Towards Swap-Free, Continuous Ballooning for Fast, Cloud-Based Virtual Machine Migrations

Kevin Alarcón Negy
Exostellar, Inc.
kevin@exostellar.io

Tycho Nightingale
Exostellar, Inc.
tycho@exostellar.io

Hakim Weatherspoon
Exostellar, Inc.
hweather@exostellar.io

Zhiming Shen
Exostellar, Inc.
zshen@exostellar.io

ABSTRACT

We have a production need to reduce the time for customers to live migrate their application virtual machine (VM) in the cloud. A single customer of ours migrates their nested, cloud-based, user virtual machines tens of thousands of times a month.

Ballooning is one technique for modifying the size of a virtual machine and has been used to speed up VM migration and increase VM consolidation. However, it has a significant risk: the ominous out-of-memory (OOM) error. The issue is that it is infeasible to use ballooning during high-risk scenarios, namely during giant memory spikes and during live migration, for fear of swapping or worse, OOM errors.

We advance the state of the art by optimizing the Linux balloon driver for VM migration in a non-overcommitted context, resulting in being able to handle both high-risk scenarios without relying on swapping and without causing OOM errors. We add a user-space continuous ballooning program that, in tandem with our balloon driver modifications, can handle memory spikes of hundreds of gigabytes, as well as survive an indefinite number of migrations.

In this paper, we discuss our minimal changes to Linux, describe our continuous ballooning program, and evaluate our now in-production, cloud solution on real-world applications. Our tests are designed to measure resilience in the face of several memory spikes and live migrations. In our tests, we add at most 8% overhead, yet can provide a migration speedup of at least 52% for giant VMs with memory intensive applications reaching almost 600 GB.

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698543

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Allocation / deallocation strategies**; **Virtual machines**.

KEYWORDS

Virtual Machine Migration, Ballooning, Memory Management, Cloud Computing

ACM Reference Format:
Kevin Alarcón Negy, Tycho Nightingale, Hakim Weatherspoon, and Zhiming Shen. 2024. Towards Swap-Free, Continuous Ballooning for Fast, Cloud-Based Virtual Machine Migrations. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3698038.3698543

1 INTRODUCTION

For decades, research on cloud computing and virtualization has focused on how to best utilize all the underlying physical hardware housing virtual machines. The goal has been to eliminate any idle CPU and memory resource usage. Memory has been a particularly difficult resource to handle, however. Although minimizing VM memory is desirable for many purposes, our own experience corroborated by others [8, 46] is that VMs are typically created with far more memory than they actually use. Even though memory spikes tend to be transient in nature [51], VMs are oversubscribed memory based on the fear that too little memory will cause unacceptable application degradation. Yet this leads to wasted, unused memory that could be better used to serve other VMs.

Most research on the topic has focused on memory minimization for the purposes of memory overcommitment, meaning maximizing the number of VMs running on top of a single virtual machine manager, or hypervisor, while sharing the same physical resources. We, however, focus on using memory minimization for a different purpose: optimizing VM migration in a non-overcommitted context.

Our system and solution stem from a direct need from our customers. Our industrial product gives customers the

ability to use cloud spot instances (cheap virtual machines without Service Level Agreements that can be terminated at any time with a short warning from cloud providers) for their workloads while trying to minimize their risk of termination. For instance, our product can support workloads that run continuously for days, weeks, even months with an achieved measured availability of three 9's (i.e. 1 out of 1000 jobs may terminate) which is sufficient for our customers and close enough to the cloud's four 9's [3].

To achieve a high availability, we use non-hardware accelerated, nested virtualization using the Xen hypervisor [52], which does not require cloud-provider support, to host a customer application inside a nested VM that is itself running inside of a cloud spot VM. To provide higher reliability, we preemptively migrate the nested VM containing the user application to a different cloud VM in order to prevent termination. Nested virtualization and nested migration is the only way, as far as we know, for a user to live-migrate a virtual machine instance without specific API support from a cloud provider. Therefore, the need for migrating nested VMs inside a cloud VM applies to all customers who wish to use our product.

Our product performs on the order of tens of thousands of live, nested migrations per customer per month (i.e., many hundreds of thousands to millions of migrations per customer per year). For instance, over a period of four months, a customer, typical of our user base, launched jobs in over 65,000 of our nested VMs in AWS spot instances, during which we executed an equivalent number of migrations. These migrations are entirely dependent on VM memory size, taking on average an extra minute for each additional hundred gigabytes sent over the network in our tests. The problem is, based on our experience, customers tend to overprovision memory by at least 2x, indicating a significant opportunity to speed up migration via memory minimization.

Ballooning is a common technique for minimizing VM memory, but existing solutions related to ballooning have many limitations. The KVM hypervisor and its VirtIO-balloon driver have a feature called free page reporting [40] that works well for minimizing VM memory by leveraging virtual paging, but KVM cannot be used for nested virtualization in a non-bare metal AWS instance [4]. Therefore, we looked at other ballooning solutions that we could employ in our nested virtualization context.

A few attempts at using ballooning specifically for migrations employ a one-time only ballooning operation before [25, 27] or during migration [39]. One-time only approaches are unacceptable for our context because they add the overhead of ballooning to overall migration time. Furthermore, as with most prior ballooning solutions, they typically require manual memory calculations and assume that memory usage will be consistent throughout the migration.

The most relevant academic work to ours automatically balloons out free memory to speed up migration, albeit in a non-traditional migration approach, called post-copy migration [16]. In spite of this difference in migration strategy (Xen uses pre-copy live migration), we share the same ultimate goal. However, their solution has a major limitation that we address in this paper. Ballooning can cause intense swapping or, worse, out-of-memory (OOM) errors, even in a non-overcommitted host. Swapping leads to terrible application degradation and requires significant disk space, sometimes the equivalent to the entire memory size of the VM. Meanwhile, OOM errors in Linux can trigger the dreaded OOM killer, a kernel thread that selects a high-memory usage program to kill, normally an important user-application that is memory-starved. The authors acknowledge this limitation by limiting balloon inflation to 95% of free memory in the hopes that the leftover free memory will stave off an OOM error. While that memory buffer may have been sufficient for tests of up to a single gigabyte, it is not sufficient in our context when we are migrating VMs with hundreds of GBs.

This major limitation of ballooning is not unique to [16]. Most prior work on the subject assume swapping and OOM errors are a given with ballooning and merely try to avoid them by setting overly-cautious memory targets. A few even use "inevitable" swapping and OOM errors as metrics for determining when to stop ballooning [7, 8, 44].

In this paper, we leverage our non-overcommitted context to modify the balloon driver so that we can aggressively balloon without requiring swap space and without triggering any OOM errors, even in the face of extreme memory spikes and multiple migrations. We design a continuous, self-ballooning mechanism around our swap-free, OOM-free balloon driver so that we can minimize VM memory throughout its lifetime. Our evaluation shows that our self-ballooning mechanism adds minimal (less than 8%) overhead in exchange for speeding up migrations by at least 52% for our real-world applications. Our solution is currently in production and has been OOM free for several months.

We make several contributions:

1) We modify the Linux balloon driver for Xen so that ballooning operations are OOM-free and do not cause swapping, as long as the hypervisor has free memory to give, even during intense memory spikes and live migrations.

2) We implement a continuous, self-ballooning, user-space program that balloons transparently, meaning other applications are not even aware ballooning has occurred and can allocate freely up to the max.

3) Unlike most prior work which focused on ballooning less than 1 GB, we evaluate our solution on large workloads of hundreds of gigabytes.

2 BACKGROUND AND MOTIVATION

In this section, we provide an overview of our target hypervisor, Xen, and its migration strategy. Then we discuss a few alternative approaches to dealing with spot termination warnings issued by cloud providers. Finally, we give a background on the balloon driver, the focus of our solution.

2.1 Xen

Xen [5] is a type-1, bare-metal hypervisor that hosts paravirtualized (modified) virtual machines. We use Xen because it can also run as a nested hypervisor in AWS on instances of any type [52], not just bare metal.

Xen refers to the VMs it hosts as domains. A single privileged domain from which Xen and other VMs can be controlled is referred to as domain 0, or dom0. All other unprivileged domains are considered guest user-domains, or domUs.

