# Effective Huge Page Strategies for TLB Miss Reduction in Nested Virtualization

Weiwei Jia*, Jiyuan Zhang*, Jianchen Shan, and Xiaoning Ding

**Abstract**—Huge page strategies, such as Linux Transparent Huge Page (THP), have become a prevalent solution to mitigate the performance bottleneck caused by increasingly high memory address translation overhead. However, in cloud environments, virtualization presents a two-fold challenge, exacerbating address translation overhead and undermining the effectiveness of huge page strategies. To effectively reduce address translation overhead, huge page strategies in the host and guest virtual machines (VMs) must work in concert for "proper huge page alignment", i.e., huge pages in guest VMs being backed by host huge pages. This requires a cross-layer coordinating mechanism, which has been designed targeting non-nested virtualization settings.
The paper introduces xGEMINI as an efficient solution targeting nested virtualization settings, where addressing these issues is particularly challenging, given the additional obstacles in creating synergy between host and guest VMs, due to an extra layer of page mappings by guest hypervisors. xGEMINI addresses these challenges by improving the shadow paging mechanism. Evaluation based on the KVM/Linux prototype implementation and diverse real-world applications shows xGEMINI greatly reduces TLB misses and enhances application performance in nested virtualization.

**Index Terms**—TLB, Nested Virtualization, Huge Pages, Memory Management

## 1 Introduction

In modern computer systems, translation lookaside buffer (TLB) capacity cannot scale at the same rate as memory capacity [1]. Many workloads, particularly big memory workloads, suffer frequent TLB misses, making virtual-to-physical address translations a serious performance bottleneck [2]. This bottleneck becomes even more pronounced on virtualized platforms, such as clouds, because address translations take much longer time than those on bare-metal platforms. The use of hardware-assisted address translation for memory virtualization (i.e., nested paging) requires a two-dimensional page walk performed upon each TLB miss on virtualized systems, which can be up to 6 times more costly than the one-dimensional page walks in bare-metal systems [3].

To reduce TLB misses, a practical and widely adopted solution is huge page strategies, with Linux Transparent Huge Page (THP) being one of them. They use multiple page sizes simultaneously in the same system and store large data chunks in huge pages [1]. Correspondingly, the virtual-to-physical page mappings utilized in address translations also have multiple granularities, one for each page size. For the two typical page sizes in mainstream systems, i.e., 4KB base pages and 2MB huge pages, two types of page mappings are used in address translations: *huge page mappings* between virtual and physical pages of 2MB, and *base page mappings* between virtual and physical pages of 4KB. For the data saved in a huge page, one huge page mapping is used to translate all the addresses within this huge page. When this mapping is cached in the TLB, visiting any addresses within this huge page does not incur TLB misses.

The effectiveness of huge page strategies relies on 1) establishing huge page mappings in page tables and 2) installing and caching huge page mappings in the TLB. Existing huge page strategies focus only on the former, assuming that the latter will be achieved automatically with the former. This assumption only holds on bare-metal systems, where huge page mappings maintained in page tables are loaded directly to the TLB when the corresponding pages are accessed.

However, the assumption does not hold on virtualized systems, where the page mappings used in the TLB are synthesized during the aforementioned two-dimensional page walks. For brevity, we call them *direct page mappings*, since they can be directly used to translate virtual addresses used by applications into physical addresses used by hardware to locate data in the memory. Each is synthesized from two *component page mappings*, one from the Guest Page Table (GPT) on the guest (i.e., VM), and the other from the extended page table (EPT) on the host, allowing the direct mapping from a guest virtual page (GVP) to a host physical page (GPP). If one of these component page mappings is a base page mapping and the other is a huge page mapping, the synthesis cannot generate a TLB cache-able page mapping. The reason is straightforward: a mapping can only map a virtual page to a physical page of the same size; hardware does not handle complex mappings that map multiple virtual pages to the same physical page or a virtual page to multiple physical pages.

We name this issue **Huge Page Misalignment (HPM) Problem**, since a huge page is not backing or is not being backed by another huge page, i.e., misalignment of huge pages at two layers. Misaligned huge pages do not contribute to reducing TLB misses; instead, they increase TLB misses since they prevent the generation of TLB cache-able page mappings. Existing huge page strategies independently create huge pages (i.e., huge page mappings) within individual layers. Some huge pages are aligned and help reduce TLB misses, and some others are mis-aligned

- *W. Jia is with Electrical, Computer, and Biomedical Engineering Department, The University of Rhode Island, Kingston, RI 02881. E-mail: weiwei.jia@uri.edu*
- *J. Zhang is with the Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, IL 61801. E-mail: jiyuanz3@illinois.edu*
- *J. Shan is with Computer Science Department, Hofstra University, Hempstead, NY 11549. E-mail: Jianchen.Shan@hofstra.edu*
- *X. Ding is with Computer Science Department, New Jersey Institute of Technology, Newark, NJ 07102. E-mail: xiaoning.ding@njit.edu*

*Manuscript received May 03, 2024.*

*. equal contribution

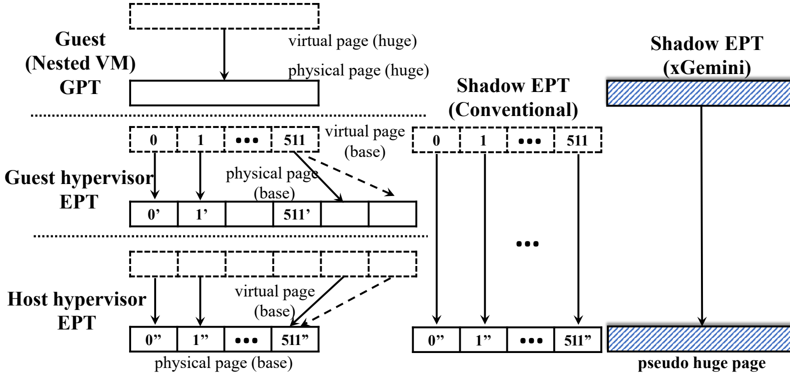



Fig. 1: The concept of pseudo huge page.

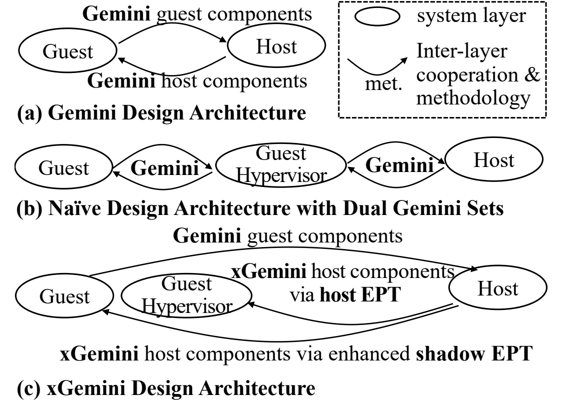



Fig. 2: Cross-layer cooperation in Gemini, a Naive solution, and xGemini.

and increase TLB misses. Because there is not a cross-layer mechanism to reduce the number of mis-aligned huge pages, their overall effectiveness in reducing TLB misses is low.

Our previous work designs GEMINI as a solution to coordinate the huge page strategies in the host and guests, and make them mutually cooperative to "properly align" huge pages (i.e., the huge pages within a guest being backed by huge pages on the host). With GEMINI, the guest has the information about which memory regions are being backed by the host with huge pages, and with the information, it preferentially allocates or promotes huge pages in these regions; meanwhile, the host has the information about which memory regions in the guest are huge pages, and tries to use huge pages (via allocation or promotion) to back these memory regions.

GEMINI has proven to be effective in non-nested virtualization environments, where guest VMs run directly on the host hypervisor (*host* for brevity). This paper introduces and evaluates XGEMINI as a solution designed specifically for nested virtualization contexts, where guest VMs run on guest hypervisors and guest hypervisors run on the host. Nested virtualization has become an indispensable configuration and is now offered in most public clouds, including Azure, Google Cloud, and Amazon AWS. It is used to support some important scenarios, such as micro-services in cloud-native environments [4], organizing and migrating multiple VMs together [5], using special purpose hypervisors, e.g., Hyper-V for running legacy applications in Microsoft Windows 11 [6], as well as formally-verified hypervisors for improved VM security [7].

Given the essential role of nested virtualization, it is imperative to address the issues associated with the huge page strategies in these environments, so as to reduce TLB misses and consequently enhance application performance. GEMINI creates synergy only between the guest and the host in synthesizing direct huge page mappings. It is not effective under the nested virtualization settings, where the synergy of all three system layers is required, because a direct page mapping has to be the combination of three component page mappings, one from each layer, i.e., guest page table (GPT), guest hypervisor EPT, and host hypervisor EPT, as shown in Figure 1 (left part). Base page mappings at any of these layers can prevent the synthesis of direct huge page mappings.

