dirty bits for

[1] Triggered, for example, when two guests allocate (what they perceive to be) pinned memory that collectively amounts to 1.5x of the physical memory available to the hypervisor.

[2] Conversely, when a guest is cooperative, the explicit purpose of inflating the balloon is to prompt the guest to swap out pages, in which case double paging is probably more likely.

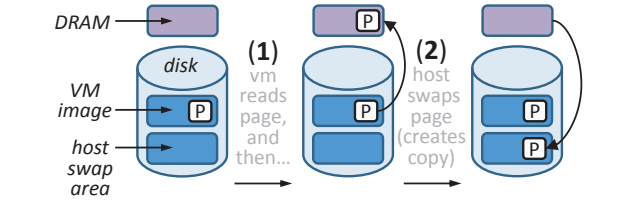

Figure 6. *Silent swap writes.*

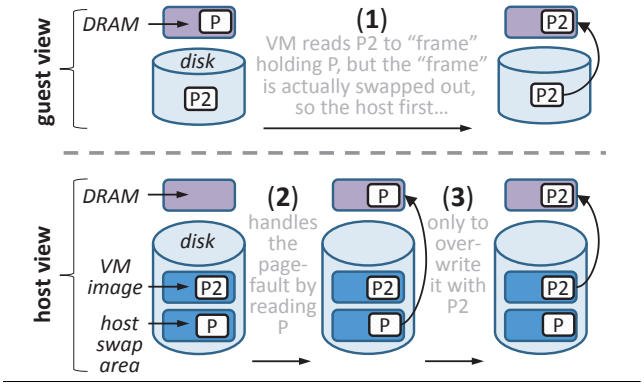

Figure 7. *Stale swap reads.*

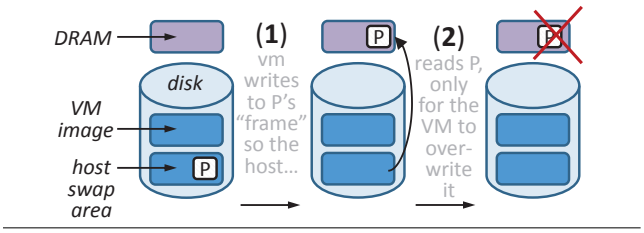

Figure 8. *False swap reads.*

guest pages,[3] so hosts assume that reclaimed pages are *always* dirty. Since hosts write to disk data that is already there, we denote this activity as *silent swap writes* (see Figure 6).[4]

Stale Swap Reads: Suppose a guest generates an explicit I/O request to read some block from its (virtual) disk into one of its memory pages, denoted *P*. The virtual I/O operation generated by the guest triggers an exit to the host that generates a corresponding physical I/O request directed at the physical disk. Page *P* is hence designated to be the destination of the physical I/O operation as well.

Consider what happens if *P* was previously reclaimed by the host. In such a case, the host would experience a page fault as part of the processing of the virtual I/O request, before the corresponding physical request is issued. *P*'s old content would thus be faulted-in, only to be overwritten shortly after by the physical I/O operation and the newly read block. We denote such host reads, whose outcome is never read and is instantly superseded by subsequent reads, as *stale swap reads* (see Figure 7).

Note that after a file block is read, so long as the uncooperative guest keeps it in its file cache, it will never again be accompanied by a stale read, because, by definition, stale reads only occur due to explicit guest I/O requests.

False Swap Reads: Memory management performed by guests includes activities like zeroing pages before they are (re)allocated [57], copying memory pages on write (COW) [34], and migrating pages from one DRAM location to another due to memory compaction [15], e.g., for super paging [23, 51]. Whether by copying memory or zeroing it, guests often overwrite full pages without regard to their old content. Such activity has no undesirable side effects in bare metal setups. But in virtual setups with uncooperative swapping, the target page being overwritten might be swapped out, generating an outcome similar to that of stale reads. Namely, the old content would be read and immediately overwritten. We denote such useless reads as *false swap reads* (see Figure 8).

The difference between stale and false reads is the computational entity that does the overwriting. It is the disk device that overwrites the stale reads via direct memory access (DMA). And it is the (guest) CPU that overwrites the false reads by copying or zeroing content. Clearly, it will be harder to identify and eliminate false reads, because the host has no a priori knowledge about whether the CPU is going to overwrite an entire target page or only part of it; in the latter case, the reads are necessary and hence are not false.

Decayed Swap Sequentiality: OSes perform file prefetching to alleviate the long latencies that programs endure when forced to wait for disk reads. The most rewarding and straightforward read pattern to anticipate is sequential access. It is easy for the OS to notice. And it is easy to issue reads for subsequent parts of the file beforehand. Additionally, contiguous file pages tend to be contiguous on disk, minimizing the movement of the head of the hard drive and thus making prefetching relatively inexpensive.

Being an OS, the guest does its own prefetching from its virtual disk. The host merely acts as proxy by issuing the I/O operations generated by the guest. But things change when memory becomes scarcer under uncooperative swapping. When the host reclaims pages, it swaps their content out. And from this point onward, any prefetch activity related to those pages is inevitably performed only by the host, as the uncooperative guest is not even aware that the pages are not there. Importantly, the swap prefetch activity is *exclusively* initiated by the host page fault handler when it must swap in previously swapped out content. Namely, (swap) file prefetching is in fact a memory management issue.

The problem that consequently arises is the outcome of a detrimental guest-host interaction. Unaware that memory is scarce, the guest too aggressively prefetches/caches file content from its virtual disk. So the host swaps out some other

[3] The expected features of Intel's next generation "Haswell" server architecture (to be released not before the end of 2014 [86]) include support for access and dirty bits for guest pages [72].

[4] The chosen term is analogous to "silent stores," which characterize cases whereby a value being written by the store machine operation matches the exact value already stored at that corresponding memory location [42].

guest pages to accommodate the excessive memory demand. It therefore happens that cached file content from the guest virtual disk ends up in host swap area. But whereas the content blocks are contiguous on the virtual disk, they become scattered and uncoupled in the swap area, because spatial locality is secondary when victimizing pages for reclamation (as opposed to usage, which is primary). Host swap prefetching therefore becomes ineffective, such that the longer the execution, the more pronounced the effect. We call this phenomenon *decayed swap sequentiality*.

False Page Anonymity: Memory pages backed by files are called *named pages*. Such are the pages of loaded executables and of files mapped to memory [52]. Conversely, memory pages not backed by files are called *anonymous pages*. Such are the pages of heaps and stacks of processes. Note that any page that could be moved to the swap area is anonymous, or else it would have been backed by some other (non-swap) file. As explained above, all the guest disk image pages are classified by the host as anonymous. This (mis)classification turns out to have negative consequences.

OSes are generally configured to have some preference to evict named pages when the need arises, because they can be reclaimed faster without write-back to swap, and because file access patterns typically exhibit more spatial locality than access to pages residing in the swap [55], making named pages easier to prefetch. Alas, guests are unable to enjoy such a preferential reclamation with uncooperative swapping, as the host (mis)classifies all their pages as anonymous. Worse, when the hypervisor is hosted (as is the case with QEMU/KVM), the hypervisor executable code within the otherwise-anonymous guest address space is classified as named, making the host OS inclined to occasionally reclaim these vital pages, thereby hindering performance further. We characterize this deficiency as *false page anonymity*.

