A SOFTWARE APPROACH TO ACCELERATING MEMORY TRANSLATION FOR VIRTUALIZED CLOUDS

BY

JIYUAN ZHANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Adviser:

Assistant Professor Tianyin Xu

ABSTRACT

Expensive page table walks triggered by frequent translation lookaside buffer (TLB) misses have incurred major performance bottlenecks for data-intensive workloads that are dominated by memory accesses with weak locality. Since it is hard to reduce TLB misses for such workloads, reducing page table walk overhead (i.e., the overhead of each TLB miss) is an increasingly important direction for improving application performance. The direction of reducing page table walk overhead is more compelling for workloads running in virtual machines. In virtualized environments, each TLB miss triggers a two-dimensional page table walk, which has a significantly higher overhead than that on native systems.

However, a major caveat of research in this area is that most designs require changes in computer hardware. Yet, for practical applications, this requirement is often untenable. To this end, research on methods to reduce page walk overhead without hardware changes becomes a valuable topic. Taking this path, our study proves that even for a hardware-defined page walk flow, it is still possible to improve address translation performance by purely software means.

This thesis presents HugeGPT, a software approach to reducing two-dimensional page table walk overhead in virtualized environments. HugeGPT ensures that page tables used in guest systems are physically held in the huge pages formed in the host system. This brings two-fold benefits: 1) the number of steps walking down the host page table is reduced; 2) the misses of page walk caches incurred by accessing the leaf nodes on host page tables can be eliminated. Extensive evaluation based on the prototype implementation and diverse real-world applications shows that HugeGPT can efficiently reduce address translation overhead and improve application performance in virtualized clouds, resulting in up to 50% application performance improvement compared to vanilla Linux/KVM.

To my parents, for their love and support.

ACKNOWLEDGMENTS

Given this opportunity, I would like to express my earnest gratitude to my advisor Assistant Professor Tianyin Xu. I am utmostly glad I have made the choice to work with him. From day one, he has encouraged and guided my research and provided all the help he could. In every project I have worked with him, I have always been deeply impressed by his exceptional knowledge and attitude. More importantly, he taught me the ability and mindset to transform research into practical and aspirational beings, which I find most invaluable. In the end, his guidance and support have been crucial to my growth as a researcher. His insight and passion always motivate me to keep going. I am incredibly grateful to be his student and to work with him both in the past and future.

I would like to express my sincere gratitude to Assistant Professor Weiwei Jia. He has been an amazing research mentor and collaborator to me for a long time. Weiwei introduced me to the world of system research and made me love this area. It is a great fortune for me to start my research journey with him. Countless times, he guided me out of confusion and helped me advance. His profound knowledge, detailed thought, and constant support are always the catalyst of my research. I have learned a lot from him both academically and personally. Without him, my research would not have been as productive and fruitful. It would be a great honor for me if I could collaborate with him for the years to come.

I want to thank Yiming Du and Peizhe Liu for their efforts and contributions to the implementation and evaluation of the HugeGPT Linux prototype.

I want to thank Associate Professor Xiaoning Ding for his help, guidance, and support throughout my undergraduate life and in previous research.

I also want to thank Siyuan Chai, Jongyul Kim, Zhaoxi Shi, and Assistant Professor Jing Li and Jianchen Shan for their help and inspiration.

My sincere gratitude also goes to all my friends, labmates and roommates. The time I spent with them helped me both fill my spirit and stomach. Without them, I couldn't live the comfy carless life I am living now.

In the end, I would like to express my dearest love to my parents. They have constantly supported and helped me unconditionally. It is them who make it possible for me to chase my dreams. Thank you for trusting me and showing me true support all the way along. No matter the distance, I will be with you.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 MOTIVATION

Address translation has become the major performance bottleneck for workloads with big memory footprints [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]. Previous works [2, 19, 20] show that the performance of big memory workloads can be degraded by as much as 50% due to the high overhead incurred by page walks happening after translation lookaside buffer (TLB) misses. This problem becomes more pronounced in clouds and may keep increasing in future computer systems. In clouds, hardware-supported memory virtualization (i.e., nested paging such as Intel extended page tables [21] and AMD nested page tables [22]) enables two-dimensional page walks to resolve TLB misses. This increases the address translation overhead by up to 6x [11, 20, 21]. With the upcoming 5-level page tables [23, 24], this increase is more than 8x.

Reducing the overhead incurred by address translation heavily relies on the hardware designs in memory management units (MMU). Thus, existing research mostly concentrates on new hardware designs, which reduces either the number of TLB misses [2, 4, 7, 8, 9, 11, 12, 14, 19, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38] or the overhead of each TLB miss [20, 39, 40].

However, it usually takes a long time before new hardware designs become available in real systems. Thus, to reduce address translation overhead on existing hardware, the mainstream approach is to allocate and map huge pages (e.g., 2 MB page) [4, 7, 14, 16, 17, 18, 33] for user applications. A TLB entry buffering the address mapping for a huge page has a much larger coverage than that for a base page (4 KB page) — with an entry for a huge page, accessing any addresses within this huge page will not incur TLB misses. With the larger coverage, TLB misses may be significantly reduced and address translations are accelerated.

Though using huge pages proves to be very effective for data accesses with a strong locality (e.g., accesses repeatedly hitting the same huge page), it is usually considered to be ineffective in accelerating the address translation for the accesses with a weak locality. For example, huge pages can hardly reduce TLB misses for random or quasi-random accesses (e.g., modern applications like graph computing) that seldom hit the same huge page. For data with weak locality, using huge pages is even considered to be harmful due to increased memory fragmentation and false sharing [38, 41]. Moreover, for scenarios that require page migration or copy-on-write, the larger

[1] This research has been published in Proceedings of the 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT '23) and is adapted here with permission. © 2023 IEEE. Reprinted, with permission, from [1].

page size of huge pages increases operation granularity and batch size, and may consequentially lead to reduced end-to-end performance [42, 43, 44, 45].

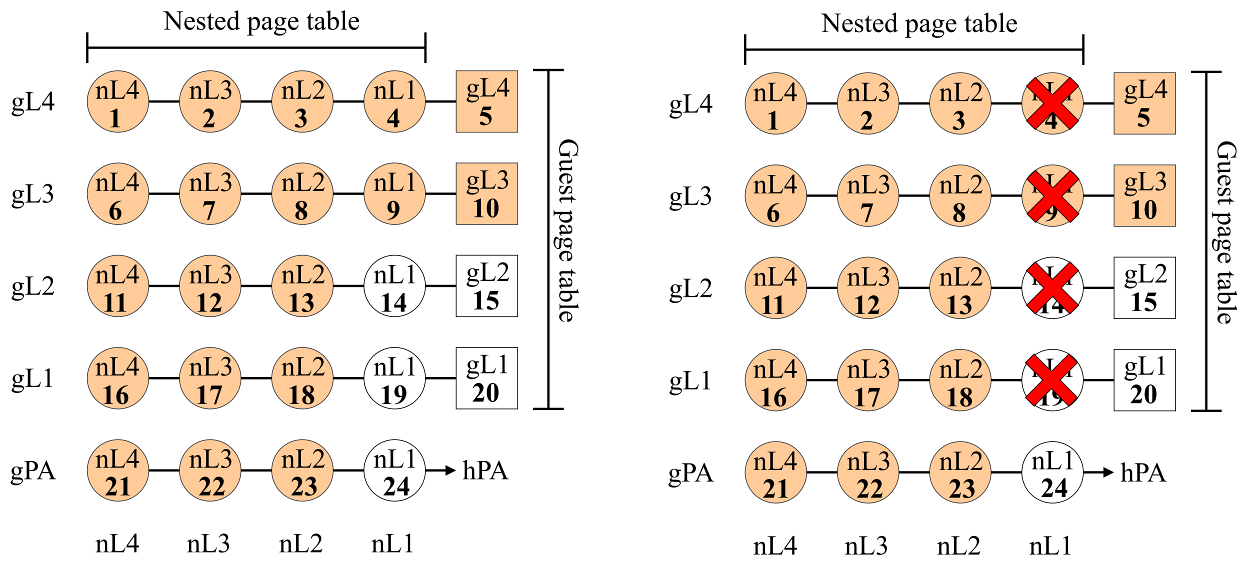
1.2 CONTRIBUTIONS

Our research is dedicated to improving the performance of address translation in virtualized environments without changing the hardware. Although the process of address translation is defined by hardware, the translation overhead can still be reduced by judiciously utilizing caching and architectural features. Our observation is that, unlike native environments, host-side huge pages under virtualization can not only translation lookaside buffer (TLB) reaches, but also affect the page walk process. First, page walk caches may have huge page awareness, and thus, their page walk cache hit rate may be higher for guest page table pages stored on host huge pages. Second, nested page walks are affected by huge pages, which will reduce the number of memory accesses, leading to a shorter page walk process.

In this thesis, we show that actually huge pages can be used to effectively accelerate address translation for weak locality data and the adverse effect is minimal. We achieve this by using huge pages in a substantially different way from conventional huge page approaches. We name our approach HugeGPT. While conventional approaches use huge pages to reduce TLB misses, for the effectiveness on weak locality data, HugeGPT “exploits” a different capability of huge pages — their capability to substantially reduce the overhead of the two-dimensional page walk, i.e., the overhead of each TLB miss in virtualized clouds. While conventional approaches use huge pages to save *data*, HugeGPT uses huge pages to save *metadata* — the page tables used in the guest operating system to manage the memory of a virtual machine. Different from traditional huge page usages, our method in no way affects the page size the user application uses. This means HugeGPT does not suffer the granularity increase seen in conventional huge page usages. Thus, HugeGPT does not incur the adverse effects that are caused by conventional approaches by saving weak locality data on huge pages.

Our insight is that most overhead in a two-dimensional page walk is incurred by walking down the host page table to resolve the entry addresses of the guest page table. This can be illustrated using Figure 1.1 (a), which shows that a two-dimensional page walk may incur as many as 24 memory accesses. Among these memory accesses, 16 are incurred by resolving the entry addresses of the guest page table, i.e., 1∼4 for resolving $gL4$ of the guest page table, 6∼9 for resolving $gL3$, 11∼14 for $gL2$, and 16∼19 for $gL1$.

Based on our insight, to reduce the overhead of two-dimensional page walks, the most effective method is to reduce the overhead incurred by resolving the entry addresses of the guest page table.



(a) Existing two-dimensional page walks: 24 memory references in the worst case (shaded page table entries are usually cached by TLB and page walk caches)

(b) Two-dimensional page walks with proposed approach: 20 memory references in the worst case

Figure 1.1: The proposed approach can substantially reduce page walk latency of the two-dimensional page walks because the lower-level page table entries shaded in the figure are usually cached by TLB and page walk caches. The proposed approach slightly changes the software, i.e., only storing guest page table data on host huge pages.

We propose HugeGPT as a software approach to save **g**uest **p**age **t**ables into host **huge** pages. This can reduce this overhead in two ways, as shown by the steps that are crossed out in Figure 1.1 (b). First, it eliminates page walk cache (PWC) misses. There is no need to buffer crossed-out steps in page walk caches. This not only eliminates the PWC misses caused by these steps but also reduces the pressure of PWCs on buffering other steps. Second, it reduces the steps to walk down the host page table upon a PWC miss at an earlier step. For example, upon a PWC miss at Step 12, in Figure 1.1 (a) 3 steps (i.e., Step 12, Step 13, and Step 14) are required to get the address of $gL2$; in Figure 1.1 (b), only 2 steps (i.e., Step 12 and Step 13) are required.

To realize HugeGPT, our basic idea is to let the guest operating system (OS) notify the host OS only to store guest page tables on host huge pages. In the default virtualized system, page faults for allocating guest page table pages at the guest level need to trap to the host level and allocate the host physical pages to back the guest page table pages. Taking this opportunity, HugeGPT allocates host huge pages to back the guest page table pages.