In line with traditional virtualization techniques, virtual memory in Xen has several layers. Guest domains have their own view of memory as being split into the usual virtual address space mapped to their own underlying physical address space. However, this guest physical address space is actually virtualized underneath by Xen which maps it to its own machine address space. Xen refers to a guest physical address (which is virtualized) as `pfn`, meaning "physical frame number", while a hypervisor machine address is referred to as `mfn`, meaning "machine frame number". We refer to the domU physical pages as either `pfns` or as "guest-physical" pages, and we refer to the Xen underlying page frames as "machine" pages or `mfns`.

Typically, domU migrations from one machine to another are triggered from dom0. Xen migrations employ a traditional pre-copy migration strategy, which is a form of live migration that minimizes downtime. This migration is split up into three phases. The first phase is the pre-copy phase, during which the guest domain continues running while its memory is forwarded over the network to the destination machine. Memory forwarding can take several rounds, the first round consisting of sending all pages residing in memory, and each subsequent round sending newly-dirtied pages. The second phase is called suspend-and-copy. As the name implies, while the VM is suspended, its remaining dirty pages are sent over along with other necessary data so that the domain can be resumed on the destination machine. Finally, during the resume phase, the domU VM is started and its old state on the source machine is cleaned up, which for domains with large amounts of memory can take tens of seconds. Although during this cleanup the VM is running as normal on the destination, for our evaluation, we consider migration time to end when the cleanup of the VM ends on the source. If migration fails for some reason during the pre-copy or suspend phase, the domain typically can be resumed safely on the source.

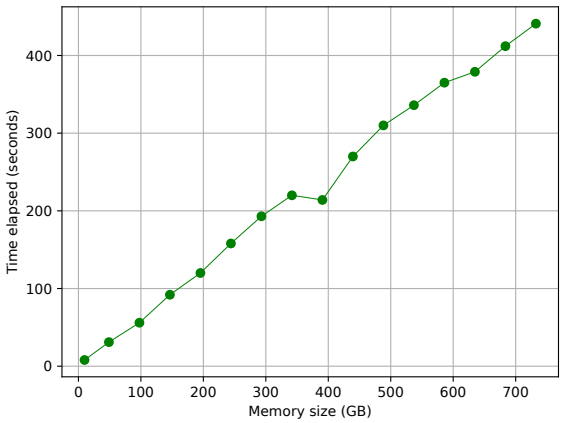

Figure 1: Average migration completion times based on idle VM memory size.

2.2 Minimizing Memory for Fast Migrations

Our focus is on decreasing overall migration time because we need to move large VMs out of spot instances to prevent them from being terminated and unrecoverable. To be clear, in this paper, we do not require migration times to be less than a cloud provider's termination warning period to consider our solution a success. Decreasing migration time in general is a significant benefit in increasing VM reliability. To this end, we view memory minimization, the focus of this paper, as one tool among many to help prevent termination.

Figure 1 shows the relation between VM memory sizes and migration completion times. For each memory size, we create a new, idle virtual machine. We then migrate ten times and calculate the average migration time. These averages are lower bounds for migration times; VMs with running processes will have more dirty pages per round and need extra time to complete a migration. Nonetheless, the figure shows that memory is linearly related to migration time and that memory minimization is a viable way of optimizing migrations.

To decrease memory dynamically, we looked into swap-based techniques for reducing memory, but they presented several drawbacks. Although we could decrease memory enough to force the VM to engage in traditional swapping, the act of writing pages to disk is unacceptably slow compared to in-memory operations. Furthermore, traditional swapping is unable to completely prevent out-of-memory

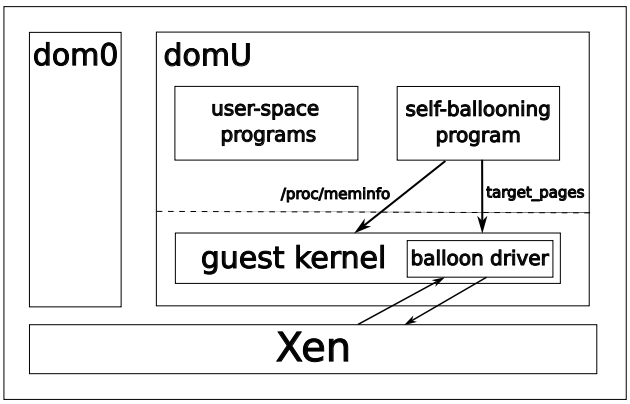

Figure 2: Our continuous, ballooning architecture. Shows one cloud instance with the Xen hypervisor, a dom0, and a domU VM. The self-ballooning program in user space of domU collects data from /proc/meminfo and sets the balloon driver memory target. The balloon driver then interacts with Xen to either inflate of deflate the balloon.

(OOM) errors if memory is unswappable, such as kernel or pinned memory.

To avoid disk usage, we could have used a hypervisor-level swapping solution to swap memory to free hypervisor memory, but hypervisor-level swapping is known to cause the *double-page problem* [34]. This is when the hypervisor swaps out a guest physical page and then the guest OS (which thinks the page is still in its own memory) decides to swap out that same page, unintentionally causing the hypervisor to bring back the page just for it to be swapped out again. More importantly, this would not decrease the overall memory that must be forwarded during a migration.

Additionally, network swapping [51] was another option for sending memory to a remote host, but instead of using network resources for sending memory over the network and adding an extra copy to migration (one copy to swap, one copy back to the VM when it is page faulted in), we instead focus on minimizing the number of forwarded pages during a migration using the balloon driver.

Finally, other optimizations could be used to minimize in-use memory sent over the network, such as run-length encoding and ignoring disk-backed pages, such as in [35]. These optimizations would require more complex implementations and focus instead on leveraging the existing balloon driver to minimize unused memory. We leave these optimizations for future work.

2.3 Ballooning

Our goal is to design a continuous ballooning solution for use in speeding up migrations. Ballooning was introduced by Waldspurger in 2002 [48]. The concept behind ballooning is that within the guest kernel, a "balloon" driver interacts with the underlying hypervisor. This driver can reduce the VM memory by *inflating* the balloon, which involves requesting memory from the guest OS memory allocator as if it were a normal process and then notifying the hypervisor that the memory can be unmapped and reclaimed for other purposes. During times of memory pressure, the balloon driver can *deflate* by requesting the hypervisor to provide page frames for the ballooned pages and then returning those pages to the guest OS using normal *free* memory operations. The advantage of ballooning is that memory can be modified dynamically at runtime, as opposed to having to shutdown the VM and reboot it with a new memory size. It also lets the OS decide which memory to balloon instead of an external program making the decision with limited knowledge of the internal memory usage.

Although ballooning is a useful technique for minimizing memory when memory usage is stable, overly-aggressive ballooning and/or spiky memory usage is known to cause intense swapping or OOM errors. To leverage ballooning to its full extent, we implement a ballooning mechanism that completely prevents OOM errors without swap space as long the hypervisor has free memory to give. In line with public cloud providers, which to our knowledge, never overcommit memory, our assumption is that we are never memory overcommitted at the hypervisor level and therefore Xen always has enough memory to satisfy VM memory spikes up to their maximum memory.

3 BALLOONING ARCHITECTURE

Figure 2 provides an architecture overview showing a cloud instance running the Xen hypervisor underneath two VMs, dom0 and a domU. The domU has its own guest kernel that contains our modified balloon driver. In user space, our self-ballooning thread gathers memory information and communicates memory targets to the balloon driver.

In this section we discuss our kernel and balloon driver modifications for a swap-free, OOM-free ballooning tool. Then we highlight our self-ballooning user space thread and describe its behavior.

3.1 Kernel changes

3.1.1 Handling memory spikes. Ideally, ballooning would always return memory fast enough to prevent OOM errors, especially given that there is a gap between the time the OS commits memory to a process and when the process actually uses the memory, giving ballooning a potential head start. In our experience, however, our attempts at making ballooning faster were futile. We tried to increase the amount of pages the balloon driver handled at a time, at one point even

making ballooning multithreaded. Although we saw some improvement, the performance was not reliable enough to prevent the worst memory spikes from triggering errors. Another naive approach was to try to slow down memory requests to give ballooning a chance to catch up during moments of intense memory pressure and failing requests, but these efforts were fruitless, not to mention introducing unwanted application slowdown.