3.1 Demonstration

Having enumerated the problems in uncooperative swapping that we have identified, we now experimentally demonstrate the manifestation of each individual problem in isolation. We use a simple experiment whereby one guest iteratively runs a Sysbench benchmark configured to sequentially read a 200MB file. The guest believes it has 512MB, whereas in fact it is allocated only 100MB and all the rest has been reclaimed by the host. The results are depicted in Figure 9.

The performance of baseline uncooperative swapping is roughly U-shaped (Figure 9a), taking about 40 seconds in the first iteration, which are halved in the second iteration, only to gradually work their way back to 40 seconds in the final iteration. The first iteration performance is largely dominated by stale swap reads. Those occur when the guest performs explicit I/O reads from its virtual disk, bringing content into memory that has been reclaimed by the host. The counterproductive activity is evident when examining the number of page faults experienced by the host while executing its own code in service of the guest (Figure 9b; first iteration). From the second iteration onward, no stale reads occur because the guest stops generating explicit I/O requests, believing it caches the entire file in memory and servicing all subsequent reads from its file cache. Thus, it is the absence of stale reads from all iterations but the first that accounts for the left side of the aforementioned U-shape.

With the exception of the first iteration, all page faults shown in Figure 9b are due to the hosted hypervisor code faulted-in while it is running. The code was swapped out because it was the only named part of the guest's address space, a problem we have denoted as false page anonymity. The problem becomes more pronounced over time as evident by the gradual increase in the baseline curve in Figure 9b.

Contributing to the gradually worsened performance (second half of U-shape) is the increasingly decayed sequentially of the host swap area. The file content is read to memory and then swapped out to disk, over and over again. As the content moves back and forth between disk and memory, it gradually loses its contiguity. Special locality is diminished, and host swap prefetching becomes ineffective. This negative dynamic is evident when plotting the number of page faults that fire when the guest accesses its memory in Figure 9c. Such faults occur due to non-present GPA⇒HPA mappings[5] while the *guest* is running (as opposed to Figure 9b, which depicts faults that occur while the *host* is running, servicing explicit virtual I/O requests generated by the guest). Every such page fault immediately translates into a disk read from the host swap area, which may or may not succeed to prefetch additional adjacent blocks. Importantly, only if the prefetch is successful in bringing the next file block(s) to be accessed will the next memory access(es) avoid triggering another page fault. Thus, amplified sequentiality decay implies greater page fault frequency, which is what we see in Figure 9c for the baseline curve. (Conversely, the VSWAPPER curve implies no decay, as it is horizontal.)

Baseline uncooperative swapping copies unchanged blocks of data to the swap area, although the corresponding source blocks are identical and stored within the guest disk image. We have denoted this phenomenon as silent swap writes. We find that the volume of this activity is significant, but that it contributes equally to the degraded performance exhibited by all iterations (Figure 9d).

The remaining problem we have not yet demonstrated is that of false reads, which occur when the guest attempts to overwrite memory pages that have been reclaimed by the host, e.g., when reallocating and initializing a page that previously held some information the guest no longer needs. We did not encounter false reads in the above Sysbench benchmark because it reads, rather than writes. We therefore extend the benchmark such that after it finishes all the read activity, it forks off a process that allocates and sequentially accesses 200MB. The simplest way to get a sense of the

[5] See Figure 1 for the meaning of GPA and HPA.

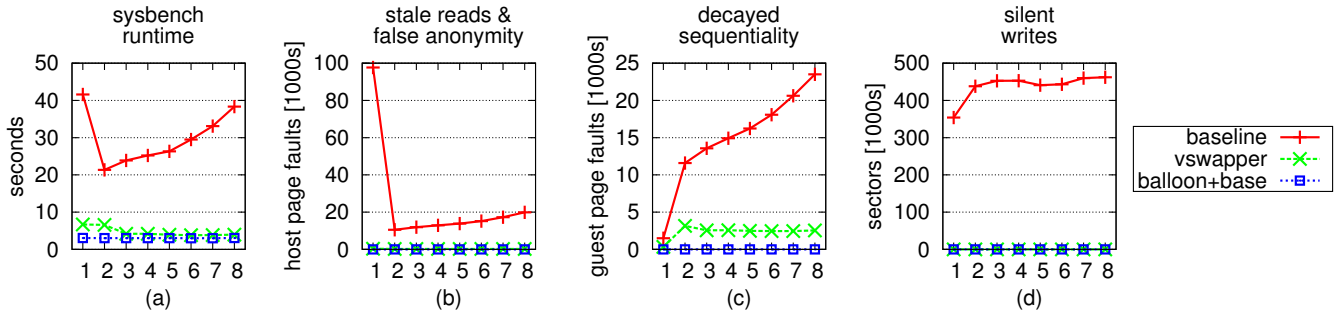

Figure 9. *Sysbench iteratively reads a 200MB file within a 100MB guest that believes it has 512MB. The x-axis shows the iteration number. The y-axis shows: (a) the benchmark's runtime; (b) the number of page faults triggered while the host code is running (first iteration faults are caused by stale reads, and the rest are due to false page anonymity); (c) number of page faults triggered while the guest code is running (a result of decayed sequentially); and (d) the number of sectors written to the host swap area (a result of silent writes).*

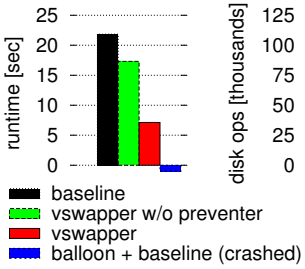

Figure 10. *Effect of false reads on a guest process that allocates and accesses 200MB.*

effect of false reads on this newly added microbenchmark is to measure its performance when using VSWAPPER without and with its False Reads Preventer component. Figure 10 shows the results. (The balloon performance is missing since it crashed the workload due to over-ballooning.) Comparing the two VSWAPPER configurations, we see that enabling the Preventer more than doubles the performance and that the performance is tightly correlated to the disk activity.

4. VSWAPPER Design and Implementation

We mitigate the problems of uncooperative swapping by introducing two new mechanisms. The first is the Swap Mapper (§4.1), which tracks the correspondence between disk blocks and guest memory pages, thereby addressing the problems of silent writes, stale writes, decayed sequentially, and false page anonymity. The second mechanism is the False Reads Preventer (§4.2), which temporarily buffers data written by unaware guests to swapped out pages, thereby addressing the problem of false reads.

4.1 The Swap Mapper

The poor performance of uncooperative swapping is tightly related to how guests utilize page caches, storing in memory large volumes of currently unused disk content because they (wrongfully) believe that memory is plentiful. Hypervisors need to learn to cope with this pathology if they are to efficiently exercise uncooperative swapping. The Swap Mapper achieves this goal by tracking guest I/O operations and by maintaining a mapping between corresponding disk and memory locations. When used carefully (so as to avoid subtle consistency issues), the mapping equips the hypervisor with the much-needed ability to treat relevant guest pages as file-backed. This ability counteracts the harmful effect of large page caches within unaware guests, because it allows the hypervisor to reasonably identify the pages that populate the cache and to efficiently discard them when the need arises, without undesirable consequences.