Creating the synergy of three layers (XGEMINI) is significantly more complex than creating the synergy between two layers (GEMINI), as the number of interactions, conflicting objectives, and constraints all grow. To reduce the complexity, rather than ensuring that all three layers are mutually cooperative, XGEMINI minimizes cross-layer cooperation and only maintains the following cooperation (shown in Figure 2).

- The mutual cooperation between the guest and the host: direct huge page mappings map huge guest virtual pages to huge host physical pages, requiring huge pages formed and aligned in both the guest and the host. Thus, the mutual cooperation between these two layers is indispensable.
- The cooperation of the host towards the guest hypervisor: This is to ensure the synergy of all three layers. Even if the guest hypervisor uses base pages to back a huge page in a guest, the host can intervene and rectify the situation by remapping these base pages in the guest hypervisor to the huge page on the host, such that the huge page in the guest is essentially backed by the host page on the host, as shown in Figure 1.

This design is partially motivated by two system designs for nested virtualization. One is the Direct Virtual Hardware (DVH) architecture [8], where the host directly provides virtual hardware to guest VMs bypassing guest hypervisors. The other is xPlace [9], where the host cooperates towards both guest hypervisors and guests in page placement mechanisms.

In addition to reduced complexity and overhead, this design brings multi-folds of other benefits. First, this design avoids the changes to guest hypervisor, which might not be feasible for closed-source or formally-verified hypervisors, or are intrusive even if they are feasible. Second, the guest components GEMINI can be reused in XGEMINI. These components are to achieve cooperation towards the host (shown on the top of Figure 2(c)). This not only simplifies the design but also allows the same guest to run on both GEMINI and XGEMINI platforms. Third, this design mirrors the de facto memory virtualization solution, simplifying its implementation on mainstream systems. To support multi-dimensional paging (i.e., EPT-on-EPT [5]), because only one EPT can be used in mainstream hardware at a time, a shadow EPT [10] is created to compress the EPTs in the guest hypervisor and the host hypervisor to allow direct mapping from GPA to HPA in the nested virtualization environments, as shown in Figure 1 (middle part). XGEMINI can leverage and enhance the shadow ETP mechanism to achieve the cooperation of the host towards guests in synthesizing direct huge page mappings (shown at the bottom of Figure 2(c)).

The above XGEMINI design does need to address a major challenge: some inherent flaws in the existing shadow EPT concept and designs impede the host from being cooperative with the

guests in forming huge page mappings. First, the existing shadow EPT mechanisms are huge-page unfriendly. They either do not support huge pages (i.e., synthesizing only base page mappings, e.g., that in Xen) or cannot effectively synthesize huge page mappings (e.g., that in KVM). With these designs, the host cannot effectively generate huge page mappings to cooperate towards guests. More importantly, in existing system designs, shadow EPTs are created and utilized as the synthesis of the EPTs in guest hypervisors and the EPTs in the host hypervisors. With this concept, the sizes of the mappings (base vs. huge) in a shadow EPT are dependent on the page allocations in the guest hypervisor. As shown in Figure 1 (left and middle parts), if the guest hypervisor allocates base pages, the mappings in the shadow EPT have to be base page mappings.

To remedy these flaws, XGEMINI first substantially renovates the shadow EPT concept. In XGEMINI, the shadow EPT is built as a special data structure that book-keep the address mappings between the guest physical pages and the corresponding host physical pages by introducing a concept of "pseudo huge pages". A pseudo huge page is a huge-page-sized region in the host's physical memory space that can be used to back a huge page in the guest. As shown in Figure 1 (right part), although this region may consist of base pages and cannot be combined into a real huge page from the perspective of a host EPT (because virtual pages are not contiguous), the shadow EPT can treat this region as a huge page to create a huge page mapping, which is then used by the MMU to form a huge page mapping from GVAs to HPAs.

To maintain pseudo huge pages, XGEMINI host components need to detect and respond to the changes in guest hypervisor EPTs. When a guest hypervisor attempts to change a page mapping that may affect a pseudo huge page (e.g., the dotted arrow shown in the guest hypervisor EPT in Figure 1), the host needs to change the host EPT accordingly (e.g., the dotted arrow shown in the host EPT in Figure 1). This design is shown in Figure 2 as the cooperation of the host towards the guest hypervisor.

With the above renovations, the granularities of the mappings in shadow EPTs can be completely controlled by the guests and the host. Thus, the huge page misalignment problem can be solved using a similar approach as GEMINI. However, different from the host components in GEMINI, the host components in XGEMINI need to use both shadow EPTs and host EPTs as primary data structures when allocating or promoting pseudo huge pages. Specifically, to form well-aligned huge pages, XGEMINI periodically scans the guest page table and the shadow EPT to identify and rebuild the overlooked huge page mappings by updating the host EPT and shadow EPT. This is shown in Figure 2(c) as the cooperation of the host towards the guest. Note that both XGEMINI and GEMINI aim to form more direct huge page mappings that are cache-able in TLB without requiring any changes to how multiple page granularities are implemented in TLB hardware.

The paper makes the following contributions. First, it analyzes and identifies the unique challenges in addressing the huge page misalignment problem in the nested virtualization environments. Second, it proposes XGEMINI as an effective solution that can efficiently reduce TLB misses and the address translation cost for nested virtualization; XGEMINI addresses a few technical challenges in implementing XGEMINI in nested virtualization environments. Finally, we have implemented XGEMINI based on KVM in Linux kernel 4.19 and tested it with diverse applications in the nested virtualization environments. Our tests show XGEMINI can significantly reduce TLB misses and effectively improve application performance and system efficiency.

## 2 Background

### 2.1 Address Translation in Native Environment

Most modern architectures use radix tree data structures known as page tables, to perform address translation. On native x86 systems, as shown in Figure 3, a standard page table has four levels and can map a 48-bit virtual address space. Each virtual address is 48 bits in size, with the lower 12 bits (bit 0∼bit 11) being the page offset and the upper 36 bits (bit 12∼bit 47) being the virtual page number. These 36 bits are further divided into four 9-bit fields.

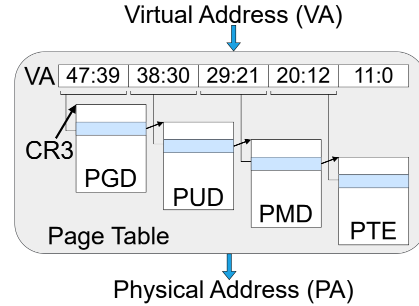



**Fig. 3: Address translation in the native environment.**

To translate a virtual address, the *page walker* navigates from the radix tree's root node to a leaf node. This process is known as a *page walk*. Each node is a memory page that includes an index table with 512 (i.e., $2^9$) entries. The CR3 register stores the root node's location, while the locations of intermediate or leaf nodes are determined by selecting an entry from the index table of an upper-level node, using the respective 9-bit field from the virtual address, as shown with the blue squares in Figure 3. In the last step of the page walk, the physical page number is obtained as the result of the page walk from the leaf node using the lowest 9-bit field. The complete physical address is then formed by appending the page offset (bits 11:0 of the VA) to the physical page number.

With a 4-level page table, a page walk may incur 4 memory accesses. To support emerging applications with terabytes of memory, architectures including x86 have started to support 5-level page tables, where a page walk may incur 5 memory accesses.

### 2.2 Address Translation in Non-Nested Virtualization

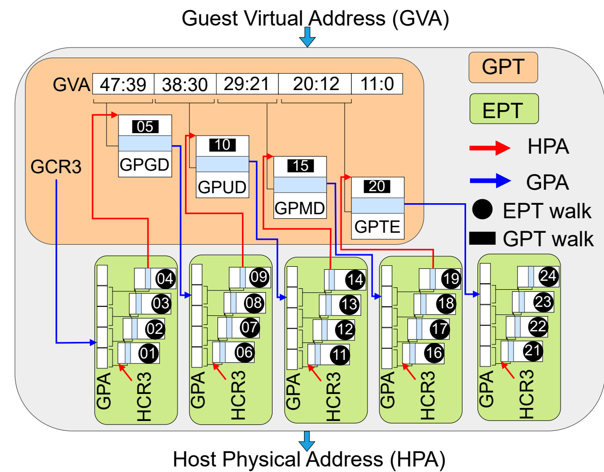



**Fig. 4: Address translation in the non-nested virtualization environment. Each blue square represents one possible memory access. Host EPT is rotated by 90 degrees.**

With hardware supports (e.g., Intel EPT [3] and AMD NPT [11]), *nested paging* is used for address translation on virtualized platforms. It allows guests and the host to each manage and map memory independently using their own page tables. A guest OS maintains *guest page tables* (GPTs) mapping guest virtual addresses (GVAs) to guest physical addresses (GPAs), and the host manages *extended page tables* (EPTs) mapping GPAs to host physical addresses (HPAs).

