HugeGPT: Storing Guest Page Tables on Host Huge Pages to Accelerate Address Translation

Weiwei Jia[1*], Jiyuan Zhang[2*], Jianchen Shan[3], Yiming Du[1], Xiaoning Ding[4], Tianyin Xu[2]
[1]The University of Rhode Island [2]University of Illinois at Urbana-Champaign
[3]Hofstra University [4]New Jersey Institute of Technology

***Abstract*—Expensive page table walks triggered by frequent 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 is more compelling for workloads running in virtual machines (VMs). In virtualized environments, each TLB miss triggers a two-dimensional page table walk, which has a significantly higher overhead than that on native systems.**

This paper 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. 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.

***Index Terms*—Virtualization, Memory Management, Page Tables, Operating Systems, TLB**

I. Introduction

TLB misses have become the major performance bottleneck for the workloads with big memory footprints [1]–[17]. Previous works [1], [18], [19] show that performance of big memory workloads can be degraded by as much as 50% due to the high overhead incurred by 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 [20] and AMD nested page tables [21]) enables two-dimensional page walk to resolve TLB misses. This increases the TLB miss overhead by up to 6x [10], [19], [20]. With the upcoming 5-level page tables [22], [23], this increase is more than 8x.

Reducing the overhead incurred by TLB misses heavily relies on the hardware designs in memory management units (MMUs). Thus, existing research mostly concentrates on new hardware designs, which reduce either the number of TLB misses [1], [3], [6]–[8], [10], [11], [13], [18], [24]–[37] or the overhead of each TLB miss [19], [38].

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 use huge pages (e.g., 2MB page) [3], [6], [13], [15]–[17], [32]. A TLB entry buffering the address mapping for a huge page has a much larger coverage than that for a base page (4KB 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 the data accesses with 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 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 [39], [40].

In this paper, 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 *meta data* — the page tables used in the guest OS to manage the memory of a VM. Thus, HUGEGPT does not incur the adverse effects that are caused by conventional approaches through 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 (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 walk, the most effective method is to reduce

*equal contribution

the overhead incurred by resolving the entry addresses of the guest page table. 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 (b). First, it eliminates most 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 (a) 3 steps (i.e., Step 12, Step 13, and Step 14) are required to get the address of $gL2$; in Figure 1 (b), only 2 steps (i.e., Step 12 and Step 13) are required.

To realize HUGEGPT, our basic idea is to let guest OS notify 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.

To store guest page tables on host huge pages, the host needs to allocate huge pages to back guest physical memory regions that 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. 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.

The paper 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 §VI-D).

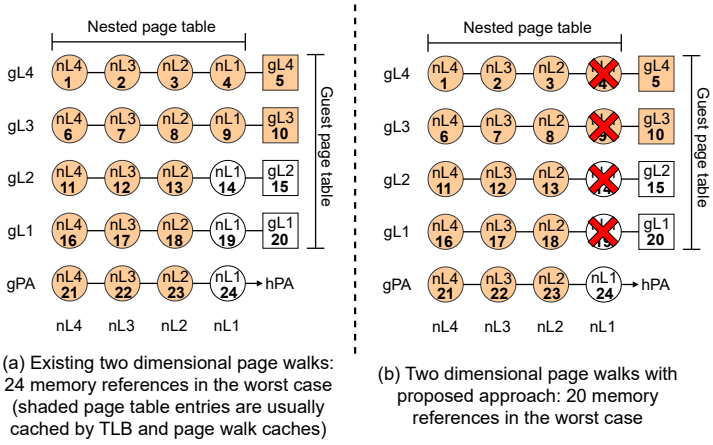

Fig. 1: The proposed approach can substantially reduce page walk latency of the two-dimensional page walks because the lower level page table entries shaded in the figure are usually cached by TLB and page walk caches. The proposed approach slightly changes the software, i.e., only storing guest page table data on host huge pages.

II. Background and Motivation

This section first introduces how the two-dimensional page walk works (§II-A). 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 (§II-B).

A. 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 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 [20]) and AMD nested page table (AMD NPT [21]), enables the two-dimensional page translation.

Figure 1 (a) shows how the two-dimensional page translation works. The two-dimensional page translation needs to walk two page tables (the guest/host page table maintained by the guest/host OS) 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 (a)) host physical addresses (Step 1-4, 6-9, 11-14, and 16-19 in Figure 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 (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 [19], [41], [42].

As today's data-intensive applications are pervasive and usually need large memory space to hold their working set, Intel releases the design of 5-level page table [23], which significantly increases the addressable memory [22]. With such 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.

B. Inefficient Two Dimensional Page Walk

In modern systems architecture, 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 [1], [4], [5], [43]. 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 [10], [20], as introduced in §II-A.