The Mapper's goal is to bridge a semantic gap. It needs to teach the hypervisor which guest pages are backed by which disk blocks. This goal could in principle be achieved through intrusive modifications applied to the guest [61] or through exhaustive memory/disk scans. But we favor a much simpler approach. Our Mapper leverages the fact that guest disk I/O is overwhelmingly implemented via emulation [67] or paravirtualization [5, 56], whereby the hypervisor serves all I/O request directed at the virtual disk. The Mapper can thus interpose on this activity and maintain the required association between disk blocks and memory pages.

The core of our Mapper implementation is simple. In the KVM/QEMU environment, each guest VM resides within (and is served by) a regular user-level QEMU process. The guest I/O requests are trapped by QEMU, which uses standard `read` and `write` system calls to satisfy the requests. Our Mapper replaces these reads/writes with `mmap` system calls [52], which provide the basic functionality of mapping guest pages to disk blocks, out of the box. The pages thus become named and are treated by the host Linux kernel accordingly. We establish "private" mappings (via standard `mmap` flags), which preserve the per-page disk association only so long as the page remains unchanged. A subsequent write instruction directed at the page will prompt the host kernel to copy-on-write the page and to make it anonymous. Thus, the disk-to-memory association is correctly maintained only as long as the content of the memory page is identical to the cor-

responding disk blocks. Future memory store operations by the unaware guest will *not* alter the disk.

As a result of this change, native uncooperative swapping done by the host Linux kernel automagically becomes more effective. Since the pages are named, they are evicted more frequently than anonymous pages, as their reclamation and retrieval is more efficient [55] (no false page anonymity). Specifically, when the kernel's page frame reclaiming mechanism selects a guest page for eviction, it knows the page is backed by a file, so it discards the page by discarding the mapping, instead of by swapping the page out (no silent swap writes). Later on, if/when the reclaimed page is accessed, the kernel knows from where to retrieve it using the information associated with the faulting non-present page table entry. Realizing the page is backed by a file, the kernel re-maps it instead of swapping it in, disregarding the specific target memory frame (no stale swap reads). At that point, host prefetch mechanisms perform disk read-ahead, benefiting from the sequential structure of the original guest disk image (no decayed swap sequentiality).

Data Consistency: While the core idea is simple, we need to resolve several problems to make the Mapper correct, as the mmap mechanism was not designed to be used in such a way. The first problem stems from the fact that a memory mapped file region can, in parallel, be written to through ordinary I/O channels. To better understand this difficulty, suppose that (1) a named page P with content C_0 is mapped to memory, that (2) P previously resided in DRAM because the guest accessed it via the memory, that (3) C_0 currently resides on disk because P's frame was reclaimed by the host, and that (4) the guest has now issued an explicit disk I/O write directed at the blocks holding C_0 in order to write to them new content C_1. In this situation, it would be an error to naively process the latter I/O write, because, later, if the guest reads P via memory, it will rightfully expect to get C_0, but it will instead erroneously get C_1 (after P is faulted in).

To solve the problem, we modify the host open system call to support a new flag, used by QEMU when opening the guest virtual disk file. The flag instructs the kernel to invalidate page mappings when associated disk blocks are being written to through the corresponding file descriptor. Invalidation involves reading C_0 and delaying the processing of C_1 until C_0 is fetched. Then, the mapping is destroyed and C_1 is finally written to disk. The host kernel (not QEMU) is the natural place to implement this semantic extension, as the kernel maintains the page mappings.

Host Caching & Prefetching: It is generally recommended to turn off host caching and prefetching for guest disk images [30, 41, 63, 64]. The reasoning is that guest OSes do their own caching/prefetching, and that they are inherently better at it because the hypervisor suffers from a semantic gap. (For example, a guest knows about files within its virtual disk, whereas, for the hypervisor, the disk is just one long sequence.) For this reason, all non-VSWAPPER configurations in the evaluation section (§5) have host caching disabled. Conversely, VSWAPPER must utilize the host "caching" for the prosaic reason that mmaped pages reside in the host page cache. Our implementation, however, carefully makes sure that, beyond this technicality, the host page cache never truly functions as a cache; namely, it holds virtual disk blocks only if they are currently residing in guest memory. Thus, when a guest writes to a host filed-backed page, the page is COWed (due to being privately mapped), and then VSWAPPER removes the source page from the host page cache.

In all configurations, host prefetching activity is prompted by page faults. It is limited to reading content that is already cached by the guest and has been reclaimed due to uncooperative swapping. But whereas non-VSWAPPER configurations only prefetch from their host swap area, the VSWAPPER design allows it to prefetch these pages from the disk image.

Using the host page cache does not break crash consistency guarantees of guest filesystems. QEMU supports crash consistency by default with its "writethrough" disk caching mode, which synchronizes writes to the disk image upon guest flush commands [32]. Guests are notified that their flushes succeed only after the synchronization, thereby ensuring the Mapper does not deny crash consistency.

Guest I/O Flow: Explicit disk read requests issued by the guest are translated by QEMU to a preadv system call invocation, which reads/scatters a contiguous block sequence to/within a given vector of guest pages. There is no mmapv equivalent. So, instead, the Mapper code within QEMU initiates reading the blocks to the page cache by invoking readahead (an asynchronous operation). It then iteratively applies mmap to the pages, using the "populate" mmap flag. The latter ensures that the readahead completes and that the pages are mapped in QEMU's page tables, thereby respectively preventing future major and minor page faults from occurring when QEMU accesses the pages. Alas, an undesirable side-effect of using "populate" is that the pages will be COWed when they are first accessed. We therefore patch the host's mmap to support a "no_COW" flag and thus avoid this overhead. Lastly, the Mapper iteratively invokes ioctl, requesting KVM to map the pages in the appropriate GPA⇒HPA table so as to prevent (minor) page faults from occurring when the guest (not QEMU) accesses the pages.

A second problem that immediately follows is how to correctly mmap a guest page P that is being written to a disk block B via a write request that has just been issued by the guest. Due to our newly added open flag (see "Data Consistency" above), B is not mmaped right before the request is processed, even if B was accessed in the past. Conversely, we want B to be mmaped right after the request is processed, such that, later, if P is reclaimed, we will not need to swap it out. The Mapper therefore: (1) writes P into B using the write system call, (2) mmaps P to B, and (3) only then notifies the guest that its request is completed.

Page Alignment: An inherent constraint of file-backed memory is that it mandates working in whole page granularity. The standard mmap API indeed dictates that both the file offset and the mapped memory address should be 4KB-aligned. The Mapper therefore must arrange things such that virtual disk requests coming from guests will comply with this requirement. Our Mapper imposes compliance by informing the guest that its virtual disk uses a 4KB logical sector size upon the creation of the virtual disk image. This approach will not work for preexisting guest disk images that utilize a smaller block size. Such preexisting images will require a reformat. (We remark that disks are expected to gradually shift to employing a 4KB *physical* block size [13].)

4.2 The False Reads Preventer

A dominant contributor to the poor performance of uncooperative swapping is the host's inability to know a-priori when guests overwrite entire pages and discard their old content. Such events routinely happen, e.g, when guests allocate pages to new processes. Unaware, the hypervisor needlessly reads the old content if it happens to be swapped out, a costly operation paid only because the host does not understand the guest semantics. The False Reads Preventer alleviates this problem by trapping guest write instructions directed at a swapped out pages, emulating them, and storing their result in page-sized, page-aligned buffers. If a buffer fills up, the Preventer maps it to the guest, thereby eliminating the extraneous disk accesses, which we have denoted as "false reads."