The thesis makes the following contributions. First, to our best knowledge, this is the first work that studies how to store guest page table data on host huge pages to accelerate two-dimensional page walks in virtualized clouds. Second, we have proposed HugeGPT as an efficient system solution that can effectively reduce page walk cache misses and the steps to walk the two-dimensional page tables for workloads with weak memory access locality. Finally, we have implemented HugeGPT based on Linux/KVM, tested it with diverse real-world applications and extensive experiments comprehensively, and also compared HugeGPT with related systems. Our tests show HugeGPT can greatly reduce two-dimensional page walk overhead, resulting in up to 50% application performance improvement compared to vanilla Linux/KVM. HugeGPT also performs better than related systems (confirmed in Section 6.4).

1.3 THESIS ORGANIZATION

The rest of the thesis is organized as follows. Chapter 2 gives an introduction to existing translation support for virtualized environments and their performance characteristics. Chapter 3 introduces the main idea to solve our research problem and outlines the major challenges in implementing the idea. Chapter 4 provides an overview of our solution design and ways to handle the design challenges. Chapter 5 further breaks down the design and introduces each component in the proposed system. Chapter 6 gives a quantitive analysis of the performance aspects of the proposed system and its applicability and overhead. Chapter 7 discusses potential considerations and improvements of the system. Chapter 8 covers related works in adjacent and connecting research areas. Chapter 9 concludes the thesis and discusses future work.

CHAPTER 2: BACKGROUND AND MOTIVATION

This chapter first introduces how the two-dimensional page walk works (Section 2.1). Then, it explains why the two-dimensional page walk is inefficient and experimentally confirms that the inefficiency can greatly increase average page walk latency and reduce application performance in virtualized clouds (Section 2.2).

2.1 HARDWARE SUPPORTED MEMORY VIRTUALIZATION

In the native system, the page walker walks the page table to translate the virtual address to the physical address upon a translation lookaside buffer (TLB) miss. The translation requires up to four memory references for the 4-level x86 page table structure, which is used by most modern architectures. In the virtualized system, the hardware-supported memory virtualization, i.e., nested paging such as Intel extended page table (Intel EPT [21]) and AMD nested page table (AMD NPT [22]), enables the two-dimensional page translation.

Figure 1.1 (a) shows how the two-dimensional page translation works. The two-dimensional page translation needs to walk two page tables (the guest and host page table maintained by the respective guest and host operating systems) to translate a guest virtual address (GVA) of an application running in the guest level to its corresponding host physical address (i.e., the real physical address) in the host level. Specifically, the guest page table and the host page table are first used to translate the guest virtual address (GVA) to the guest physical address (GPA) in the guest level (Step 1-20). To obtain the GPA of the GVA, the page walker needs to walk the host page table to obtain the guest page table entries' (gL4, gL3, gL2, and gL1 in Figure 1.1 (a)) host physical addresses (Step 1-4, 6-9, 11-14, and 16-19 in Figure 1.1 (a)). Finally, the GPA of the GVA is translated to the final HPA by walking the host page table (Step 21-24 in Figure 1.1 (a)). Since the guest page table and the host page table are both 4-level page table structures, the two-dimensional page translation requires up to 24 memory references [20, 46, 47].

As today's data-intensive applications are pervasive and usually need large memory space to hold their working set, Intel releases the design of the 5-level page table [24], which significantly increases the addressable memory [23]. With such a 5-level page table structure, a two-dimensional page translation requires up to 35 memory references. This further exacerbates the address translation overhead in virtualized clouds.

Workload locality	**High level idea**	**Previous works**	
		Hardware approaches	Software approaches
Strong locality	Reducing TLB misses	ASAP [2], POM-TLB [25], CA-paging [50], RMM [26]	Gemini [51], Transparent Huge Page [4, 33]
Weak locality	Reducing page walk overhead	FPT [39], DMT [40], Compendia [49]	**Our proposed approach (HugeGPT)**

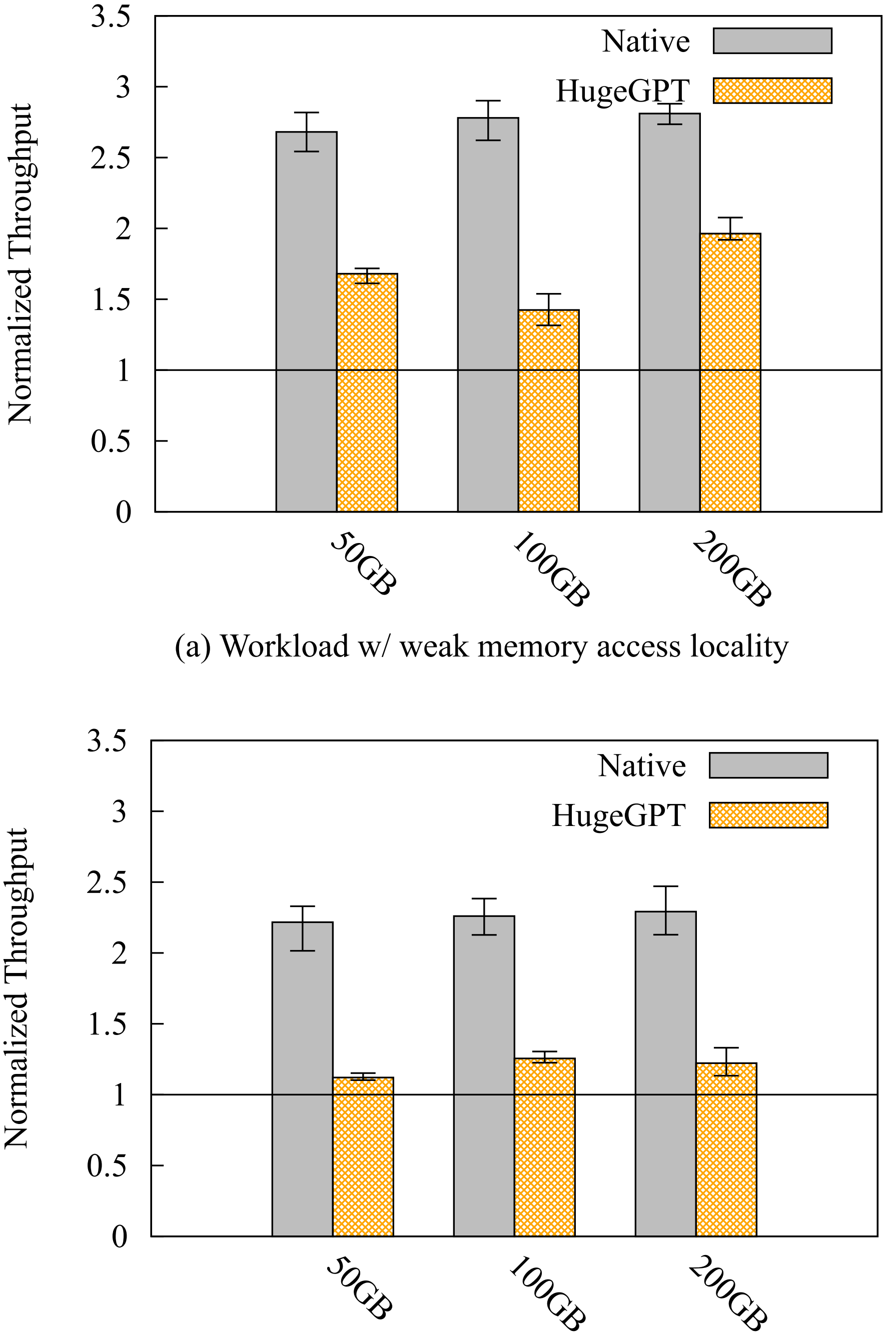
Table 2.1: A summary of related works based on the locality of workload memory access patterns. Please note that transparent huge pages are usually used to store application data on huge pages, so as to reduce TLB misses and their overhead.

2.2 INEFFICIENT TWO-DIMENSIONAL PAGE WALK

In modern systems architecture, translation lookaside buffer (TLB) capacity cannot scale at the same rate as memory capacity. TLB misses and address translation overhead have become a major performance bottleneck for workloads with weak memory access locality [2, 5, 6, 48]. This problem becomes even more pronounced in virtualization environments, as a TLB miss needs to walk through two layers of page tables, and the cost can be 6x as much as walking through one layer of page table in native environments [11, 21], as introduced in Section 2.1.

Existing research proposals on reducing address translation overhead mainly fall into two categories: reducing TLB misses and their overhead for applications with strong locality [2, 25, 26], and reducing page walk cache misses and their overhead for applications with weak locality [39, 49], as summarized in Table 2.1. HugeGPT falls into the second category. In this category, existing works need to modify hardware [20, 39, 40, 47, 49]. For instance, FPT [39] flattens the page table by merging adjacent layers of the page table. For the x86 4-level page table, it flattens the page global directory and the page upper directory, as well as the page middle directory and the page table entry, thereby translating 18 bits in a single memory access instead of the traditional 9 bits each in two memory accesses. It changes the page table structure and the page table walker to implement the flattened page tables. Commodity cloud servers are hard to integrate the approaches that need to modify the hardware in the near future. Therefore, we pursue a software solution that does not need to modify hardware and incur high overhead.

To illustrate the problem, we designed and implemented two micro-benchmarks. The first micro-benchmark shows almost no memory access locality. The micro-benchmark randomly accesses the memory with a total size of 50 GB, 100 GB, and 200 GB, respectively. The second

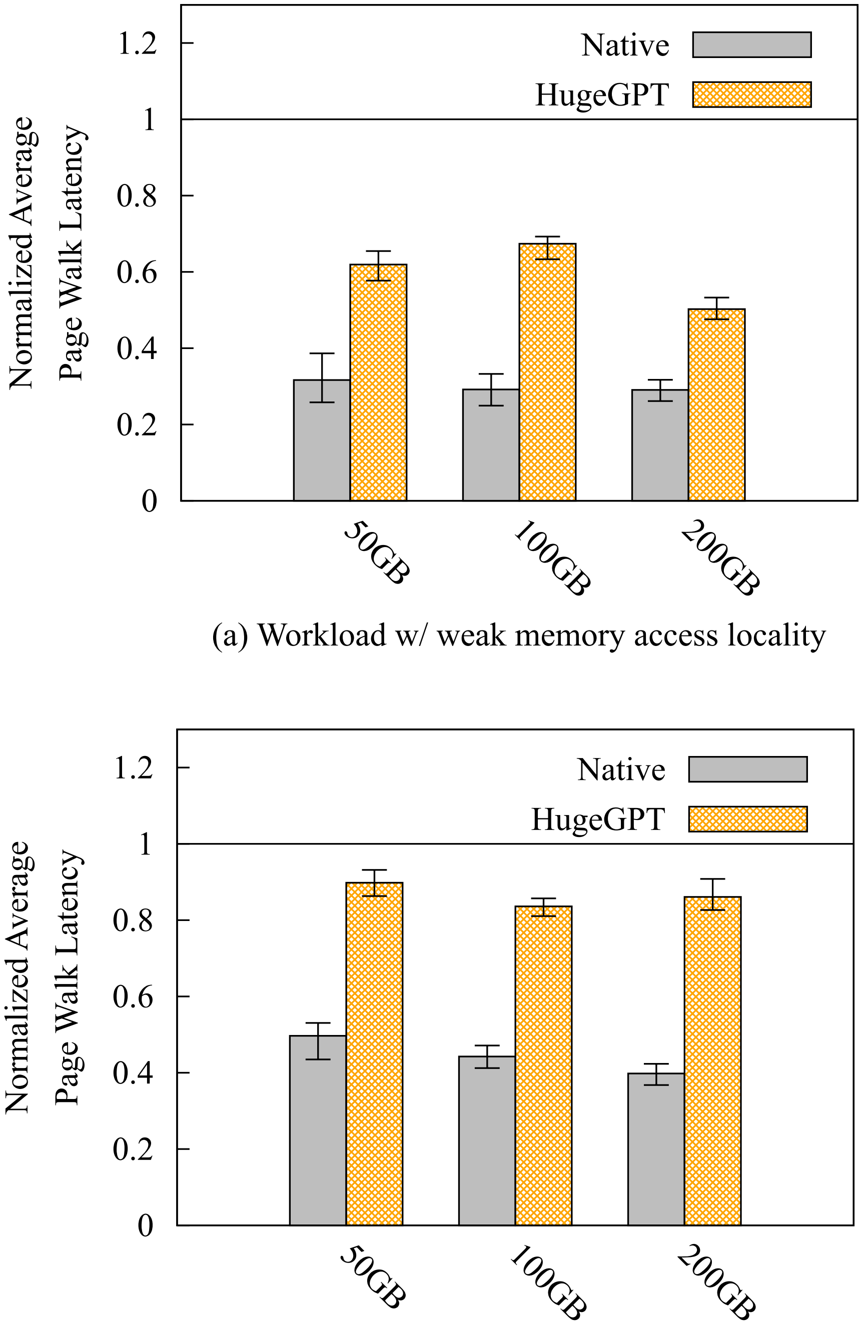


(a) Workload w/ weak memory access locality

(b) Workload w/ no memory access locality

Figure 2.1: Throughputs of native system, and HugeGPT. Throughputs are normalized to vanilla Linux/KVM.

micro-benchmark shows weak spatial locality. It accesses each 4 KB memory page once with the same set of working set sizes as those in the first micro-benchmark. We follow the same approach



(a) Workload w/ weak memory access locality

(b) Workload w/ no memory access locality

Figure 2.2: Average page walk latency of native system and HugeGPT. Average page walk latencies are normalized to vanilla Linux/KVM.

in the previous work [52] to generate workloads with weak memory access locality. To measure the throughputs of micro-benchmarks, we measure the memory accesses performed per second.

