PTEMagnet: Fine-Grained Physical Memory Reservation for Faster Page Walks in Public Clouds

PTEMagnet: Fine-Grained Physical Memory Reservation for Faster Page Walks in Public Clouds

Artemiy Margaritov
University of Edinburgh
United Kingdom
artemiy.margaritov@ed.ac.uk

Dmitrii Ustiugov
University of Edinburgh
United Kingdom
dmitrii.ustiugov@ed.ac.uk

Amna Shahab
University of Edinburgh
United Kingdom
amna.shahab@ed.ac.uk

Boris Grot
University of Edinburgh
United Kingdom
boris.grot@ed.ac.uk

ABSTRACT

The last few years have seen a rapid adoption of cloud computing for data-intensive tasks. In the cloud environment, it is common for applications to run under virtualization and to share a virtual machine with other applications (e.g., in a virtual private cloud setup). In this setting, our work identifies a new address translation bottleneck caused by memory fragmentation stemming from the interaction of virtualization, colocation, and the Linux memory allocator. The fragmentation results in the effective cache footprint of the host PT being larger than that of the guest PT. The bloated footprint of the host PT leads to frequent cache misses during nested page walks, increasing page walk latency.

In response to these observations, we propose PTEMagnet, a new software-only approach for reducing address translation latency in a public cloud. PTEMagnet prevents memory fragmentation through a fine-grained reservation-based allocator in the guest OS. Our evaluation shows that PTEMagnet is free of performance overheads and can improve performance by up to 9% (4% on average). PTEMagnet is fully legacy-preserving, requiring no modifications to either user code or mechanisms for address translation and virtualization.

CCS CONCEPTS

• **Computer systems organization** → **Serial architectures**; • **Software and its engineering** → **Virtual memory**; Allocation / deallocation strategies.

KEYWORDS

virtual memory, operating system, virtualization

ACM Reference Format:
Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. 2021. PTEMagnet: Fine-Grained Physical Memory Reservation for Faster Page Walks in Public Clouds. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3445814.3446704

ASPLOS '21, April 19–23, 2021, Virtual, MI, USA

1 INTRODUCTION

Cloud computing offers great flexibility through on-demand resource scaling, high resource utilization and low operating costs. Businesses deploy their services in the public cloud in order to enjoy these benefits as reflected in the global cloud computing market growing from $350 billion in 2020 to an anticipated $800 billion by 2025 [37]. To ensure safety, isolation and to hide the complexity of managing physical machines, cloud resources operate under virtualization and rent virtual machines (VMs) to cloud users.

The applications that commonly run in the cloud, such as data analytics frameworks, key-value stores, and databases, operate on massive – and continually expanding – in-memory datasets. The large footprint of the datasets pushes beyond TLB reach and, together with irregular memory access patterns, reduces the efficacy of the processor TLBs, resulting in frequent TLB misses. Each TLB miss triggers a page walk – a long pointer chase through the page table (PT). When operating under virtualization, a page walk requires traversing both the guest and host PTs (i.e., a *nested* page walk) thereby incurring a particularly high latency as noted by prior works [36]. The nested page walk latency is further amplified when multiple applications are colocated within a single VM. Prior work uses software support and customized hardware to shorten the latency of traversing both the guest and host PTs by leveraging a more favorable layout of PTs [36] and application data in the physical memory [2]. However, these works do not analyze which of the accesses in the multiple levels of guest and host PTs contribute most to the overall nested page walk latency and should be prioritized for acceleration over the other accesses.

In this work, we investigate which accesses in a nested page walk are most significant for the overall latency by examining from where in the memory hierarchy these accesses are served. We observe that while accesses to the guest PT are often served by higher levels of the memory hierarchy (closer to the core), a significant fraction of accesses to host PT levels are served by the main memory, which results in long latencies in the nested page walk. We study the discrepancy in behaviour of accesses to the guest and host PTs and find that cache behaviour (hit or miss) of page walks is defined by spatial locality of page table entries (PTEs) that reside in the leaf PT level, which dominates the overall PT footprint. While guest PTEs corresponding to nearby virtual addresses reside in the same cache block, host PTEs corresponding to these guest virtual addresses are often scattered among multiple cache blocks.

To understand why PTEs may reside in different cache blocks, consider a simple case where applications are running natively. A PT is indexed through virtual addresses, where the PTEs for two adjacent virtual pages A and A+1 sit in neighbouring leaf nodes and within the same cache block. While the A and A+1 are adjacent

in virtual address space, their virtual-to-physical mappings are determined by the memory allocator. If the memory allocator is allocating memory for a single application, adjacent virtual pages are likely to be mapped to adjacent physical pages, carrying over their spatial locality to physical address space. However, if the memory allocator is allocating memory for multiple applications, the allocations for virtual pages A and A+1 may be interleaved by memory allocation requests for co-running application(s). In this case, A and A+1 are unlikely to be mapped to adjacent physical pages and their spatial locality is lost in the physical address space. In the worst case, a set of pages that are contiguous in the virtual space may be allocated to physical pages that are entirely non-contiguous. This results in application's memory being fragmented in the physical address space.

Under virtualization, when the memory allocator in a VM is stressed by colocated applications, each individual application's memory is fragmented in the guest OS's physical address space. The host OS deals with the VM like another process and treats the guest physical address space as the VM process' virtual memory. Problematically, the fragmentation in the guest physical memory carries over to the host virtual memory. Guest virtual pages A and A+1 which were mapped to non-adjacent guest physical pages will now be non-adjacent in the host virtual address space. As a result, they will not occupy neighbouring PTEs in the host page table, and will not reside in the same cache block. This increases the footprint of the host page table nodes corresponding to each application running inside the VM.

In response to these observations, we introduce PTEMagnet, a legacy-preserving software-only technique to reduce page walk latency in cloud environments by improving locality of host PTEs. Cache locality of host PTEs can be improved by limiting memory fragmentation in the guest OS. We show that prohibiting memory fragmentation within a small contiguous region greatly increases locality for host PTEs. PTEMagnet uses this observation and employs a custom guest OS memory allocator which prohibits fragmentation within small virtual address regions mapped to guest physical address space. PTEMagnet improves locality of the host PTEs and thus accelerates nested page walks.

Based on this insight, we propose PTEMagnet – a reservation-based approach to prevent fragmentation. To determine the optimal reservation granularity, we note that a 64B cache block can fit a maximum of 8 host PTEs, assuming an 8-bytes PTE, typical for x86. The 8 adjacent host PTEs represent a contiguous $8 \times 4\text{KB} = 32\text{KB}$ memory region in the VM's virtual memory and, in turn, the guest OS's physical memory. We find that by prohibiting fragmentation in each 32KB guest physical memory region, host PTEs can enjoy the benefits of maximum spatial locality from a cache block. PTEMagnet deploys a custom OS memory allocator that, on the first page fault to a given 32KB region, allocates the full 32KB (8 pages) but returns only 4KB (1 page) to an application, keeping the rest reserved for later use. On subsequent page faults within a reserved region, PTEMagnet's custom allocator instantly returns the already-reserved memory to the application.

Using a diverse set of cloud applications, including SPEC and big-memory applications, executing in a virtualized environment under aggressive colocation on real hardware, we make the following contributions :

- We observe that under virtualization and colocation, page walks within the host PT incur 4.4× more cache misses than page walks within the guest PT.
- We show that the guest OS memory allocator, when operating under colocation, fragments the guest physical memory across the colocated applications. This results in host PTEs corresponding to each application being scattered over multiple cache blocks. For `pagerank` colocated with memory-intensive co-runners, the guest OS memory allocator fragments over 63% of `pagerank`'s contiguous memory regions, scattering their host PTEs over many cache blocks.
- We propose PTEMagnet, a legacy-preserving software-only technique that prevents scattering host PTEs across multiple cache blocks by prohibiting fragmentation within small memory regions using a reservation-based allocation approach.
- We demonstrate that PTEMagnet achieves 4% performance improvement on average (9% max) for big-memory applications sharing a VM with other workloads. Critically, applications that do not benefit from PTEMagnet are never slowed down by it, making PTEMagnet broadly attractive for cloud deployment.