Existing research proposals on reducing address translation overhead mainly fall into two categories: reducing TLB misses and their overhead for applications with strong locality [1], [24], [25], and reducing page walk cache misses and their overhead for applications with weak locality [38], [44], as summarized in Table I. HUGEGPT falls into the second category. In this category, existing works need to modify hardware [19], [38], [42], [44]. For instance, FPT [38] flattens the page table through merging adjacent layers of the page table. For 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 50GB, 100GB, and 200GB, respectively. The second micro-benchmark shows weak spatial locality, a better locality than the first micro-benchmark. It accesses each 4KB memory page once with the same different working set sizes as those in the first micro-benchmark. We follow the same approach in the previous work [47] to generate workloads with weak memory access locality. To measure micro-benchmarks' throughputs, we measure the memory accesses performed per second.

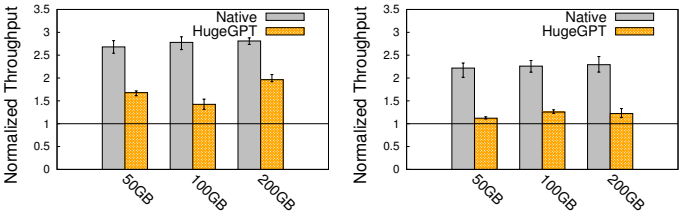

Fig. 2: Throughputs of native system, and HUGEGPT. Throughputs are normalized to vanilla Linux/KVM.

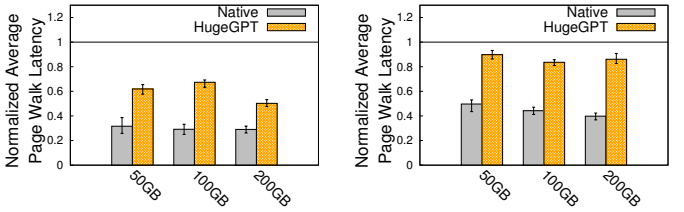

Fig. 3: Average page walk latency of native system and HUGEGPT. Average page walk latencies are normalized to vanilla Linux/KVM.

Workload locality	High level idea	Previous works	
		Hardware approaches	Software approaches
Strong locality	Reducing TLB misses	ASAP [1], POM-TLB [24], `CA-paging` [45], RMM [25]	Gemini [46], Transparent Huge Page [3], [32]
Weak locality	Reducing page walk cache (PWC) misses	FPT [38], Compendia [44]	**Our proposed approach (HUGEGPT)**

TABLE I: 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.

Figure 2 shows the throughputs of the two micro-benchmarks when they are tested with 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-level page walk used in the native environments, the inefficiency of 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 3. Compared to vanilla Linux/KVM, HUGEGPT reduces the average page walk latency by 41% for workload with weak memory access locality and 14% for workload with almost no memory access locality on average.

III. Main Idea and Technical Challenges

As explained and confirmed in §II, two-dimensional page walk used by vanilla Linux/KVM incurs much longer average page walk latency compared to the one-level page walk used

in 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 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 GPA 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 (b).

Intuitively, guest OS accesses application's whole working set and has worse locality compared to 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; and columns nL4, nL3, and nL2 of the host page table may be cached. This is also corroborated by the previous work [38]. We shaded the cached page table entries in Figure 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 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.

With HUGEGPT, translating the GPA of the GVA to the final HPA (Step 21-24 in Figure 1) still needs to walk the 4-level host page table as shown in the last row of Figure 1. This is because user application data is still stored on base pages (4KB pages) to avoid the adverse effects caused by transparent huge pages [48]–[51]. Since the size of the guest page table data is much smaller than user application data size (around 200MB page table data for 100GB user application data), the adverse effects of using huge pages are negligible.

Figures 2 and 3 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 workload with weak memory access locality than it for the workload 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 (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.

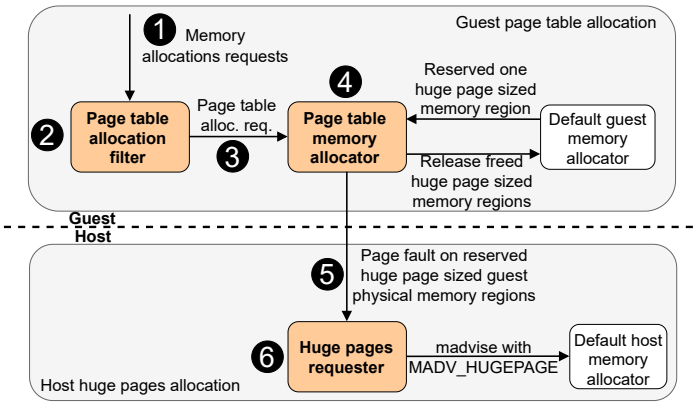

Fig. 4: HUGEGPT system overview. Key components are shaded in orange.