Figure 2.1 shows the throughputs of the two micro-benchmarks when they are tested with a native system, vanilla Linux/KVM, and HugeGPT, respectively. HugeGPT offers 44% more throughput compared to vanilla Linux/KVM on average. This shows the inefficiency of the two-dimensional page walk used by vanilla Linux/KVM. Compared to the one-dimensional page walk used in the native environments, the inefficiency of the two-dimensional page walk becomes even worse. To further understand the inefficiency, we profile the average page walk latency of the three systems. We show the test results in Figure 2.2. Compared to vanilla Linux/KVM, HugeGPT reduces the average page walk latency by 41% for workloads with weak memory access locality and 14% for workloads with almost no memory access locality on average.

CHAPTER 3: MAIN IDEA AND TECHNICAL CHALLENGES

3.1 MAIN IDEA

As explained and confirmed in Chapter 2, the two-dimensional page walk used by vanilla Linux/KVM incurs much longer average page walk latency compared to the one-dimensional page walk used in the native system. The reason is that vanilla Linux/KVM incurs more page walk cache misses and steps to walk page tables for workloads with weak memory access locality in comparison to the native system.

To reduce page walk cache misses and the number of memory references incurred by two-dimensional page walk, our main idea is to store the guest page table data on the host huge page, such that the steps to walk two-dimensional page tables and the page walk cache misses can be reduced. Since guest page tables are stored on host huge pages, to obtain the guest physical address of the guest page table entry, it only needs to walk the 3-level host page table, improving the page walk cache capability and shortening the 24 memory references in the two-dimensional page walk to 20 memory references in the worst case, as shown in Figure 1.1 (b).

Intuitively, the guest operating system (OS) accesses the application's whole working set and has a worse locality compared to the host OS that only accesses the page table of the application. Therefore, rows gL4 and gL3 of the guest page table may be cached, as shown in Figure 1.1; and columns nL4, nL3, and nL2 of the host page table may be cached. This is also corroborated by the previous work [39]. We shaded the cached page table entries in Figure 1.1.

Empirically, we profile the average memory references in the two-dimensional page walk. We first get the total memory references by collecting the last level cache misses ($total\_ref$). Then, we calculate the memory references of accessing application data by using the total working set size divided by page size ($data\_ref = total\_working\_set\_size/4KB$). Next, we remove memory references incurred by accessing application data from the total memory references and get the total memory references incurred by page walks ($total\_ref - data\_ref$). Finally, we use total memory references incurred by page walks divided by the total number of page walks and get the memory references of each page walk ($per\_pw\_mem\_ref = (total\_ref - data\_ref)/num\_of\_pw$). The test results show that each page walk incurs about 5 memory references, which are consistent with the memory references that are not shaded in Figure 1.1.

With HugeGPT, translating the guest physical address of the guest virtual address to the final host physical address (Step 21-24 in Figure 1.1) still needs to walk the 4-level host page table as shown in the last row of Figure 1.1. This is because user application data is still stored on base pages (4 KB pages) to avoid the adverse effects caused by transparent huge pages [42, 44, 53, 54].

Since the size of the guest page table data is much smaller than the size of user application data (around 200 MB page table data for 100 GB user application data), the adverse effects of using huge pages are negligible.

Figures 2.1 and 2.2 confirm the effectiveness of the proposed approach, i.e., HugeGPT. Compared to vanilla Linux/KVM, HugeGPT offers up to 96% more throughput and 50% lower average page walk latency. HugeGPT provides more performance improvement for workloads with weak memory access locality than it for the workloads with no memory access locality. This is because it is easier to cache root page table entries (e.g., nL4, nL3, gL4, and gL3 as shown in Figure 1.1 (a)) for weak memory access locality workload compared to no memory access locality workload, such that removing the leaf page table entries can bring more benefits. However, in the random memory access (no memory access locality), upper-level page table entries may be poorly cached, so the effectiveness of removing the leaf page table entries is reduced.

3.2 TECHNICAL CHALLENGES

To realize the proposed approach, there are two main technical challenges. To form host huge pages for storing guest page table data, it needs to form host huge pages based on the huge page sized guest physical memory regions that are used to store guest page table data. There are two technical challenges to achieving it.

The first challenge is how to filter out guest page table data and store it on specific guest physical memory regions at the guest level. The guest memory allocator does not distinguish memory allocations for guest page table data and other application/system data, such that guest page table pages are mixed with other application/system data pages and randomly scattered in the guest physical memory space. Since we need to store guest page table pages on host huge pages, we have to filter out memory allocations of guest page table pages. To address this challenge, HugeGPT modifies the guest memory subsystem to filter out memory allocations for guest page table data and allocate huge page sized guest physical memory regions to store the guest page table data.

The second challenge is how to identify the guest physical memory regions that store the guest page table data at the host level. To back guest page tables with host huge pages, the host needs to figure out the guest physical memory regions that store guest page table data and create host huge pages to back these regions. However, due to the semantic gap between the guest and the host, the host memory allocator cannot figure out the guest physical memory regions that are used to store guest page table data. To address this challenge, HugeGPT marks all huge page sized guest physical memory regions that store guest page table data, such that the host can form host huge pages based on these guest physical memory regions upon the first page faults on these regions.

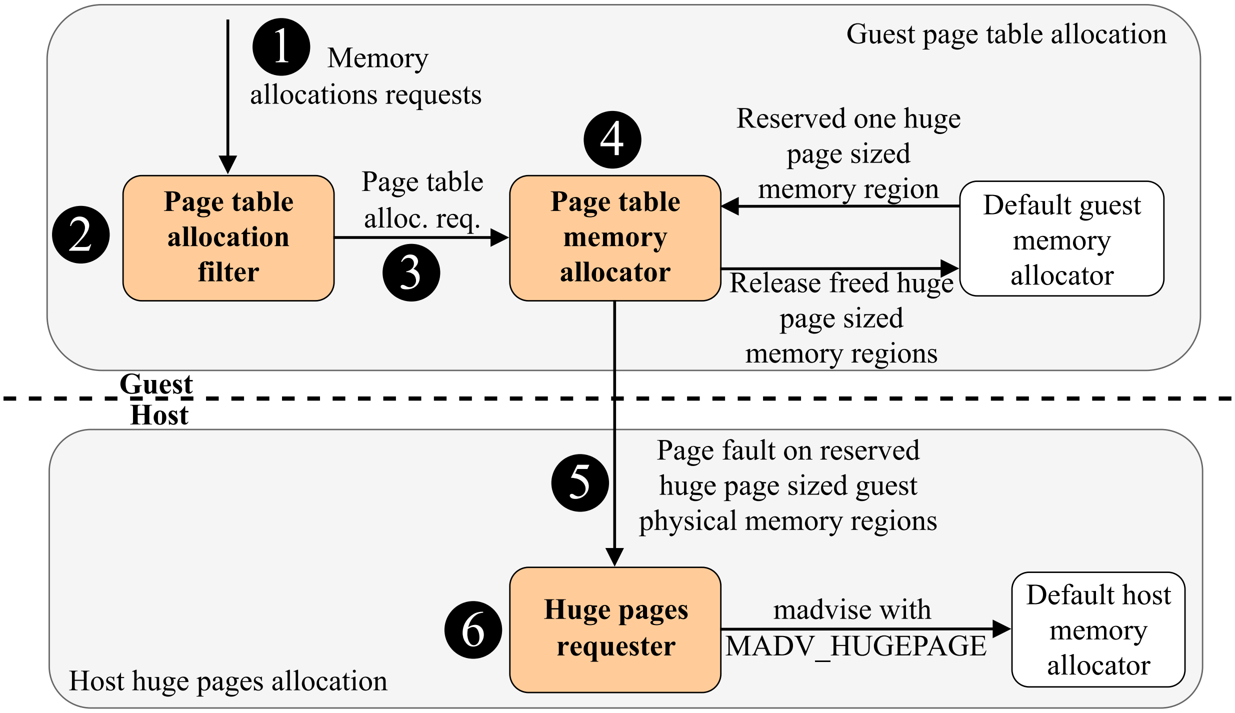

Guest page table allocation
1 Memory allocations requests
2
Page table allocation filter
Page table alloc. req.
3
4
Page table memory allocator
Reserved one huge page sized memory region
Default guest memory allocator
Release freed huge page sized memory regions
Guest
Host
5
Page fault on reserved huge page sized guest physical memory regions
6
Huge pages requester
madvise with MADV_HUGEPAGE
Default host memory allocator
Host huge pages allocation

Figure 4.1: HugeGPT system overview. Key components are shaded in orange.

This chapter gives a design overview of the HugeGPT system. It also explains the two-phase workflow of the HugeGPT system.

Figure 4.1 shows the system architecture of HugeGPT. HugeGPT includes three key components that are shaded in orange. Page table allocation filter is used to filter out memory allocations of page table pages. Page table memory allocator is used to allocate page table pages onto the assigned huge page sized guest physical memory regions. This is to ensure guest page tables can be stored on host huge pages. Huge pages requester forms huge pages for the designated huge page sized guest physical memory regions, such that the guest page table pages on these huge page sized guest physical memory regions can be backed by the host huge pages.

HugeGPT works in two phases. In the first phase, a host huge page is created upon the first page fault that is requested from the memory allocation of guest page table page. The memory allocations mixed with user application data pages, page table pages, and others are generated (❶). Memory allocation requests of page table pages are filtered out by the page table allocation filter (❷). To allocate guest physical pages to store guest page tables, the page table allocation requests are sent to the page table memory allocator (❸). Then, the page table memory allocator issues the page fault with a reserved huge page sized guest memory region which was assigned by the default guest memory allocator beforehand (❹ and ❺). At last, the huge pages requester sends a `madvise` request with `MADV_HUGEPAGE` command to the default host memory allocator to form a host huge page based on the reserved huge page sized guest physical memory region (❻). When the `madvise` is called with `MADV_HUGEPAGE` command, the system will try to directly allocate huge pages if the guest physical memory region is aligned to huge pages.

In the second phase, as the huge page sized guest physical memory region has been backed by the host huge page, the following memory allocations of guest page table pages will be stored on this reserved huge page sized guest physical memory region. Specifically, the page table memory allocator will not issue page fault requests to the host operating system if the reserved huge page sized guest physical memory region is not used up (❹). As a side effect, the VM exits caused by page faults are minimized.

CHAPTER 5: DESIGN AND IMPLEMENTATION DETAILS

This chapter first introduces the initialization of HugeGPT upon the system starts. Then, it explains how the guest page table memory allocator and guest page table allocation filter work. It also presents how host huge pages are created based on the huge page sized guest physical memory regions. At last, it covers how the HugeGPT can be implemented in the Linux kernel.