The Preventer does not utilize any knowledge about guest OS internals, nor does it resort to paravirtual guest/host collaboration that others deem necessary [61]. Instead, it optimistically intercepts and emulates guest write instructions directed at swapped out pages, hoping that all bytes comprising the page will soon be overwritten, obviating the need to read the old content from disk. When that happens, the Preventer stops emulating and repurposes its write buffer to be the guest's page.

The Preventer can sustain the emulation of all writes and all reads directed at already-buffered data. But emulation is slow, so we stop emulating a page when a predetermined interval has elapsed since the page's first emulated write (1ms), or if the write pattern is not sequential. We further avoid emulating a newly accessed page if too many pages are already being emulated (32). (The two values—1ms and 32—were empirically set.) In both cases, the corresponding missing page is read asynchronously. The guest is allowed to continue to execute so long as it does not read unavailable data; if it does, then the Preventer suspends it. When the disk content finally arrives, the Preventer merges the buffered and read information, and it resumes regular execution.

The Preventer design is architected to avoid a data hazard created by the fact that, in addition to the guest, the guest's memory pages can also be directly accessed by QEMU, which is the user-level part of the hypervisor that resides in an ordinary (non-virtualized) process. To preserve correctness and consistency, QEMU must observe exactly the same data as its guest, motivating the following design.

component	*user (QEMU)*	*kernel*	*sum*
Mapper	174	235	409
Preventer	10	1,964	1,974
sum	184	2,199	2,383

Table 1. *Lines of code of* VSWAPPER.

Let P be a reclaimed page frame that is being emulated. The data we maintain for P includes the time of P's first emulated write, a page-sized buffer that stores emulated writes at the same offset as that of the real writes, the number of buffered bytes, and a bitmap marking all the buffered bytes, utilized to decide if reads can be emulated and to determine how to merge with the original disk content. The data structure also contains a reference to the original memory mapping of the reclaimed page (a vm_area_struct denoted here as M_{old}), to be used for reading the preexisting data in case a merge is required. Upon the first emulated write to P, we break the association between P and M_{old}, and we associate P with a new vm_area_struct (denoted M_{new}).

Note that P is respectively accessed by QEMU and the guest via HVAs and GVAs (see Figure 1), such that the two types of accesses trigger different page fault handlers. Faulting HVAs trigger a "regular" handler (denoted h), whereas faulting GVAs trigger a special virtualization handler (denoted g).[6] The Preventer associates M_{new} with an h handler that, when invoked, terminates the emulation by merging the buffer with the old content, reading the latter via M_{old} if it is needed; QEMU is suspended until h finishes, ensuring it will always get up-to-date data when it faults. In contrast, the Preventer patches g to sustain the emulation by buffering writes and serving reads if their data has been previously buffered. When g decides to terminate the emulation (e.g., because 1ms has elapsed since the first write), it initiates the termination by invoking h.

We have identified a number of emulated instructions that allow the Preventer to recognize outright that the entire page is going to be rewritten, when the x86 REP prefix is used [33]. The Preventer short-circuits the above mechanism when such instructions are encountered. We expect that advanced binary translation techniques [1] could do better.

The number of lines of code of VSWAPPER is detailed in Table 1. The VSWAPPER source code is publicly available [3].

5. Evaluation

We implement VSWAPPER within QEMU [65] and KVM, the Linux kernel-based hypervisor [37]. We run our experiments on a Dell PowerEdge R420 server equipped with two 6-core 1.90GHz Intel Xeon E5-2420 CPUs, 16GB of memory,

[6] This handler serves "extended page table (EPT) violations," which occur when the hardware traverses GPA⇒HPA page table entries (bottom of Figure 1) that are marked non-present, e.g., due to uncooperative swapping.

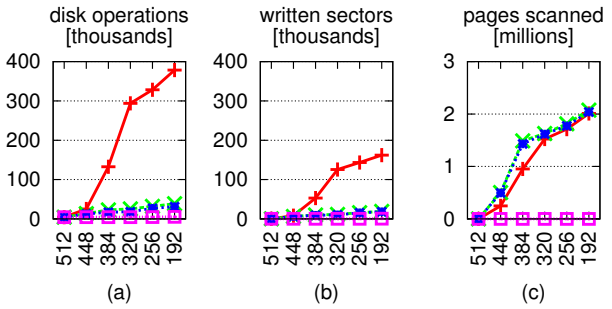

Figure 11. *Pbzip's 8 threads compressing Linux within a guest whose actual memory size is displayed along the X axis (in MB).*

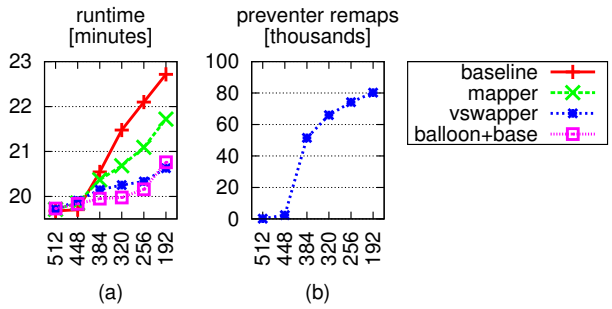

Figure 12. *Compiling the Linux kernel source code with Kernbench. (The X axis is the same as in Figure 11.)*

and a 2TB Seagate Constellation 7200 enterprise hard drive. Host and Linux guests run Ubuntu 12.04, Linux 3.7, and QEMU 1.2 with their default settings. The Windows guest runs Windows Server 2012. Guests have 20GB raw image disk drives, paravirtual disk controllers, and 1–2 VCPUs as indicated. We disable host kernel memory deduplication (KSM) and compression (zRAM) to focus on ballooning. We constrain guest memory size using container groups ("cgroups") as recommended [40]. The host caching policy is as specified in §4.1.

We evaluate five configurations: (1) "baseline," which relies solely on uncooperative swapping; (2) "balloon," which employs ballooning and falls back on uncooperative swapping; (3) "mapper," which denotes VSWAPPER without the Preventer; (4) "vswapper," which consists of both Mapper and Preventer; and (5) "balloon + vswapper," which combines ballooning and VSWAPPER. We typically run each experiment 5 times and present the average. When balloon values are missing it is because the workload crashed due to over-ballooning (§2.4).

5.1 Controlled Memory Assignment

We begin by executing a set of experiments whereby we systematically reduce and fix the size of the memory assigned to a 1-VCPU Linux guest, such that the guest believes it has 512MB of memory but it may actually have less. Balloon configurations communicate this information to the guest by appropriately inflating the balloon driver, whereas baseline and VSWAPPER configurations leave the guest unaware. The exact memory size allocated to the guest is displayed along the X-axis of the respective figures.

In this subsection, the results of the two balloon configurations (with and without VSWAPPER) were similar, so the respective figures display the balloon + baseline configuration only, to avoid clutter.