Instead of trying to speed ballooning up to win in the race against memory spikes, we make a fundamental decision to make the balloon driver act as a memory allocator itself. Memory requests in the standard Linux memory allocator have a fast path [19] and a slow path [20] for being satisfied, each with more desperate attempts at being satisfied before ending up calling the OOM killer to free up memory. We modify the slow path to add a hook into our balloon driver. This way, most memory can be satisfied by the fast path, but failing requests that reach the slow path can be satisfied by the balloon driver. The hook is added before any swapping or direct reclaim [10] and before any calls to the OOM killer. This way the balloon driver can satisfy requests before any disk usage is needed and OOM errors only occur if the VM (balloon driver and main memory allocator) is truly out of memory.

The failing requests hook into the balloon driver by calling our new function, `get_page_from_balloon()`, from the memory allocator. The only parameter needed is the page order of the request, indicating the number of pages requested (2^{order} pages). The function then deflates the balloon to satisfy the request and passes back a `struct page*` pointing to the first page of the request, mimicking exactly how the standard memory allocator behaves.

Although the solution seems simple on the surface, we made small, but significant changes to the balloon driver to support this new ballooning behavior. We continue the section describing some of the hurdles and respective solutions.

3.1.2 Circular memory allocations. Since our balloon driver can handle failing memory requests from the memory allocator, we need to make sure it is not possible for balloon inflation to submit allocation requests that fail and then end up back in the balloon driver. If this happens, the balloon driver could pass memory back to itself repeatedly in an attempt to shrink memory. To prevent this, we only call our balloon driver function from the memory allocator if the memory request flags are not the same as the `GFP_BALLOON` flag that is used for balloon deflation. `GFP_BALLOON` is the combination of the `GFP` flags `HIGHUSER`, `NOWARN`, `NORETRY`, and `NOMEMALLOC`, which are meant to be low priority flags.

3.1.3 Contiguous, high-order requests. Now that the balloon driver acts as a memory allocator, the existing data structure for storing ballooned pages, a single linked list, is insufficient for handling memory requests. The list is used to add and remove pages one at a time. As a result, pages on the list are not guaranteed to be contiguous. This presents an issue for memory requests of more than one page that are expected to be contiguous. Although most page requests tend to be of order 0 (a single page), it is possible for high order memory requests to come in and cause kernel errors down the line when processes access memory as if the pages they have received are contiguous.

We considered various solutions for satisfying contiguous page requests. The naive solution was a bounded linear search through the ballooned pages list for contiguous pages. This was insufficient to satisfy all requests, especially at higher orders of 4 or more (16 pages+). Attempts at sorting the ballooned pages list helped with finding contiguous pages, but resulted in all ballooning operations stalling for several seconds while the list was sorted.

To solve the issue, we changed the data structure for storing ballooned pages to a buddy data structure exactly like the one used in the Linux memory allocator. The buddy data structure is an array of lists, where each index represents an order of the pages in the corresponding list. For instance, at index 2, the first page in the subsequent list is the first page of a contiguous set of order 2 (4 pages). In the best case, contiguous pages of the desired order can be pulled off to satisfy the request immediately. If the correct order is not found, each larger order must be searched incrementally. Once a set of pages larger than the requested order is found, the pages must be broken into chunks. All the excess chunks of pages are placed back in the data structure at their new smaller index. Conversely, when pages are added into the buddy structure, such as during inflation, they can be placed directly into the index corresponding with their order. However, if the neighboring set of contiguous pages, i.e. their "buddy", is present in the structure, these two sets can be merged to create a set that is one order higher. This logic repeats with the goal of storing the largest contiguous sets of pages possible.

3.1.4 Hugepage compatibility. Our in-house Linux kernel and Xen hypervisor have support for transparent hugepages which means pages can automatically be combined into hugepages without the application knowing, providing optimized TLB lookup [22]. However, the existing balloon driver does not support hugepages [50]. To support this feature, we modify the normal balloon driver operations of inflating and deflating memory to operate at hugepage granularity, which is a set of 512 pages or 2 MB. Otherwise, during deflation, the balloon driver would return individual pages (4 KB each) that the OS would then try to promote to hugepages without the guarantee that their underlying machine addresses are contiguous, causing memory errors. Note that this contiguous

guarantee is needed at the machine address level, as opposed to the guest-physical address contiguity needed in 3.1.3.

Therefore, we modify the balloon driver inflation and deflation functions to allocate and free a single hugepage at a time. Although modifying inflation granularity is not strictly necessary, we modify it anyway for parity with deflation and to minimize fragmentation. In fact, despite [43] stating that the biggest issue with ballooning is the fragmentation it causes, our modification for hugepages makes the balloon driver far less likely to cause system-wide fragmentation than before.

The largest modification for supporting hugepages is in our `get_page_from_balloon()` function. Although normal balloon operations now handle hugepage granularity, the `get_page_from_balloon()` function must still handle memory requests of any size. Since all pages stored in the balloon are hugepages or bigger, this could lead to extra overhead when a small request incurs the penalty of processing an entire hugepage. Further, since this function returns only the memory requested, the function must handle any excess pages from the hugepage that have been processed but not yet freed.

One option for handling these excess pages is to just free them while completing a request. This would mean that each small order request would have to deal with the overhead of freeing all excess pages from a hugepage one by one. Given how critical this function is to satisfy already failing memory requests, this is unacceptable. Instead, we opt to hold onto remaining pages which have already been mapped by Xen and put them into a separate buddy data structure we name `balloon_cache`. Any subsequent small order request first checks if they can be satisfied immediately with memory from this secondary buddy struct.

Although adding a secondary structure to hold excess pages introduces additional fragmentation, this fragmentation is minimal. Ultimately, the maximum amount of pages that could be in this `balloon_cache` is 511 pages, one less than a complete hugepage. This is because there will only be at most one buddy at each index of the cache, up to order 8. In this state, the next small order request will not cause further fragmentation since it will be satisfied by one of the existing buddies. Furthermore, in the worst case, each buddy in the cache will correspond to a separate hugepage. With the cache being limited to order < 9, this means at most 10 hugepages could be fragmented between the balloon driver and the memory allocator, equivalent to 20 MB. This amount is insignificant compared to modern-day memory sizes.

We note that although our hugepage balloon driver modifications only introduce minimal fragmentation, it is generally possible for the memory allocator itself to be fragmented because of normal process allocations, so much so that the balloon driver is unable to shrink memory. For instance, we noticed that for a simple multithreaded `malloc` program, it was possible for some threads to have freed their portion of memory and for other threads to be holding onto their own portion, causing fragmentation of hugepages. As such, there were short periods of time where the balloon driver was unable to shrink memory despite total free memory clearly being enough to shrink. Normally, this resolved itself after several seconds, when more memory was freed. However, the worst case would be if processes never relinquished their memory, making further shrinking impossible. There may be some ways to alleviate this issue by manually triggering memory compaction, but we did not explore these options. For the time being, we accept this limitation since it only makes ballooning less effective; it does not change application performance.

3.1.5 Lock usage. The vanilla balloon driver used one mutex for protecting both its ballooned pages list and its stat-tracking struct. A mutex was sufficient because the existing driver was not used in interrupt contexts and could safely sleep. Now that our driver directly handles memory requests sent from the memory allocator, sleeping in the `get_page_from_balloon()` function with the existing mutex becomes unacceptable. The function could receive atomic memory requests or could be called in an interrupt context, both unable to sleep. Allowing these threads to sleep could cause deadlocking or other serious system failures.

To handle these non-sleepable contexts, we add interrupt-safe spinlocks to protect the balloon buddy data structure. Although many memory requests could overload the balloon driver and burn up CPU cycles from waiting on the spinlock, the function satisfies memory requests quickly (or fails quickly if there is no memory), so all requests will eventually be handled. More typically, when there is no memory pressure, the main balloon kernel thread will never be blocked by the spinlock.

3.2 Continuous ballooning design decisions

To accompany our modified balloon driver, we design a continuous, self-ballooning process to constantly monitor memory usage and set memory targets accordingly. Typically, the balloon driver is controlled from dom0 by passing a new target to the guest through a shared database for communication between Xen and its guests. Putting our self-ballooning process in dom0 could leverage this existing mechanism to set new memory targets, but this would require dom0 and Xen to be privy to internal memory data from the guest, an issue usually referred to as the *semantic gap* [18, 29]. Additionally, during migrations we only migrate the guest domain; a self-ballooning process sitting in dom0 would not be migrated along with the domain it controls.