5.1 SYSTEM INITIALIZATION

The goal of the initialization is to set up HugeGPT before it is used. The initialization is conducted immediately after the system starts. In the initialization, the HugeGPT guest page table memory allocator first pre-allocates several (configurable) huge page sized memory regions from the default guest memory allocator. These reserved guest physical memory regions are used to store guest page table data of applications running in virtual machines. Please note that the size of the reserved guest physical memory regions is small as 100 GB application data only needs around 200 MB page table data. Then, the HugeGPT guest page table memory allocator notifies the guest physical addresses of these pre-allocated guest physical memory regions to the HugeGPT huge pages requester in the host level. Since the notification is not frequent, the communication overhead between the guest and the host is small (confirmed in Section 6.6). This lets the host know of the guest physical locations of these huge page sized guest physical memory regions that are used to store guest page table data, such that the host can later form huge pages based on these huge page sized guest physical memory regions.

5.2 GUEST PAGE TABLE ALLOCATION

Guest page table allocation in HugeGPT is designed to filter out guest page table data and allocate guest page table pages on the pre-allocated huge page sized guest physical memory regions. To achieve the goal, we modified kernel functions for allocating and freeing page table pages in the guest operating system (i.e., `pte_alloc_one` and `free_pmds`), such that they will pass the page allocation and free requests to HugeGPT page table memory allocator. This will not only filter out page allocations for page table data but also store page table data on reserved huge page sized guest physical memory regions. Since page table pages are allocated and freed with dedicated kernel functions, our approach can make sure that the HugeGPT page table memory allocator is only used to manage page table data. Upon the allocation requests for page table pages, the HugeGPT page table memory allocator returns free pages from the memory pool of the pre-allocated huge page sized guest physical memory regions. After the page table pages are allocated, the page table entries are updated. This may trigger the first page fault on the huge page sized guest physical memory region that stores the page table pages. Upon the first page fault on the huge page sized guest physical memory region, the host is notified to allocate the host physical frame to back the huge page sized guest physical memory region. When page table pages are freed, they are returned to the HugeGPT guest physical memory pool.

5.3 HOST HUGE PAGES ALLOCATION

HugeGPT's host huge pages allocation is designed to create host huge pages based on the reserved huge page sized guest physical memory regions that are used to store guest page table data. After HugeGPT initialization, the host huge page allocation component records the reserved guest physical locations of the huge page sized guest physical memory regions. Upon the first page fault of each huge page sized guest physical memory region, the host huge page allocation component allocates huge page sized host physical memory region to back huge page sized guest physical memory region, such that host huge pages are formed. HugeGPT realizes it through leveraging the `madvise` mechanisms. Specifically, using `MADV_HUGEPAGE` command in `madvise` can form huge pages with designated guest physical addresses. This can minimize the modifications to both the guest and the host operating systems.

HugeGPT re-executes the initialization process once the memory pool of pre-allocated huge page sized guest physical memory regions is run out of space. In addition, when HugeGPT fails to allocate host huge pages (e.g., severe memory fragmentation), HugeGPT host huge pages allocation will fallback to allocate 4 KB base pages, in order to make systems run correctly.

5.4 IMPLEMENTATION

HugeGPT can be practically implemented in the Linux kernel without introducing breaking changes. We have implemented HugeGPT prototypes in Linux/KVM 5.15 and Linux/KVM 6.1. We added and changed around 710 lines of source code mainly in the page table allocation of the kernel memory management subsystem. We added a new kernel file (`mm/hugegpt.c`) to implement the guest-side memory allocator (around 460 lines of source code). For the host-side component, we added around 175 lines of code in `arch/x86/kvm/x86.c` to realize it.

To allocate and organize guest page table pages, we create a custom memory allocator that builds a physically contiguous memory pool at a user-configurable size during guest system boot. The default pool size is 1% of the total memory allocated to the virtual machine. The pool is allocated using the Linux contiguous page allocator, alloc_contig_pages, implemented in the buddy allocator with GFP_PGTABLE_USER.

We replace all calls to the default Buddy allocator in the user page table allocation routines so that the HugeGPT allocator will handle these requests. We left the kernel page table allocation routines unchanged. When allocating a new page table page, the allocator will prefer the most recently freed page to maximize the host translation lookaside buffer and host page walk cache hit rate. If the memory pool is depleted, the HugeGPT allocator will reinvoke alloc_contig_pages to create another memory pool.

To make sure the guest page table memory pool is backed by huge pages, the host HugeGPT kernel invokes do_madvise with advise MADV_HUGEPAGE once the pool is created. The guest HugeGPT allocator will then touch each memory page in the pool to fault them as huge pages. After the faulting is done, the host system will then walk the host page table for the memory pool region using walk_page_range to check for any failed huge page allocations. If any, madvise_collapse will be invoked to try defragmenting the memory and promoting the failed regions to huge pages on best effort.

The operations of guest and host operating systems are coordinated through two hypercalls we added to the system, named KVM_HC_HGPT_REQ_MAP and KVM_HC_HGPT_REQ_GATHER respectively. The KVM_HC_HGPT_REQ_MAP hypercall is used to notify the host kernel when creating and expanding the memory pool, which eventually triggers the do_madvise operations. The KVM_HC_HGPT_REQ_GATHER hypercall is used to notify the host that the guest-side faulting is completed, which triggers the page table check and potentially madvise_collapse.

CHAPTER 6: EVALUATION

We have implemented a HugeGPT prototype based on Linux/KVM 5.15 for evaluation. We have evaluated HugeGPT extensively with a diverse set of workloads and compared HugeGPT to native systems (without virtualization), vanilla Linux/KVM, Linux transparent huge page (THP) [33] and Gemini [51]. The objective of the evaluation is four-fold: 1) to show that HugeGPT can improve throughput for throughput-oriented workloads compared to vanilla Linux/KVM (Section 6.2), 2) to show that HugeGPT can reduce mean and tail application response latency of latency-sensitive workloads compared to vanilla Linux/KVM (Section 6.3), 3) to compare HugeGPT with related systems (Section 6.4), and 4) to evaluate the applicability and overhead of HugeGPT (Section 6.5 and Section 6.6).

Workload Name	**Workload Description**	**Working Set Size**
Sphinx	Speech recognition like Apple Siri [55].	30 GB
Moses	Real time translation like Google translate [56].	25 GB
Masstree	In memory key-value store (50% GET, 50% SET) [57].	25 GB
Specjbb	Industry-standard JAVA middleware benchmark [58].	60 GB
Shore	Transactional database with TPCC [59].	30 GB
Redis	Serve requests (random keys, 50% SET, 50% GET) [60].	155 GB
Memcached	Serve requests (random keys, 50% SET, 50% GET) [61].	95 GB
Canneal	Chip design optimizer [62].	62 GB
Graph500	Graph analysis.	123 GB
GUPS	Giga Updates Per Second benchmark [63].	128 GB
XSBench	Monte Carlo neutron transport compute kernel [64].	84 GB
BTree	Index lookup benchmark [5].	125 GB

Table 6.1: Programs and workloads used to test HugeGPT.

6.1 EXPERIMENT SETTINGS

Our evaluation was conducted on a Hewlett Packard Enterprise (HPE) ProLiant DL580 Gen10 server with four Intel Xeon Gold 6138 processors, 256 GB memory, and two 2 TB solid-state drives. Each processor has 20 cores. With KVM-accelerated QEMU, we built virtual machines (VMs), each VM with 40 virtual CPUs (vCPUs) and 240 GB memory. We set the number of

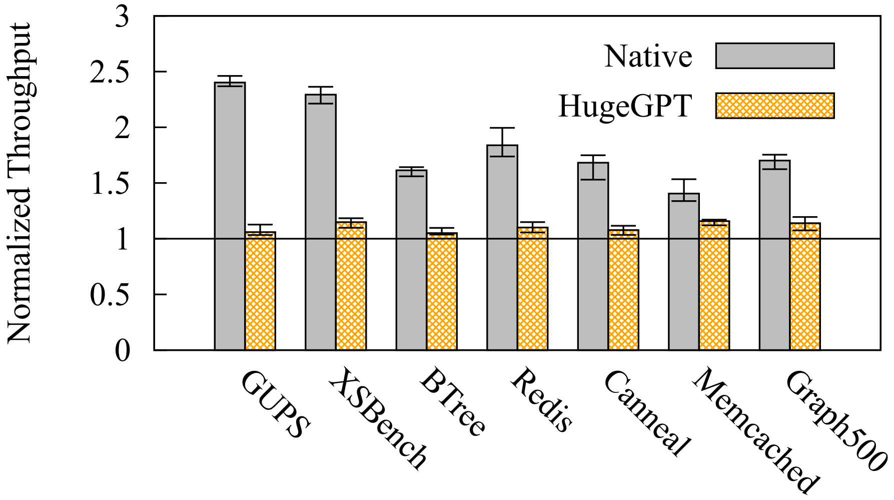


Figure 6.1: Throughputs of throughput-oriented workloads. Throughputs are normalized to vanilla Linux/KVM.

application threads equal to the number of vCPUs. Both the host and guest operating systems are Ubuntu Linux 20.04 with Linux kernel 5.15. We test HugeGPT with a large and diverse set of workloads generated by typical applications from different domains (e.g., database server, key/value store, AI workload, scientific applications, etc.), as summarized in Table 6.1. We profile these workloads using the Linux Perf tool to read performance hardware counters. It shows that these workloads all spend a significant part of execution time (>20%) on page walks. Hence, these workloads have weak memory access locality. Two workloads (i.e., Swaptions and Raytrace) are not translation lookaside buffer (TLB) sensitive and page walk intensive. They are used to test the overhead of HugeGPT. In the experiments, each VM encapsulates one workload.

We categorize the benchmarks into two types: throughput-oriented benchmarks (e.g., GUPS, XSBench, and BTree) and latency-sensitive benchmarks (e.g., Sphinx, Moses, and Masstree). We first measure the throughputs of throughput-oriented workloads reported by these workloads. Then, we collect average and tail latencies reported by the latency-sensitive workloads. Some workloads (e.g., Redis and Memcached workloads in YCSB [65]) report both throughputs and latencies, so we present both of them in the test results. The performance measurements may vary significantly across different workloads. When we present them in figures, for clarity, we normalize them against those of vanilla Linux/KVM, as indicated in the figures.

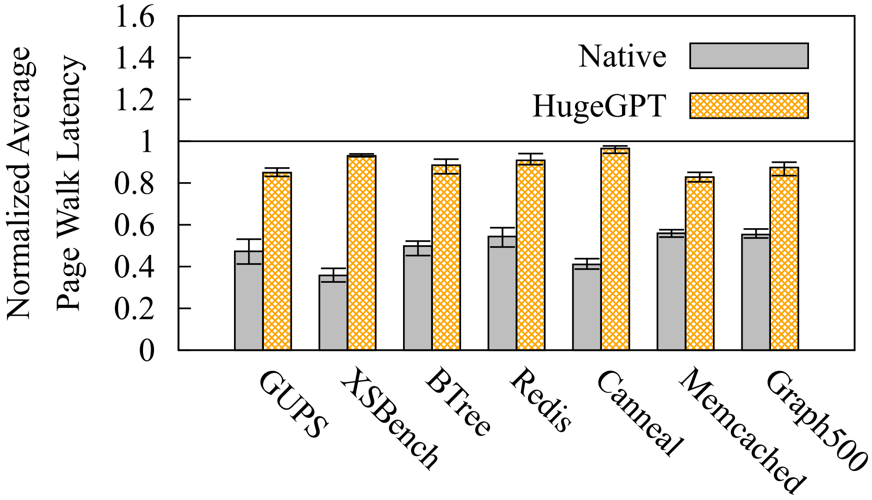


Figure 6.2: Average page walk latencies of throughput-oriented workloads. Average page walk latencies are normalized to vanilla Linux/KVM.