On virtualized systems, the address translation is to obtain the HPA for a given GVA. This involves the use of two mappings maintained in a GPT and an EPT in a *two-dimensional (2D) page walk*, as illustrated in Figure 4. We use circular boxes to denote the steps in EPT (01–04, 06–09, 11–14, 16–19, and 21–24), and rectangle boxes to denote the steps in GPT (05, 10, 15, and 20). First, to obtain the GPA from GVA, the hardware needs to walk over each level of the GPT, from the guest page global directory (GPGD) to guest page table entry (GPTE), i.e., the rectangle boxes (Steps 5, 10, 15, and 20). However, each of these steps needs to use the corresponding HPA, to obtain which a conventional page walk is incurred in EPT from host page global directory (HPGD) to host page table entry (HPTE), i.e., the circular boxes (Steps 1–4, 6–9, 11–14, or 16–19). Second, to obtain the final HPA from GPA, the hardware needs to walk over the EPT for one more time (Steps 21–24) to locate the PTE and obtain the physical page number.

For 4-level radix page tables, the 2D page table walk requires up to 24 *sequential* memory accesses [12] , as shown in Figure 4. With 5-level page table enabled on both guest and host, it takes up to 35 *sequential* memory accesses.

### 2.3 Address Translation in Nested Virtualization

In nested virtualization, guest hypervisors run in VMs to host guest VMs inside VMs [5]. Nested virtualization has been offered by major cloud vendors to enable important use cases for higher cost-effectiveness, compatibility [5], [6], or security [13]. For instance, Microsoft Windows 11 provides virtualization-based kernel integrity protection [14] and operating system interoperability [15]; Linux also has a similar implementation in progress [16]. Nested virtualization must be enabled to run these systems in guest VMs [6].

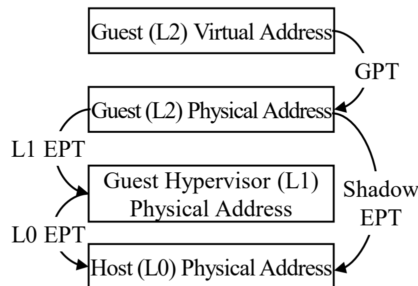



**Fig. 5: Address translation in nested virtualization.**

Nested virtualization requires multi-dimensional paging. A popular solution is EPT-on-EPT [5], as shown in Figure 5. L2, L1, and L0 refer to the guest, guest hypervisor, and host hypervisor layers throughout the paper, respectively. At L1, to run a guest VM (L2), a L1 EPT is emulated to translate GPA to the guest hypervisor physical address (GHPA). At L0, to run a guest hypervisor (L1), a L0 EPT is emulated to translate GHPA to HPA. The address translation requires a three-dimensional page walk across 3 layers of page tables (i.e., GPT, L1 EPT, and L0 EPT).

Currently, hardware cannot support such page walks, which are untenable due to excessive memory accesses for each address translation. Existing solutions use a combination of hardware-assisted nested paging and software-assisted shadow paging [17], [18]. They map three layers of page tables onto two layers and leverage 2D-page walks for address translation (described in §2.2). Specifically, the page table in the guest hypervisor (i.e., L1 EPT) and the page table in the host hypervisor (i.e., L0 EPT) are squashed into one shadow EPT, Then, a 2D page walk can translate a GVA to HPA using the GPT and the shadow EPT. To keep consistency between the emulated EPTs and shadow EPTs, the host needs to monitor any changes to the emulated EPTs.

### 2.4 Translation Lookaside Buffer (TLB)

To minimize page walks, modern processors use translation lookaside buffers (TLBs) to cache and reuse the results of earlier page walks, i.e., the mappings between the virtual page numbers and the physical page numbers, the virtual page numbers being the keys/tags for TLB look-ups and physical page numbers being the values returned by the look-ups and used to assemble physical memory addresses. Upon a memory access, when the mapping required for the address translation is already cached in the TLB (i.e., a TLB hit), no page walk is needed. Otherwise (i.e., a TLB miss), a page walk must be conducted to locate the PTE and load the virtual-to-physical page mapping to the TLB.

TLBs usually have small sizes to keep the address translation latencies low. TLB misses are a serious performance bottleneck on many systems. To reduce this overhead, various techniques have been integrated, such as page walk cache (PWC) for caching the intermediate results of page walks, host TLB for caching guest physical to host physical mapping, and caching page table entries in L2 and L3 caches. But TLB misses still can take up to 50% of application execution time under nested paging [19].

### 2.5 Huge Page Strategies

Huge pages (e.g., 2MB pages on x86 platforms), sometimes also called superpages, can reduce address translation overhead in two ways. First, huge page mappings can reduce TLB misses in TLB lookups. A TLB entry for a huge page mapping can be used to translate addresses for an increased amount of data (e.g., 2MB with a huge page PTE vs. 4KB with a base page PTE). This significantly increases TLB coverage and reduces TLB misses. Second, huge page mappings can reduce the steps in a page walk and the corresponding memory references. In this paper, unless specified otherwise, the huge page refers to a 2MB page. Thus, for an address in a huge page, its lower 21 bits are the page offset, and the rest 27 bits are 3 fields, 9-bit each. Thus, a page walk needs only 3 steps (i.e., at most 3 memory accesses), one for each field.

To leverage huge pages to reduce address translation overhead, huge page mappings must be established. This can be achieved by directly allocating huge pages upon page faults or "assembling" base page mappings into a huge page mapping through a process called *huge page promotion*. Huge page allocation starts by having a huge virtual page, which is then mapped to a huge physical page. Huge page promotion starts by having 512 base virtual pages that are sequentially organized in memory, with the starting address aligned to 2MB. If these pages are directly mapped to physical pages in the same sequential order, with the starting address also 2MB aligned, then an *in-place huge page promotion* can be performed. This involves merging these base virtual pages into one

huge virtual page and similarly combining the physical pages into one huge physical page, followed by updating the mappings in the page table accordingly. If the physical pages cannot meet the above criteria, an *out-of-place huge page promotion* is required. This involves remapping the base virtual pages to physical pages that do meet the criteria, transferring the data, and then performing the same steps as in an in-place promotion. When a huge page is under-utilized, huge page demotion can reverse the promotion operation and split a huge page back into base pages. Huge page promotions and demotions, particularly out-of-place promotions, are expensive because costly operations, such as memory copying and TLB flushing, are involved. Thus they are typically executed asynchronously in the background to minimize the impact on system performance, as seen with Linux's khugepaged daemon.

Applications can request huge page allocation or promotion via system calls (e.g., *madvise*). However, most applications rely on system-level huge page strategies, such as Transparent Huge Page (THP) support in Linux and FreeBSD, to automatically manage the allocation and promotion of huge pages and enhance performance transparently without the need of code changes from the applications.

# 3 Research Problem and Challenges

## 3.1 Huge Page Misalignment in Virtualized Systems

On a native system, when accessing the data in any huge pages, the mappings between the virtual and physical huge pages can be cached in TLB to reduce TLB misses. Thus, the more huge pages are created and used, the more address translation overhead can be reduced. However, on virtualized platforms, guests and hosts have their own huge page strategies. They manage huge page allocation and promotion independently using different page tables. Thus, the same guest virtual page may be backed at different layers by physical pages of different sizes. For example, a huge page in the guest may be backed by base pages in the host, as shown in Figure 6. In this case, for this guest virtual page, there is no direct page mapping that can be used to translate the addresses within this page directly to host physical addresses. Depending on specific designs, a TLB may choose not to cache a direct mapping in this case or cache direct mappings in a smaller granularity (i.e., at the size of base pages [11]). Either way, huge page mappings formed at individual layers cannot help reduce TLB misses. With fewer mappings being cached, the former design would even increase TLB misses. The paper refers to such huge pages as *misaligned huge pages* and this problem as *huge page misalignment problem*. Using huge pages can reduce TLB misses only when a virtual huge page is backed by a physical huge page at both the guest and the host layers. For brevity, we refer to such huge pages as *well-aligned huge pages*.

On virtualized platforms, the huge pages may still be "well-aligned". However, it is largely by chance. Though this chance increases when more huge pages are created in every layer, the pressure to reduce the adverse effects of huge pages (e.g., space waste and paging overhead) caps the chance, limiting the effectiveness of huge page strategies.