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. The first technical challenge is how to filter out guest page table data and store it on specific guest physical memory regions. Memory allocations of page table pages are mixed with other memory allocation requests. 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. The second technical challenge is how to identify the guest physical memory regions that store guest page table data at the host level. Due to the semantic gap between the guest and the host, it is challenging to obtain the guest information in the host.

IV. System Overview

This section gives a system overview of HUGEGPT. and explain how HUGEGPT works in two phases.

Figure 4 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 table 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, 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 *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 *madvise* is called with *MADV_HUGEPAGE* command, the system will 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, page table memory allocator will not issue page fault request to the host OS 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.

V. Design Details

This section first introduces the initialization of HUGEGPT upon the system starts. Then, it explains how guest page table memory allocator and guest page table allocation filter work. At last, it presents how host huge pages are created based on the huge page sized guest physical memory regions.

A. System Initialization

The goal of the initialization is to setup HUGEGPT before it is used. The initialization is conducted immediately after the system starts. In the initialization, 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 100GB application data only needs around 200MB page table data. Then, HUGEGPT guest page table memory allocator notifies the guest physical addresses of these pre-allocated guest physical memory regions to 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 §VI-E). This makes 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.

B. 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 OS (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, 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.

C. 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, 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, 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 OS.

HUGEGPT re-executes the initialization process once the memory pool of pre-allocated huge page sized guest physical memory regions are 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 4KB base pages, in order to make systems run correctly.

VI. Evaluation

We have implemented HUGEGPT prototype based on Linux/KVM 5.15. We added and changed around 640 lines of source code mainly in the page table allocation of the kernel memory management subsystem. We added a new kernel file (`arch/x86/mm/hugegpt-guest.c`) to implement the guest memory allocator (around 560 lines of source code). For the host huge pages requester, we added around

Workload Name	Workload Description	Working Set Size
Sphinx	Speech recognition like Apple Siri [52].	30GB
Moses	Real time translation like Google translate [53].	25GB
Masstree	In memory K/V store (50% GET, 50% SET) [54].	25GB
Specjbb	Industry-standard JAVA middleware benchmark [55].	60GB
Shore	Transactional database with TPCC [56].	30GB
Redis	Serve requests (random keys,50% SET,50% GET) [57].	155GB
Memcached	Serve requests (random keys,50% SET,50% GET) [58].	95GB
Canneal	Chip design optimizer [59].	62GB
Graph500	Graph analysis.	123GB
GUPS	Giga Updates Per Second benchmark [60].	128GB
XSBench	Monte Carlo neutron transport compute kernel [61].	84GB
BTree	Index lookup benchmark [4].	125GB

TABLE II: Programs and workloads used to test HUGEGPT.

80 lines of code in `arch/x86/kvm/x86.c` to realize it. HUGEGPT works at the VM/process granularity. The Page Table Allocation Filter of HUGEGPT is used to identify different VMs/processes and decides whether HUGEGPT should be enabled on each of them.

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 (Linux THP [32]) and Gemini [46]. 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 (§VI-B), 2) to show that HUGEGPT can reduce mean and tail latency of latency-sensitive workloads compared to vanilla Linux/KVM (§VI-C), 3) to compare HUGEGPT with related systems (§VI-D), and 4) to evaluate applicability and overhead of HUGEGPT (§VI-E).

A. Experiment Settings

Our evaluation was conducted on a Hewlett Packard Enterprise (HPE) ProLiant DL580 Gen10 server with four Intel Xeon Gold 6138 processors, 256GB memory, and two 2TB SSDs. Each processor has 20 cores. With Linux QEMU/KVM, we built virtual machines (VMs), each VM with 40 virtual CPUs (vCPUs) and 240GB memory. We set the number of application threads equal to the number of vCPUs. Both host OS and guest OS 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 II. 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 are with weak memory access locality. Two workloads (i.e., `Swaptions` and `Raytrace`) are not 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 critical 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 [62]) 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.

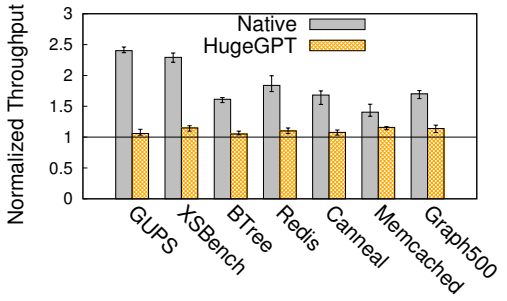

Fig. 5: Throughputs of throughput-oriented workloads. Throughputs are normalized to vanilla Linux/KVM.