2 BACKGROUND

2.1 Virtual Memory Basics

Virtual memory provides each process with the illusion of having access to a full address space, thus mitigating the reality of limited physical memory. The operating system (OS), using a combination of hardware and software, maps memory addresses used by a program called (i.e., virtual addresses) into physical addresses in memory. The OS assigns memory on a page granularity. Translation of virtual-to-physical address happens on every memory access. In this paper, we focus on Linux/x86, which is a common platform for public cloud solutions. The remainder of this subsection describes Linux memory allocation and translation mechanisms in detail.

2.2 Memory Allocation Mechanism

In Linux, a process can request memory from Linux, by executing `mmap()` or `brk()` system calls. After executing these calls, the process immediately (eagerly) receives the requested virtual memory region. In contrast to the allocation of virtual space, physical memory allocation is performed lazily: the OS assigns physical memory to a process on-demand and on a page-by-page basis. The OS allocates memory for the process with a page of physical memory upon the first access that triggers a page fault and inserts the corresponding virtual-to-physical mapping, that is a PTE, into the PT.

2.3 Memory Allocation Granularity

Linux/x86 supports several page sizes, namely 4KB (small), 2MB (large), and 1GB pages. Pages of different sizes are managed at different levels of the PT radix tree in a recursive fashion. For example, translations of 4KB pages are stored at the fourth level (leaf) of the PT. Translations also can be stored at the third level

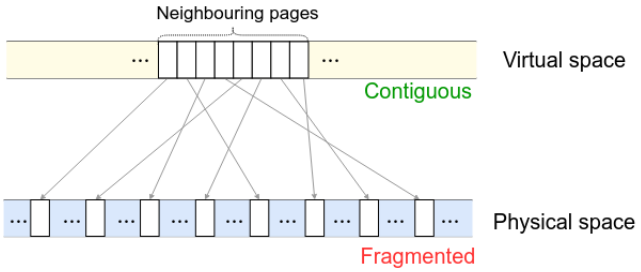

Figure 1: Difference in contiguity in virtual and physical address spaces.

of the PT and correspond to a 2MB page that is equivalent to a collection of 512 adjacent 4KB pages that are managed as a whole.

Large pages, being a big hammer solution, come at the cost of various problems. First, only a fraction of a large page may be in use by an application that results in internal memory fragmentation. Second, prior work shows that using large pages also increases external memory fragmentation, may cause performance anomalies, and is not always possible under high memory pressure [32, 40]. Finally, many of the current kernel mechanisms (e.g., page cache [28]) still do not fully support large pages and continue to rely on baseline 4KB pages.

Due to the combination of issues listed above, large pages are often disabled in production systems, e.g. in AWS EC2 [8]. Some cloud applications experience degraded performance with transparent huge pages (THP) as reported by Netflix [17], Redis [29] and MongoDB [38]. As a result, THP is set to `madvise` by default in Amazon Linux. Hence, cloud customers need to manually analyze their applications with regards to large pages and, then, customize the EC2 instances according to the needs of the applications that can be scheduled to run on these instances. Custom instance optimizations make the server fleet heterogeneous and require more complicated cluster scheduling policies. Instead, cloud services, like Amazon Batch [7], aim to simplify the task of launching and scheduling a job in the cloud by removing the need to manually set up cluster infrastructure [21]. Thus, cloud customers often resort to using default EC2 instances with 4KB pages.

2.4 Memory Fragmentation

Eager allocation of virtual address space in conjunction with lazy physical address space allocation can create a drastic difference in the spatial locality of the two address spaces. In Linux, both `mmap()` and `brk()` system calls are implemented to return contiguous virtual memory regions whereas physical memory pages are allocated ad hoc, as the Linux buddy allocator is optimized for fast physical memory allocation instead of contiguity. In a multi-tenant system, due to the fact that page faults from different applications are coming asynchronously, the Linux memory allocator fragments the physical memory space. As a result, addresses that are contiguous in the virtual address space often correspond to pages arbitrarily placed in physical memory [3, 36, 50]. Thus, under aggressive colocation, which is typical in the public cloud, the Linux memory allocator interspersedly allocates physical memory to different applications, destroying the contiguity present in the virtual space. Figure 1 demonstrates the difference in contiguity between virtual and physical address spaces.

2.5 Address Translation Mechanism

Linux keeps the mapping of the virtual address space of a running process onto physical memory in a per-process structure, called a Page Table (PT). The PT maintains the mappings at a page granularity: for each page in the virtual space, there is a PT entry (PTE) that contains a virtual-to-physical address translation for all virtual addresses within a single page, as well as other important metadata including page access permissions. Before accessing any memory location in physical memory, the CPU must look up the address translation of the virtual address and check its validity with respect to access permissions.

In x86 architecture, a PT is commonly organized as a multi-level radix tree, where the leaf level contains the actual PTEs. Upon a memory access, the CPU needs to perform a page walk, i.e., traverse the PT level-by-level from its root to the appropriate leaf with the corresponding translation. As Linux uses four-level PTs – with a planned migration to five-level PTs in the near future [25] – a page walk, which is a sequence of serialized memory accesses to the nodes of a radix-tree PT, can be a long-latency operation in an application with a large memory footprint [11, 36, 46].

To reduce address translation overheads, modern CPUs deploy a wide range of machinery that includes both address-translation-specific and general-purpose caching. First, multi-level Translation-Lookaside Buffers (TLBs) cache address translations to recently accessed memory locations. Upon a TLB miss (if no TLBs have the required translation), a hardware-based page walker performs a PT look-up, bringing the corresponding translation from memory, potentially raising a page fault if the translation does not exist, and installing it in the TLB for future reuse. PTs are located in conventional physical memory so that page walks can benefit from the regular CPU cache hierarchy that accelerates accesses to the recently used parts of the PT. Besides the conventional caches, CPUs deploy Page-Walk Caches (PWCs) that hold recently accessed intermediate nodes of the radix-tree PTs, allowing the page walker to skip one or more PT levels in a page walk.

In a virtualized scenario, which is typical in a public cloud, guest and host OS-es manage their own sets of page tables, together comprising so-called nested page tables [18]. Upon a TLB miss, a virtualized process has to perform a 2D (or *nested*) page walk. A 2D page walk is comprised of a conventional 1D page walk in the guest PT where each access to the nodes of the guest PT triggers a complete 1D page walk in the host PT. The latter is required to find the location of the node in the next level of the guest PT in host-physical memory. Thus, in order to translate a guest virtual address to a guest physical address, in addition to the 4 memory accesses to the guest PT, the page walker needs to perform up to 4 memory accesses to the host PT for each access to the guest PT. Moreover, after getting a guest physical address, the page walker needs to perform one more 1D page walk in the host PT to figure out the location of the data in the host physical memory. Thus, up to 24 accesses to the memory hierarchy may be required during a 2D page walk [14].

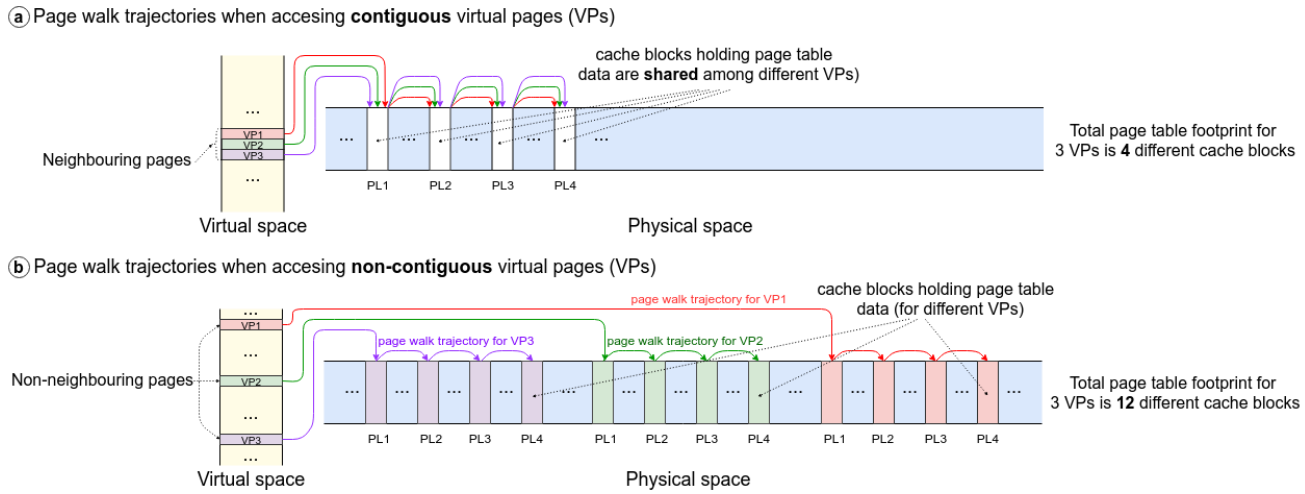

Figure 2: Page walk trajectories in physical memory when accessing *a* – non-contiguous and *b* – contiguous virtual pages.