Though the misaligned huge pages still can help reduce page walk overhead, they increase TLB misses. Thus, they can hardly reduce address translation overhead when the benefits of reducing page walk overhead is largely offset by increased misses. Using well-aligned huge pages can substantially improve performance by reducing both TLB misses and page walk overhead.

Our previous work, GEMINI [20], identifies this huge page misalignment problem and analyzes the causes. It reveals and experimentally confirms that only huge guest pages backed by huge host pages can effectively reduce address translation overhead. Existing huge page strategies only aim to increase huge pages at each layer, and fail to consider this cross-layer requirement on the alignment of huge pages.

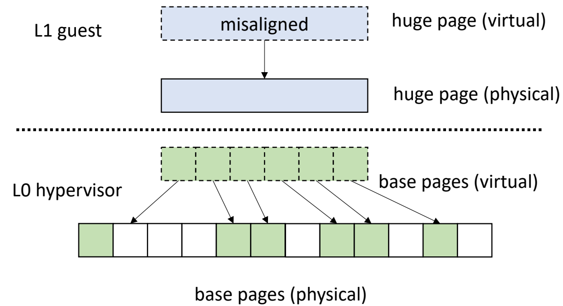



**Fig. 6: An example of a misaligned huge page. Note that an actual 2MB huge page contains 512 base pages.**

To show how virtualization affects the effectiveness of huge page management, we measure the performance of a micro-benchmark when misaligned huge pages and well-aligned huge pages are used, respectively. We show the results in Figure 7. The micro-benchmark running in a virtual machine randomly accesses a data set. When the data set is small, well-aligned huge pages show similar performance as the baseline; however, the performance of misaligned huge pages is even worse than the baseline. This is because misaligned huge pages incur more TLB misses compared to the baseline. When the data set is large, well-aligned huge pages can greatly improve performance, because they can reduce TLB misses and address translation overhead. Misaligned huge pages can hardly improve performance compared to the baseline, as the benefits of reducing page walk overhead are largely offset by increased TLB misses.

**Fig. 7: Well-aligned huge pages can effectively reduce address translation overhead. Throughput is normalized to that of vanilla Linux/KVM.**

To address this issue, GEMINI designs a cross-layer solution for non-nested virtualization environments. It guides the allocation and promotion of huge pages in guests and the host. With GEMINI, the guest has the information about which memory regions are being backed by the host with huge pages, and with the information, it preferentially allocates or promotes huge pages in these regions; meanwhile, the host has the information about which memory regions in the guest are huge pages, and tries to use huge pages (via allocation or promotion) to back these regions. Because huge pages are preferentially formed and allocated from these regions and less from other regions, more well-aligned huge pages can be

formed without aggravating the adverse effects incurred by excessive huge pages. However, GEMINI cannot be directly applied in nested virtualization as we will explain below. Our evaluation also shows directly applying GEMINI in nested virtualization cannot effectively resolve the huge page misalignment problem (see §5).

### 3.2 Research Challenges in Nested Virtualization

To address the huge page misalignment problem in nested virtualization environments, we need to overcome three main challenges. The first major challenge is the interposition of guest hypervisors between the guest OS and the host OS. Under non-nested virtualization environments, the host can directly control the memory page allocations to VMs to facilitate the creation of well-aligned huge pages with GEMINI. However, this becomes very challenging in nested virtualization environments. Theoretically, we can apply two Gemini systems to three layers in the nested environment (i.e., Gemini guest component in the nested VM, paring with Gemini host component in the guest hypervisor, and another Gemini guest component in the guest hypervisor, pairing with another Gemini host component in the host hypervisor) as shown in Figure 2(b). With slight changes, the two separate Gemini systems can communicate and coordinate to make well-aligned huge pages across three layers. However, this design is not viable for two reasons. First, it requires the changes of guest hypervisors (not possible for closed-source hypervisors or prohibitive for formally verified hypervisors). Second, coordinating three layers is inherently more challenging and costly than coordinating two layers. It is harder to reach an agreement since the huge page alignment made by one Gemini system between two layers may not be ideal for the other Gemini system to enforce alignment between the other two layers. For example, one Gemini system may make a huge page in the nested VM backed by a huge page in the guest hypervisor which, however, is backed by base pages that are not contiguous on the host. In such a case, the other Gemini system has to perform costly out-of-place huge page promotion using page migration to let the huge page in the guest hypervisor be backed by a huge page on the host. Moreover, frequent changes would be made to the guest hypervisor EPT, triggering expensive VMexits to update the shadow EPT. The high overhead would offset the benefits of reduced TLB misses.

The second major challenge is that the shadow EPT used for nested virtualization does not well support huge pages. To our knowledge, some hypervisors do not support huge pages in their shadow page table mechanisms [21]. Other hypervisors may support huge pages only when a guest hypervisor huge page is backed by a host hypervisor huge page, such that the huge page mappings in both guest hypervisor EPT and the host hypervisor EPT can be merged into the shadow EPTs without incurring any issues. This may miss some opportunities when a guest huge page is mapped to multiple guest hypervisor base pages that are actually backed by a huge page or base pages in a huge page aligned and sized region in the host. In this case, the shadow EPT should also form a well-aligned huge page to translate guest physical address to host physical address.

The last major challenge is how to fully unlock the performance benefits of XGEMINI. Nested virtualization incorporates four layers of address space, including guest virtual address (GVA) space, guest physical address (GPA) space, guest hypervisor physical address (GHPA) space, and host physical address (HPA) space, as shown in Figure 5. If guest virtual addresses of an application is not allocated and aligned with huge pages, it cannot benefit from the well-aligned huge pages created by XGEMINI. To make guest virtual addresses aligned with huge pages, we need to improve the application's virtual memory allocation mechanisms (*malloc*) by modifying the guest OS because GVAs are initiated by the application and actually allocated by the guest OS. Modern OS usually uses virtual memory areas (VMAs) to organize application virtual address space. VMA is an OS abstraction of contiguous regions in the virtual address space of a program. Each VMA contains a set of virtual pages with the same protection, representing a local data section (e.g., code, data, heap, stack, or a memory-mapped file).It has a base virtual address and size of the mapped region, along with other metadata. Collectively, VMAs in a program constitute the program's working set [22]. It is challenging to modify guest OS to allocate huge page aligned VMAs to maximize the performance of XGEMINI.

## 4 XGEMINI Design

### 4.1 System Overview

Figure 8 shows XGEMINI's system architecture. XGEMINI is designed for nested virtualization and built based on GEMINI [20] that was designed for non-nested virtualization. XGEMINI includes three new components (highlighted with blue in Figure 8): 1) Huge Aligner (HA) allocates huge page aligned guest virtual memory for application data to maximize XGEMINI's performance; 2) Nested Enhanced Memory Allocator (NEMA) enhances the memory allocator to create huge pages from the memory regions reserved by the Huge Booking in the nested virtualization environments. 3) Nested Well-aligned Huge Page Creator (NWHPC) scans the guest page table and the shadow EPT to identify the misaligned huge pages and create well aligned huge pages by modifying the corresponding mappings in the host hypervisor EPT and the shadow EPT.

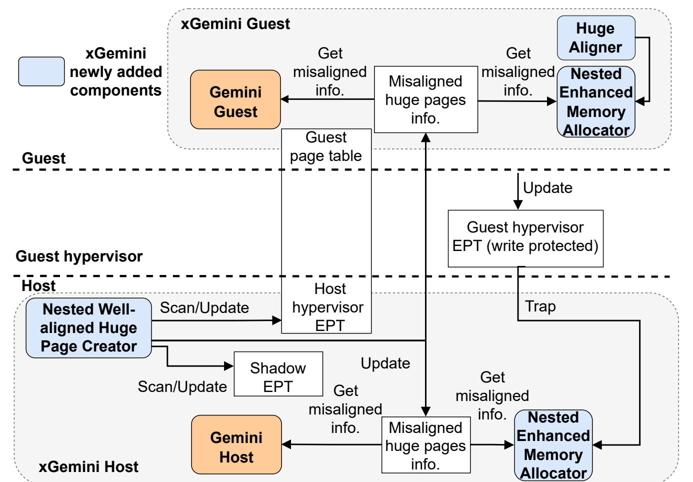



**Fig. 8: XGEMINI System Overview.**

XGEMINI reuses some components from GEMINI, such as Huge Booking and Misaligned Huge Page Promoter, as represented by the orange boxes in Figure 8. Huge Booking temporarily reserves well-aligned huge-page-sized memory regions between the guest and the host upon the VMA being touched for the first time in a non-nested virtualized environment. Misaligned Huge Page Promoter preferentially promotes misaligned huge pages in guest- and host-level in non-nested virtualization environments. Next, we detail the three newly added components in XGEMINI and explain their interactions with the reused GEMINI components.