Pbzip2: In our first set of experiments, the guest runs pbzip2, which is a parallel implementation of the bzip2 block-sorting file compressor [21]. We choose this multi-threaded benchmark to allow the baseline configuration to minimize uncooperative swapping overheads by leveraging the "asynchronous page faults" mechanism employed by Linux guests [50]. This mechanism exploits intra-guest parallelism to allow guests to continue to run despite experiencing page faults caused by host swapping (the host delivers a special page fault exception advising the guest to context switch or else it would block). We evaluate the performance by applying pbzip2 to the Linux kernel source code.

The execution time is shown in Figure 5, indicating that despite the asynchronous faults, the baseline performance rapidly worsens with memory pressure, yielding an execution time up to 1.66x slower than ballooning. VSWAPPER and its mapper-only configuration improve upon the baseline, respectively yielding performance within 1.03–1.08x and 1.03–1.13x of ballooning, since they greatly reduce the number of disk operations (Figure 11a). Baseline disk operations include a notable component of writes, which is largely eliminated by VSWAPPER (Figure 11b), thus making it beneficial for systems that employ solid state drives (SSDs).

Kernbench: For our second benchmark evaluation, we reproduce an experiment reported in a VMware white paper [76] in which the authors executed Kernbench—a standard benchmark measuring the time it takes to build the Linux kernel [38]—inside a 512MB guest whose actual memory allocation was 192MB. Relative to the runtime measured when the guest was allocated the entire 512MB, the authors reported 15% and 4% slowdowns with baseline uncooperative swapping and ballooning, respectively. Although our experimental environment is different, we observe remarkably similar overheads of 15% and 5%, respectively (Figure 12a).

The performance of the baseline, mapper, and VSWAPPER configurations relative to ballooning is 0.99–1.10x, 1.00–1.05x, and 0.99–1.01x faster/slower, respectively. The Preventer eliminates up 80K false reads (Figure 12b), reducing guest major page faults by up to 30%.

Eclipse: Our final set of controlled memory experiments executes Eclipse workloads that are part of the DaCapo Java benchmark suite [7]. (Eclipse is a popular integrated development environment.) Java presents a challenge for virtual environments, as its garbage collector subsystem causes an LRU-related pathological case of degraded performance

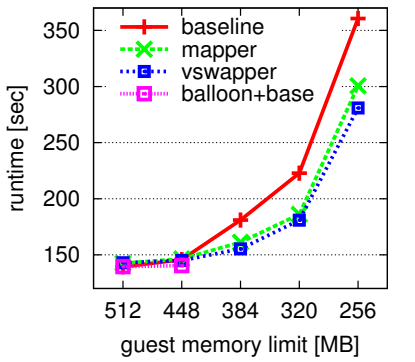

Figure 13. *Eclipse IDE workload from the DaCapo benchmark suite.*

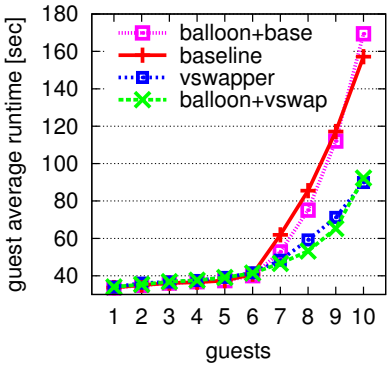

Figure 14. *Phased execution of multiple guests running the MapReduce runtime.*

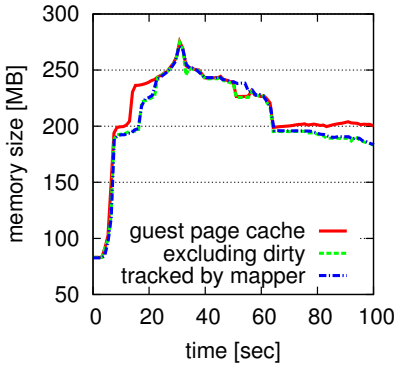

Figure 15. *Size of page cache as time progresses. (From the Eclipse benchmark.)*

when the physical memory allocated to the guest is smaller than the Java virtual machine (JVM) working set [76].

Figure 13 depicts the benchmark results, executed using OpenJDK and a 128MB heap. While it manages to run, ballooning is 1–4% faster than the other configurations, but Eclipse is occasionally killed by the ballooning guest when its allocated memory is smaller than 448MB. Relative to VSWAPPER, the baseline and mapper configurations are 0.97–1.28x and 1.00–1.08x faster/slower, respectively.

5.2 Dynamic Memory Assignment

So far, we have utilized benchmarks whereby the amount of memory allocated to guests is fixed. Virtualization setups, however, commonly run multiple guests with memory demands that dynamically change over time. To evaluate the performance in this scenario, we execute a workload whose dispatch is phased so that guests start the benchmark execution one after the other ten seconds apart. Handling such workloads is challenging for balloon managers, yet similar resource consumption spikes in virtualization environments are common [66].

In our dynamic experiment set, we vary the number of guests from one to ten. Each guest runs the Metis Mapreduce runtime for multicores [9, 47, 54], which comes with a benchmark suite comprised of 8 applications. The results we present here are of the word-count application, which uses a 300MB file holding 1M keys. The memory consumption of Metis is large, as it holds large tables in memory, amounting to roughly 1GB in this experiment. We assign each guest with 2 VCPUs and 2GB of memory, and we limit the host memory to 8GB so that it will eventually overcommit. (We note that each guest virtual disk is private, and so VSWAPPER does not exploit file caching to improve performance by saving fewer data copies.)

We employ MOM, the Memory Overcommitment Manager [43], to manage and adapt the balloon sizes. MOM is a host daemon which collects host and guest OS statistics and dynamically inflates and deflates the guest memory balloons accordingly. MOM requires that we use libvirt [8], a virtualization API for controlling virtual machines.

Figure 14 presents the average runtime as a function of the number of guests comprising the experiment. Running seven or more guests creates memory pressure. From that point on we observe a cascading effect, as guest execution is prolonged due to host swap activity and therefore further increases memory pressure. Clearly, the slowdown is lowest when using VSWAPPER, whereas memory ballooning responds to guest memory needs belatedly. Relative to the combination of ballooning and VSWAPPER, we get that: ballooning only, baseline, and VSWAPPER are 0.96–1.84x, 0.96–1.79x, and 0.97–1.11x faster/slower, respectively, suggesting that the combination is the preferable configuration.

5.3 Overheads and Limitations

Slowdown: VSWAPPER introduces slowdowns which might degrade the performance by up to 3.5% when memory is plentiful and host swapping is not required. The slowdowns are mostly caused by our use of Linux mmap, which was advantageous for simplifying our prototype but results in some added overheads. Firstly, because using mmap is slower than regular reading [29]. And secondly, because a COW (which induces an exit) is required when a named page is modified, even if there are no additional references to that page. The latter overhead could be alleviated on hardware that supports dirty bits for virtualization page tables, which would allow VSWAPPER to know that pages have changed only when it needs to, instead of immediately when it happens.

We uncovered another source of overhead introduced by the page frame reclamation mechanism, which scans the pages in search for eviction candidates when the need arises. Due to subtleties related to how this mechanism works in Linux, the impact of VSWAPPER is such that it up to doubles the length of the mechanism traversals when memory pressure is low (Figure 11c).