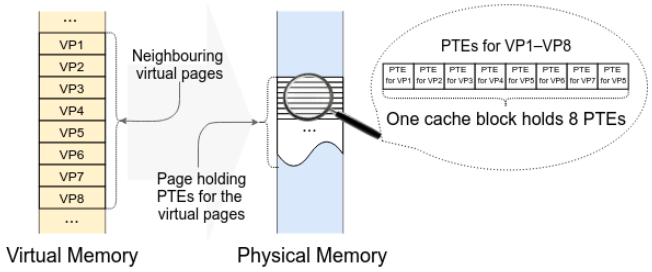

Figure 3: Packing of PTEs of neighbouring virtual pages in one cache block.

2.6 Spatial Locality in the Page Table

Prior work has demonstrated that abundant spatial locality exists in access patterns of many cloud applications [49]. Spatial locality means that nearby virtual addresses are likely to be accessed together In native execution, PT accesses also inherit the spatial locality of accesses to data. Indeed, being indexed by virtual addresses, PTEs corresponding to adjacent pages are tightly packed in page-sized chunks of memory. As a result, TLB misses to adjacent pages in the virtual space result in page walks that traverse the PT radix-tree to adjacent PTEs. Dense packing of such PTEs amplifies the efficiency of CPU caches that manage memory at the granularity of cache blocks: up to eight adjacent 8-byte PTEs (which are likely to be accessed together due to applications' spatial locality) may reside in a single cache block. Figure 3 represents the layout of the PT, highlighting the tight packing of PTEs of neighbouring pages in a cache block.

Figure 2 shows trajectories of consecutive page walks for (a) contiguous pages and (b) non-contiguous pages. One can see that page walks for contiguous virtual pages experience spatial locality as page walks for all pages access the same cache blocks. In contrast, page walks for non-contiguous pages go through different cache blocks, which necessarily increases the cache footprint of the PT and increases the page walk latency. As a result, the locality in access patterns and contiguity in the virtual space can make page walks faster. Thus, in the absence of virtualization, by creating contiguity in the virtual space, the memory allocation mechanism naturally helps to make address translation faster[1].

3 CHALLENGES FOR SHORT PAGE WALK LATENCY UNDER VIRTUALIZATION AND COLOCATION

In this section, we demonstrate that the combination of virtualization and workload colocation causes fragmentation of the guest physical address space, which diminishes the efficiency of caching of host PTEs and leads to elevated page walk latencies.

3.1 Fragmentation in the Host Virtual Address Space

Virtualization blurs the clear separation between virtual and physical address spaces. Modern virtualization solutions, e.g., Linux KVM hypervisor, are integrated with the host OS kernel allowing the host OS to reuse the bulk of existing kernel functionality, including memory allocation, for virtual machines. Hence, a virtual machine appears as a mere process for the host OS that treats the virtual machine's (guest) physical memory as a single contiguous virtual memory region [18]. As a result, one can think of the guest physical memory as the virtual memory for the host. Just like physical memory for any other virtual memory regions in the host OS, the physical memory for the guest OS is allocated on-demand and page-by-page when the guest actually accesses its physical pages.

As described in Section 2.2, while virtual address space features high contiguity, physical address space is highly fragmented, especially when under aggressive workload colocation. As a result, with virtualization and workload colocation, the host virtual address space, being similar to the guest physical address space, is

[1]Note that fragmentation in the physical space has no effect on page walk latency because the locality in the PT stems *only* from contiguity in the virtual space and not physical.

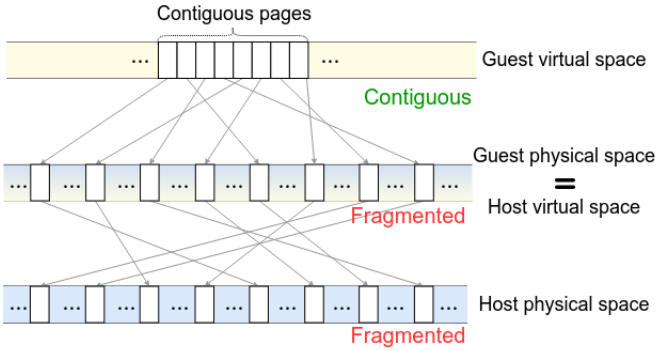

Figure 4: Contiguity (or lack of it) in virtual and physical address spaces under virtualization.

highly fragmented too. Thus, the combination of virtualization and workload colocation breaks contiguity in the host virtual address space while keeping the guest virtual address space contiguous. Figure 4 represents contiguity (or lack of it) in virtual and physical address spaces under virtualization.

3.2 Consequences of Fragmentation in the Host Virtual Address Space

Lack of contiguity in the host virtual address space impairs spatial locality in the host PT. To understand this effect, consider a scenario when several applications run in the same virtual machine. Further, assume that each application allocates a region of guest-virtual memory that spans eight or more pages. Although pages in each of these regions are adjacent in guest-virtual memory, they are scattered across guest-physical memory. This is attributable to the fact that, after allocation of the regions, the applications start to access them concurrently, resulting in the interleaving of page faults to these regions. As a result, the Linux memory allocator in the guest OS fails to preserve guest-physical memory contiguity, assigning arbitrary guest-physical addresses to each of these pages so that these pages are distant from each other in the guest physical address space. As explained in Section 3.1, if the guest physical address space is fragmented, the host virtual address space is fragmented too.

Let's consider page walks performing address translation for a memory region allocated by an application in the scenario described in the previous paragraph. Page walks involve accessing guest and host PTEs (termed *gPTE* and *hPTE*, respectively). As discussed in Section 2.6, a single cache block with PTEs contains PTEs of eight pages neighbouring in a virtual address space. Due to spatial locality of the application's access patterns, the pages accessed by the application are likely to be close to each other in the guest virtual address space. Thanks to spatial locality, within a short period of time, a cache block holding gPTEs is likely to be accessed by multiple page walks that perform address translation for neighboring pages. In contrast, due to fragmentation in the host virtual address space, the hPTEs corresponding to pages neighbouring in the guest virtual address space are not located close to each other but scattered over different cache blocks. As a result, while accesses to gPTEs can benefit from spatial locality of application's access patterns, accesses to hPTEs cannot since fragmentation in the host virtual address space prohibits propagation of the spacial locality from guest virtual to host virtual address space.

The difference in the ability of accesses to gPTEs and hPTEs of exploiting spatial locality results in two consequences. Firstly, during a page walk, hPTEs are more likely to be fetched from the main memory than gPTEs. Secondly, such a difference makes the footprint of hPTEs larger than the footprint of gPTEs. In the extreme case, page walks of a group of eight pages would touch one cache block with gPTEs and eight cache blocks with hPTEs. We find the number of cache blocks with hPTEs that are touched by page walks for a group of eight pages neighbouring in the guest virtual address space to be a useful metric. Hereafter, we characterize the level of fragmentation in the host PT by calculating the average number of cache blocks with hPTEs that correspond to gPTEs that are packed into a single cache block.