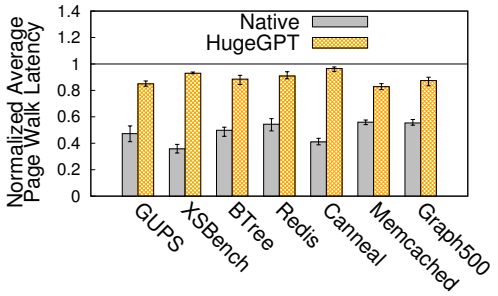

Fig. 6: Average page walk latencies of throughput-oriented workloads. Average page walk latencies are normalized to vanilla Linux/KVM.

B. Experiments with Throughput Oriented Workloads

Figure 5 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, 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, native system outperforms HUGEGPT by 68%. This is because TLB 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 native system, we profile the average page walk latency when the workload is tested in different systems. We show the results in Figure 6. 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 native

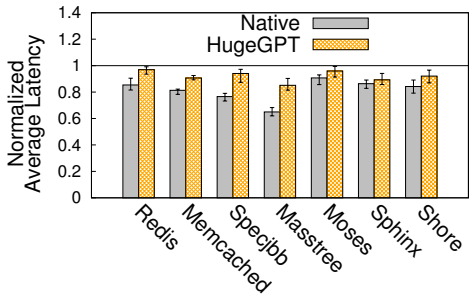

Fig. 7: Average latencies of latency sensitive workloads. Average latencies are normalized to vanilla Linux/KVM.

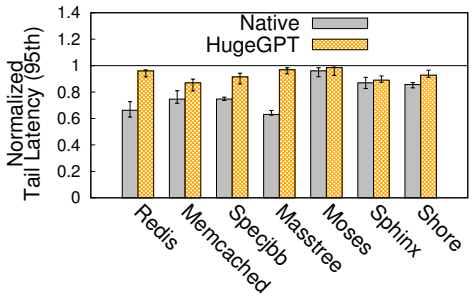

Fig. 8: 95th percentile tail latencies of latency sensitive workloads. Tail latencies are normalized to vanilla Linux/KVM.

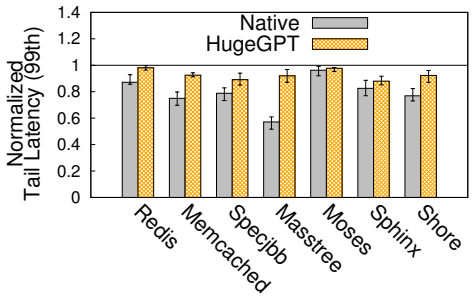

Fig. 9: 99th percentile tail latencies of latency sensitive workloads. Tail latencies are normalized to vanilla Linux/KVM.

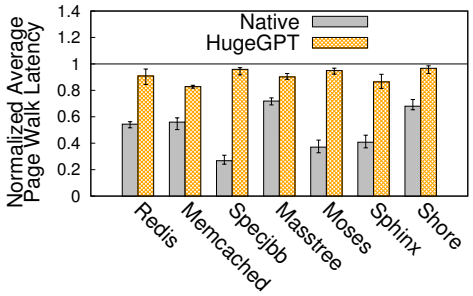

Fig. 10: Average page walk latencies of latency sensitive workloads. Average page walk latencies are normalized to vanilla Linux/KVM.

system on average. This confirms HUGEGPT's effectiveness on improving application throughput by reducing the overhead of two dimensional page walks in vanilla Linux/KVM.

Figure 5 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 TLB and page walk caches compared to random memory accesses. Therefore, reducing the leaf page table entries of the nested page table in HUGEGPT shows more performance improvement. This is consistent with the performance observation in §II-B. Since `GUPS` and `BTree` workloads conduct randomly memory accesses, HUGEGPT's performance improvement on these workloads is less. For instance, `GUPS` is calculated by identifying 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.

C. Experiments with Latency Sensitive Workloads

Figure 7 shows the average latencies of different systems when they are tested with latency sensitive workloads. On average, native system shows the lowest average latency as most page table entries can be cached while walking the one level page table. In the worst case, native system only incurs four memory references. Relative to 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 §II-B. HUGEGPT reduces page walk cache misses and the number of memory references in two dimensional page walks from 24 to 20 in the worst case.

To further pinpoint why HUGEGPT increases the average latency compared to 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. We show the profiling results in Figure 10. On average, HUGEGPT increases the average page walk latency by 62% compared to native system and decreases the average page walk latency by 8% compared to vanilla Linux/KVM. This is consistent with the average latency results and also shows HUGEGPT's effectiveness on reducing the overhead of two dimensional page walks for latency sensitive workloads in comparison to vanilla Linux/KVM.

Figure 8 and Figure 9 show the 95th percentile tail latencies and the 99th percentile tail latencies, respectively, when the latency sensitive workloads are tested with the three systems. On average, HUGEGPT provides 8% lower 95th percentile tail latency 8% lower 99th percentile tail latency compared to vanilla Linux/KVM, and 32% higher 95th percentile tail latency and 30% higher 99th percentile tail latency relative to native system. The tail latency test results are consistent with the average page walk latency of the three systems as shown in Figure 10.