We choose for our self-ballooning process to reside in the user-space of the guest whose memory needs to be minimized. This allows us to leverage internal metrics from `/proc/meminfo` without extra modifications. It also means that our process is coupled with the VM during migration, which is conceptually and implementation-wise easier.

3.2.1 Exposing the balloon driver to user-space. To accurately make memory targeting decisions in user-space and to trigger ballooning in the driver, we needed an interface between the two layers. We leverage an existing `/proc/sys/xen/balloon/` directory and add three variables: `max_pages`, `current_pages`, and `target_pages`. The first two read-only variables allow our user-space process to have an accurate view of system memory. The last variable is a read/write variable that is used to set new memory targets and wake up the balloon driver thread. We limit `target_pages` to a maximum of `max_pages` and a minimum of one, so that a malicious user will only harm themself if they set memory too low and also stay within the VM maximum if they set memory too high. However, normal applications will not know to look for these variables at all.

3.2.2 Setting memory targets. We set our self-ballooning program to recalculate the memory target every second. We feel this is a good compromise between not so frequent that our program takes up excessive CPU time, but not so infrequent that it cannot react within a reasonable time to handle memory changes. With our balloon driver changes, however, it is not vital for the self-ballooning mechanism to respond faster than a second to handle memory spikes; any failing memory requests will be directly satisfied by the balloon driver anyway.

We use a simple policy for determining memory targets. First, we turn on OS-level memory-overcommitting so that programs are able to allocate memory even if current system memory is small. As long as the overcommitment is less than the maximum memory, the allocation should be satisfied. As such, our target is set to current memory minus available memory plus overcommitted memory. Current memory is read from the balloon `/proc/` variable, `current_pages`. Available memory comes from `meminfo`, and is the memory available for applications without swapping [21]. We calculate overcommitted memory as `Committed_AS` minus active, inactive, and swapped memory. `Committed_AS` is the amount of memory theoretically needed in RAM to avoid OOM errors [14]. In other words, it is all the memory in the system that is currently allocated and also overcommitted (reserved, but not yet allocated). Therefore we need to remove active, inactive, and swapped memory to calculate overcommitted memory.

3.2.3 Transparent ballooning. A unique feature we implement for our ballooning mechanism is what we call *transparent ballooning*. Typically, memory changes from the balloon driver are reflected in `/proc/meminfo`. Some programs dynamically decide how much memory to allocate based on this information. Since we can handle memory spikes without OOM errors, we make ballooning transparent, meaning we modify `meminfo` to show the full memory that the VM has access to without showing the memory that has been ballooned out. That way apps can behave as if they have access to the full memory and do not restrict their own memory usage.

3.2.4 Blocking/non-blocking. When tackling the problem, initially we tried the suggestions in [25, 27], i.e. one major balloon inflation right before a migration. This proved problematic for several reasons. We made setting the memory target a blocking operation until the balloon driver reached its memory target. This led ballooning to be slow to react to memory changes. When memory spikes came in, we would set the target and then wait for seconds or even minutes while memory increased, even if the memory spike was transient. It also meant that setting the memory was not guaranteed to terminate if the target could not be reached because of continuous memory fluctuations.

We decided to make setting the memory target a non-blocking operation that would merely send a wake up signal to the balloon thread. This allowed our self-ballooning program to respond immediately to memory changes in case the changes were short-lived.

3.2.5 Migration obstacles. Our version of Xen does not support shrinking during migration. During testing, we noticed that just a single migration caused about 20 GB to go missing if the entire migration was spent inflating the balloon. The missing memory would appear when reading the guest memory from the dom0 Xen tool, but the guest itself did not have internal records of the memory. This effect was compounded across multiple migrations.

The issue is that when pages are ballooned out on the source host during migration, the migration mechanism does not notify the destination host. If newly-ballooned pages were previously sent to the destination in an earlier pre-copy migration round, the destination will not be notified that these pages should now be removed from the pool of incoming-VM memory. This leaves the host with allocated pages for the VM that the VM itself thinks are ballooned away.

To preemptively avoid any memory discrepancy due to shrinking, we disable shrinking as soon as a migration begins. Only after the migration has completed successfully do we re-enable shrinking. This means guest memory minimization during migration loses some efficiency, especially for large

guests where migration may take several minutes, with each minute having been able to shrink about 84 GB of memory. We intend to fix this bug in a future iteration of our solution.

4 IMPLEMENTATION

We implement our x86 balloon driver modifications on our in-house kernel which is based on Linux kernel version 4.19.49. However, our modifications are applicable to upstream Linux, as of version 6.7, and are not affected by the recent kernel features MGLRU [12] and folios [11], since ballooning is page replacement independent and, as far as we know, needs no adjustments for folios. Our changes overall are minimal, requiring about 700 additional lines, mostly to the balloon driver file. No modifications to Xen itself were required. We implement our user-space, self-ballooning program as a Golang program (about 350 LOC).

All our experiments were run using AWS EC2 r5.24xlarge instances, which have a total of 96 CPUs and 768GB of memory. We install our customized Xen 4.12 hypervisor and use Centos7 based VMs with our kernels. Our dom0 VM always has 8 CPUs and 4096 MB of memory. We assume only one domU exists on each instance at a time. Unless otherwise mentioned, we initialize all domU VMs with about 732 GB (750000 MB)[1] and 88 CPUs. We choose these starting parameters because we assume that real-world users will submit various jobs with different memory requirements that can take advantage of parallelism. We also assume users do not know beforehand the memory usage of their programs. Therefore we maximize the number of CPUs and memory available to applications. We set the frequency of our memory target updates to every second. During each update, we log the current memory size (based on our exposed balloon driver variable, `current_pages`) and our latest memory target.

5 EVALUATION

To evaluate our ballooning implementation, we attempt to answer four questions.

1) How much memory can we safely minimize?

2) How well can we handle memory spikes of hundreds of GBs?

3) How much overhead is added from constant minimization?

4) How much speedup do we get because of ballooning in terms of migration and overall application performance?

5.1 How much can we safely minimize?

Throughout the history of attempts at dynamic memory adjustments, calculating an appropriate memory target, also framed as calculating the working set size, was seen as essential for maintaining high performance and not causing debilitating system failures. With our ballooning modifications, however, we asked *how far could we truly minimize memory?* Our modifications were designed to satisfy failing requests and prevent grave system memory issues. Therefore, we hypothesized that we could set our target extremely low without issue.

To test our theory, we started a new VM with 732 GB and set the memory target to an absurdly low 4KB (1 page) of memory, despite our self-ballooning program suggesting 19.5 GB as the minimum. Interestingly, the VM memory decreased until around 17.5 GB and then stopped. From there, the VM became noticeably slower, but amazingly did not crash and was still responsive to our terminal commands. Each `ls` or `cat` command immediately had failing memory requests that went to our balloon driver to be satisfied, but all commands progressed and ultimately completed. We then increased the memory half a GB at a time and found that the VM returned to normal (i.e., became responsive to simple Linux commands without triggering any failing memory requests handled by the balloon driver) at 18.5 GB, one GB more than the true minimum and one less than our self-ballooning calculated minimum.

Takeaway: From this test, we draw several conclusions. The first is that for a VM that is 732 GB, Linux requires almost 20 GB to run. About 11 GB of that is just for the `struct page` metadata that describes all pages in memory. Although it is expected that not all system memory will be usable by applications, this minimum memory requirement is surprisingly high and should be accounted for when configuring appropriately-sized VMs to run memory-intensive applications. The second is that our self-ballooning calculation is fairly accurate, relying only on `/proc/meminfo` and a few exposed variables from the balloon driver to find a target. Finally, we learn that even if our calculations had been wrong (i.e. estimating too little memory to satisfy all applications), our balloon driver resiliently brings the memory back to appropriate levels. This makes calculating memory targeting trivial; choose any low memory target knowing that the balloon driver will protect the VM from crashes if the calculation was too aggressive.