### 4.2 Huge Aligner

The application running in guest VM invokes *malloc* (or other memory allocation interfaces) to allocate guest virtual address space. To allocate the GVA space, the OS returns virtual memory areas (VMAs), which are contiguous regions in the virtual address space of the application. VMAs are dynamically changed. For instance, a small VMA of one application can be expanded into a larger one as the workload executes and requests.

Figure 9a shows how default guest virtual address space is allocated for an application. The application first requests some virtual memory space and the OS returns the requested VMA, as highlighted with blue in Figure 9a. Later, the application wants more GVA space by calling the system call (e.g., *mremap*) to expand the VMA; and the OS expands the VMA, as highlighted with orange in Figure 9a. Such VMA expansion may bring a serious issue. Once the huge page aligned virtual memory region is all allocated, it needs to pay a high cost to form a huge page (i.e., out-of-place huge page promotion) [23]. The main reason for the high overhead is that the out-of-place huge page promotion incurs costly memory copy, as base pages backing the huge page aligned guest virtual memory region need to be copied to another huge page sized physical memory region for promotion.

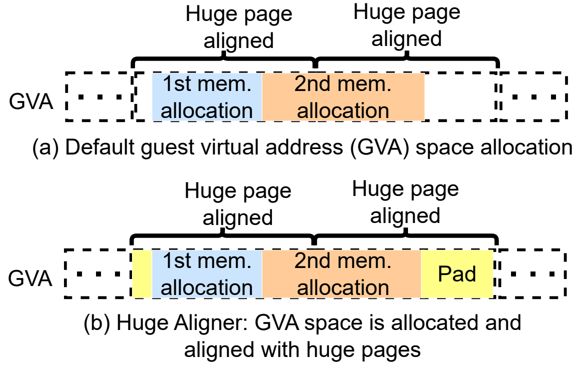



**Fig. 9: Huge Aligner.**

To address the problem and create more well aligned huge pages in nested virtualization, XGEMINI proposes Huge Aligner, as shown in Figure 9b. The main idea is to allocate the whole huge page aligned guest virtual memory region when it is touched for the first time. XGEMINI's Nested Enhanced Memory Allocator allocates corresponding guest physical memory and host physical memory aligned with the guest virtual memory allocation while reserving the corresponding padding regions, such that the huge page promotion cost can be eliminated, as we will explain in Section 4.3. The solution is practical and effective (confirmed in §5), albeit it may waste some guest virtual memory space. However, we argue that the virtual memory space of an application is usually large and we have not found any issues in our implementations and evaluations. Moreover, if this may be an issue, we may consider some mechanisms to allocate the whole huge page aligned guest virtual memory region only when the size of the memory allocation space is larger than a threshold (e.g., 70%).

### 4.3 Nested Enhanced Memory Allocator

The main goal of the Nested Enhanced Memory Allocator (NEMA) is to form well-aligned huge pages without changing the guest hypervisor in the nested virtualization environments. The number of well-aligned huge pages can be significantly increased, if HPA, GPA, and GVA are well aligned to huge pages when the VMA is touched for the first time. We do not need to align the guest hypervisor physical address (GHPA) to huge pages because hardware only supports two-dimensional page walks by walking the guest page table and the shadow EPT to realize three dimensional page walks for nested virtualization, as introduced in §2.3. We show how NEMA works in Figure 10.

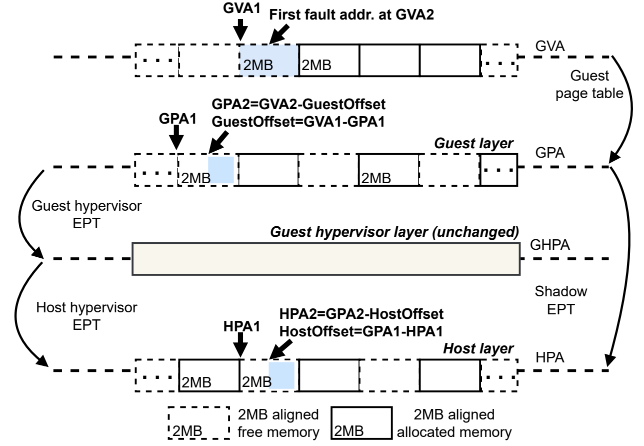



**Fig. 10: Nested Enhanced Memory Allocator (NEMA).**

Thanks to the Huge Aligner, NEMA only needs to align GPA and HPA to GVA upon the first page fault to the virtual memory area (VMA). This is because Huge Aligner makes the VMA in the GVA space already aligned with huge pages. When the VMA (highlighted with blue in GVA in Figure 10) is touched at GVA2 for the first time, NEMA allocates guest physical memory space starting at GPA2. GPA2 is aligned to GVA2 based on huge pages.

Particularly, NEMA guest component first locates the starting address of the Huge Page Aligned and Sized (HPAS) region (GVA1 in Figure 10) that the VMA's starting address (GVA2) belongs to. Then, NEMA finds a free HPSA region in GPA that can fit the VMA. Next, NEMA locates the starting address of the free HPSA region found in GPA (GPA1). Finally, NEMA calculates the offset between GVA1 and GPA1 (i.e., $GuestOffset = GVA1 - GPA1$) to maintain a one-to-one mapping between the HPSA regions in GVA and GPA, facilitating any allocation in that HPSA region in GVA in calculating where to allocate GPA (e.g., GPA2).

Subsequently, the guest hypervisor would respond to the page fault by modifying the guest hypervisor EPT, which triggers a VM exit trapping to the host where NEMA host component can find a corresponding free HPSA region in HPA. The starting address of HPA is fault at HPA2 that is aligned to GPA2 using the offset in the host level that is calculated in the same way as it is in the guest level (i.e., $HostOffset = GPA1 - HPA1$). For forthcoming memory allocations, $GuestOffset$ is used as the guest level offset to calculate where to allocate GPA; HostOffset is used as the host level offset to calculate where to allocate HPA. These two types of offset are calculated and maintained individually by the NEMA guest and host components. There is no need to pass the GuestOffset to the host or verse-visa. This process is transparent to the guest hypervisor and the memory management in the guest hypervisor is untouched. Please note that XGEMINI does not allocate a huge page if the touched memory space is smaller than a huge page. The memory size allocated by XGEMINI depends on the size of the touched memory space in the VMA (e.g., memory space marked as blue in GPA and HPA, respectively). The Huge Booking component [20] is used to temporarily reserve these HPSA regions to form well-aligned huge pages.

Host memory allocations are triggered in two ways. First,

**TABLE 1: Configurations of the evaluation platform.**

| Parameter | Configuration |
|---|---|
| Machine Type | Dell EMC PowerEdge T630 |
| L0 Processor | Intel® Xeon® E5-2620 v4 @ 2.10GHz (2 sockets) |
| L0 Memory | 32GB DDR4 2400 MT/s (2 per socket, 128GB total) |
| L0 Kernel | Linux 4.19.60 |
| L0 Host Hypervisor | QEMU/KVM |
| L1 VM Processor | 32 vCPU |
| L1 VM Memory | 110 GB |
| L1 Kernel | Linux 4.19.60 |
| L1 Guest Hypervisor | QEMU/KVM |
| L2 VM Processor | 32 vCPU (Single VM) / 16 vCPU (Colocated VMs) |
| L2 VM Memory | 100 GB (Single VM) / 50 GB (Colocated VMs) |
| L2 Kernel | Linux 4.19.60 |

**TABLE 2: Workloads used in the evaluation.**

| Name | Suite | Usage | Domain |
|---|---|---|---|
| Img-dnn | Tailbench | Latency, Throughput | Image recognition |
| Masstree | Tailbench | Latency, Throughput | Key-value store |
| Moses | Tailbench | Latency, Throughput | Real-time translation |
| Silo | Tailbench | Latency, Throughput | In-memory database |
| Specjbb | Tailbench | Latency, Throughput | Java middleware |
| Sphinx | Tailbench | Latency, Throughput | Speech recognition |
| Xapian | Tailbench | Overhead | Online search |
| Canneal | PARSEC | Latency, Throughput | Simulated annealing |
| Facesim | PARSEC | Throughput | Motion simulation |
| Raytrace | PARSEC | Throughput | Real-time raytracing |
| Streamcluster | PARSEC | Throughput | Online clustering |
| Dedup | PARSEC | Throughput | Data deduplication |
| x264 | PARSEC | Throughput | Video encoding |
| FFT | PARSEC | Throughput | Scientific computation |
| Ferret | PARSEC | Overhead | Content search |

when the guest hypervisor physical page is touched for the first time without being backed by host physical memory page, host conducts the aforementioned procedure to allocate host physical HPSA memory region and modifies the shadow EPT to reflect such change. Second, the guest hypervisor physical page is touched for the first time and has been backed by the host physical memory page. This may happen because, in the virtualization environments, the address mapping between GHPA and HPA remains as long as the guest hypervisor is alive or until the host OS reclaims it [24], [25]. In this case, when the guest hypervisor modifies the guest hypervisor EPT, it traps to the host to update the shadow EPT, as the GHPT is write-protected. If this is the first fault for the corresponding VMA, the aforementioned host physical memory allocation is conducted. This may need to modify the mapping between the GPA space to the HPA space by allocating new host HPSA memory region to accommodate and create well-aligned huge pages (i.e., pseudo huge pages as shown in Figure 1) but the overhead is acceptable (confirmed in §5) as it does not involve data copying.