Figure 7, Figure 8, and Figure 9 also show that HUGEGPT shows small performance advantage for some workloads (e.g., `Moses` and `Masstree`) and large performance advantage 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 §II-B. Memory access patterns in `Moses` and `Masstree` workloads are more random than `Specjbb` and `Sphinx`. HUGEGPT does not show good performance with

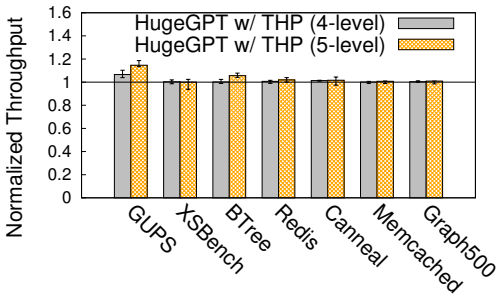

Fig. 11: HUGEGPT's throughput improvement compared to Linux transparent huge page (THP) [32] when 4-level and 5-level page table are used respectively. Throughputs are normalized to Linux THP.

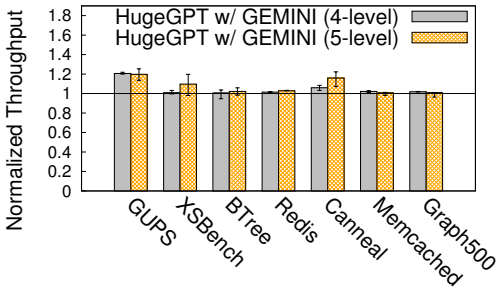

Fig. 12: HUGEGPT's throughput improvement compared to Gemini [46] when 4-level and 5-level page table are used respectively. Throughputs are normalized to Gemini.

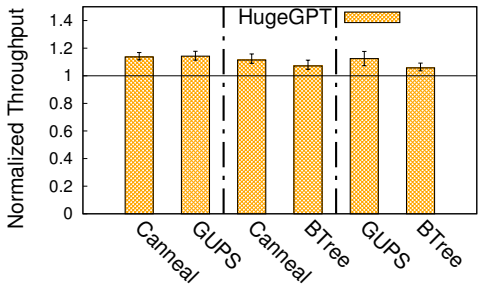

Fig. 13: Throughputs of throughput oriented workloads when they are colocated on the same server. Throughputs are normalized to vanilla Linux/KVM.

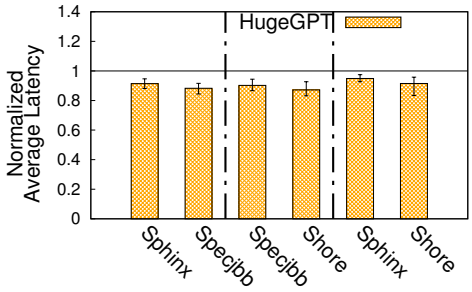

Fig. 14: Average latencies of latency sensitive workloads when they are colocated on the same server. Average latencies are normalized to vanilla Linux/KVM.

workloads with random memory access patterns as lower page table entries may not be cached.

D. Comparisons with Related Systems

We compare HUGEGPT with Linux transparent huge page (THP) and Gemini [46] on x86 4-level page table and 5-level page table, respectively. To support 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 TLB entries, and 36MiB last level CPU cache), 256GB of DRAM, and 2TB SSD. With Linux QEMU/KVM, we built the virtual machine with 32 vCPUs, and 240GB memory. Both host OS and guest OS are Ubuntu Linux 20.04 with the same Linux 5.10 kernel and software configuration, unless otherwise indicated.

Figure 11 and Figure 12 compare HUGEGPT's throughput with that for Linux transparent huge page (THP) and Gemini [46], respectively, when 4-level page table and 5-level page table are used. When the 4-level page table is used, HUGEGPT outperforms THP by up to 6% and Gemini 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 §II-B, 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.

E. Applicability and Overhead

To evaluate HUGEGPT's applicability, we colocate two virtual machines (VMs) on the server and test HUGEGPT's performance when multiple VMs are colocated on the same server. We choose this test scenario as VMs colocation on the same server is pervasive in clouds. We mainly test three settings. In the first setting, two throughput oriented applications running in VMs are colocated on the same server. In the second setting, two latency sensitive applications running in VMs are colocated on the same server. In the last setting, six throughput oriented applications running in VMs are colocated on the same server.

Figure 13 shows throughputs of throughput-oriented workloads when HUGEGPT and vanilla Linux/KVM are tested under the aforementioned first setting. Under this setting, HUGEGPT outperforms vanilla Linux/KVM by 12% on average. This shows HUGEGPT can improve application performance by reducing two dimensional page walk overhead when multiple page walk intensive throughput-oriented applications are collocated on the same server. These experiments also show HUGEGPT's effectiveness on multi-threaded applications, multiple processors, and multiple VMs consolidated on the same server.