6.2 EXPERIMENTS WITH THROUGHPUT-ORIENTED WORKLOADS

Figure 6.1 shows the throughputs of throughput-oriented workloads when three systems (i.e., native system, HugeGPT, and vanilla Linux/KVM) are tested with these workloads. On average, HugeGPT offers 10% more throughput compared to vanilla Linux/KVM. With HugeGPT, the page walker does not need to walk the leaf page table entries of the nested page table while walking the two-dimensional page tables, so HugeGPT reduces the page walk cache misses and performs better than vanilla Linux/KVM. For the average throughput, the native system outperforms HugeGPT by 68%. This is because translation lookaside buffer and page walk caches may cache most page table entries in the native system.

To further understand why HugeGPT's throughput is better than vanilla Linux/KVM and worse than the native system, we profile the average page walk latency when the workload is tested in different systems. We show the results in Figure 6.2. As we expected, HugeGPT reduces the average page walk latency by 12% compared to vanilla Linux/KVM and increases the average page walk latency by 92% compared to the native system on average. This confirms HugeGPT's effectiveness in improving application throughput by reducing the overhead of two-dimensional page walks in vanilla Linux/KVM.

Figure 6.1 also shows that HugeGPT increases the throughput by the largest percentage (16%) for the Memcached workload and the smallest percentage (5%) for the BTree and GUPS workloads. For the Memcached workload, it strides the memory with weak memory access locality so more page table entries may be cached by translation lookaside buffer and page walk caches compared to random memory access patterns. Therefore, reducing the leaf page table entries of

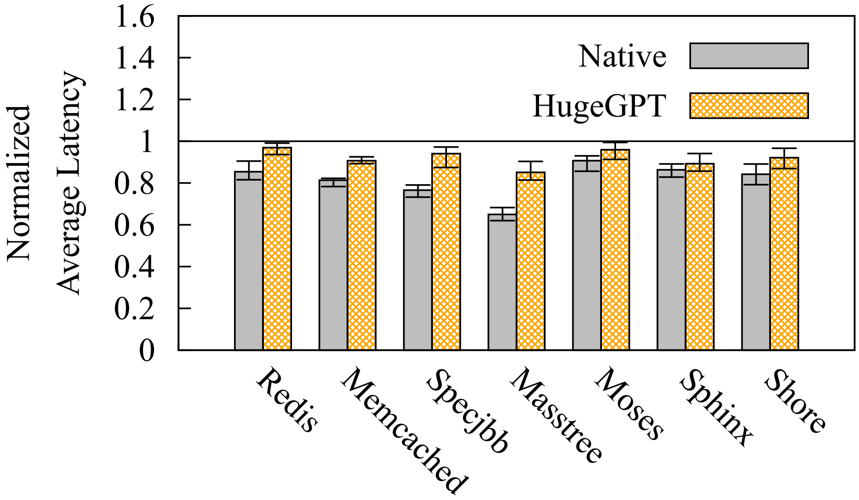


Figure 6.3: Average latencies of latency-sensitive workloads. Average latencies are normalized to vanilla Linux/KVM.

the nested page table in HugeGPT shows more performance improvement. This is consistent with the performance observation in Section 2.2. Since GUPS and BTree workloads conduct random memory accesses, HugeGPT's performance improvement on these workloads is less. For instance, GUPS is designed to measure the number of memory locations that can be randomly updated in one second, so it shows almost no memory access locality such that it may be hard to cache lower-level page table entries.

6.3 EXPERIMENTS WITH LATENCY-SENSITIVE WORKLOADS

Figure 6.3 shows the average application response latencies of different systems when they are tested with latency-sensitive workloads. On average, the native system shows the lowest average latency as most page table entries can be cached while walking the one-dimensional page table. In the worst case, the native system only incurs four memory references. Relative to the native system, HugeGPT increases the average latency by 16% on average. Compared to vanilla Linux/KVM, HugeGPT reduces the average latency by 8% on average. This is because HugeGPT reduces the average page walk latency of the two-dimensional page walks by up to about 50% as explained in Section 2.2. HugeGPT reduces page walk cache misses and the number of memory references in two-dimensional page walk from 24 to 20 in the worst case.

To further pinpoint why HugeGPT increases the average application latency compared to the native system and reduces the average latency compared to vanilla Linux/KVM, we profile the average page walk latency when the latency-sensitive workloads are tested with the three systems.

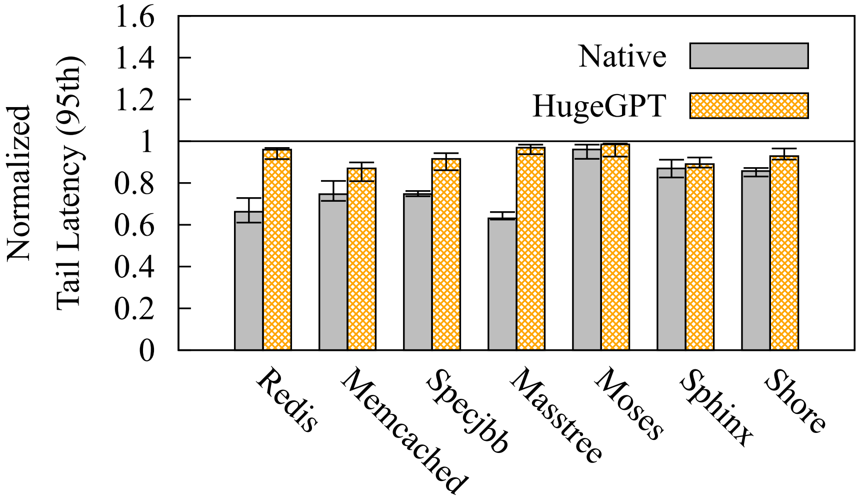


Figure 6.4: 95th percentile tail latencies of latency-sensitive workloads. Tail latencies are normalized to vanilla Linux/KVM.

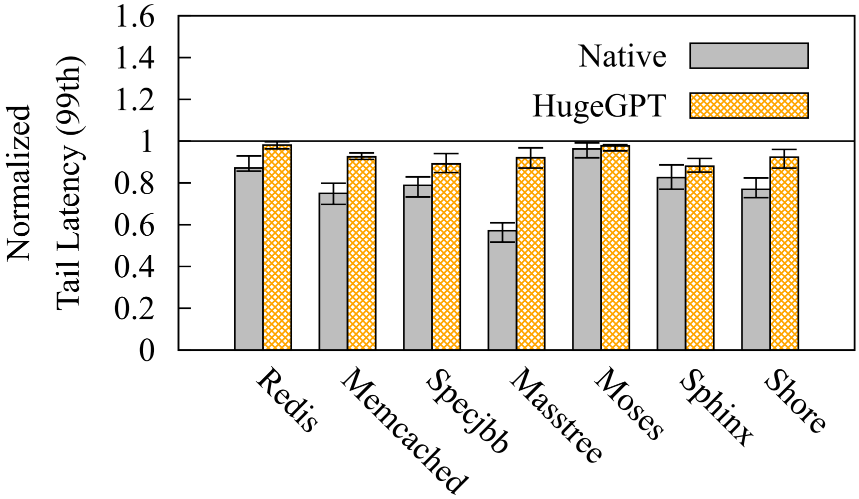


Figure 6.5: 99th percentile tail latencies of latency-sensitive workloads. Tail latencies are normalized to vanilla Linux/KVM.

We show the profiling results in Figure 6.6. On average, HugeGPT increases the average page walk latency by 62% compared to the native system and decreases the average page walk latency by 8% compared to vanilla Linux/KVM. This is consistent with the average application latency results. The result also shows HugeGPT's effectiveness in reducing the overhead of two-dimensional page walks for latency-sensitive workloads in comparison to vanilla Linux/KVM.

Figure 6.4 and Figure 6.5 show the 95th percentile tail application response latencies and the 99th percentile tail latencies, respectively, when the latency-sensitive workloads are tested with

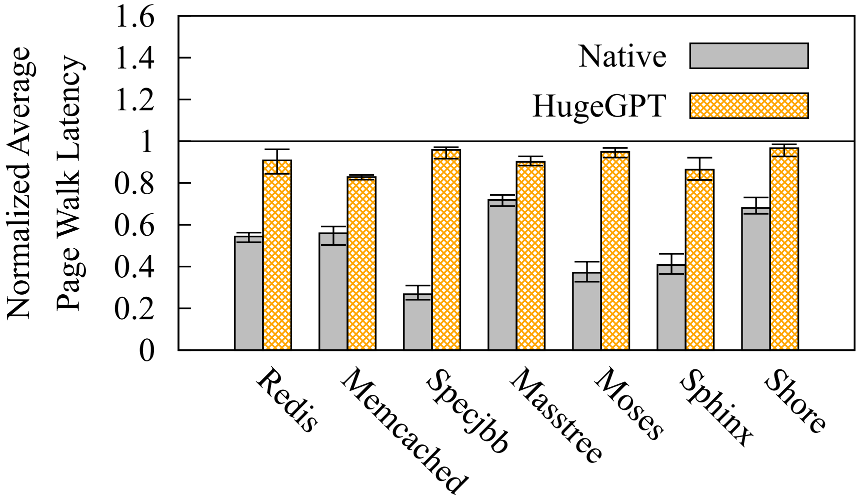


Figure 6.6: Average page walk latencies of latency-sensitive workloads. Average page walk latencies are normalized to vanilla Linux/KVM.

the three systems. On average, HugeGPT provides 8% lower 95th percentile tail latency 8% lower 99th percentile tail latency compared to vanilla Linux/KVM. On the other hand, HugeGPT shows 32% higher 95th percentile tail latency and 30% higher 99th percentile tail latency relative to the native system. The tail latency test results are consistent with the average page walk latency of the three systems as shown in Figure 6.6, where the average page walk latency of HugeGPT is 62% higher than the native system, but 8% lower than vanilla Linux/KVM.

Figure 6.3, Figure 6.4, and Figure 6.5 also show that HugeGPT shows small performance advantages for some workloads (e.g., Moses and Masstree) and large performance advantages for some other workloads (e.g., Specjbb and Sphinx). This is because Specjbb and Sphinx show weak memory access locality. HugeGPT performs better on these workloads as explained in Section 2.2. Memory access patterns in Moses and Masstree workloads are more random than Specjbb and Sphinx. HugeGPT does not show good performance with workloads with random memory access patterns as lower page table entries may not be cached.

6.4 COMPARISONS WITH RELATED SYSTEMS

We compare HugeGPT with Linux transparent huge page (THP) and Gemini [51] on x86 4-level page table and 5-level page table, respectively. To support the 5-level page table and compare these systems in a fair manner, we change our platform to a DELL PowerEdge R750 server with two Intel Xeon Gold 6346 processors (32 cores, 2046 translation lookaside buffer entries, and 36 MB last-level CPU cache), 256 GB memory, and 2 TB solid-state drive. With KVM-accelerated QEMU,

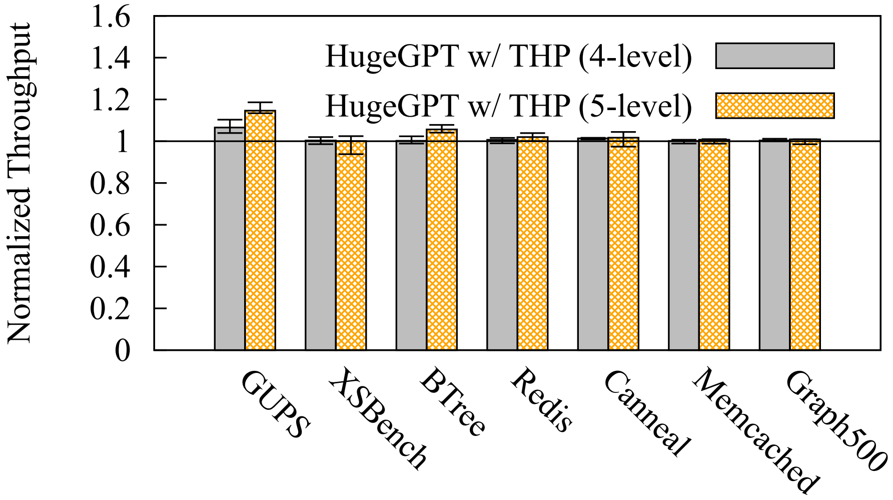


Figure 6.7: HugeGPT's throughput improvement compared to Linux transparent huge page (THP) [33] when 4-level and 5-level page table are used respectively. Throughputs are normalized to Linux THP.