### 4.4 Nested Well-aligned Huge Page Creator

This component is used to further increase the rate of well-aligned huge pages without incurring extra overhead in the nested virtualization environments. With the aforementioned mechanisms, huge pages at the guest level backed by huge pages at the host level may still not form well-aligned huge pages. This is because the shadow EPT is formed by combining the guest hypervisor EPT and the host hypervisor EPT; and if a guest huge page is backed by guest hypervisor base pages, it may not form well-aligned huge pages in the shadow EPT. Even worse, shadow EPTs in some hypervisors do not support huge pages [21]. To address this issue, we propose Nested Well-aligned Huge Page Creator (NWHPC). Specifically, NWHPC periodically scans the guest page table and the shadow EPT and forms well-aligned huge pages only when a guest-level huge page is backed by a host-level pseudo huge page. This is doable because the shadow EPT is used to map a guest physical address to a host physical address.

## 5 Evaluation

We have implemented xGEMINI prototype based on our previous work, GEMINI [20]. We added and modified around 800 LoC mainly in the Linux kernel memory management and KVM kernel module. We conducted our evaluation on a Dell PowerEdge machine as shown in Table 1 where the details, such as the guest VM kernel, guest hypervisor, and host hypervisor are listed. The evaluation is conducted under three huge page strategies:

- Vanilla Nested Linux/KVM: As ineffective huge page strategies can hurt the performance, to show the effectiveness of the proposed huge page strategy, the transparent huge page is disabled in this setting to serve as the baseline. Although applications can still allocate huge pages by invoking the madvise interface, most applications do not use that interface.
- GEMINI: We deploy a dual GEMINI setup as shown in Figure 2(b). One GEMINI guest component is in the nested VM, paring with GEMINI host component in the guest hypervisor, and another GEMINI guest component is in the guest hypervisor, pairing with another GEMINI host component in the host hypervisor. The two components at the guest hypervisor layer can communicate and coordinate to make well-aligned huge pages across three layers. Though we still call this scenario GEMINI for brevity, note that deploying a single GEMINI is ineffective in reducing TLB misses.
- xGEMINI: xGEMINI is deployed to guest and host only. No change is made to the guest hypervisor. GEMINI was built for non-nested virtualization. A comparison between xGEMINI and dual GEMINI can illustrate the superior effectiveness of xGEMINI in nested virtualization.

The objective of our evaluation is three-fold: 1) to show that xGEMINI can improve the throughputs of throughput-oriented workloads compared to vanilla nested Linux/KVM and GEMINI (§5.1), 2) to show that xGEMINI can reduce mean and tail latencies of latency-sensitive workloads compared to vanilla nested Linux/KVM and GEMINI (§5.2), and 3) to evaluate the applicability and overhead of xGEMINI (§5.3).

A diverse set of workloads are used in our evaluation, as listed in Table 2. Specifically, we evaluated xGEMINI with TLB-sensitive workloads in Tailbench [26] and PARSEC [27] benchmark suites. To test xGEMINI's applicability and overhead, we use two TLB-nonsensitive workloads, i.e., Xapian and Ferret. In our evaluation, each nested VM encapsulates one workload.

We categorized the benchmarks into two types: throughput-oriented benchmarks (provided by the PARSEC benchmark suite) and latency-critical benchmarks (provided by the Tailbench benchmark suite). We tested the workloads with memory fragmentation and without memory fragmentation, respectively. We care more about xGEMINI's performance when memory is fragmented because previous works [23], [28] show that memory quickly fragments in multi-tenant virtualized cloud environments. We first measured the throughputs of the throughput-oriented workloads. Then, we collected average and tail latencies reported by the latency-sensitive workloads. Since Tailbench workloads also report throughputs, we show them in the figures related to throughput-oriented workloads. The performance results may vary significantly across different workloads. When we present

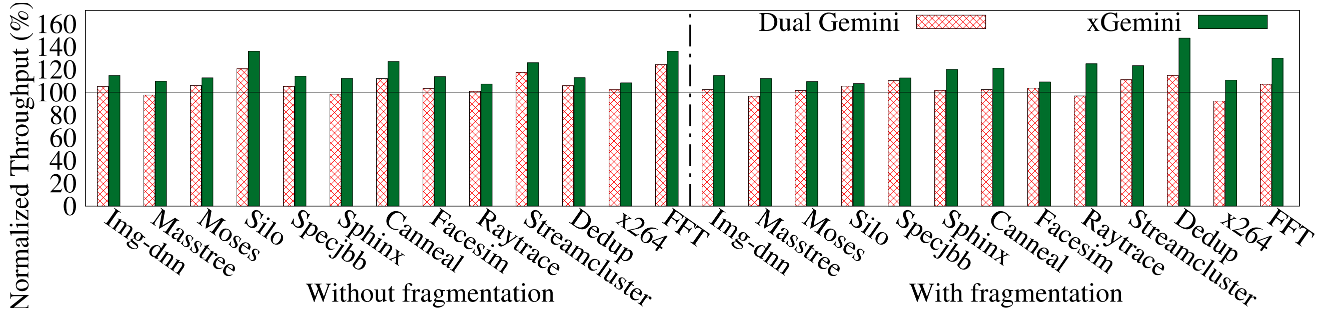



**Fig. 11: Throughputs of GEMINI and XGEMINI. Throughput is normalized to that of vanilla nested Linux/KVM.**

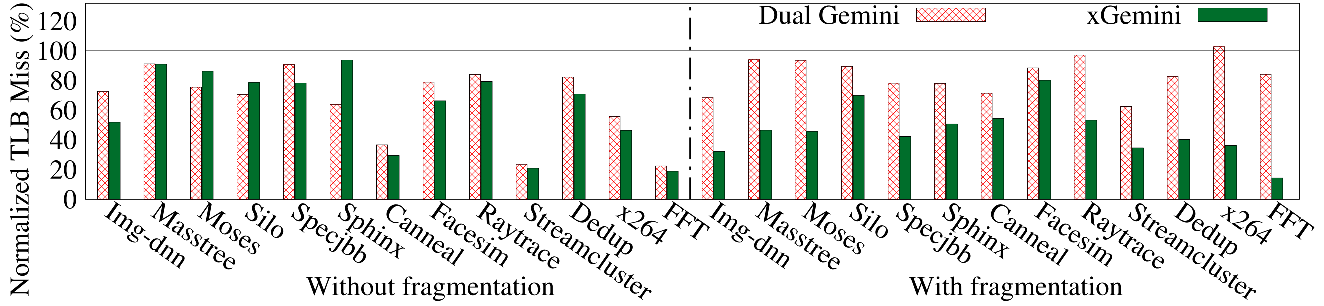



**Fig. 12: TLB misses of GEMINI and XGEMINI. TLB miss is normalized to that of vanilla nested Linux/KVM.**

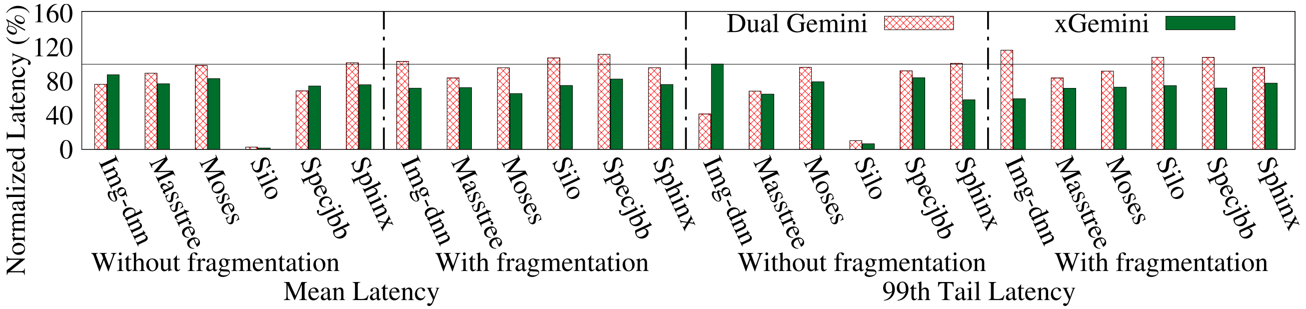



**Fig. 13: The mean and tail latencies of GEMINI and XGEMINI. Latency is normalized to that of vanilla nested Linux/KVM.**

these results in figures, for clarity, we normalized the performance results of XGEMINI and Dual GEMINI against those of vanilla nested Linux/KVM, as indicated in the figures.