Figure 14, Figure 15, and Figure 16 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 tail latency by 12%, and the 99th percentile tail latency by 9% on average,

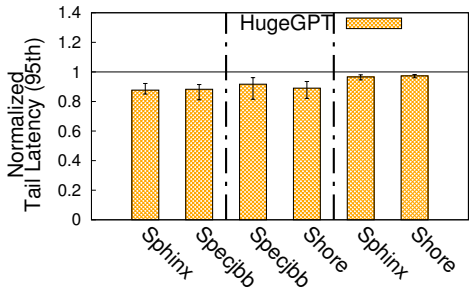

Fig. 15: 95th percentile tail latencies of latency sensitive workloads when they are colocated on the same server. Tail latencies are normalized to vanilla Linux/KVM.

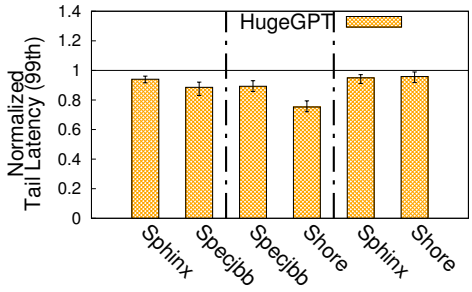

Fig. 16: 99th percentile tail latencies of latency sensitive workloads when they are colocated on the same server. Tail latencies are normalized to vanilla Linux/KVM.

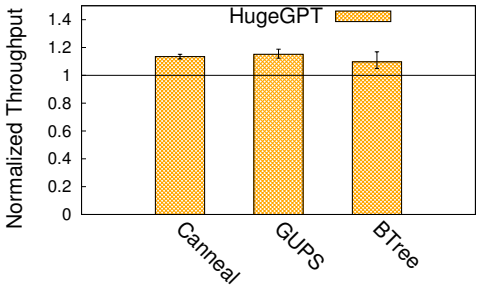

Fig. 17: Throughputs of six throughput oriented workloads colocated on the same server. We run two copies of each workload. Throughputs are normalized to vanilla Linux/KVM.

in comparison to vanilla Linux/KVM. This shows HUGEGPT can reduce average and tail latencies when latency sensitive workloads are colocated on the same server.

Figure 17 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. The VM running each workload has 12 vCPUs and 40GB memory. The working set size of each workload is kept around 35GB. 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 13.

Figure 18 shows HUGEGPT's throughput for different page sizes. We run `XSBench` to test HUGEGPT's throughput. We choose 4KB, 2MB, and 1GB memory page sizes because current x86 CPU only supports those page sizes. As the page

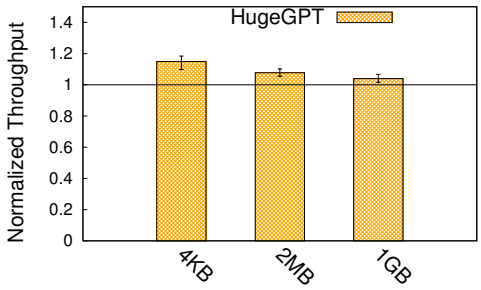

Fig. 18: HUGEGPT's throughputs with different memory page sizes. Throughputs are normalized to vanilla Linux/KVM.

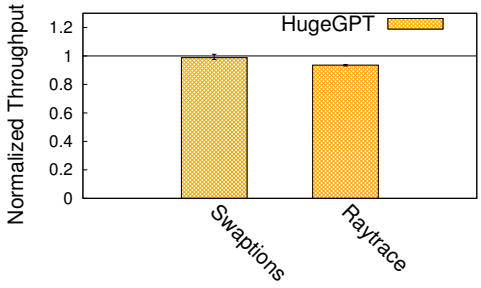

Fig. 19: HUGEGPT's overhead. `Swaptions` and `Raytrace` are page walk non-intensive workloads. Throughputs are normalized to vanilla Linux/KVM.

size increases from 4KB to 1GB, HUGEGPT's throughput improvement relative to vanilla Linux/KVM degrades from 15% to 4%. This is because huge pages (e.g., 1GB) can shorten page table walk. For instance, the page table for 1GB huge pages does not need the last two levels that are present in page tables for 4KB pages. As a result, HUGEGPT cannot obtain more benefits when the page size becomes very large. On the other hand, 1GB huge pages are not widely used as they incur large overhead such as memory fragmentation and CPU waste for defragmentation [6].

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 performance results in Figure 19. 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 19 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 OS.

VII. Discussion

Live Migration. HUGEGPT can support live migration and restore from a snapshot. It needs the destination host OS to conduct system initialization as described in §V-A.