5.2 Extreme memory spikes

Next, we create a simple program that is designed to test our code against extreme memory spiking. This program spawns 27 threads that each allocates 25 GB of memory for a total of 675 GB allocated. Each thread uses `malloc` to reserve its memory. Then to ensure the memory is actually allocated, each thread writes to one byte per page and then holds onto the memory for a minute. Finally, the threads

[1] Our dom0 Xen tool uses MB for all commands. Therefore, for ease we start VMs at 750000 MB and collect all measurements in MB, but in this paper we convert all results to GB for legibility.

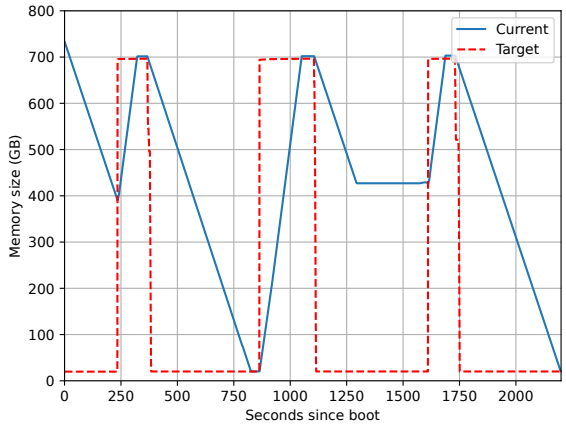

Figure 3: Simple memory spike program with a single migration. Shows both the current memory of the VM and the memory target set by our self-ballooning program.

free their memory, sleep for eight minutes, and then restart the cycle. We run this test for 24 hours with migrations throughout, each set to start 15 minutes after the end of the previous migration.

Figure 3 shows just the first 2200 seconds (about 36 minutes) of the test. This captures the VM memory size from bootup to three full memory spikes, including a single migration. To start, the VM begins with its full memory allotment of 732 GB. Immediately our self-ballooning tool sets the memory target for the idle VM to 19.5 GB.

After a few minutes, we start the memory spike program and the minimum memory target rises to about 700 GB. The balloon driver immediately reverses course and returns enough memory to satisfy the VM. It deflates the balloon at a rate of 3.7 GB/s, including both normal balloon deflation and failing memory requests satisfied directly by the balloon driver. Eventually, all threads are able to complete at roughly the same time, causing the memory target to drop immediately. The threads then sleep for eight minutes. With the threads asleep, the VM is essentially idle and the balloon driver continues shrinking memory at a rate of about 1.4 GB/s. The second memory spike occurs right after the balloon driver is able to reach its idle memory target. Immediately, the target reflects the updated `Committed_AS` and memory begins increasing. Between the second and third spike is the first migration. As mentioned in section 3.2.5, Xen migration is not equipped to handle balloon inflation and we disable shrinking during migration. This makes it clear when the migration starts: when the current memory flatlines even though the target is lower. The migration continues for about two minutes without being able to shrink. This shows an opportunity for future migration optimization. Nonetheless, the migration completes right before the third memory spike. This time the threads do not all complete at the same time because migration slows down the VM causing some threads to wait slightly more for page allocations than others. This explains the slight break in the target decrease at the end of the spike.

Takeaway: Given the cyclical nature of this test, the rest of the data demonstrates the same behavior, except for the spike that occurred after the VM initialized. Nonetheless, after 24 hours, no OOM error messages appeared in the kernel logs, nor did the application fail at all. This demonstrates that our ballooning mechanism is able to handle extreme memory spikes despite the balloon driver typically being too slow to handle such intense memory pressure situations.

5.3 Benchmarks

We tested our solution on the Phoronix Test Suite (PTS) [41]. We used Phoronix version 10.8.4, which is the latest release from 2022. We test on VMs that have 2 CPU cores and restrict memory to 12 GB. Although we focus on benchmarks that are memory related, we find that none surpass more than a few GBs. Therefore, we use these benchmarks only to compare self-ballooning overheads with no ballooning and do not test with migrations.

The first Phoronix benchmark we use is Memcached. Memcached is an in-memory key-value store and the Phoronix test tests different ratios of set and get operations. We tested on all set:get test ratios, including 1:1, 1:5, 5:1, 1:10, and 1:100. Based on the metric of operations per second, we find that there is essentially no difference between having our self-ballooning on or off. The average difference in operations per second across all tests was -0.04%.

The Phoronix Memory test suite is a more general set of memory tests that test memory speed and bandwidth. Here, we test on all included benchmarks. Although each test used a different performance metric, including MB/s and total time, no individual test differed by more than 0.35% between self-ballooning and native. The average percent difference across all tests in the memory suite amounted to -0.2%.

Takeaway: For these benchmarks, overall we see virtually no overhead from self-ballooning. Given the low memory usage of the tests, we turn to evaluating our system on larger tests on the order of hundreds of gigabytes.

5.4 Real-world applications

Although our solution has been evaluated on many types of applications such as web servers, Bitcoin miners and validators, electronic design automation, and computer aided

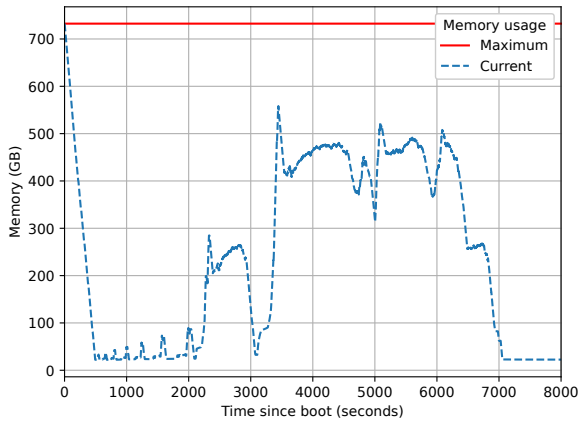

Figure 4: EM_sim memory usage compared to the maximum memory.

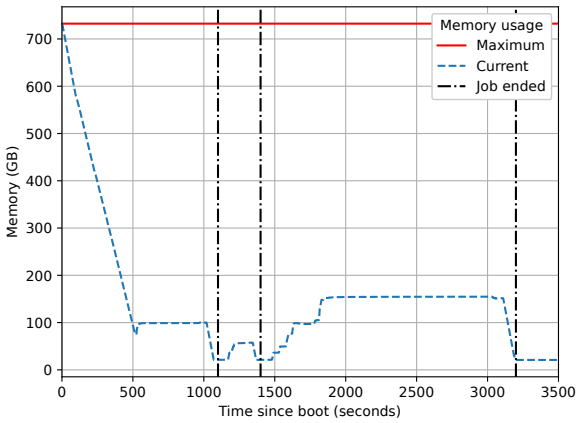

Figure 5: Memory usage of fluid_sim when run as three separate jobs with different input compared to the maximum memory.

design, to name a few, for this evaluation we focus on proprietary high performance application (HPC) workloads taken directly from our customers since these use significant amounts of memory and CPUs, which represent the most challenging, real-world workloads. The datasets and configuration settings we used for each workload were from our customers, as well.

Thus, for our next tests, we use two proprietary programs. One is an electromagnetic (EM) simulation. The other is a fluid simulation software. We refer to each as EM_sim and fluid_sim, respectively. We setup each program to run on one VM, utilize all 88 CPUs, and use up to 95% of system memory. Thus, these programs share CPU time with our ballooning code. We run each test three times to calculate averages.

First, we aim to determine the memory usage of these real-world applications to see how effective ballooning could be. Figure 4 shows the memory usage of EM_sim during one run, measured by using the ballooning code to shrink the VM and log its current memory. The program goes through two phases which have different memory requirements. The first phase ends at about 2200 seconds in and fluctuates between small memory spikes and virtually no memory usage. The peak usage during this phase is about 88 GB. The second phase consumes much more memory, using as much as 562 GB. Thus for a 732 GB VM hosting this program, our ballooning code could shrink between 23-97% memory, a wide range. Even if we had started the VM slightly above its peak memory usage, as is common when application memory requirements are known beforehand, in phase one we still could have minimized at least 84% memory. In phase two, although the memory usage is much greater, there is still ample opportunity for minimizing memory since the peak only lasts a few seconds and is reached only once during the run.