The aforementioned 3.5% overhead can be compared to the overhead of "Geiger," a guest page cache monitoring mechanism by Jones et al., which introduced overheads of

up to 2% [35]. Part of the 1.5% difference is probably accounted for by the fact that Geiger was evaluated on a system that did not support guest memory virtualization in hardware, forcing the hypervisor to write-protect newly mapped pages in the baseline setup and thereby creating exits that Geiger leveraged for tracking.

Memory Consumption: The Mapper's use of the native Linux memory area data structures (vm_area_struct and i_mmap) increases memory consumption and might fragment the hypervisor address space. These structures consume 200 bytes, so theoretically, in the worst case, the overhead might be 5% of the guest memory size, if every 4KB page requires its own structure. Underlying this upper bound is our decision to use the already existing mmap mechanism. A dedicated mechanism for tracking guest page caches can achieve a similar goal with only 20 bytes per page [35]. Empirically, the Mapper consumed not more than 14MB across all of our experiments.

The Mapper is quite successful in tracking only the memory pages that reside in the guest page cache. The Mapper's effectiveness is illustrated in Figure 15, which shows that the memory size it tracks coincides with the size of the guest page cache excluding dirty pages. The Mapper correctly avoids tracking dirty pages, as they do not correspond to disk blocks. The Mapper occasionally tracks more pages than found in the page cache (time≈50 in Figure 15), because guests sometimes repurpose pages holding disk content for other uses. The Mapper will break the disk association of these pages when they are subsequently modified.

The Mapper will lose track of disk-backed pages if guests perform memory migration, e.g., for superpages [23, 51] or due to NUMA considerations. This drawback can be eliminated in a straightforward manner via paravirtual interfaces [73]; the question of whether it could be efficiently done for fully virtualized guests remains open, but the wealth of efficient memory deduplication techniques [25] suggests the answer is yes.

5.4 Non-Linux Guests and Hosts

Windows: We validate the applicability of VSWAPPER to non-Linux guests using a VM running Windows 2012 Server. Windows does not align its disk accesses to 4KB boundaries by default. The hypervisor should therefore report that the disk uses 4KB physical *and* logical sectors to enforce the desired behavior. Our patched QEMU reports a 4KB physical sector size. But alas, QEMU virtual BIOS does not support 4KB logical sectors. We therefore formatted the Windows guest disk before installation as if the BIOS reported 4KB logical sectors by: creating aligned partitions, setting the cluster size to be 4KB, and using large file record segments (FRS). We remark that despite this configuration we still observed sporadic 512 byte disk accesses.

Our first experiment consists of Sysbench reading a 2GB file from within a single VCPU 2GB guest that is allocated only 1GB of physical memory. The resulting average runtime without VSWAPPER is 302 seconds, reduced to 79 seconds with VSWAPPER. Our second experiment consists of bzip2 running within the same guest with 512 MB of physical memory. The average runtime without and with VSWAPPER is 306 and 149 seconds, respectively.

	balloon enabled	balloon disabled
runtime (sec)	25	78
swap read sectors	258,912	1,046,344
swap write sectors	292,760	1,042,920
major page faults	3,659	16,488

Table 2. *Runtime and swap activity of executing a 1GB sequential file read from within a Linux VM on VMware Workstation.*

VMware: In our last experiment we attempt to demonstrate that the benefit of using a VSWAPPER-like system is not limited to just KVM. We use the VMware Workstation 9.0.2 hypervisor as an example, running the Sysbench benchmark to execute a sequential 1GB file read from within a Linux guest. We set the host and guest memory to 512MB and 440MB, and we reserve a minimum of 350MB for the latter. The results (Table 2) indicate that disabling the balloon more than triples the execution time and creates substantial additional swap activity, coinciding with VMware's observation that "guest buffer pages are unnecessarily swapped out to the host swap device" [76]. (It is interesting to note that a similar benchmark on KVM using VSWAPPER completed in just 12 seconds.)

6. Related Work

Many paravirtual memory overcommitment techniques were introduced in recent years. Memory ballooning is probably the most common [5, 84]. CMM2 is a collaborative memory management mechanism for Linux guests that makes informed paging decisions on the basis of page usage and residency information [61]. CMM2 can discard free and file-backed guest page frames and thereby eliminate undesirable swap writes, yet it requires substantial and intrusive guest modifications. Transcendent memory [46] uses a pool of underutilized memory to allow the hypervisor to quickly respond to changing guest memory needs. This approach does not suit situations in which multiple guests require more memory all at once [4]. Application-level ballooning [58] can mitigate negative side effects of ballooning on applications that manage their own memory. Like OS-level ballooning, this approach does not render host swapping unnecessary. In contrast to all these paravirtual approaches, VSWAPPER does not require guest modifications.

Memory overcommitment can also be performed by emulating memory hotplug. The hypervisor can inform the guest about "physical" removal/addition of DRAM by emulating the corresponding architectural mechanisms. This approach is fully-virtual, but it has several shortcomings: it takes a

long time to hot-unplug memory due to memory migration, the operation might fail [60], and it is unsupported by popular OSes, such as Windows. Like memory ballooning, memory hotplug cannot cope with memory consumption spikes of guests, and therefore requires host-swapping fallback for good performance under high memory pressures [28].

Regardless of how memory is overcommitted, improved memory utilization can lead to greater server consolidation. Unrelated mechanisms that help to achieve this goal include transparent page sharing [12], cooperative page sharing [49], memory deduplication [70], and sub-page level sharing [25]. All are complementary to VSWAPPER (and to ballooning).

Improving uncooperative host swapping performance was discussed before in Cellular Disco [24], in which two cases of undesirable disk traffic in virtualization environments were presented and addressed: writes of unallocated guest pages to the host swap, and double paging (explained in §2.1). Neither of these cases are addressed by VSWAPPER. Cellular Disco requires guest OS annotations to avoid writing unallocated guest pages to the host swap. VSWAPPER requires no such cooperation.

Our Mapper monitoring techniques resemble those used by Jones et al. for guest buffer cache monitoring [35]. That work used the monitoring techniques to estimate the guest working set size. Lu and Shen used similar techniques for guest memory access tracing [45]. Disco, by Bugnion et al., used "COW disks" for efficient disk sharing across multiple guests in order to eliminate memory redundancy [12]. To this end, they used memory mappings of a single shared disk. The Mapper uses similar techniques for monitoring, but it leverages them for a different purpose: improving host swapping reclamation and prefetching decisions by "teaching" the host memory management subsystem about guest memory to disk image mappings.

Useche used asynchronous page faults and write buffering in OSes to allow non-blocking writes to swapped-out pages [71]. His work implied that such methods have limited potential, as the additional overhead often surpasses the benefits obtained by reducing I/O wait time. Conversely, our work shows that write buffering is beneficial when deployed by hypervisors to enhance uncooperative swapping performance. The opposing conclusions are due to the different nature and purpose of the systems. Useche strives to handle page faults asynchronously in bare metal OSes, whereas VSWAPPER reduces the faults for hypervisors.