Memory Consumption. HUGEGPT consumes negligible extra memory space to store page table data compared to vanilla Linux/KVM. In our evaluation, for 100GB of application data, vanilla Linux/KVM needs around 217MB of memory space to store page table data, and HUGEGPT needs around 221MB.

In comparison to vanilla Linux/KVM, the extra memory consumption of HUGEGPT is below 2%.

Memory Fragmentation. HUGEGPT relies on the vanilla Linux mechanisms for memory defragmentation. To defragment 200MB of memory (100 2MB pages) in a highly fragmented environment, it needs less than 200ms. 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.

VIII. Related Works

Hardware-Assisted Approaches. Prefetched address translation [1] prefetches page table entries by creating direct mappings from virtual addresses to corresponding entries. Flat nested page table [63] leverages the direct mapping idea for nested page walks. FPT [38], [44] flattens the page table through merging the adjacent layers of the page table. POM-TLB [24] uses part of the DRAM space as a very large level-3 TLB to mitigate address translation overhead. Agile Paging [18] mitigates two dimensional page walks overhead by leveraging the nested paging and the shadow paging at the same time. Gandhi et al. [64] apply direct segment [65] in virtualized systems; and it requires large contiguous physical memory space to hold application's entire dataset. Elastic cuckoo hashing [19], [42] extends and implements cuckoo hashing [66] in virtualized environments. `CA-paging` [45] mitigates the address translation overhead with software and hardware codesign. Redundant memory mappings [25] enables ranges of an arbitrary number of virtually and physically contiguous pages to increase TLB reach and speedup address translation. Midgard [26] proposes a new virtual cache mechanism that maps virtual memory areas (VMAs) 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 [27]–[29] increases TLB efficiency by exploiting the contiguity in virtual-to-physical mappings and merging their TLB entries into a single entry. Barr et al. [30] study different designs of MMU caches and conclude that the most effective one is translation cache (e.g., page walk caches). Hashed page tables [31], [67] challenge this conclusion and propose to use the hashing scheme to shorten the page walk latency.

Compared to 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 [18], [68] 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 (VM exits) to the host and update the SPT, in order to keep consistency between the emulated page tables and the SPT. The overhead caused by the synchronization is large [69]. Hardware assisted memory virtualization technology (nested paging) is proposed to resolve the overhead.

Huge Pages. Many research proposals [3], [6], [13], [15]–[17], [32] focus on optimizing huge page mechanisms to reduce address translation overhead. `Ingens` [32] identifies several issues in existing Linux huge page mechanisms and addresses them correspondingly. `HawkEye` [6] further optimizes `Ingens`. Illuminator [13] shows that unmovable pages (e.g., OS 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. [3] 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. [14] comprehensively analyze huge page mechanisms and propose Quicksilver to optimize memory bloat and fragmentation problems. Temeraire [16] allocates huge pages with different sizes based on application memory requests to mitigate memory fragmentation. `Gemini` [46] forms well aligned huge pages between guest OS and host OS to improve TLB efficiency in virtualized clouds.

Existing huge page mechanisms may cause memory fragmentation [13]. Since HUGEGPT only stores guest page tables on host huge pages and the size of guest page tables is very small (200MB for 100GB application data), the adverse effect is negligible. Yet, HUGEGPT can work with these approaches to achieve better performance.

IX. Conclusion and Future Work

This paper 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.

Nested virtualization has been widely used to achieve a variety of purposes, such as migrating multiple VMs together [70], supporting legacy applications [71], as well as securing systems and applications [72], [73]. As future work, we plan to study how to apply HUGEGPT in nested virtualization and seek the adoption of HUGEGPT in systems and architectures utilized in industry.

Acknowledgments

We thank the anonymous reviewers for their constructive comments, and Dr. Nilay Vaish for his helpful suggestions as the shepherd for this paper. Jiyuan Zhang and Tianyin Xu are supported in part by NSF CNS-1956007. Jianchen Shan is supported in part by NSF CNS-2324923.

References

[1] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1023–1036.

[2] 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," *Acm sigplan notices*, vol. 47, no. 4, pp. 37–48, 2012.

[3] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 89–104, 2002.

[4] 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*, 2020, pp. 283–300.

[5] 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*, 2021.

[6] 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*, 2019, pp. 347–360.

[7] 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*, 2015, pp. 39–51.

[8] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. Lui, "Smartmd: A high performance deduplication engine with mixed pages," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 733–744.

[9] A. Margaritov, D. Ustiugov, A. Shahab, and B. Grot, "Ptemagnet: Fine-grained physical memory reservation for faster page walks in public clouds," in *The 26th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021*, 2021.

[10] T. Merrifield and H. R. Taheri, "Performance implications of extended page tables on virtualized x86 processors," in *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2016, pp. 25–35.

[11] 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*, 2015, pp. 1–12.