If users always ran one application per VM and happened to know their application's peak memory usage, perhaps they could always start their VM with just enough memory to handle its peak usage. In reality, we have seen anecdotally that many companies run workload managers that perform job scheduling, often running multiple jobs back to back on the same VM. Furthermore, even when the same application is run with different input data, its memory usage may vary. Figure 5 shows such a scenario. Here we run three fluid_sim tests using different customized input as if we were using a workload manager. Again, these tests are configured to utilize all 88 CPUs. This program has a more stable memory usage than the EM_sim program; fluid_sim allocates memory and tends to hold onto the memory for the duration of the program. In the first job, it allocates about 98 GB and finishes after 1000 seconds. After a two-minute gap, we start the second job, which finishes much faster, about two minutes. Its peak memory usage is about 58 GB. Finally, the third run has the highest memory usage of about 150 GB. Despite maximizing CPU usage, collectively these programs use less than a quarter of the maximum memory. The takeaway here is that although some programs do not have very spiky memory usage patterns, if a workload manager schedules jobs on the same VM, there is still ample opportunity for memory minimization.

Finally, we add migrations to these tests. We migrate periodically with 15 minutes between each migration, which is more frequent than what we expect to run in production. We compare our EM_sim program with our ballooning code

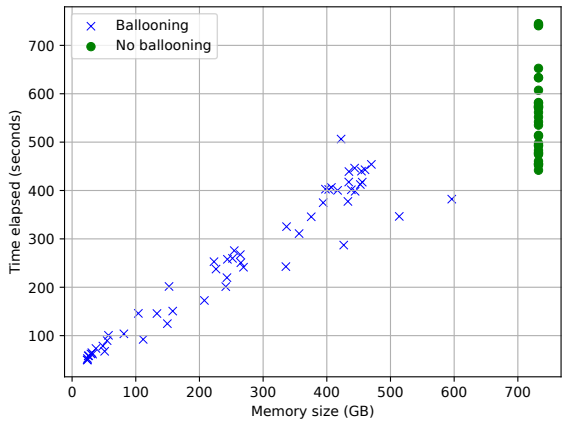

Figure 6: Migration completion times based on VM memory size for EM_sim.

turned off and then with it turned on. Figure 6 shows a scatterplot of all migrations based on memory usage at the start of the migration and how much time the migration took to complete. The scatterplot shows a very similar trend to the expected migration times from Figure 1, albeit with more variation. This variation could be from migrations that experienced memory spikes which increased the amount of memory sent over the network. Alternatively, a busy VM will dirty more pages during a migration, leading to the same effect. Nonetheless, the trend in the figure is still apparent. The worst-performing migrations during the ballooning test were on par with the fastest performing migrations in the non-ballooning test, taking about 7.5 minutes. However, the best migrations took under two minutes, typically when the VM memory size was less than 100 GB. We observed similar results with the fluid_sim tests. Thus it is clear that our ballooning mechanism is an effective tool for optimizing migrations.

Table 1 summarizes our findings across the EM_sim program and several runs of the fluid_sim application with different input files. EM_sim has the highest ballooning overhead when there are no migrations (7.8%) because it has the most varied memory usage including several large memory spikes. During these spikes, many memory requests must go through the slow path to be satisfied in the balloon driver. It also has the lowest average migration speedup of 52.6% because it uses the most memory of all the test programs. We also show the migration speed up between the fastest migration when our code is disabled and when it is enabled. For instance, in the EM_sim the fastest migration in both cases was when the memory usage is practically zero. When our code is disabled, the fastest migration took 7 minutes and 35 seconds; with our code enabled, the fastest migration took 49 seconds total. Overall, that is an 89% speedup for the fastest migration case.

Despite the less than 8% overhead with our modifications when there is no migration at all, when migrations do occur, we see the opposite; there is actually a performance increase of about 3.2% overall. The increase is from less time spent in a migration, when the VM typically slows down. This performance increase is amplified for fluid_sim tests (closer to 10%) because its memory usage is lower, resulting in faster migrations when self-ballooning is on. It also has very few memory spikes so ballooning itself causes less overhead. In general, however, measuring speedup of applications experiencing migration is difficult because speedup is highly dependent on the number of migrations executed during the application lifetime and its memory usage during a migration.

Takeaway: Overall, we find that in exchange for less than 8% overhead in our workloads, we can effectively decrease migration times by at least 52%. Furthermore, in spite of the overhead caused by our ballooning code, when migrations are expected, the gains from our optimizations can actually outweigh the overhead and lead to a slight application performance boost.

6 DISCUSSION

We have demonstrated that our ballooning modifications can handle memory spikes of several hundreds of gigabytes by acting as a memory allocator itself. As long as the hypervisor has free memory to satisfy the memory demands, we do not require swap space and yet will not trigger any out-of-memory errors. Furthermore, our continuous, self-ballooning tool is able to respond to memory fluctuations and set appropriate memory targets across the lifetime of a VM. In this section, we discuss generalizability, limitations, and future work.

6.1 Generalizability

First, we discuss the generalizability of our findings. Regarding overhead, we know that most of our overhead comes during memory spikes, when memory is in such high demand that the normal balloon thread alone cannot keep up, causing failing memory requests to call into our balloon driver to be satisfied. Therefore, applications that have more frequent intense memory spikes will have higher overhead. As such, the 8% overhead for the EM_sim (without migrations) is, unsurprisingly, the highest we measured since that test has the most memory spikes and largest memory usage of our real workloads.

However, we anticipate this overhead to be tolerable or even diminished for a wide variety of applications when migrations are expected, as it is in our organization. This is

Table 1: Summary of ballooning results. We omit migration results for fluid_sim_C because the program is short-lived.

	Avg. mem. usage	Peak mem. usage	Overhead w/o migr.	Overhead w/ migr.	Avg. migr. speedup	Fastest migr. speedup
EM_sim	264.5 GB	562.5 GB	7.80%	-3.2%	52.60%	89%
fluid_sim_A	121 GB	153.5 GB	1.60%	-9.6%	64%	72.70%
fluid_sim_B	78 GB	100 GB	3%	-11.8 %*	68.7%	85.4%
fluid_sim_C	47 GB	57 GB	4.60%	-	-	-

* Tested with only one migration during each run.

because we expect that even when users know their peak application memory usage, it is not a given that this peak will reflect the average, particularly when running batch jobs. Thus, we expect ample opportunity for memory minimization in a variety of contexts.

Second, we consider whether our ballooning solution could be beneficial in a more normal use case. In the general case, ballooning is typically done with one-time only operations outside of migration (hence our trouble in 3.2.5). This normally requires manually calculating a safe memory size that does not risk an out-of-memory error. Even with just our minimal kernel balloon driver modifications, one could set memory targets fearlessly; an overly aggressive target will be gracefully stabilized to an appropriate memory level almost immediately. However, for more exact target calculations, our user space process accurately and automatically determines appropriate minimum memory levels. In the general case, our solution makes ballooning safer and automatic.

Finally, we acknowledge that our solution, as presented, is not generalizable to the overcommitted case. That is, our assumption that the hypervisor always has extra memory to satisfy balloon requests is not true in an overcommitted context. Specifically when multiple VMs reside on a single machine, memory may be overcommitted and the hypervisor will be unable to satisfy requests. In this case, swap space may still be necessary to prevent failing applications. Of course, some memory overcommitment solutions have maintained a percentage of free hypervisor memory specifically to prevent this scenario. Nonetheless, even if our solution alone does not solve memory overcommitment outright, we envision our modifications could be used as one piece of a larger solution.

6.2 Limitations

One limitation of our solution is that we trust other user-space applications not to override our memory targets through `/proc/` interface. Fortunately, a bad actor can only harm itself or make our ballooning solution ineffective by hogging maximum memory and preventing minimization. For situations when user-space cannot be trusted, the self-ballooning program could be placed in dom0 and use the Xen tool to set memory targets instead. However, this would require extra modifications to Xen so that the self-ballooning program could be coupled with the VM during migrations.

Another limitation is that our solution may not be viable for applications that benefit from the Linux page cache. Linux is known to use excess memory as a cache for improving future memory accesses. To accommodate this feature, our memory target calculation could be modified so that we add a configurable buffer size to the target. More evaluation is needed to find the optimal buffer size for improving application performance while still keeping memory low.

6.3 Future Work

We leave a few optimizations to our solution as future work. For even faster ballooning, at the expense of CPU usage, the balloon driver could be made multithreaded. As mentioned, Xen migration could be modified to handle balloon inflation throughout which would optimize migration even more. And finally, we could make VM bootup time faster by having a different maximum memory than starting memory, instead of booting with the maximum memory and then shrinking.