3.3 Quantifying Effects of Fragmentation in the Host PT

Fragmentation of the host PT significantly increases its footprint in the CPU cache hierarchy (i.e., the number of cache blocks containing PTEs). A large PT cache footprint is obviously undesirable, since it presents a capacity challenge that is further amplified by cache contention (by application's code and data, as well as by co-running applications). Misses for PTEs in the CPU cache hierarchy necessarily go to memory, thus increasing page walk latency and hurting performance.

To showcase the effect of fragmentation in the host PT on performance, we construct an experiment where we run a representative `pagerank` benchmark from the GPOP graph workload suite [33] inside a virtual machine in isolation and in colocation with a memory-intensive co-runner. As a co-runner, we use `stress-ng` [19], configured to run 12 threads that continuously allocate and deallocate physical memory. As a consequence of colocation, the host virtual address space gets fragmented, which results in fragmentation in the host PT, thus increasing page walk latency. As a metric of performance, we measure execution time. Using *perf*, we also measure different hardware metrics to validate the fact that the change in performance stems from fragmentation in the host PT (see §5 for details of the complete setup). We analyze the source code of `pagerank` to identify a moment of execution by which `pagerank` finishes allocation of physical memory (namely, when it completes initializing all allocated data structures). Before collecting measurements, we stop the co-runner after `pagerank` finishes allocation of physical memory, because by that moment the co-runner already caused fragmentation in the host PT, which is the intended effect. As a result, when measuring `pagerank`'s performance, there is no contention for shared resources, such as LLC capacity, between `pagerank` and the co-runner.

Table 1 represents changes in values of the measured metrics caused by fragmentation in the host PT. We observe that fragmentation in the host PT, caused by colocation with the memory-intensive co-runner, increases execution time by 11%. We find that while not affecting the number of cache and TLB misses, fragmentation in the host PT increases the number of page walk cycles by 61%. Therefore, we conclude that the performance degradation is attributed to the change in the overhead of address translation.

We observe that colocation affects 63% of pagerank's contiguous memory regions, scattering their hPTEs to 8 distinct cache blocks.

Table 1: Changes in metrics of `pagerank` in colocation with `stress-ng` as compared to standalone execution.

Metric	Change
Execution time	+11%
Cache misses	<1%
TLB misses	<1%
Page walk cycles	+ 61%
Cycles spent traversing the host page table	+117%
Guest page table accesses served by main memory	+3%
Host page table accesses served by main memory	+283%
Host page table fragmentation (defined in Sec 3.2)	+242%

Overall, colocation raises the host PT fragmentation metric to 6.8, a significant increase from 2.8 observed in isolation. We find that fragmentation in the host PT has a nominal effect on the number of guest PT accesses served from memory. In contrast, the number of accesses to the host PT served from memory increases by 283%. As a result, in colocation, page walk incurs misses to host page table 4.4× more frequently than to the guest PT. Since a main memory access has higher latency than a cache access, with memory fragmentation, traversing the host PT takes more time, which increases page walk latency. Indeed, we observe a 117% increase in the number of cycles spent while traversing the host PT.

Our experiment shows that memory fragmentation under virtualization and colocation inside the same virtual machine can significantly increase page walk latency and degrade application performance.

3.4 Virtual Private Clouds: Virtualization + Colocation

While it is common knowledge that virtualization is a foundational technology in cloud computing, an astute reader might ask how likely are multiple applications to be colocated inside a single virtual machine. This section addresses this question.

Colocation in the same virtual machine is common-place in public clouds due to prevalence of services known as virtual private cloud (VPC), inlcuding Amazon VPC [6] and Google VPC [23]. These services allow internal and external users to run their applications on a cluster of virtual machines using an orchestration framework.

In a VPC, colocation of different applications in the same virtual machine occurs as a result of a combination of three factors. Firstly, a VPC typically includes a virtual machine that has a large number of virtual CPUs (vCPUs) and thus is capable of colocation. Such *large virtual machines* are needed to run large-memory applications as cloud providers tend to offer virtual machines with a fixed RAM-to-CPU ratio [26]. However, that is not the only scenario when a VPC includes a large virtual machine. Another case for including a large virtual machine in a VPC is reducing the costs by constructing a VPC from lower-cost available transient virtual machine instances. Such a policy selects the cheapest configuration of a virtual machine that can happen to be a large virtual machine [27]. Secondly, to increase utilization and reduce costs, cloud customers tend to run multiple different applications on a cluster at the same time [10]. Thirdly, cluster orchestration frameworks, such as Kubernetes [5, 22, 31], manage resources by a small unit of compute, typically one vCPU [45]. As a result of these factors, a machine with many vCPUs can receive a command to execute multiple different applications at the same time.

Colocation in the same VM is widely employed in Amazon Elastic Container Service (Amazon ECS) [4]. The bin packing task placement strategy that aims to fit Kubernetes tasks in as few EC2 instances as possible can easily cause a colocation of multiple applications in the same VM [48]. As a result, *any* applications can be colocated together in the same VM. Amazon ECS powers a growing number of popular AWS services including Amazon SageMaker, Amazon Polly, Amazon Lex, and AWS Batch, and is used by hundreds of thousands of customers including Samsung, GE, Expedia, and Duolingo [21].

3.5 Summary

Public clouds ubiquitously employ both virtualization and colocation inside the same virtual machine. Our analysis of the virtual memory subsystem in such an environment reveals that a lack of coordination between different parts of the memory subsystem – namely, the OS physical memory allocator and the address translation mechanism – leads to memory fragmentation in the host PT. This fragmentation increases the cache footprint of host PTEs, which results in elevated cache misses during page walks, thus increasing page walk latency and hurting application performance. Preventing fragmentation of the host PT in such an environment can thus improve cache locality for PTEs and increase performance.

4 PTEMAGNET DESIGN

In this work, our goal is to prevent fragmentation of the host PT and reduce the latency of page walks under virtualization and colocation. We aim to achieve the latency reduction by leveraging existing CPU capabilities and without disrupting the existing software stack.

4.1 Design Overview

As shown in Section 3, the page walk overhead comes from the lengthy pointer chase through the fragmented host PT (hPT) due to its poor utilization of caches. Our key insight is that it is possible to reduce the page walk latency by increasing the efficiency of hPTE caching, namely grouping hPTEs corresponding to neighbouring application's pages in one cache block. Such placement can be achieved by propagating the contiguity that is naturally present in the guest PT (gPT) to the hPT.

To exploit the contiguity potential presented inside the guest-virtual address space for compacting hPTEs inside one cache block, one needs to guarantee that adjacent guest-virtual addresses are mapped contiguously onto adjacent host-virtual addresses or, equivalently, onto guest-physical addresses. This mapping criterion requires prohibiting fragmentation and introducing contiguity within small regions in the guest physical space. Since a CPU cache block contains eight 8-byte PTEs, to achieve the maximum locality for hPTEs, the contiguity degree in the guest physical space should be at least eight pages (see Figure 3). This means that, with 4KB pages, the size of the region contiguously allocated in the guest physical space should be 32KB. Meanwhile, the Linux/x86 page fault handler requests a single page from the buddy allocator on each page fault.

To eliminate fragmentation of the hPT, we introduce PTEMagnet – a new Linux memory allocator. PTEMagnet increases the current memory allocation granularity in the guest OS to the degree that maximizes cache block utility in the existing CPU hierarchy – that is, eight adjacent pages that correspond to eight adjacent hPTEs packed into a single cache block.

To support a different memory allocation granularity, we draw inspiration from the superpage (i.e., as in large page) promotion/demotion mechanism in FreeBSD [39] that relies on allocation-time physical memory reservations. Upon the first page fault to an eight-page virtual memory range, PTEMagnet reserves an eight-page long contiguous physical memory range inside the kernel so that future accesses to this page group get allocated to their corresponding pages inside the reserved range. This approach guarantees zero fragmentation for allocated hPTEs inside a cache block and minimizes the hPTEs footprint in the CPU memory cache hierarchy.

The allocation-time reservation approach adopted by PTEMagnet avoids costly memory fragmentation that is associated with using large pages because the OS always keeps track of reserved pages and can quickly reclaim them in case of high memory pressure.