7. Future Work

OSes gather knowledge about their pages and use it for paging decisions. Although such information is located in intrinsic OS data structures, the hypervisor may be able to infer some of it and base its paging decisions on common OS paradigms. For instance, since OSes tend not to page out the OS kernel, page tables, and executables, the hypervisor may be able to improve guest performance by adapting a similar policy. The hypervisor can acquire the information by write-protecting and monitoring the guest state upon guest page faults. Alternatively, the hardware could be enhanced to perform such tracking more efficiently, by supporting additional usage flags beyond "accessed" and "dirty." The hardware could, for example, indicate if a page was used: in user mode (which means it is not a kernel page); for page walks while resolving a TLB miss (which means it is a page table); and for fetching instructions (which means it is part of an executable).

VSWAPPER techniques may be used to enhance live migration of guests and reduce the migration time and network traffic by avoiding the transfer of free and clean guest pages. Previous research suggested to achieve this goal via guest cooperation by inflating the balloon prior to live migration [26], or by migrating non-page-cache pages first [2]. The Mapper and the Preventer techniques can achieve the same goal without guest cooperation. Hypervisors that migrate guests can migrate memory mappings instead of (named) memory pages; and hypervisors to which a guest is migrated can avoid requesting memory pages that are wholly overwritten by guests.

8. Conclusions

To this day, uncooperative host swapping is considered a necessary evil in virtual setups, because it is commonly unavoidable but induces high overheads. We isolate, characterize, name, and experimentally demonstrate the major causes for the poor performance. We propose VSWAPPER to address these causes, and we show that VSWAPPER is highly effective in reducing the overheads without or in addition to memory ballooning.

Availability

The source code of VSWAPPER is publicly available [3].

Acknowledgments

We thank Ilya Kravets for his technical help. We thank the reviewers and especially Carl Waldspurger, our shepherd, for their insightful feedback, which has helped to improve this paper greatly. This research was partially supported by The Israel Science Foundation (grant No. 605/12) and by the Hasso Plattner Institute.

References

[1] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *USENIX Annual Technical Conference (ATC)*, volume 12, 2011.

[2] S. Akiyama, T. Hirofuchi, R. Takano, and S. Honiden. Fast wide area live migration with a low overhead through page cache teleportation. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 78–82, 2013.

[3] Nadav Amit. VSwpapper code. http://nadav.amit.to/vswapper, 2014.

[4] Ishan Banerjee, Fei Guo, Kiran Tati, and Rajesh Venkatasubramanian. Memory overcommitment in the ESX server. *VMware technical journal (VMTJ)*, Summer, 2013.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[6] Robert Birke, Lydia Y Chen, and Evgenia Smirni. Data centers in the wild: A large performance study. Technical Report RZ3820, IBM Research, 2012. http://tinyurl.com/data-centers-in-the-wild.

[7] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[8] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. Non-intrusive virtualization management using libvirt. In *Conference on Design, Automation and Test in Europe*, pages 574–579, 2010.

[9] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2010.

[10] Rich Brambley. Why do I need to install VMware tools? http://vmetc.com/2008/08/30/why-do-i-need-to-install-vmware-tools/, 2008.

[11] Mary Brandel. The trouble with cloud: Vendor lock-in. http://www.cio.com/article/488478/The_Trouble_with_Cloud_Vendor_Lock_in, 2009.

[12] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, November 1997.

[13] P. Chicoine, M. Hassner, E. Grochowski, S. Jenness, M. Noblitt, G. Silvus, C. Stevens, and B. Weber. Hard disk drive long data sector white paper. Technical report, IDEMA, 2007.

[14] YP Chien. The yin and yang of memory overcommitment in virtualization: The VMware vSphere 4.0 edition. Technical Report MKP-339, Kingston Technology Corporation, Fountain Valley, CA, 2010. http://media.kingston.com/images/branded/MKP_339_VMware_vSphere4.0_whitepaper.pdf.

[15] Jonathan Corbet. Memory compaction. http://lwn.net/Articles/368869/, 2010.

[16] Oracle Corporation. Virtual box manual. https://www.virtualbox.org/manual/, 2013.

[17] Orcale Corporation. Project: Transcendent memory – new approach to managing physical memory in a virtualized system. https://oss.oracle.com/projects/tmem/, 2010. Visited: Dec 2013.

[18] Johan De Gelas. Hardware virtualization: the nuts and bolts. ANANDTECH, 2008. http://www.anandtech.com/show/2480/10.

[19] Frank Denneman. Impact of memory reservation. http://frankdenneman.nl/2009/12/08/impact-of-memory-reservation/, 2009.

[20] Michael Factor. Enterprise cloud clients tend to prefer unmodified guest virtual machines. Private communication, 2013.

[21] Jeff Gilchrist. Parallel data compression with bzip2. In *IASTED International Conference on Parallel and Distributed Computing and Systems (ICPDCS)*, volume 16, pages 559–564, 2004.

[22] Robert P. Goldberg and Robert Hassinger. The double paging anomaly. In *ACM National Computer Conference and Exposition*, pages 195–199, 1974.

[23] Mel Gorman and Andy Whitcroft. Supporting the allocation of large contiguous regions of memory. In *Ottawa Linux Symposium (OLS)*, pages 141–152, 2007.

[24] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 154–169, 1999.

[25] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Communications of the ACM (CACM)*, 53(10):85–93, 2010.

[26] Michael R Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 51–60, 2009.

[27] Michael R Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Dong Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with Ginkgo. In *IEEE Cloud Computing Technology and Science (CloudCom)*, pages 130–137, 2011.

[28] Eric Horschman. Hypervisor memory management done right. http://blogs.vmware.com/virtualreality/2011/02/hypervisor-memory-management-done-right.html, 2011.

[29] Yiming Hu, Ashwini Nanda, and Qing Yang. Measurement, analysis and performance improvement of the Apache web server. In *IEEE International Performance Computing & Communications Conference (IPCCC)*, pages 261–267, 1999.

[30] Khoa Huynh and Stefan Hajnoczi. KVM / QEMU storage stack performance discussion. In *Linux Plumbers Conference*, 2010.

[31] Woomin Hwang, Yangwoo Roh, Youngwoo Park, Ki-Woong Park, and Kyu Ho Park. HyperDealer: Reference pattern

aware instant memory balancing for consolidated virtual machines. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 426–434, 2014.

[32] IBM documentation. Best practice: KVM guest caching modes. http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liaat/liaatbpkvmguestcache.htm. Visited: Dec 2013.

[33] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2013.

[34] Francisco Javier, Thayer Fábrega, Francisco, and Joshua D. Guttman. Copy on write. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.3144, 1995.

[35] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 14–24, 2006.

[36] Hwanju Kim, Heeseung Jo, and Joonwon Lee. XHive: Efficient cooperative caching for virtual machines. *IEEE Transactions on Computers*, 50(1):106–119, Jan 2011.

[37] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)*, 2007.

[38] Con Kolivas. KernBench project. freecode.com/projects/kernbench.

[39] Ilia Kravets and Dan Tsafrir. Feasibility of mutable replay for automated regression testing of security updates. In *Runtime Environments/Systems, Layering, & Virtualized Environments workshop (RESoLVE)*, 2012.

[40] KVM. Tuning kernel for KVM. http://www.linux-kvm.org/page/Tuning_Kernel.

[41] KVM. Tuning KVM. http://www.linux-kvm.org/page/Tuning_KVM.

[42] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 182–191, 2000.