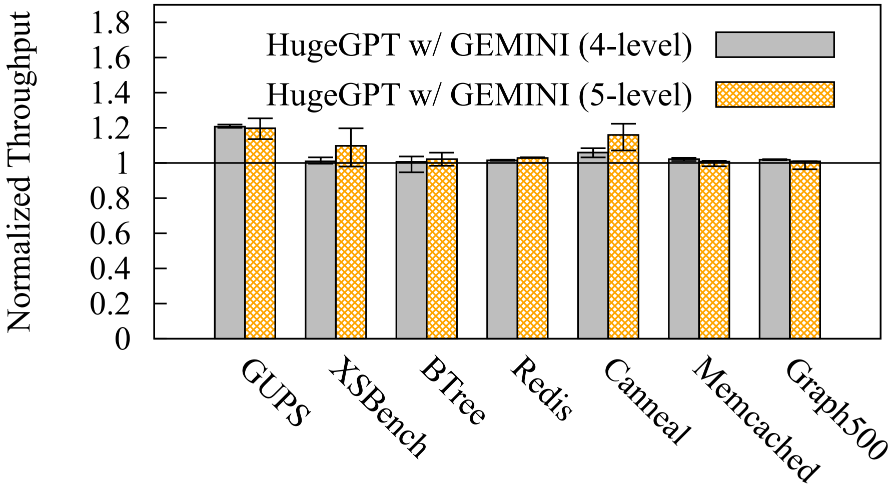


Figure 6.8: HugeGPT's throughput improvement compared to Gemini [51] when 4-level and 5-level page table are used respectively. Throughputs are normalized to Gemini.

we built the virtual machine with 32 vCPUs and 240 GB memory. Both host and guest operating systems are Ubuntu Linux 20.04 with the same Linux 5.10 kernel and software configuration, unless otherwise indicated.

Figure 6.7 and Figure 6.8 compare HugeGPT's throughput with that for Linux transparent huge page (THP) and Gemini [51], respectively, when the 4-level page table and the 5-level page table are used. When the 4-level page table is used, HugeGPT outperforms THP by up to 6% and Gemini

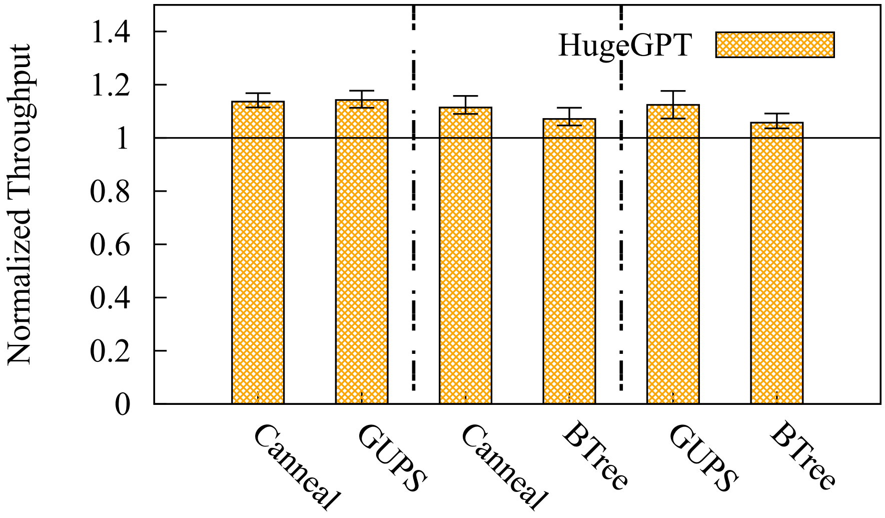


Figure 6.9: Throughputs of throughput-oriented workloads when they are co-located on the same server. Throughputs are normalized to vanilla Linux/KVM.

by up to 21%. When the 5-level page table is used, HugeGPT offers up to 15% and 20% more throughput, compared to THP and Gemini, respectively. HugeGPT shows better performance with the 5-level page table because the 5-level page table incurs more page walk overhead. This gives HugeGPT more potential to obtain benefits. The comparison also confirms that HugeGPT can further improve the throughput of workloads with weak memory access locality after THP or Gemini is used. As introduced in Section 2.2, HugeGPT is complementary to THP and Gemini, as they mainly target workloads with strong memory access locality, and HugeGPT mainly optimizes workloads with weak memory access locality.

6.5 APPLICABILITY

To evaluate HugeGPT's applicability, we co-locate two virtual machines (VM) on the server and test HugeGPT's performance when multiple VMs are co-located on the same server. We choose this test scenario as VM colocation on the same server is pervasive in clouds. We mainly test three settings: 1) two throughput-oriented applications running in VMs are co-located on the same server; 2) two latency-sensitive applications running in VMs are co-located on the same server; and 3) six throughput-oriented applications running in VMs are co-located on the same server.

Figure 6.9 shows throughputs of throughput-oriented workloads when HugeGPT and vanilla Linux/KVM are tested under the system setting shown in Section 6.1. Under this setting, HugeGPT outperforms vanilla Linux/KVM by 12% on average. This shows that HugeGPT can improve

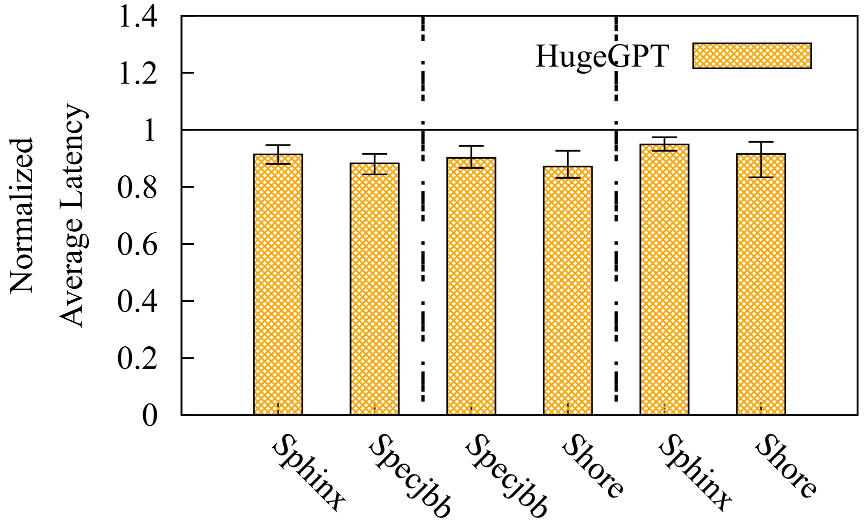


Figure 6.10: Average latencies of latency-sensitive workloads when they are colocated on the same server. Average latencies are normalized to vanilla Linux/KVM.

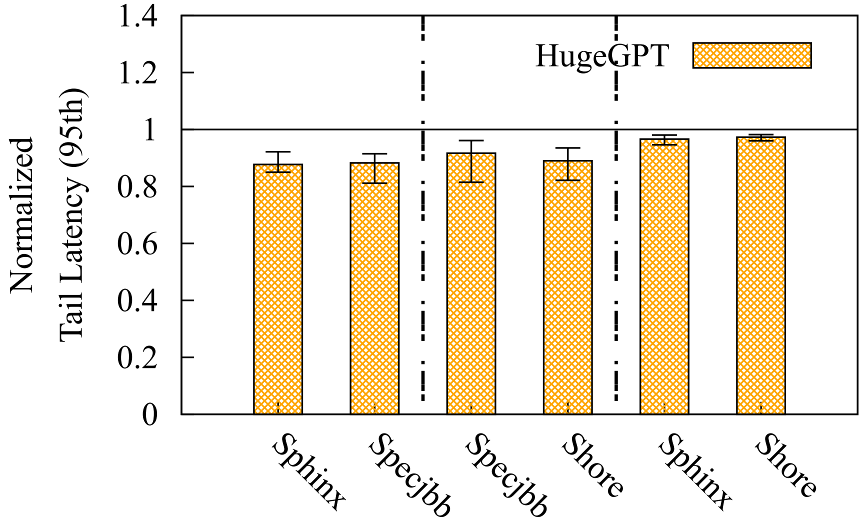


Figure 6.11: 95th percentile tail latencies of latency-sensitive workloads when they are colocated on the same server. Tail latencies are normalized to vanilla Linux/KVM.

application performance by reducing two-dimensional page walk overhead when multiple page walk intensive throughput-oriented applications are co-located. These experiments also show HugeGPT's effectiveness in multi-threaded applications and multiple processors.

Figure 6.10, Figure 6.11, and Figure 6.12 show the average latency, 95th percentile tail latency, and 99th percentile tail latency, respectively, when HugeGPT and vanilla Linux/KVM are tested under the second setting. HugeGPT decreases the average latency by 11%, the 95th percentile

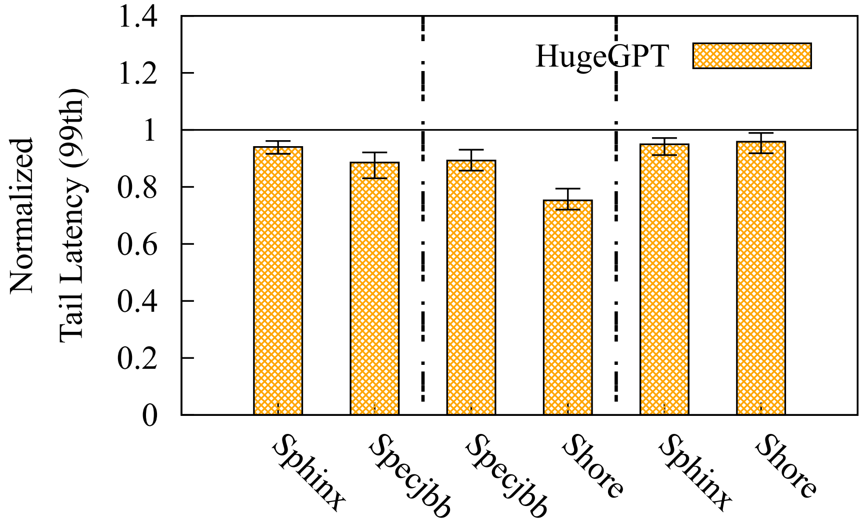


Figure 6.12: 99th percentile tail latencies of latency-sensitive workloads when they are colocated on the same server. Tail latencies are normalized to vanilla Linux/KVM.

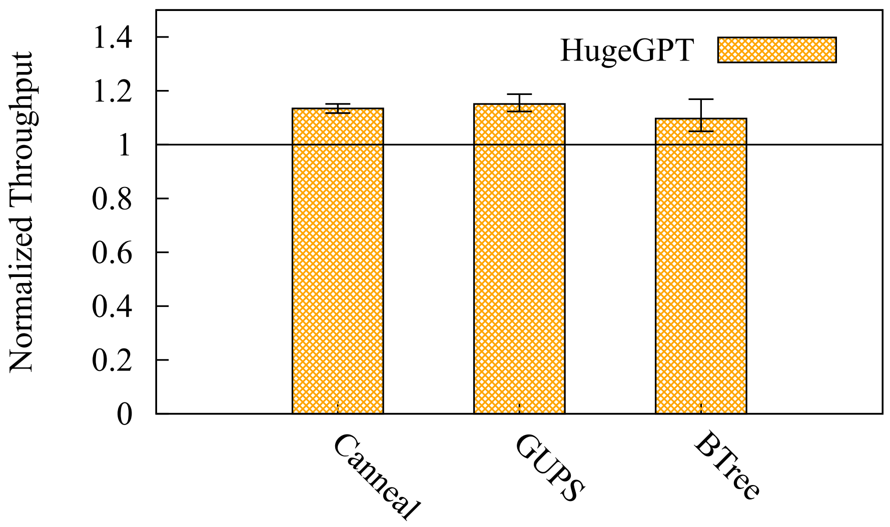


Figure 6.13: Throughputs of six throughput-oriented workloads colocated on the same server. We run two copies of each workload. Throughputs are normalized to vanilla Linux/KVM.

tail latency by 12%, and the 99th percentile tail latency by 9% on average, in comparison to vanilla Linux/KVM. This shows that HugeGPT can reduce average and tail latencies when latency-sensitive workloads are colocated on the same server.