In the remainder of this section, we discuss the key aspects of PTEMagnet design that includes the reservation mechanism and its key data structures, and the pages reclamation mechanism.

4.2 Page Group Reservation

PTEMagnet attempts to reserve physical memory for adjacent virtual page groups eagerly, upon the first page fault to any of the pages within that group. Upon such a page fault, a contiguous eight-page group is requested from the buddy allocator while only one virtual page (corresponding to the faulting page) is mapped to a physical memory page, creating a normal mapping in the guest and host PTs. The other seven physical pages inside that reservation are not mapped until the application accesses them. Although these physical pages are taken from the buddy allocator's lists, these pages are still owned by the OS and can be quickly reclaimed in case of high memory pressure.

To track the existing reservations, PTEMagnet relies on an auxiliary data structure, called Page Reservation Table (PaRT). PaRT is queried on every page fault. A look-up to PaRT succeeds if there already exists a reservation for a group of eight virtual pages that includes the faulting page. If not found in PaRT, the page fault handler takes a contiguous chunk of eight pages from the buddy allocator and stores the pointer to the base of the chunk in a newly created PaRT entry. In addition to the pointer, the entry includes an 8-bit mask that defines which pages in the group are used by the application.

Upon a page fault, the faulting virtual address is rounded to 32KB (i.e., eight 4KB pages) before performing a PaRT lookup. If the reservation exists, then a page fault can be served immediately, without a call into the buddy allocator, by creating a PTE that maps to one of the reserved pages. Thus, the extra work upon the first page fault to reserve eight pages can be largely amortized with faster page faults to the rest of the pages in a reservation. Once all the reserved pages inside a reservation are mapped, their PaRT entry can be safely deleted.

PaRT is implemented as a per-process 4-level radix tree that is indexed with a virtual address of the page fault. A leaf PaRT node corresponds to one reservation and holds a pointer to the base of a chunk of physical memory, an 8-bit mask for tracking mapped pages, and a lock. To guarantee the safety and avoid a scalability bottleneck that may appear when a large number of threads spawned by one process concurrently allocate memory, the radix tree must support fast concurrent access. Thus, to reduce lock contention and maximize inter-thread concurrency, we implement fine-grain locking with one lock per node of the PaRT radix tree.

4.3 Reserved Memory Reclamation

Reservations can be reclaimed in one of two ways: (1) by the application, once it freed all eight pages in a reservation, by calling `free()`; or (2) by the OS, when the system is under memory pressure. To avoid unnecessary complexity, the OS reclaims a reservation entirely and returns all the physical pages in the group back to the buddy allocator's free list. If an application explicitly frees all pages in the group, the last call results in the deletion of the reservation. If a PaRT entry was removed because all reserved pages had been mapped, freeing of the associated memory (if and when it happens) is performed as in the default kernel, without involving PTEMagnet.

Under memory pressure, the OS must be able to reclaim the reserved physical memory pages. Similar to the `swappiness` kernel parameter [44], we introduce a configurable threshold that, when reached, triggers a daemon process that walks through all reservations in PaRT of a randomly selected application, returning all reserved pages to the buddy allocator. The demon keeps releasing reservations until the overall memory consumption goes below the threshold. Note that when the OS has to free the reserved pages, the affected application(s) still continue to benefit from the shorter page walks to pages that have previously been allocated via PTEMagnet.

We expect no noticeable performance degradation from the PTEMagnet's reclamation mechanism, as the reclamation is a mere `free()` call to the buddy allocator. In contrast, other similar reclamation mechanisms, such as in Transparent Huge Pages or in FreeBSD, are associated with the demotion of large pages into collections of small pages. Such a demotion requires PT updates and TLB flushes, which can delay application's access to its memory and lead to performance anomalies [32, 39]. PTEMagnet's reclamation mechanism does not change the PT content and does not lock memory pages used by the application.

4.4 Discussion

Fork and copy-on-write. Reservations are not copied, only individual pages. On a page fault in a child process, the reservation map of a parent can be checked to see if this page was allocated or not. If the requested page is not allocated by a parent (or other children), a page from a parent's reservation is returned to the child. This works well as the majority of pages shared between a parent and a child processes are read-only pages. Shared read-only pages don't invoke copy-on-write (COW), stay contiguous and benefit from faster page walks, accelerated by PTEMagnet. Children processes cannot create new reservations in the parent's reservation map.

PTEMagnet cannot enhance contiguity among COWs. However, we observed that less than 0.1% of all pages are COW'ed for the

Table 2: Parameters of the platform used for the evaluation.

Parameter	Value/Description
Processor	Dual Intel® Xeon® E5-2630v4 (BDW) 2.40GHz, 20 cores, 2 threads/core
Memory size	128GB/socket
Hypervisor	QEMU 2.11.1
Host OS	Ubuntu 18.04.3, Linux Kernel v4.15
Guest OS	Ubuntu 16.04.6, Linux Kernel v4.19
Guest configuration	20 vCPUs and 64GB RAM

Table 3: Evaluated benchmarks and co-runners.

Name	Description
Benchmarks	
mcf, gcc, xz, omnetpp	SPEC'17 benchmarks (ref input)
cc, bfs, nibble, pagerank	GPOP graph analytics benchmarks [33], 16GB dataset (scaled from Twitter)
Co-runners	
objdet	MLPerf [43] object detection benchmark, SSD-MobileNet [24], COCO dataset [34]
chameleon	HTML table rendering
pyaes	Text encryption with an AES block-cipher
json_serdes	JSON serialization and de-serialization
rnn_serving	Names sequence generation (RNN, PyTorch)
gcc, xz	SPEC'17 benchmarks (ref input)

applications we have studied. Our evaluation in Section 6.1 includes the effect of COWs.

Swap and THP. If the OS chooses a reserved page for swapping or THP compaction, it triggers a reclamation of the reservation.

Security implications. PTEMagnet does not violate existing security barriers. Similarly to accesses to the guest PT, accesses to the data structure holding reservations are performed within the kernel code on behalf of the memory owner process only.

System interface for enabling PTEMagnet. It is possible to limit the set of applications for which PTEMagnet is used. By design, PTEMagnet improves the performance of big-memory applications – applications experiencing a large number of TLB misses. To conditionally enable PTEMagnet, a mechanism can be implemented through `cgroups` as follows. In a public cloud, the orchestrator (e.g., Kubernetes) usually specifies the maximum memory usage for each deployed container, by setting `memory.limit_in_bytes`. If this parameter is set, and it is above a predefined threshold, the OS can enable PTEMagnet for the target process. While evaluating PTEMagnet, we find that PTEMagnet does not cause a slow down even for applications that exhibit infrequent TLB misses and hence limited benefits of faster page walks (see Section 6.1). Consecutively, limiting the set of applications for which PTEMagnet is optional.

Fragmentation of the memory reclaimed from reservations. Under low memory pressure, the non-allocated physical pages within reservations do not adversely affect the system. Under high memory pressure, PTEMagnet's reclamation mechanism will release non-allocated pages within a reservation. Since a reservation is an aligned contiguous eight-page group with at least one allocated physical page, the set of reclaimed pages does not comprise a group of pages that can be used to form a new reservation. This results in a particular type of memory fragmentation whose consequence is that PTEMagnet cannot allocate reservations and preserve contiguity within the reclaimed memory. Moreover, the pages of the reclaimed memory can be allocated to different applications, which, in turn, may release pages at different times. As a result, fragmentation of the reclaimed memory can persist in the system for a long time, limiting PTEMagnet's ability to create new reservations.

To summarize, under high memory pressure, a release of non-allocated physical pages within reservations can result in fragmentation of reclaimed memory. The more unused reservations are reclaimed, the more memory can be fragmented. The effect of fragmentation by reclamation is identical to the effect of fragmentation that occurs due to application colocation without PTEMagnet. However, while PTEMagnet prevents fragmentation stemming from colocation, it cannot prevent or remove fragmentation originating from memory reclamation. We evaluate the incidence of unused pages within reservations in Section 6.2.