### 5.1 Experiments with Throughput-Oriented Workloads

Figure 11 shows the throughputs of all the evaluated workloads, when the three systems are tested with memory fragmentation and without memory fragmentation, respectively.

Without memory fragmentation, XGEMINI offers 16.5% more throughput on average, compared to vanilla nested Linux/KVM; this is because XGEMINI reduces TLB misses by 35.0% on average relative to vanilla nested Linux/KVM, as shown in Figure 12. With memory fragmentation, XGEMINI increases throughput by 18.8% and reduces TLB misses by 53.8%, compared to vanilla nested Linux/KVM. On average, GEMINI outperforms vanilla nested Linux/KVM by 16.6% and 3.5% with memory fragmentation and without memory fragmentation, respectively. The main reason is that GEMINI increases more huge pages in each virtualization layer, such that the TLB misses are reduced, albeit this also incurs much overhead.

In comparison to GEMINI, XGEMINI improves throughput by 9.4% and decreases TLB misses by 3.7% on average when memory is not fragmented. With memory fragmentation, on average, XGEMINI provides 15.3% more throughput and 37.7% lower TLB misses relative to GEMINI. There are two main reasons. First, XGEMINI increases the number of well aligned huge pages through better huge page management mechanisms specifically designed for nested virtualization, which can significantly reduce TLB misses and improve application throughput especially when memory is fragmented. Second, by aligning huge pages across different virtualization layers, XGEMINI makes different virtualization layers work on the same huge page, mitigating redundant page faults and thus reducing shadow paging overhead, which is mainly generated by the VMExits upon handling nested page faults [6].

For some applications (e.g., Silo and Sphinx), XGEMINI outperforms GEMINI but shows more TLB misses in the non-fragmentation environments. This is due to GEMINI's extra control at the guest hypervisor layer. When a workload leads to constant changes to the guest hypervisor EPT, XGEMINI has to constantly modify the host hypervisor EPT to maintain the direct huge page mappings. On the contrary, the dual Gemini setup can well control the guest hypervisor EPT to reduce the misalignment in such a case by creating more huge pages in the guest hypervisor, possibly leading to fewer TLB misses. However, GEMINI provides worse throughput compared to XGEMINI due to the extra overhead. For instance, producing more huge pages leads to frequent changes to guest hypervisor EPT, triggering expensive VMexits to update the

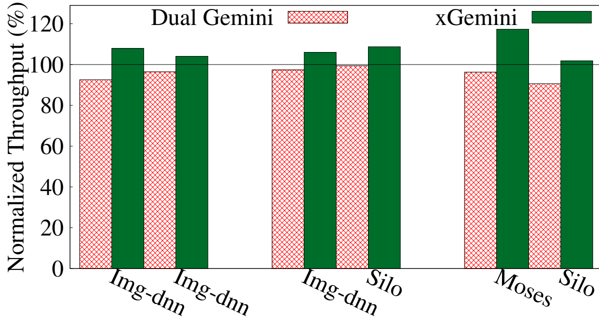



**Fig. 14: Throughputs of GEMINI and XGEMINI when multiple nested VMs are colocated together. Throughput is normalized to that of vanilla nested Linux/KVM.**

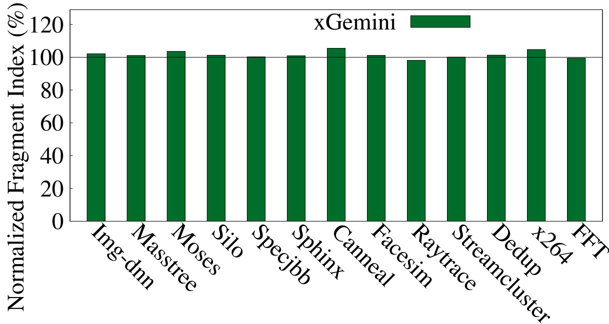



**Fig. 15: Fragmentation caused by XGEMINI. Fragmentation index is normalized to that of vanilla nested Linux/KVM.**

shadow EPT. XGEMINI forms well-aligned huge pages without creating more huge pages and incurring much overhead thanks to the direct coordination between the guest and the host.

### 5.2 Experiments with Latency-Sensitive Workloads

Figure 13 shows the mean and tail latencies of different systems when they are tested with memory fragmentation and without memory fragmentation. Relative to vanilla nested Linux/KVM, XGEMINI reduces the mean latency by 28.0% and the 99th tail latency by 28.4% on average, when memory is not fragmented. With memory fragmentation, XGEMINI offers 26.1% lower mean latency and 28.5% lower 99th tail latency on average compared to vanilla nested Linux/KVM. Compared to GEMINI, XGEMINI reduces the mean latency by 25.7% and the 99th tail latency by 28.0% on average, when memory is fragmented.

To understand why XGEMINI provides lower mean and tail latencies compared to vanilla nested linux/KVM and GEMINI, we profile the TLB misses when the three systems are tested with memory fragmentation and without memory fragmentation. We show the test results in Figure 12. On average, XGEMINI reduces the TLB misses by 35.0% without memory fragmentation and 53.8% with memory fragmentation, compared to vanilla nested linux/KVM. This is consistent with the latency test results and also shows XGEMINI's effectiveness in reducing the latency for latency-sensitive workloads as well as the overhead of TLB misses. XGEMINI shows lower latencies compared to GEMINI because XGEMINI forms more well aligned huge pages in nested virtualization, such that it can further reduce the mean and tail latencies through decreasing TLB misses and address translation cost.

To understand whether XGEMINI causes more memory fragmentation compared to vanilla nested Linux/KVM, we measure

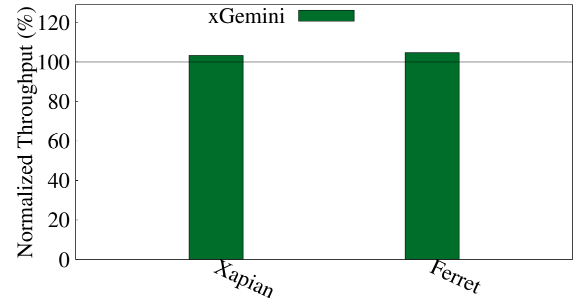



**Fig. 16: Overhead of XGEMINI. We leverage TLB non-intensive workloads to measure XGEMINI's overhead. Throughput is normalized to that of vanilla nested Linux/KVM.**

their fragmentation using the free memory fragmentation index (FMFI) [1], [29] while testing the benchmarks. FMFI is a value between 0 (unfragmented) and 1 (heavily fragmented). We show the test results in Figure 15. Compared to vanilla nested Linux/KVM, XGEMINI increases the fragmentation by 2% on average. XGEMINI incurs negligible extra memory fragmentation because it reuses the booking timeout adjustment algorithm, as described in Algorithm 1 in GEMINI [20]. The main idea of the algorithm is to reduce memory fragmentation without decreasing the effectiveness of forming well-aligned huge pages in virtualization environments.

### 5.3 Applicability and Overhead

To evaluate XGEMINI's applicability, we co-locate two nested virtual machines (VMs) on the same server. We want to test XGEMINI's performance when multiple nested VMs are co-located together, as VM co-location is pervasive in clouds. We show our test environment in Table 1.

Figure 14 shows throughputs of colocated workloads when they are tested with different systems. XGEMINI outperforms vanilla nested Linux/KVM and GEMINI by 7.6% and 12.2% on average, respectively. This shows XGEMINI can improve application performance when multiple nested VMs are collocated on the same server. Surprisingly, vanilla nested Linux/KVM performs better than GEMINI when nested VMs are colocated together. The main reason might be that resource contention increases alignment efforts and associate overhead in GEMINI. This also shows XGEMINI performs better than GEMINI as XGEMINI can efficiently avoid overhead by simplifying the alignment in nested virtualization environments.