Figure 6.13 shows HugeGPT's throughput when six workloads are colocated on the same server. We run two copies of each workload (Canneal, GUPS, and BTree). Since copies of the same workload have similar throughput, we plot the average throughput for the copies of each workload.

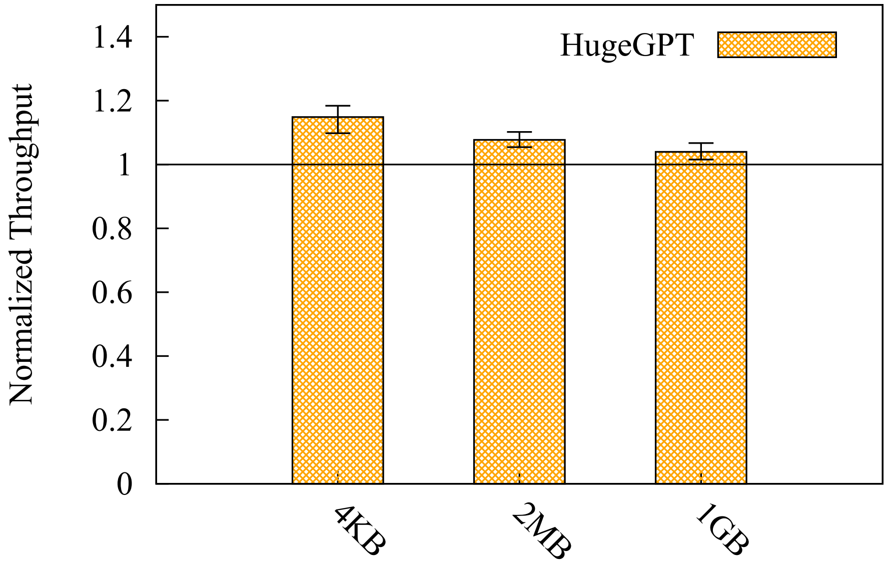


Figure 6.14: HugeGPT's throughputs with different memory page sizes. Throughputs are normalized to vanilla Linux/KVM.

The VM running each workload has 12 vCPUs and 40 GB memory. The working set size of each workload is kept around 35 GB. This prevents the total workload working set size from exceeding the server's memory capacity. On average, HugeGPT outperforms vanilla Linux/KVM by 13%. This is consistent with the test results when two workloads are consolidated on the same server, as shown in Figure 6.9.

Figure 6.14 shows HugeGPT's throughput for different page sizes. We run XSBench to test HugeGPT's throughput. We choose 4 KB, 2 MB, and 1 GB memory page sizes because the current x86 CPU only supports those page sizes. As the page size increases from 4 KB to 1 GB, HugeGPT's throughput improvement relative to vanilla Linux/KVM degrades from 15% to 4%. This is because huge pages (e.g., 1 GB) can shorten page table walk. For instance, the page table for 1 GB huge pages does not need the last two levels that are present in page tables for 4 KB pages. As a result, HugeGPT cannot obtain more benefits when the page size becomes very large. On the other hand, 1 GB huge pages are not widely used as they incur large overheads such as memory fragmentation and CPU waste for defragmentation [7].

6.6 OVERHEAD

To evaluate HugeGPT's overhead, we test the performance of HugeGPT and vanilla Linux/KVM with two page walk non-intensive workloads, i.e., Swaptions and Raytrace. We show the perfor-

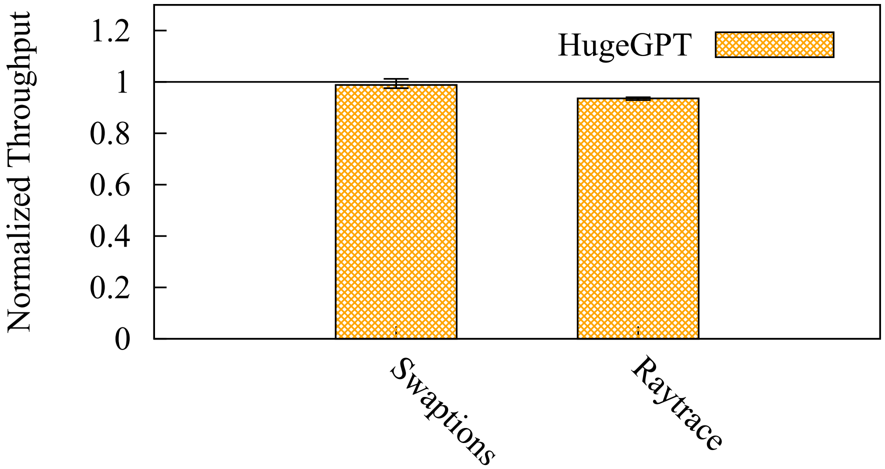


Figure 6.15: HugeGPT's overhead. Swaptions and Raytrace are page walk non-intensive workloads. Throughputs are normalized to vanilla Linux/KVM.

mance results in Figure 6.15. When the workload is page walk non-intensive, there is almost no space for HugeGPT to improve application performance compared to vanilla Linux/KVM, and the performance difference between HugeGPT and vanilla Linux/KVM shows HugeGPT's overhead. Figure 6.15 shows that HugeGPT does not introduce much performance overhead (3% on average). HugeGPT may introduce overhead as it needs to identify guest page table pages and allocate huge pages in the host operating system.

CHAPTER 7: DISCUSSION

Live Migration. HugeGPT can support live migration and restore from a snapshot. It needs the destination host operating system to conduct system initialization as described in Section 5.1. The destination host system shall also support the migration of transparent huge pages.

Memory Consumption. HugeGPT consumes negligible extra memory space to store page table data compared to vanilla Linux/KVM. In our evaluation, for 100 GB of application data, vanilla Linux/KVM needs around 217 MB of memory space to store page table data, and HugeGPT needs around 221 MB. In comparison to vanilla Linux/KVM, the extra memory consumption of HugeGPT is below 2%. Note that, despite HugeGPT reserves memory pools, HugeGPT will release unused pooled memory when the system memory pressure is high. Hence the above memory consumption only considers unreleasable memory usage.

Memory Fragmentation. HugeGPT relies on the vanilla Linux mechanisms for memory defragmentation. To defragment 200 MB of memory (100 2 MB pages) in a highly fragmented environment, it needs less than 200 ms. Memory can be defragmented when HugeGPT is in the system initialization phase or in an asynchronous manner. This can further minimize performance interference to application performance, when memory is heavily fragmented.

Difference from Existing Huge Page Provisioning Mechanisms. Some virtual machine (VM) hypervisors, such as QEMU, may provide huge pages for VMs by default [66]. The system administrators may also enable the transparent huge page system-wide [67]. These mechanisms overlap to some extent with the goal of HugeGPT, i.e. storing guest page table pages on host huge pages. However, note that such mechanisms cannot distinguish between user data pages and page table pages in the VMs. Therefore, their huge page provisioning is non-discriminatory. Yet, it is known that these approaches may lead to a negative performance impact [68]. Moreover, since host system memory may be fragmented, the availability of huge pages may be limited. Under a high-fragmentation scenario, the design of HugeGPT makes it prioritize storing the guest page table pages on host huge pages, thus potentially obtaining a higher performance benefit than the existing approaches, as confirmed in Section 6.4.

Generalizability of HugeGPT. The idea of HugeGPT can be applied to translation designs other than the x86 page table design. For example, for other tree-based page tables such as FPT [39, 49], HugeGPT may help reduce the tree depth and thus reduce the memory access count for each page table walk. On the other hand, for hashing-based and mapping-based designs, such as ECPT [20, 47] or DMT [40], HugeGPT may still improve the hit rate of their page walk caches, thus reducing the address translation overhead.

Software Compatibility of HugeGPT. HugeGPT is transparent to the user applications. This is because HugeGPT only changes the way the application page tables are stored, which is beyond the scope of observation of user applications. The HugeGPT system can run in environments that do not support this design. During system initialization, HugeGPT detects the availability of necessary hypercalls. If the current environment does not support HugeGPT, the HugeGPT system will automatically resort to the behavior of the vanilla system. To the unsupported counterparts, the behavior of the HugeGPT system is no different from the vanilla system.

Real-world Application. HugeGPT can be practically applied to real-world scenarios. We are currently working with industrial partners to deploy HugeGPT in production environments. We also plan to upstream the design into the mainline Linux kernel in the future.

CHAPTER 8: RELATED WORK

Hardware-Assisted Approaches. Prefetched address translation [2] prefetches page table entries by creating direct mappings from virtual addresses to corresponding entries. Flat nested page table [69] leverages the direct mapping idea for nested page walks. FPT [39, 49] flattens the page table through merging the adjacent layers of the page table. DMT [40] directly fetches the last-level translation entries by creating a direct mapping from each virtual page to its last-level PTE in memory. POM-TLB [25] uses part of the DRAM space as a very large level-3 translation lookaside buffer (TLB) to mitigate address translation overhead. Agile Paging [19] mitigates two-dimensional page walks overhead by leveraging the nested paging and the shadow paging at the same time. Gandhi et al. [70] apply direct segment [71] in virtualized systems, and it requires large contiguous physical memory space to hold the application's entire dataset. Elastic cuckoo hashing [20, 47] extends and implements cuckoo hashing [72] in virtualized environments. CA-paging [50] mitigates the address translation overhead with software and hardware co-design. Redundant memory mappings [26] enables ranges of an arbitrary number of virtually and physically contiguous pages to increase TLB reach and speed up address translation. Midgard [27] proposes a new virtual cache mechanism that maps virtual memory areas (VMA) to a single unified Midgard address space. Since each process usually has a few frequently used VMAs, Midgard's TLB coverage is larger than traditional TLB. TLB coalescing [28, 29, 30] increases TLB efficiency by exploiting the contiguity in virtual-to-physical mappings and merging their TLB entries into a single entry. Barr et al. [31] study different designs of memory management unit (MMU) caches and conclude that the most effective one is the translation cache (e.g., page walk caches). Hashed page tables [32, 73] challenge this conclusion and propose to use the hashing scheme to directly reduce the page walk latency.

Compared to the above approaches, HugeGPT is designed to reduce page walk cache misses for workloads with weak memory access locality. HugeGPT only needs to slightly change software and can be easily used in virtualized clouds.

Shadow Paging. Shadow paging [19, 74] is the software approach to facilitate memory virtualization. It emulates the guest page table to run the application and the host page table to run the virtual machine. The page walker walks the shadow page table (SPT) that merges the address mappings in the guest page table and the host page table. Any update in the guest page table (write protected) needs to trap to the host and update the SPT to keep consistency between the emulated page tables and the SPT. The overhead caused by the synchronization is large [75]. Hardware-assisted memory virtualization technology (nested paging) is proposed to resolve the overhead.

Huge Pages. Many research proposals [4, 7, 14, 16, 17, 18, 33] focus on optimizing huge page mechanisms to reduce address translation overhead. Ingens [33] identifies several issues in existing Linux huge page mechanisms and addresses them correspondingly. HawkEye [7] further optimizes Ingens. Illuminator [14] and Contiguitas [76] shows that unmovable pages (e.g., system kernel pages) can greatly increase memory fragmentation when huge pages are used. To address this issue, it proposes to manage movable, unmovable, and hybrid memory regions separately. Navarro et al. [4] propose reservation-based huge page management, huge pages with very large sizes, and a novel contiguity-aware page replacement algorithm to control memory fragmentation. Zhu et al. [15] comprehensively analyze huge page mechanisms and propose Quicksilver to optimize memory bloat and fragmentation problems. Temeraire [17] allocates huge pages with different sizes based on application memory requests to mitigate memory fragmentation. Gemini [51] forms well-aligned huge pages between guest and host operating systems to improve TLB efficiency in virtualized clouds.

Existing huge page mechanisms may cause memory fragmentation [14]. Since HugeGPT only stores guest page tables on host huge pages and the size of guest page tables is very small (200 MB for 100 GB application data), the adverse effect is negligible. Yet, HugeGPT can work with these approaches to achieve better performance.