5 METHODOLOGY

System setup. We prototype PTEMagnet in Linux kernel v4.19. We assume public cloud deployment (as with Amazon VPC [6] or Google VPC [23]) where multiple jobs are scheduled on top of a fleet of virtual machines. We model the cloud environment by using QEMU/KVM for virtualized execution and by running multiple applications inside one virtual machine at the same time. As a metric of performance, we evaluate the execution time of an application in the presence of co-runners. Table 2 summarizes the configuration details of our experimentation system.

Our evaluation primarily focuses on small (4KB) pages, since fine-grain memory management delivers greater flexibility, better memory utilization, and better performance predictability (Section 2.3).

Benchmarks. We select a set of diverse applications that are representative of those run in the cloud and that exhibit significant TLB pressure. Our set of applications includes benchmarks from SPEC'17 and GPOP graph analytics framework [33]). Table 3 lists the benchmarks.

Co-runners. We select a set of diverse applications from domains that are typically run in a public cloud such as data compression, machine learning, compilation, and others. The full list of co-runners is shown in Table 3. The list includes benchmarks from SPEC'17, graph analytics, MLPerf and other benchmarks.

6 EVALUATION

6.1 PTEMagnet's Performance Improvement

We evaluate PTEMagnet in two colocation scenarios. In the first scenario, we study a colocation with a co-runner which has the highest

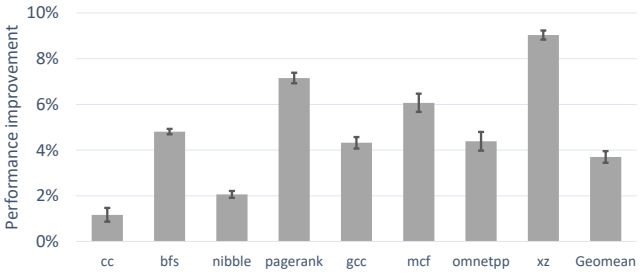

Figure 6: Performance under colocation with `objdet`.

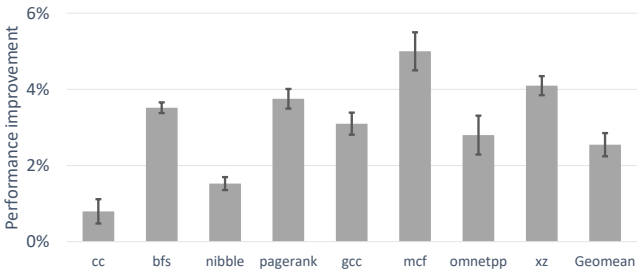

Figure 7: Performance under colocation with a combination of co-runners.

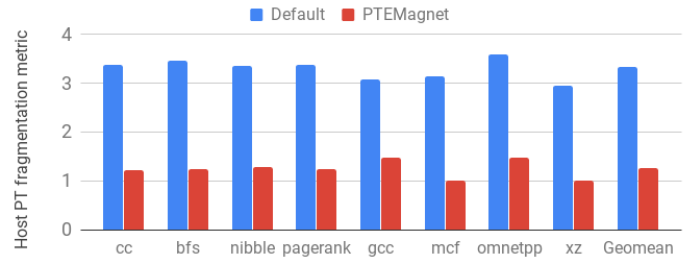

Figure 5: Host PT fragmentation in colocation with `objdet` (lower is better).

page fault rate among all the co-runners listed in Table 3, which is the 8-threaded `objdet` from MLPerf. In the second scenario, we colocate an application with a combination of all co-runners listed in Table 3. In both scenarios, we evaluate 8-threaded applications. To minimize performance variability in the system stemming from contention for hardware resources, in both scenarios, we pin applications' and co-runners' threads to different CPU cores. The standard deviation of the execution time calculated over 40 runs for each application does not exceed 2% for all applications.

PTEMagnet in colocation with `objdet`. Figure 5 represents fragmentation in the host PT measured in colocation with `objdet` with and without PTEMagnet. We observe that PTEMagnet reduces fragmentation in the host PT to almost 1 for all evaluated benchmarks. The host fragmentation metric shows how many cache blocks on average hold hPTEs corresponding to gPTEs stored in one cache block. The results show that PTEMagnet prevents fragmentation in the host PT, reducing the footprint of the host PT, and enhances spatial locality across page walks, reducing their latency.

We evaluate performance improvement stemming from accelerated page walks. To calculate performance improvement, we measure the execution time averaged over 40 runs. Figure 6 shows performance improvement delivered by PTEMagnet in comparison to the default Linux kernel. The baseline corresponds to execution in colocation with `objdet` without PTEMagnet. PTEMagnet increases performance by 4% on average and by up to 9% in the best case (on `xz`).

To highlight the fact that PTEMagnet is an overhead-free technique, we measure how PTEMagnet affects the performance of applications that do not experience high TLB pressure. We evaluate PTEMagnet on all SPEC'17 Integer benchmarks. We find that on these benchmarks PTEMagnet delivers performance improvement in the range of 0-1% (not shown in Figure 6).

Crucially, we find that none of the applications experience any performance degradation, underscoring that PTEMagnet can be widely deployed without concern for specifics of the application or the colocation setup.

PTEMagnet in colocation with a combination of different co-runners. Figure 7 represents improvement delivered by PTEMagnet in comparison to the default Linux kernel. On average, PTEMagnet improves performance by 3%, with a maximum gain of 5% achieved with `mcf`. A large number of co-runners increases contention for the capacity of shared caches. Due to increased contention, the application's cache blocks with hPTEs have higher chances to be evicted and reduced opportunity to experience locality. Our results show that even under high cache contention, PTEMagnet is capable to speed up execution, losing just 1% of performance improvement on average, in comparison to colocation with lower cache pressure – with `objdet` only.

6.2 Incidence of Non-Allocated Pages Within Reservations

As discussed in Section 4.4, if reclaimed under high memory pressure, non-allocated pages within reservations can result in fragmentation of memory that PTEMagnet can not prevent or remove. To understand the memory overhead of non-allocated pages within reservations, we study how many pages within reservations are not used by the evaluated benchmarks. For each of the evaluated benchmarks, we measure the number of non-allocated pages within reservations every second throughout the entire benchmark execution.

We find that during the execution of each evaluated benchmark, the non-allocated pages within reservations never exceed 0.2% of the benchmark's physical memory footprint size. Such a low level of non-allocated pages within reservations indicates that 1) invocation of the reclamation mechanism due to PTEMagnet's memory overhead is highly unlikely as the total benchmark's physical memory usage is on par with the baseline; and 2) applications tend to quickly allocate pages within newly created reservations, thus reducing the likelihood that the reclamation mechanism will destroy contiguity under high memory pressure.

While our evaluation shows that in practice the incidence of non-allocated pages within reservations is low, in theory, it is possible

Table 4: Changes in metrics of `pagerank` in colocation with `objdet` with PTEMagnet compared to the default kernel.

Metric	Change
Host page table fragmentation (defined in Sec 3.2)	-66%
Execution time	-7%
Page walk cycles	-17%
Cycles spent traversing the host page table	-26%
Guest page table accesses served by main memory	-1%
Host page table accesses served by main memory	-13%

to design an application that has a large fraction of non-allocated pages within reservations. For example, an application that uses only every eighth page it allocated with malloc() can have 7× more non-allocated pages within reservations than its actual physical memory footprint. If the memory reclamation mechanism releases memory reserved by such an application, PTEMagnet can not use the reclaimed memory for reservations of other applications running on the system as this memory would be highly fragmented. As a result, such an application can prevent other applications from benefiting from PTEMagnet.

6.3 Page Walk Cycles with PTEMagnet

In this section, we evaluate a reduction in page walk cycles caused by PTEMagnet by collecting hardware performance counters data with *perf*. We also measure other hardware metrics previously studied in Section 3.3. We measure the metrics for `pagerank` application running in colocation with `objdet` with and without PTEMagnet[2]. Table 4 lists the evaluated metrics and changes in their value delivered by PTEMagnet.

We find that PTEMagnet reduces the host page table fragmentation from 3.4 to 1.2, shortening execution time by 7%. Performance counters report that with PTEMagnet the CPU spends by 26% fewer cycles traversing the host page table, resulting in by 17% fewer cycles in page walks in total. This result is confirmed by a 13% reduction in the number of host page table accesses served by the main memory.