7 RELATED WORK

Virtualization, memory minimization, and ballooning have been studied for several decades. In this section we break down prior work on memory management solutions, ballooning, and industrial attempts at self-ballooning.

7.1 Memory management solutions

Memory management has been looked at from various levels of the virtualization stack. Some have looked at it from a cloud-platform level to control VMs on multiple physical servers [36, 37], others focus on implementing a scheduler for a single physical host managing multiple VMs [1, 15, 23, 34, 54, 55]. At a higher level, some target memory minimization within each VM at the guest kernel level [8]. Finally, a few papers have argued that the application level is the best place for memory decisions since applications have the most knowledge of their own needs [17, 42]. Our work fits into literature focusing on guest kernel modifications for both memory data collection and management because we do

not want to modify applications directly, but we are able to modify the guest kernels that we use.

A major problem for memory management is determining appropriate memory targets for VMs. Many papers specifically focus on improving memory targeting algorithms [24, 30, 38, 45, 49]. We focus on making the underlying ballooning mechanism resilient enough that targeting becomes a low-risk, trivial operation and therefore can use a simple heuristic to maintain minimum memory sizes.

7.2 Ballooning

Ballooning was introduced in 2002 by Waldspurger for use in the VMWare ESX hypervisor [48]. They use statistical sampling of memory accesses to determine memory targets during periods of overcommitment. For safety, they require swap space of size `max_memory - min_memory`, which for our tests would lead to hundreds of GBs of swap space. Since then, many papers have tried to handle memory overcommitment. Many focus on improving memory target calculations. For instance, a few employ machine learning [24, 45], while others attempt to create a more accurate page-miss rate curve [30, 49] to aid in calculations. However, all of these works rely on swap space, some even using swap activity as an indicator for calculating the memory target [7, 8, 44]. Swapping is so prevalent in the ballooning literature, that some have focused specifically on optimizing swapping itself [2, 53]. In our non-overcommitted context, we do not consider any swapping and yet can set our memory targets as aggressively as we want without fear of OOM errors.

Ballooning is typically discussed in the literature as it relates to the memory overcommitment problem. However, it has been employed in a couple other contexts occasionally. It has been used for virtual machine checkpointing as a way to decrease space requirements for large VMs [28]. However, this kind of ballooning is a one-time only operation right before checkpointing and not investigated under the intense memory situations we investigate.

Virtual machine migration is a well studied area, yet only a few papers as far as we know have incorporated ballooning as a fundamental part of the solution. A few papers have employed a one-time only ballooning operation before migration, again not dealing with extreme cases [25, 27].

Hines et al. [16] are the most similar to our work. They introduce a post-copy approach to live VM migration, where most of the VM memory is page faulted in to the destination. They design a Dynamic Self Ballooning (DSB) process that minimizes memory to speed up migration. We also use a self-ballooning mechanism with the aim of speeding up migration. However, we implement our solution to work on Xen's pre-copy migration strategy. Their solution limits ballooning to consuming 95% free memory in the hopes that the remaining 5% will prevent OOM errors. They evaluate their solution on up to a single gigabyte. We are able to balloon 100% of free memory because our solution does not trigger OOM in non-overcommitted hosts, even when we test on VMs that are hundreds of gigabytes in size.

7.3 Industrial self-ballooning efforts

Several attempts at self-ballooning are either currently in use or have never left experimental phases. VMWare vSphere contains a self-ballooning mechanism, called `vmmemctl` [47], but user-guides for the feature recommend setting a maximum balloon size in order to not trigger OOM issues. It also requires swap space greater than the working set size of the VM [9]. Microsoft's Hyper-V contains something similar, called Dynamic Memory, which requires users to specify a minimum memory usage for their VM and restricts ballooning from inflating past this value [33]. Both products are meant for memory overcommitment and require users to have prior knowledge of their VM memory requirements. We do not require any prior knowledge of memory usage requirements. Since we are not in an overcommitted context, essentially any low target can be set and the balloon driver will bring the memory back to a stable minimum.

Transcendent memory (tmem) [32] was an experimental feature added to Xen/Linux that allowed extra hypervisor memory to be used as an extra memory layer between guest RAM and disk for use as a cache and swap space. Alongside tmem was a self-ballooning mechanism to encourage guests to send pages to the tmem pool. This mechanism utilized the `Committed_AS` variable for setting memory targets [31], similar to our memory targeting. However, this ballooning feature was removed after tmem was removed, since the former depended on the latter [13]. We differ in that we use ballooning for optimizing migrations and we use ballooning independently of swapping or cache solutions.

Two experimental projects tried to create a self-ballooning tool for KVM. One was a project by Luiz Capitulino where they used memory pressure states to dictate iterative memory adjustments [6]. Another project at IBM created the memory overcommitment manager (MOM) to intelligently decide which VMs to take memory from if the host memory had less than 20% free [26]. Both projects were focused on ballooning for memory overcommitment, whereas we focus on ballooning for migration.

Recent efforts in KVM/VirtIO-balloon try to minimize memory without triggering OOM. Since KVM cannot be used for nested virtualization in non-bare metal AWS instances, KVM-specific solutions cannot be used in our context. However, we provide a brief comparison between KVM features and our solution.

The first effort is free page reporting [40], an API that allows the guest to notify the hypervisor of free pages it owns. The hypervisor can then repurpose the underlying memory from the guest. The memory can be page faulted back to the guest when the guest accesses it. Our solution can respond faster to memory spikes because our user-space program detects spikes before the memory is written to (see 3.2.2), whereas free page reporting waits until the pages are actually written to to fault the memory in.

The second KVM effort is the joint related features of free page hinting and deflate on OOM [40]. Free page hinting is a mechanism that shrinks the guest VM during a migration to avoid sending free pages over the network. The accompanying deflate on OOM feature prevents memory errors during this shrinking by having the balloon release some pages back to the guest if OOM is reached. These features comprise of a one-time only solution for migrations, adding the shrinking overhead to the overall migration time, repeated with every migration. Our solution continuously minimizes memory to help decrease migration time and not redo the work of shrinking/expanding the VM with every migration.

8 CONCLUSION

We have an industrial production use case that requires minimizing live migration times of cloud user, nested virtual machines.

In this paper, we open the door for more aggressive memory minimization by making the balloon driver handle otherwise failing memory requests directly, thereby preventing out-of-memory (OOM) errors and swapping. We show that continuous ballooning is a viable solution for achieving significant migration speedup.

9 ACKNOWLEDGEMENTS

We thank Maximilian Dittgen and Clara Steinhoff for their initial feedback on earlier drafts, and the anonymous reviewers for their insightful suggestions. We also thank Alain Tchana for agreeing to be our shepherd.

REFERENCES

[1] Orna Agmon Ben-Yehuda, Eyal Posener, Muli Ben-Yehuda, Assaf Schuster, and Ahuva Mu'alem. 2014. Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 41–52.

[2] Nadav Amit, Dan Tsafrir, and Assaf Schuster. 2014. VSwapper: A Memory Swapper for Virtualized Environments. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. 349–366.

[3] AWS. 2022. Amazon Compute Service Level Agreement. https://aws.amazon.com/compute/sla/. Accessed: 2024-10-10.

[4] AWS. 2024. What is KVM (Kernel-Based Virtual Machine)? https://aws.amazon.com/what-is/kvm/. Accessed: 2024-01-04.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

[6] Luiz Capitulino. 2013. Automatic ballooning. https://www.linux-kvm.org/page/Projects/auto-ballooning. Accessed: 2022-11-20.

[7] Luiz Capitulino. 2013. Automatic Ballooning slides. https://www.linux-kvm.org/images/5/58/Kvm-forum-2013-automatic-ballooning.pdf. Accessed: 2022-11-20.

[8] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. 2013. Working set-based physical memory ballooning. In *10th International Conference on Autonomic Computing (ICAC 13)*. 95–99.

[9] VMWare Customer Connect. 2014. The balloon driver, vmmemctl, is unaware of pinned pages. https://kb.vmware.com/s/article/1003586. Accessed: 2022-12-13.

[10] Jonathan Corbet. 2010. Fixing writeback from direct reclaim. https://lwn.net/Articles/396561/ Accessed: 2022-12-10.