CHAPTER 9: CONCLUSION AND FUTURE WORK

This thesis presents HugeGPT, an efficient system solution to reduce page walk cache misses and the steps to walk the nested page table in the two-dimensional address translation. HugeGPT's main idea is to store guest page table pages on host huge pages. To realize HugeGPT, it needs to overcome several technical challenges, such as filtering out memory allocations of guest page table pages and forming host huge pages based on huge page sized guest physical memory regions that store the guest page table data. The evaluation based on diverse real-world applications shows that HugeGPT can efficiently reduce address translation overhead and achieve better performance compared to vanilla Linux/KVM.

In addition to software-based solutions, another viable path is to drive an incremental evolution of the hardware. That is, to design hardware address translation schemes that support smooth transition and upward compatibility, thus allowing the existing systems and programs to operate with the opportunity to migrate to new designs with minimal modifications. As a future work, it is of equal practicality to explore such hardware designs. Also, the translation overhead problem is not limited to memory virtualization scenarios. Such a problem has variants in a variety of domains including storage and networking. Exploring similar solutions in other domains through horizontal migration is also one of the possible research directions.

REFERENCES

[1] W. Jia, J. Zhang, J. Shan, Y. Du, X. Ding, and T. Xu, “HugeGPT: Storing Guest Page Tables on Host Huge Pages to Accelerate Address Translation,” in *Proceedings of the 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT ’23)*, Oct. 2023.

[2] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, “Prefetched Address Translation,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, Oct. 2019.

[3] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware,” in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*, Mar. 2012.

[4] J. Navarro, S. Iyer, P. Druschel, and A. Cox, “Practical, Transparent Operating System Support for Superpages,” in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI ’02)*, Dec. 2002.

[5] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, “Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’20)*, Mar. 2020.

[6] A. Panwar, R. Achermann, A. Basu, A. Bhattacharjee, K. Gopinath, and J. Gandhi, “Fast Local Page-Tables for Virtualized NUMA Servers with vMitosis,” in *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’21)*, Apr. 2021.

[7] A. Panwar, S. Bansal, and K. Gopinath, “HawkEye: Efficient Fine-grained OS Support for Huge Pages,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’19)*, Apr. 2019.

[8] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee, “Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi,” in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE ’15)*, Mar. 2015.

[9] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. Lui, “SmartMD: A High Performance Deduplication Engine with Mixed Pages,” in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC ’17)*, July 2017.

[10] A. Margaritov, D. Ustiugov, A. Shahab, and B. Grot, “PTEMagnet: Fine-Grained Physical Memory Reservation for Faster Page Walks in Public Clouds,” in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’21)*, Apr. 2021.

[11] T. Merrifield and H. R. Taheri, “Performance Implications of Extended Page Tables on Virtualized x86 Processors,” in *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE ’16)*, Apr. 2016.

[12] B. Pham, J. Veselỳ, G. H. Loh, and A. Bhattacharjee, “Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have it Both Ways?” in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*, Dec. 2015.

[13] B. Pham, J. Vesely, G. H. Loh, and A. Bhattacharjee, “Using TLB Speculation to Overcome Page Splintering in Virtual Machines,” Rutgers University, Tech. Rep. DCS-TR-713, Mar. 2015.

[14] A. Panwar, A. Prasad, and K. Gopinath, “Making Huge Pages Actually Useful,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’18)*, Mar. 2018.

[15] W. Zhu, A. L. Cox, and S. Rixner, “A Comprehensive Analysis of Superpage Management Mechanisms and Policies,” in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC ’20)*, July 2020.

[16] R. Kadekodi, S. Kadekodi, S. Ponnapalli, H. Shirwadkar, G. R. Ganger, A. Kolli, and V. Chidambaram, “WineFS: a hugepage-aware file system for persistent memory that ages gracefully,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP ’21)*, Oct. 2021.

[17] A. Hunter, C. Kennelly, P. Turner, D. Gove, T. Moseley, and P. Ranganathan, “Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator,” in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’21)*, July 2021.

[18] M. Maas, C. Kennelly, K. Nguyen, D. Gove, K. S. McKinley, and P. Turner, “Adaptive Huge-Page Subrelease for Non-moving Memory Allocators in Warehouse-Scale Computers,” in *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM ’21)*, June 2021.

[19] J. Gandhi, M. D. Hill, and M. M. Swift, “Agile Paging: Exceeding the Best of Nested and Shadow Paging,” in *Proceedings of 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA ’16)*, June 2016.

[20] J. Stojkovic, D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, “Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’22)*, Feb. 2022.

[21] Intel, “Intel® 64 and IA-32 Architectures Developer’s Manual,” https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, Dec. 2023.

[22] Advanced Micro Devices, “AMD64 Architecture Programmer’s Manual,” https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf, June 2023.

[23] H. Ye, “Introduction to 5-Level Paging in 3rd Gen Intel Xeon Scalable Processors with Linux,” https://lenovopress.lenovo.com/lp1468.pdf, May 2021.

[24] Intel, “5-Level Paging and 5-Level EPT White Paper,” https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html, May 2017.

[25] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, “Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA ’17)*, June 2017.

[26] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, “Redundant Memory Mappings for Fast Access to Large Memories,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA ’15)*, June 2015.

[27] S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer, “Rebooting Virtual Memory with Midgard,” in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA ’21)*, June 2021.

[28] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, “CoLT: Coalesced Large-Reach TLBs,” in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*, Dec. 2012.

[29] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, “Increasing TLB Reach by Exploiting Clustering in Page Translations,” in *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA-20)*, Feb. 2014.

[30] C. H. Park, T. Heo, J. Jeong, and J. Huh, “Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA ’17)*, June 2017.

[31] T. W. Barr, A. L. Cox, and S. Rixner, “Translation Caching: Skip, Don’t Walk (the Page Table),” in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA ’10)*, June 2010.

[32] I. Yaniv and D. Tsafrir, “Hash, Don’t Cache (the Page Table),” in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’16)*, June 2016.

[33] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, “Coordinated and Efficient Huge Page Management with Ingens,” in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’16)*, Nov. 2016.

[34] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath, "Implementation of Multiple Pagesize Support in HP-UX," in *Proceedings of the 1998 USENIX Annual Technical Conference (USENIX ATC '98)*, June 1998.

[35] N. Ganapathy and C. Schimmel, "General Purpose Operating System Support for Multiple Page Sizes," in *Proceedings of the 1998 USENIX Annual Technical Conference (USENIX ATC '98)*, June 1998.

[36] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in Supporting Two Page Sizes," in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, June 1992.

[37] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, Mar. 2013.

[38] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quema, "Large Pages May Be Harmful on NUMA Systems," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, June 2014.

[39] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, "Every Walk's a Hit: Making Page Walks Single-Access Cache Hits," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, Feb. 2022.

[40] J. Zhang, W. Jia, S. Chai, P. Liu, J. Kim, and T. Xu, "Direct Memory Translation for Virtualized Clouds," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*, Apr. 2024.

[41] G. Choi, J. Son, J. Choi, S.-j. Cho, and Y. Won, "HPanal: A Framework for Analyzing Tradeoffs of Huge Pages," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*, Apr. 2019.

[42] Redis, "Diagnosing latency issues," https://redis.io/topics/latency, Apr. 2024.

[43] Couchbase, "Disabling Transparent Huge Pages (THP)," https://docs.couchbase.com/server/5.5/install/thp-disable.html, Apr. 2024.

[44] MongoDB, "Disable Transparent Huge Pages (THP)," https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/, Apr. 2024.

[45] Microsoft Azure, "Monitoring and troubleshooting from HANA side," https://docs.microsoft.com/en-us/azure/virtual-machines/workloads/sap/hana-monitor-troubleshoot, Apr. 2024.

[46] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "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)*, Mar. 2008.

[47] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, “Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’20)*, Mar. 2020.

[48] K. S. McKinley, “Next Generation Virtual Memory Management,” in *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE ’16)*, Mar. 2016.

[49] S. Ainsworth and T. M. Jones, “Compendia: Reducing Virtual-Memory Costs via Selective Densification,” in *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM ’21)*, June 2021.

[50] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, “Enhancing and Exploiting Contiguity for Fast Memory Virtualization,” in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA ’20)*, June 2020.

[51] W. Jia, J. Zhang, J. Shan, and X. Ding, “Making Dynamic Page Coalescing Effective on Virtualized Clouds,” in *Proceedings of the 18th European Conference on Computer Systems (EuroSys ’23)*, May 2023.

[52] L. McVoy and C. Staelin, “lmbench: Portable Tools for Performance Analysis,” in *Proceedings of the USENIX 1996 Annual Technical Conference (USENIX ATC ’96)*, June 1996.

[53] Cloudera, “Disabling Transparent Hugepages (THP),” https://docs.cloudera.com/cdp-private-cloud-base/7.1.9/managing-clusters/topics/cm-disabling-transparent-hugepages.html, Apr. 2024.

[54] IBM, “Transparent Huge Pages (THP) may cause high CPU utilization and performance issues on IBM Smart Analytics System 5600 environments using SUSE Linux Enterprise Server 11 SP2,” http://www-01.ibm.com/support/docview.wss?uid=swg21677458, Apr. 2024.

[55] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel, “Sphinx-4: A Flexible Open Source Framework for Speech Recognition,” Sun Microsystems, Tech. Rep. SMLI TR2004-0811, Dec. 2004.

[56] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens et al., “Moses: Open Source Toolkit for Statistical Machine Translation,” in *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions (ACL 2007)*, June 2007.

[57] Y. Mao, E. Kohler, and R. T. Morris, “Cache Craftiness for Fast Multicore Key-Value Storage,” in *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys ’12)*, Apr. 2012.

[58] Standard Performance Evaluation Corporation, "SPECjbb® 2015," https://www.spec.org/jbb2015/, Apr. 2024.

[59] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: A Scalable Storage Manager for the Multicore Era," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*, Mar. 2009.

[60] Redis, "Redis," http://redis.io/, Apr. 2023.

[61] Memcached, "memcached - a distributed memory object caching system," https://memcached.org, Apr. 2023.

[62] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," Princeton University, Tech. Rep. TR-811-08, Jan. 2008.

[63] HPC Challenge Benchmark, "RandomAccess: GUPS (Giga Updates Per Second)," https://hpcchallenge.org/projectsfiles/hpcc/RandomAccess.html, Aug. 2022.

[64] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," in *Proceedings of the International Conference on Physics of Reactors (PHYSOR 2014)*, Sep. 2014.

[65] Yahoo!, "Yahoo! Cloud Serving Benchmark," https://github.com/brianfrankcooper/YCSB, Oct. 2019.

[66] K. Buettner, "Benchmarking transparent versus 1GiB static huge page performance in Linux virtual machines," https://developers.redhat.com/blog/2021/04/27/benchmarking-transparent-versus-1gib-static-huge-page-performance-in-linux-virtual-machines, Apr. 2021.

[67] The kernel development community, "Transparent Hugepage Support," https://docs.kernel.org/admin-guide/mm/transhuge.html, Dec. 2023.

[68] IBM, "General system settings," https://www.ibm.com/docs/en/linux-on-systems?topic=tuning-general-system-settings, Nov. 2023.

[69] J. Ahn, S. Jin, and J. Huh, "Revisiting Hardware-Assisted Page Walks for Virtualized Systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*, June 2012.

[70] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*, Dec. 2014.

[71] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, June 2013.

[72] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[73] J. Stojkovic, N. Mantri, D. Skarlatos, T. Xu, and J. Torrellas, "Memory-Efficient Hashed Page Tables," in *Proceedings of the 29th IEEE International Symposium on High-Performance Computer Architecture (HPCA-29)*, Feb. 2023.

[74] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, Oct. 2006.

[75] VMware, "Performance Evaluation of Intel EPT Hardware Assist," https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf, Mar. 2009.

[76] K. Zhao, K. Xue, Z. Wang, D. Schatzberg, L. Yang, A. Manousis, J. Weiner, R. Van Riel, B. Sharma, C. Tang, and D. Skarlatos, "Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, June 2023.