6.4 PTEMagnet's Effect on Memory Allocation Latency

In this section, we show that the reservation mechanism itself employed by PTEMagnet is overhead-free. As explained in Section 4, on a first page fault to a 32KB region, PTEMagnet requests 32KB from the buddy allocator, replacing subsequent page faults to the reservation group by quick accesses to PaRT. To show that the reservation-based mechanism does not cause performance degradation, we study if PTEMagnet slows down physical memory allocation.

We construct a microbenchmark that allocates a 60GB array and accesses each of its pages once to invoke the physical memory allocator. We measure the execution time of the microbenchmark with and without PTEMagnet. We observe that PTEMagnet negligibly – by 0.5% – reduces the execution time of the microbenchmark. This result can be explained by the fact that PTEMagnet makes fewer calls to the buddy allocator, replacing 7 out of 8 calls to it with quick accesses to PaRT. As a result, we conclude that PTEMagnet does not increase memory allocation latency.

[2]Note that in contrast to the study in Section 3.3, in this study, the co-runner was present during the entire execution of `pagerank` in both scenarios: with and without PTEMagnet.

7 RELATED WORK

Disruptive vs incremental proposals on accelerating address translation. Prior attempts at accelerating address translation take one of two directions. One calls for a disruptive overhaul of the existing radix-tree based mechanisms in both hardware and software [1, 13, 47, 51]. The other direction focuses on incremental changes to existing mechanisms [9, 20, 42]. While disruptive proposals are potentially more attractive from a performance perspective than incremental ones, disruptive approaches entail a radical re-engineering of the whole virtual memory subsystem, which presents an onerous path to adoption. In contrast, incremental techniques, requiring fewer efforts to be incorporated into existing systems, are favoured by practitioners from OS and hardware communities. Requiring only small modifications in the Linux kernel of the guest OS, PTEMagnet falls within the incremental technique category as it can be easily added to the existing systems. Moreover, in the cloud computing platforms, e.g. at AWS or Google Cloud Platform, PTEMagnet can be enabled just by cloud customers, without the involvement of cloud providers.

Incremental techniques inducing contiguity by software means. Other researches have studied incremental techniques on enforcing contiguous mappings for reducing the overhead of address translation [2, 36, 50]. Contiguity-aware (CA) paging [2] introduces a change to the OS memory allocator to promote contiguity in the physical address space. CA paging leverages contiguity to improve the performance of any hardware technique that relies on contiguous mappings, including a speculative address translation mechanism introduced by themselves. PTEMagnet is different from CA paging in two important dimensions. Firstly, CA paging, being a no pre-allocation technique, is a best-effort approach to achieve contiguity: it does not guarantee contiguity since contiguous mapping can be impossible due to allocations of other applications running on the machine. As a result, improvements of CA paging can be significantly reduced under aggressive colocation – when there are multiple memory consumers running on the same system. In contrast, PTEMagnet guarantees contiguity by eager reservation and it is insensitive to colocation. Secondly, to deliver performance improvement, CA paging requires an advanced TLB design currently not employed by modern processors, whereas PTEMagnet reduces the overhead of address translation without a need to change hardware.

Translation Ranger [50] is another incremental technique enforcing translation contiguity by software means. Translation Ranger is designed as a software helper to increase the benefits of emerging TLB designs coalescing multiple TLB entries into a single TLB entry [41, 42]. Translation Ranger creates contiguity by employing a THP-like daemon which places pages together by copying them to a contiguous region. As a consequence, Translation Ranger has disadvantages, such as high tail latency and various performance anomalies, inherent to THP-based large-page construction

methods (see Section 2.3 for more details). In contrast to Translation Ranger, PTEMagnet creates contiguity at the moment of memory allocation and doesn't involve page copying, harmful for performance predictability. Moreover, in comparison to Translation Ranger, PTEMagnet can be straightforwardly incorporated into existing systems: Translation Ranger relies on hardware which is non-existed in the current systems, whereas PTEMagnet does not have such a requirement.

ASAP [36] is another incremental proposal aiming at reducing the overhead of address translation by enforcing contiguity through software. ASAP is different from CA paging and Translation Ranger in that it calls only for contiguity in the page table rather than contiguity in the whole memory. With ASAP, contiguity in the page table enables page table node prefetching which reduces page walk latency, accelerating address translation. Compared to PTEMagnet, ASAP has the disadvantage of requiring a change in hardware, namely the addition of the prefetching mechanism, whereas PTEMagnet can be straightforwardly used on existing machines.

Other prior incremental techniques. There is a large amount of work addressing the overhead of address translation by reducing the number of TLB misses, including such techniques as increasing TLB reach [16, 35, 42, 46], TLB prefetch [15], and speculative translation [12]. PTEMagnet is complimentary to these techniques and can be used to reduce page walk latency on a TLB miss under virtualization.

8 CONCLUSION

This work identifies a new address translation bottleneck specific to cloud environments, where the combination of virtualization and workload colocation results in heavy memory fragmentation in the guest OS. This fragmentation increases the effective cache footprint of the host PT relative to that of the guest PT. The bloated footprint of the host PT leads to frequent cache misses during nested page walks, increasing page walk latency. We introduced PTEMagnet, which addresses this problem with a legacy-preserving software-only technique to reduce page walk latency in the cloud environments. PTEMagnet is powered by an insight that grouping hPTEs of nearby application's pages in one cache block can reduce the footprint of the host PT. Such grouping can be achieved by enhancing contiguity in guest physical memory space via a light-weight memory reservation approach. PTEMagnet requires minimal changes in the Linux memory allocator and no modifications to either user code or virtual address translation mechanisms. PTEMagnet enables a significant reduction in page walk latency, improving application performance by up to 9%.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and the paper's shepherd, Professor Dan Tsafrir, as well as Priyank Faldu and Professor Edouard Bugnion for the fruitful discussions and for their valuable feedback on this work. This work was supported by the Google Faculty Research Award, the EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, and the industrial CASE studentship from Arm Ltd.

REFERENCES

[1] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself virtual memory translation. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. 457–468.
[2] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Enhancing and exploiting contiguity for fast memory virtualization. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 515–528.
[3] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. 515–528.
[4] Amazon. 2020. Amazon Elastic Container Service features. Available at https://aws.amazon.com/ecs/features/.
[5] Amazon. 2020. Amazon Elastic Kubernetes Service. Available at https://aws.amazon.com/eks.
[6] Amazon. 2020. Amazon Virtual Private Cloud. Available at https://aws.amazon.com/vpc.
[7] Amazon. 2020. AWS Batch FAQs. Available at https://aws.amazon.com/batch/faqs/.
[8] Amazon. 2020. How do I configure hugepages on my Amazon EC2 Linux instance? Available at https://aws.amazon.com/premiumsupport/knowledge-center/configure-hugepages-ec2-linux-instance/.
[9] A. Arcangeli. 2010. Transparent hugepage support. *KVMForum* (2010).
[10] AWS Admin. 2020. Multi-tenant design considerations for Amazon EKS clusters. Available at https://aws.amazon.com/blogs/containers/multi-tenant-design-considerations-for-amazon-eks-clusters.
[11] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation caching: Skip, don't walk (the page table).. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*. 48–59.
[12] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A mechanism for speculative address translation. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*. 307–318.
[13] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*. 237–248.
[14] Ravi Bhargava, Ben Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems.. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*. 26–35.
[15] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 63–76. https://doi.org/10.1145/3037697.3037705
[16] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *Proceedings of the 17th International Conference on High-Performance Computer Architecture (HPCA)*. 62–63.
[17] Brendan Gregg. 2017. How Netflix tunes EC2 instances for performance. Available at http://www.brendangregg.com/Slides/AWSreInvent2017_performance_tuning_EC2.pdf.
[18] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. *Hardware and Software Support for Virtualization.* Morgan & Claypool Publishers.
[19] Colin Ian King. 2020. Stress-ng. Available at https://kernel.ubuntu.com/~cking/stress-ng.
[20] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient address translation for architectures with multiple page sizes. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 435–448.
[21] Deepak Singh. 2020. Amazon ECS vs Amazon EKS: making sense of AWS container services. Available at https://aws.amazon.com/blogs/containers/amazon-ecs-vs-amazon-eks-making-sense-of-aws-container-services/.
[22] Google Cloud. 2020. Google Kubernetes Engine. Available at https://cloud.google.com/kubernetes-engine.
[23] Google Cloud. 2020. Google Virtual Private Cloud. Available at https://cloud.google.com/vpc.
[24] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
[25] Intel. 2017. *5-level paging and 5-level EPT.* White Paper 335252-002. Intel.
[26] Jeff Barr. 2013. Choosing the right EC2 instance type for your application. Available at https://aws.amazon.com/blogs/aws/choosing-the-right-ec2-instance-type-for-your-application.
[27] Jeff Barr. 2020. Capacity-optimized spot instance allocation in action at Mobileye and Skyscanner. Available at https://aws.amazon.com/blogs/aws/capacity-optimized-spot-instance-allocation-in-action-at-mobileye-and-skyscanner.
[28] Jonathan Corbet. 2017. Huge pages in the ext4 filesystem. Available at https://lwn.net/Articles/718102.