[12] B. Pham, J. Vesely, G. H. Loh, and A. Bhattacharjee, "Using tlb speculation to overcome page splintering in virtual machines," 2015.

[13] 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*, 2018, pp. 679–692.

[14] W. Zhu, A. L. Cox, and S. Rixner, "A comprehensive analysis of superpage management mechanisms and policies," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 829–842.

[15] 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 CD-ROM*, 2021, pp. 804–818.

[16] 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 *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 257–273.

[17] 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*, 2021, pp. 28–38.

[18] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile paging: Exceeding the best of nested and shadow paging," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 707–718.

[19] 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*, 2022, pp. 84–97.

[20] "Intel 64 and ia-32 architectures developer's manual," https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[21] "Amd64 architecture programmer's manual," https://developer.amd.com/resources/developer-guides-manuals/.

[22] "Intel five level paging," https://en.wikipedia.org/wiki/Intel_5-level_paging.

[23] "Five level paging and five level EPT white paper," https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html.

[24] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 469–480, 2017.

[25] 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," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 66–78, 2015.

[26] S. Gupta, A. Bhattacharyya, Y. Oh, A. Bhattacharjee, B. Falsafi, and M. Payer, "Rebooting virtual memory with midgard," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 512–525.

[27] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 258–269.

[28] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 558–567.

[29] 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*, 2017, pp. 444–456.

[30] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 48–59, 2010.

[31] I. Yaniv and D. Tsafrir, "Hash, don't cache (the page table)," *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1, pp. 337–350, 2016.

[32] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 705–721.

[33] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath, "Implementation of multiple pagesize support in hp-ux." in *USENIX Annual Technical Conference*, 1998, pp. 105–119.

[34] N. Ganapathy and C. Schimmel, "General purpose operating system support for multiple page sizes." in *USENIX Annual Technical Conference*, no. 98, 1998, pp. 91–104.

[35] 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*, 1992, pp. 415–424.

[36] 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," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 381–394, 2013.

[37] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on {NUMA} systems," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 231–242.

[38] 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*, 2022, pp. 128–141.

[39] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quema, "Large pages may be harmful on numa systems," in *2014 USENIX Annual Technical Conference (USENIX ATC*

14). Philadelphia, PA: USENIX Association, Jun. 2014, pp. 231–242. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/gaud

[40] 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*, 2019, pp. 1438–1443.

[41] 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*, 2008, pp. 26–35.

[42] 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*, 2020, pp. 1093–1108.

[43] K. S. McKinley, "Next generation virtual memory management," ser. VEE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 107. [Online]. Available: https://doi.org/10.1145/2892242.2892244

[44] 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*, 2021, pp. 52–65.

[45] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, "Enhancing and exploiting contiguity for fast memory virtualization," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 515–528.

[46] 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 2023)*, 2023.

[47] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[48] "Cloudera reports huge pages seriously degrade performance of Hadoop workloads," https://docs.cloudera.com/cdp-private-cloud-base/7.1.3/managing-clusters/topics/cm-disabling-transparent-hugepages.html.

[49] "Recommendation for disabling huge pages for Redis," https://redis.io/topics/latency.

[50] "MongoDB does not recommend enabling transparent huge pages," https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/.

[51] "IBM recommends turning off huge pages due to high CPU utilization," http://www-01.ibm.com/support/docview.wss?uid=swg21677458.

[52] 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," 2004.

[53] 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 ACL on interactive poster and demonstration sessions*. Association for Computational Linguistics, 2007, pp. 177–180.

[54] 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*, 2012, pp. 183–196.

[55] "The SPECjbb benchmark," https://www.spec.org/jbb2015/.

[56] 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*, 2009, pp. 24–35.

[57] "Redis In-memory Key-Value Database," http://redis.io/.

[58] "Memcached Key-Value Store," https://memcached.org.

[59] "The Princeton application repository for shared-memory computers (PARSEC)," http://parsec.cs.princeton.edu/, 2010.

[60] "HPC Challege: GUPS(Giga Updates Per Second) Random Access Workload," https://icl.utk.edu/projectsfiles/hpcc/RandomAccess/.

[61] 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," *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[62] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[63] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 476–487.

[64] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 178–189.

[65] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 237–248, 2013.

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

[67] 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.

[68] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *ACM ASPLOS 2006*, pp. 2–13. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168860

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

[70] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization." in *OSDI*, vol. 10, 2010, pp. 423–436.

[71] J. T. Lim and J. Nieh, "Optimizing nested virtualization performance using direct virtual hardware," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 557–574.

[72] Z. Mi, D. Li, H. Chen, B. Zang, and H. Guan, "(mostly) exitless {VM} protection from untrusted hypervisor through disaggregated nested virtualization," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1695–1712.

[73] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 203–216.