[43] Adam Litke. Automatic memory ballooning with MOM. http://www.ibm.com/developerworks/library/l-overcommit-kvm-resources/, 2011. Visited: Dec 2013.

[44] Pengcheng Liu, Ziye Yang, Xiang Song, Yixun Zhou, Haibo Chen, and Binyu Zang. Heterogeneous live migration of virtual machines. In *International Workshop on Virtualization Technology (IWVT)*, 2008.

[45] Pin Lu and Kai Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *USENIX Annual Technical Conference (ATC)*, pages 3:1–3:15, 2007.

[46] Dan Magen heimer, Chris Mason, Dave McCracken, and Kurt Hackel. Transcendent memory and Linux. In *Ottawa Linux Symposium (OLS)*, pages 191–200, 2009.

[47] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, Massachusetts Institute of Technology, 2010. URL http://pdos.csail.mit.edu/metis/.

[48] Cade Metz. The meta cloud—flying data centers enter fourth dimension. http://www.theregister.co.uk/2009/02/24/the_meta_cloud/, 2009.

[49] Grzegorz Miłós, Derek G Murray, Steven Hand, and Michael A Fetterman. Satori: Enlightened page sharing. In *USENIX Annual Technical Conference (ATC)*, 2009.

[50] Gleb Natapov. Asynchronous page faults - AIX did it. KVM Forum www.linux-kvm.org/wiki/images/a/ac/2010-forum-Async-page-faults.pdf.

[51] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2002.

[52] The Open Group. *mmap - map pages of memory*, 2004. The Open Group Base Specifications Issue 6. IEEE Std 1003.1. http://pubs.opengroup.org/onlinepubs/009695399/functions/mmap.html.

[53] Brent Ozar. Top 10 keys to deploying SQL server on VMware. http://www.brentozar.com/archive/2011/05/keys-deploying-sql-server-on-vmware/, 2011.

[54] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, 2007.

[55] Rikvan Riel. Linux page replacement design. http://linux-mm.org/PageReplacementDesign, 2010.

[56] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR)*, 42(5):95–103, 2008.

[57] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition*. Microsoft Press, 2004.

[58] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application level ballooning for efficient server consolidation. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 337–350, 2013.

[59] Vinay Sastry. Virtualizing tier 1 applications - MS SQL server. http://blogs.totalcaos.com/virtualizing-tier-1-sql-workloads, 2012.

[60] Joel H Schopp, Keir Fraser, and Martine J Silbermann. Resizing memory with balloons and hotplug. In *Ottawa Linux Symposium (OLS)*, volume 2, pages 313–319, 2006.

[61] Martin Schwidefsky, Hubertus Franke, Ray Mansell, Himanshu Raj, Damian Osisek, and JongHyuk Choi. Collaborative memory management in hosted Linux environments. In *Ottawa Linux Symposium (OLS)*, volume 2, 2006.

[62] L. H. Seawright and R. A. MacKinnon. VM/370—a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, Mar 1979.

[63] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *International Symposium on High Performance Distributed Computer (HPDC)*, pages 15–26, 2012.

[64] Balbir Singh. Page/slab cache control in a virtualized environment. In *Ottawa Linux Symposium (OLS)*, volume 1, pages 252–262, 2010.

[65] Balbir Singh and Vaidyanathan Srinivasan. Containers: Challenges with the memory resource controller and its performance. In *Ottawa Linux Symposium (OLS)*, page 209, 2007.

[66] Vijayaraghavan Soundararajan and Jennifer M. Anderson. The impact of management operations on the virtualized datacente. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 326–337, 2010.

[67] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference (ATC)*, pages 1–14, 2001.

[68] Taneja Group. Hypervisor shootout: Maximizing workload density in the virtualization platform. http://www.vmware.com/files/pdf/vmware-maximize-workload-density-tg.pdf, 2010.

[69] Ryan Troy and Matthew Helmke. *VMware Cookbook*. O'Reilly Media, Inc., 2009. Section 4.1: Understanding Virtual Machine Memory Use Through Reservations, Shares, and Limits.

[70] Irina Chihaia Tuduce and Thomas R Gross. Adaptive main memory compression. In *USENIX Annual Technical Conference (ATC)*, pages 237–250, 2005.

[71] Luis Useche. *Optimizing Storage and Memory Systems for Energy and Performance*. PhD thesis, Florida International University, 2012.

[72] Bob Valentine. Intel next generation microarchitecture codename Haswell: New processor innovations. In *6th Annual Intel Software Developer Conference & User Group*, June 2013. http://ftp.software-sources.co.il/Processor_Architecture_Update-Bob_Valentine.pdf. Visited: Dec 2013.

[73] Rik van Riel. KVM and memory managment updates. KVM Forumhttp://www.linux-kvm.org/wiki/images/1/19/2012-forum-memory-mgmt.pdf, 2012.

[74] Gabrie van Zanten. Memory overcommit in production? YES YES YES. http://www.gabesvirtualworld.com/memory-overcommit-in-production-yes-yes-yes/, 2010.

[75] VMware, Inc. *vSphere 4.1 - ESX and VCenter*. VMware, Inc., 2010. Section: "VMware HA Admission Control". http://tinyurl.com/vmware-admission-control.

[76] VMware, Inc. Understanding memory management in VMware vSphere 5, 2011. Technical white paper. http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf.

[77] VMware Knowledge Base (KB). Problems installing VMware tools when the guest CD-ROM drive is locked. http://kb.vmware.com/kb/2107, 2011. Visited: Dec 2013.

[78] VMware Knowledge Base (KB). Troubleshooting a failed VMware tools installation in a guest operating system. http://kb.vmware.com/kb/1003908, 2012. Visited: Dec 2013.

[79] VMware Knowledge Base (KB). Unable to upgrade existing VMware tools. http://kb.vmware.com/kb/1001354, 2012. Visited: Dec 2013.

[80] VMware Knowledge Base (KB). VMware tools may not install on a windows guest operating system after upgrading to a newer version of ESX/ESXi. http://kb.vmware.com/kb/1012693, 2012. Visited: Dec 2013.

[81] VMware Knowledge Base (KB). Troubleshooting a failed VMware tools installation in Fusion. http://kb.vmware.com/kb/1027797, 2013. Visited: Dec 2013.

[82] VMware Knowledge Base (KB). Updating VMware tools fails with the error. http://kb.vmware.com/kb/2007298, 2013. Visited: Dec 2013.

[83] VMware Knowledge Base (KB). Updating VMware tools operating system specific package fails with dependency errors and driver issues on RHEL 6 and CentOS 6. http://kb.vmware.com/kb/2051322, 2013. Visited: Dec 2013.

[84] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, volume 36, pages 181–194, 2002.

[85] Carl A. Waldspurger. Default ESX configuration for balloon size limit. Personal communication, 2013.

[86] Wikipedia. Haswell microarchitecture – expected server features. http://en.wikipedia.org/wiki/Haswell_(microarchitecture)#Expected_Server_features, 2013. Visited: Dec 2013.

[87] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. *ACM SIGOPS Operating Systems Review (OSR)*, 43:27–36, 2009.

[88] Xiaowei Yang. Evaluation and enhancement to memory sharing and swapping in Xen 4.1. In *Xen Summit*, 2011. http://www-archive.xenproject.org/xensummit/xensummit_summer_2011.html.