[29] Kirk Kirkconnell. 2014. Often overlooked Linux OS tweaks. Available at https://blog.couchbase.com/often-overlooked-linux-os-tweaks.
[30] Marios Kogias. 2018. distbenchr. https://github.com/marioskogias/distbenchr.
[31] Kubernetes. 2020. Turnkey cloud solutions. Available at https://kubernetes.io/docs/setup/production-environment/turnkey-solutions.
[32] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with Ingens.. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*. 705–721.
[33] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2019. GPOP: A cache and memory-efficient framework for graph processing over partitions. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 393–394.
[34] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft COCO: Common objects in context. In *European conference on computer vision*. 740–755.
[35] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 1 (2013), 1–38.
[36] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched address translation. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO)*. 1023–1036.
[37] MarketsandMarkets. 2020. Cloud computing market report. Available at https://www.marketsandmarkets.com/Market-Reports/cloud-computing-market-234.html.
[38] MongoDB. 2020. MongoDB administration documentation. Available at https://docs.mongodb.com/manual/tutorial/transparent-huge-pages.
[39] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. 2002. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*.
[40] Ashish Panwar, Aravinda Prasad, and K Gopinath. 2018. Making huge pages actually useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 679–692.
[41] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 558–567.
[42] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced large-reach TLBs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 258–269.
[43] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. MLPerf inference benchmark. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*. 446–459.
[44] Rik van Riel, Peter Morreale. 2018. Linux Kernel documentation for the sysctl files. Available at https://www.kernel.org/doc/Documentation/sysctl/vm.txt.
[45] Roshni Pary. 2018. Run your Kubernetes workloads on Amazon EC2 spot instances with Amazon EKS. Available at https://aws.amazon.com/blogs/compute/run-your-kubernetes-workloads-on-amazon-ec2-spot-instances-with-amazon-eks.
[46] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. 469–480.
[47] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1093–1108.
[48] Tiffany Jernigan. 2019. Amazon ECS Task Placement. Available at https://aws.amazon.com/blogs/compute/amazon-ecs-task-placement/.
[49] Dmitrii Ustiugov, Alexandros Daglis, Javier Picorel, Mark Sutherland, Edouard Bugnion, Babak Falsafi, and Dionisios N. Pnevmatikatos. 2018. Design guidelines for high-performance SCM hierarchies. In *Proceedings of the 4th International Symposium on Memory Systems (MEMSYS)*. 3–16.
[50] Zi Yan, Daniel Lustig, David W. Nellans, and Abhishek Bhattacharjee. 2019. Translation ranger: Operating system support for contiguity-aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. 698–710.
[51] Idan Yaniv and Dan Tsafrir. 2016. Hash, don't cache (the page table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 337–350.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains a Linux kernel patch for enabling PTEMagnet, shell scripts for Linux kernel compilation, a VM disk image with precompiled benchmarks, and Python/shell scripts that are expected to reproduce the results presented in Figure 6 for non-SPEC benchmarks (SPEC benchmarks cannot be distributed due to the terms of the SPEC licensing agreement).

A.2 Artifact check-list (meta-information)

- **Program:** Shell, Python, and C/C++ code
- **Compilation:** GCC 7.5.0
- **Benchmarks:** We provide binaries for GPOP [33] benchmarks and MLPerf `objdet` [43]
- **Hardware:** Dual-socket server-grade x86 machine with 20+ physical cores and more than 128GB of RAM
- **Metrics:** Execution time of benchmarks
- **Disk space required:** 150GB
- **How much time is needed to prepare infrastructure for experiments?:** 2 hours
- **How much time is needed to complete experiments?:** 24 hours
- **Publicly available?:** Yes
- **Evaluation workflow:** Clone a GitHub repository, run scripts to set up the environment and compile Linux kernels and measure execution time with different Linux kernels, inspect the results
- **Code licenses (if publicly available)?:** GNU GPL v2 (Linux), Apache 2.0 (MLPerf), MIT (GPOP)
- **Archived (provide DOI)?:** 10.5281/zenodo.4321196

A.3 Description

A.3.1 How to access the PTEMagnet's source code. The source code of our work is hosted on GitHub[3] and Zenodo[4]. The GitHub repository has the most updated version.

A.3.2 Hardware dependencies. We developed and tested the artifact on a server with a 20-core dual Intel® Xeon® E5-2630v4 (Broadwell). We expect that PTEMagnet can work on all machines running under a Linux-based OS. This artifact can run on an x86 server that runs Ubuntu 18.04 LTS and has two processors with more than 20 physical cores in total. The performance improvement of PTEMagnet on other processors can be different from one presented in Figure 6. For example, a larger improvement can be achieved on a processor with a larger LLC than one evaluated in the paper. More LLC capacity increases the chances of a cache line with a page table staying in LLC, and hence boosts the speedup delivered by PTEMagnet. In contrast, more hardware page walkers than in the processor evaluated in the paper can reduce PTEMagnet's performance improvement.

A.3.3 Software dependencies. Our work uses Linux kernel sources (version 4.19) and tools to build the kernel. We use Python 2 for our Python scripts. Our Python scripts require `numpy`, `pandas`, `fabric`, `scipy` and `distbenchr` [30]. For evaluation, we provide open-source GPOP benchmarks [33].

A.3.4 Datasets. This artifact includes a dataset for GPOP benchmarks. We provide a 16GB graph with 4 billion nodes scaled from the Twitter dataset.

A.4 Installation

The installation involves cloning the GitHub repository, running the installation script, and setting ssh keys for passwordless ssh. The detailed workflow is explained in the installation section of the repository's README file[5].

```
git clone --recurse-submodules \
  https://github.com/amargaritov/PTEMagnet_AE.git
cd PTEMagnet_AE
./install/install_all.sh <PATH> # takes 2 hours
source source.sh
```

The installation script installs relevant packages and tools, downloads and builds clean and modified Linux kernels, downloads a disk image with benchmarks and their datasets, and sets the environment for the evaluation.

A.5 Experiment workflow

To reproduce the results of Figure 6, one needs to measure the execution time of benchmarks in colocation with MLPerf `objdet` in a virtual machine on the clean version of the Linux kernel and with PTEMagnet. We provide scripts for automating the launch of the experiments. The detailed instructions on how to launch the experiments can be found in the evaluation section of the README file[5]. The scripts would run each benchmark in each configuration 5 times to average execution time across the runs, reducing the effects of jitter in a system.

```
cd PTEMagnet_AE/evaluation
. launch_all_exps_short.sh # takes about 24 hours
```

A.6 Evaluation and expected result

We provide a script that outputs average execution time among multiple runs and performance improvement delivered by PTEMagnet in comparison to the clean Linux kernel.

```
cd PTEMagnet_AE/evaluation
. show_results.sh
```

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

[3] https://github.com/amargaritov/PTEMagnet_AE
[4] https://doi.org/10.5281/zenodo.4321196
[5] https://github.com/amargaritov/PTEMagnet_AE/blob/main/README.md