[11] Jonathan Corbet. 2021. Clarifying memory management with page folios. https://lwn.net/Articles/849538/. Accessed: 2024-10-10.

[12] Jonathan Corbet. 2022. Merging the multi-generational LRU. https://lwn.net/Articles/894859/. Accessed: 2024-10-10.

[13] Juergen Gross. 2019. [1/3] xen: remove tmem driver. https://patchwork.kernel.org/project/xen-devel/patch/20190527103207.13287-2-jgross@suse.com/. Accessed: 2022-11-20.

[14] Dave Hansen. 2003. Meminfo Documentation Take 2. https://lwn.net/Articles/28345/. Accessed: 2022-11-27.

[15] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. 2009. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, 630–637.

[16] Michael R. Hines and Kartik Gopalan. 2009. Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 51–60. https://doi.org/10.1145/1508293.1508301

[17] Michael R Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Ryu, and Muli Ben-Yehuda. 2011. Applications know best: Performance-driven memory overcommit with ginkgo. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE, 130–137.

[18] Stephen T Jones, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2006. Geiger: monitoring the buffer cache in a virtual machine environment. *ACM Sigplan Notices* 41, 11 (2006), 14–24.

[19] Linux kernel source code. 2009. Page Allocation Fastpath. https://elixir.bootlin.com/linux/v4.19.49/source/mm/page_alloc.c#L4364. Accessed: 2022-12-10.

[20] Linux kernel source code. 2009. Page Allocation Slowpath. https://elixir.bootlin.com/linux/v4.19.49/source/mm/page_alloc.c#L4064. Accessed: 2022-12-10.

[21] Linux kernel source code. 2014. The /proc Filesystem. https://github.com/torvalds/linux/blob/master/Documentation/filesystems/proc.rst. Accessed: 2022-11-27.

[22] The Linux kernel user's and administrator's guide. 2018. Transparent Hugepage Support. https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html. Accessed: 2022-12-13.

[23] Jinchun Kim, Viacheslav Fedorov, Paul V Gratz, and AL Narasimha Reddy. 2015. Dynamic memory pressure aware ballooning. In *Proceedings of the 2015 International Symposium on Memory Systems*. 103–112.

[24] Kapil Kumar, Nehal J Wani, and Suresh Purini. 2015. Dynamic memory and core scaling in virtual machines. In *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 269–276.

[25] Cho-Chin Lin, Zong-De Jian, and Shyi-Tsong Wu. 2015. Live migration performance modelling for virtual machines with resizable memory. In *Computer and Information Science*. Springer, 87–100.

[26] Adam Litke. 2011. Manage resources on overcommitted KVM hosts. https://developer.ibm.com/tutorials/l-overcommit-kvm-resources/. Accessed: 2022-12-13.

[27] Haikun Liu and Bingsheng He. 2014. VMbuddies: Coordinating live migration of multi-tier applications in cloud environments. *IEEE transactions on parallel and distributed systems* 26, 4 (2014), 1192–1205.

[28] Haikun Liu, Hai Jin, and Xiaofei Liao. 2009. Optimize performance of virtual machine checkpointing via memory exclusion. In *2009 Fourth ChinaGrid Annual Conference*. IEEE, 199–204.

[29] Maxime Lorrillere, Julien Sopena, Sébastien Monnet, and Pierre Sens. 2015. Puma: pooling unused memory in virtual machines for I/O intensive applications. In *Proceedings of the 8th ACM International Systems and Storage Conference*. 1–11.

[30] Pin Lu and Kai Shen. 2007. Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache.. In *Usenix Annual Technical Conference*. 29–43.

[31] Dan Magenheimer. 2011. Transcendent memory in a nutshell. https://lwn.net/Articles/454795/. Accessed: 2022-11-20.

[32] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. 2009. Transcendent memory and linux. In *Proceedings of the Linux Symposium*. Citeseer, 191–200.

[33] Microsoft. 2016. Hyper-V Dynamic Memory Overview. https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh831766(v=ws.11). Accessed: 2022-12-13.

[34] Changwoo Min, Inhyeok Kim, Taehyoung Kim, and Young Ik Eom. 2012. VMMB: virtual machine memory balancing for unmodified operating systems. *Journal of Grid Computing* 10, 1 (2012), 69–84.

[35] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. 2011. RemusDB: Transparent high availability for database systems. *Proceedings of the VLDB Endowment* 4, 11 (2011), 738–748.

[36] Germán Moltó, Miguel Caballer, and Carlos De Alfonso. 2016. Automatic memory-based vertical elasticity and oversubscription on cloud platforms. *Future Generation Computer Systems* 56 (2016), 1–10.

[37] Germán Moltó, Miguel Caballer, Eloy Romero, and Carlos de Alfonso. 2013. Elastic Memory Management of Virtualized Infrastructures for Applications with Dynamic Memory Requirements. *Procedia Computer Science* 18 (2013), 159–168. 2013 International Conference on Computational Science.

[38] Vlad Nitu, Aram Kocharyan, Hannas Yaya, Alain Tchana, Daniel Hagimont, and Hrachya Astsatryan. 2018. Working set size estimation techniques in virtualized environments: One size does not fit all. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 1 (2018), 1–22.

[39] Oasis Open. 2022. Virtual I/O Device (VIRTIO) Version 1.2, Section 5.5.6.5. https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.pdf. Accessed: 2024-01-04.

[40] Oasis Open. 2022. Virtual I/O Device (VIRTIO) Version 1.2, Section 5.5.6.7. https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.pdf. Accessed: 2024-01-04.

[41] Phoronix. 2024. Phoronix Test Suite. https://www.phoronix-test-suite.com/. Accessed: 2024-07-14.

[42] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. 2013. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 337–350.

[43] Joel H Schopp, Keir Fraser, and Martine J Silbermann. 2006. Resizing memory with balloons and hotplug. In *Proceedings of the Linux Symposium*, Vol. 2. Citeseer, 313–319.

[44] Rebecca Smith and Scott Rixner. 2017. A Policy-Based System for Dynamic Scaling of Virtual Machine Memory Reservations. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. 282–294.

[45] Vangelis Tasoulas, Hårek Haugerundrek Haugerud, and Kyrre Begnum. 2012. Bayllocator: A proactive system to predict server utilization and dynamically allocate memory resources using Bayesian networks and ballooning. In *26th Large Installation System Administration Conference (LISA 12)*. 111–121.

[46] Luis Tomás and Johan Tordsson. 2013. Improving cloud infrastructure utilization through overbooking. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing conference*. 1–10.

[47] VMWare. 2019. Memory Balloon Driver. https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-5B45CEFA-6CC6-49F4-A3C7-776AAA22C2A2.html. Accessed: 2022-12-13.

[48] Carl A. Waldspurger. 2002. Memory resource management in VMware ESX server. https://www.usenix.org/legacy/event/osdi02/tech/waldspurger/waldspurger.pdf. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*. 181–194.

[49] Zhigang Wang, Xiaolin Wang, Fang Hou, Yingwei Luo, and Zhenlin Wang. 2016. Dynamic memory balancing for virtualization. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 1–25.

[50] Xen Project Wiki. 2015. Huge Page Support. https://wiki.xenproject.org/wiki/Huge_Page_Support. Accessed: 2023-05-28.

[51] Dan Williams, Hani Jamjoom, Yew-Huey Liu, and Hakim Weatherspoon. 2011. Overdriver: Handling memory overload in an oversubscribed cloud. *ACM SIGPLAN Notices* 46, 7 (2011), 205–216.

[52] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. 2012. The Xen-Blanket: virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems*. 113–126.

[53] Qi Zhang and Ling Liu. 2015. Shared memory optimization in virtualized cloud. In *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 261–268.

[54] Qi Zhang, Ling Liu, Jiangchun Ren, Gong Su, and Arun Iyengar. 2016. iBalloon: Efficient vm memory balancing as a service. In *2016 IEEE International Conference on Web Services (ICWS)*. IEEE, 33–40.

[55] Wei-Zhe Zhang, Hu-Cheng Xie, and Ching-Hsien Hsu. 2015. Automatic memory control of multiple virtual machines on a consolidated server. *IEEE Transactions on Cloud Computing* 5, 1 (2015), 2–14.