To evaluate XGEMINI's overhead, we tested the throughputs of two TLB non-intensive workloads, i.e., Xapian and Ferret, when they are executed with vanilla nested Linux/KVM and XGEMINI, respectively. We show the test results in Figure 16. When workloads are TLB non-intensive, XGEMINI cannot provide performance benefits by reducing TLB misses. Interestingly, we find XGEMINI can still improve their performance (4.0% on average) as it can reduce the shadow paging overhead. In all our evaluations, we didn't observe XGEMINI incurs high overhead.

## 6 Related Work

**Huge Pages.** Many research proposals focus on optimizing huge page mechanisms to reduce address translation overhead. Ingens [1] addresses several issues of Linux THP. HawkEye [29] further optimizes Ingens. Illuminator [30] proposes to manage movable, unmovable, and hybrid memory regions separately to address memory fragmentation. Navarro et al. [2] propose to

control memory fragmentation with several huge page optimizations. Zhu et al. [31] propose Quicksilver to optimize memory bloat and fragmentation problems. Temeraire [32] aggressively allocates 2MB and 1GB huge pages based on application memory allocation patterns. Perforated page [33] enables huge pages for fragmented physical memory by allowing holes in huge pages and providing alternative mappings for the holes. Gemini [20] forms well-aligned huge pages between guest and host to improve TLB efficiency for non-nested virtualization environments.

**Hardware Approaches.** Some prior works reduce address translation overhead by prefetching [19] or caching [22], [34], [35] translation entries. ASAP [19] prefetches page table entries through the mappings formed between virtual addresses and page table entries. PTEMagnet [36] preserves CPU cache locality for page table entries by reserving contiguous guest physical memory space for page table entries. POM-TLB [34] proposes to use part of DRAM space as a very large level-3 TLB to reduce address translation overhead. Midgard [22] proposes a new virtual cache mechanism that maps the virtual address to a single intermediate Midgard address space in the system. Barret al. [35] study different designs of MMU caches and conclude that the most effective one is the translation cache (e.g., page walk caches). Hashed page tables [37] challenge this conclusion and propose to use the hashing scheme to shorten the page walk latency.

Some other approaches increase TLB reach by merging multiple TLB entries into one [38] or storing application data on contiguous physical memory [24], [39], [40], [41]. As TLB capacity does not increase at the same rate as DRAM capacity, these approaches may not be scalable. Today's big memory workloads still result in frequent TLB misses. RMM [41] enables ranges of an arbitrary number of virtually and physically contiguous pages to increase TLB reach. TLB Coalescing [38] increases TLB efficiency by merging multiple TLB entries into one. Gandhi et al. [39] propose to apply direct segment [40] in virtualized systems. It requires large contiguous physical memory space to store the application's entire data set. CA-paging [24] mitigates the address translation overhead through software and hardware collaboration.

Other works on improving the translation leverage hashed page tables [37], [42], flattened page tables [43], [44] , or combined nested and shadow page table [17], [18] to accelerate address translation. FPT [43] merging adjacent page table layers. Agile Paging [17] combines the advantages of nested paging and shadow paging to speed up address translation. SHSP [18] proposes to use nested paging and shadow paging for different workloads. Flat nested page table [44] flattens the nested page table. Mosaic pages [42] verifies the feasibility of the Iceberg hashing [45].

# 7 Conclusion and Future Work

Nested virtualization becomes increasingly important in today's clouds, as it can be used to improve application and system reliability and security and many others. However, due to the huge page misalignment issue, using huge pages becomes ineffective in reducing TLB misses in nested virtualization environments. This reduces the performance of memory intensive applications, such as scientific computing programs, and hampers the adoption of nested virtualization in modern clouds. This work proposes XGEMINI as an effective system solution to address the problem in the nested virtualization environments. To realize XGEMINI, we address several technical challenges such as how to form well-aligned huge pages without modifying the guest hypervisor. Our evaluations confirm that XGEMINI can greatly reduce TLB misses and improve application performance through forming more well-aligned huge pages. In future work, we plan to test XGEMINI on ARM servers. As ARM servers have become popular in clouds, memory-intensive workloads running in VMs on these servers may also suffer from TLB ineffectiveness and application performance degradation.

# 8 Acknowledgments

We sincerely thank the anonymous reviewers for their insightful suggestions. We are equally grateful to Zhaoxi Shi for his help in the manuscript revision. The work of Weiwei Jia was supported in part by NSF grant CRII-SHF-2348066. The work of Jianchen Shan was supported in part by NSF grant CNS-2324923.

# References

[1] 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)*, 2016.

[2] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," *ACM SIGOPS Operating Systems Review*, 2002.

[3] "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.

[4] H. Huang, J. Lai, J. Rao, H. Lu, W. Hou, H. Su, Q. Xu, J. Zhong, J. Zeng, X. Wang *et al.*, "Pvm: Efficient shadow paging for deploying secure containers in cloud-native environment," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 515–530.

[5] 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 *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[6] 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 (ASPLOS)*, 2020.

[7] 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 (SOSP)*, 2011.

[8] 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 (ASPLOS)*, 2020. [Online]. Available: https://doi.org/10.1145/3373376.3378467

[9] X. Shang, W. Jia, J. Shan, X. Ding, and C. Borcea, "Reestablishing page placement mechanisms for nested virtualization," *IEEE Transactions on Cloud Computing*, 2023.

[10] J. Nakajima, "Making nested virtualization real by using hardware virtualization features," https://events.static.linuxfound.org/sites/events/files/cojp13_nakajima.pdf.

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

[12] 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)*, 2020.

[13] 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.

[14] Microsoft, "Virtualization-based Security (VBS)," https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.

[15] MicroSoft, "Frequently Asked Questions about Windows Subsystem for Linux," https://learn.microsoft.com/en-us/windows/wsl/faq.

[16] J. M. Thara Gopinath, Mickaël Salaün, "Hypervisor-Enforced Kernel Integrity (Heki)," https://lpc.events/event/17/contributions/1515/attachments/1353/2717/LPC_2023_LVBS.pdf.

[17] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile Paging: Exceeding the Best of Nested and Shadow Paging," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[18] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li, "Selective Hardware/Software Memory Virtualization," *ACM SIGPLAN Notices*, 2011.

[19] 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)*, 2019.

[20] 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.

[21] "Xen does not support huge pages in shadow page tables," https://wiki.xenproject.org/wiki/Huge_Page_Support.

[22] 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)*, 2021.

[23] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation ranger: operating system support for contiguity-aware tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[24] 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)*, 2020.

[25] X. Shang, W. Jia, J. Shan, and X. Ding, "Coplace: Effectively mitigating cache conflicts in modern clouds," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021.

[26] H. Kasture and D. Sanchez, "Tailbench: A benchmark suite and evaluation methodology for latency-critical applications," in *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016.

[27] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[28] J. Araujo, R. Matos, P. Maciel, R. Matias, and I. Beicker, "Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure," in *Proceedings of the Middleware 2011 Industry Track Workshop*, 2011.

[29] 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)*, 2019.

[30] 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)*, 2018.

[31] 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.

[32] 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)*, 2021.

[33] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh, "Perforated Page: Supporting Fragmented Memory Allocation for Large Pages," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[34] 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*, 2017.

[35] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," *ACM SIGARCH Computer Architecture News*, 2010.

[36] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[37] I. Yaniv and D. Tsafrir, "Hash, Don't Cache (the Page Table)," *ACM SIGMETRICS Performance Evaluation Review*, 2016.

[38] 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)*, 2017.

[39] 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)*, 2014.

[40] 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*, 2013.

[41] 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*, 2015.

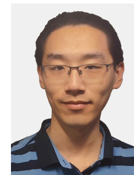
[42] K. Gosakan, J. Han, W. Kuszmaul, I. N. Mubarek, N. Mukherjee, K. Sriram, G. Tagliavini, E. West, M. A. Bender, A. Bhattacharjee, A. Conway, M. Farach-Colton, J. Gandhi, R. Johnson, S. Kannan, and D. E. Porter, "Mosaic Pages: Big TLB Reach with Small Pages," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[43] 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)*, 2022.

[44] 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)*, 2012.

[45] M. A. Bender, A. Conway, M. Farach-Colton, W. Kuszmaul, and G. Tagliavini, "All-Purpose Hashing," *arXiv preprint arXiv:2109.04548*, 2021.

**Weiwei Jia** is an Assistant Professor in the Department of Electrical, Computer, and Biomedical Engineering at The University of Rhode Island. He received his Ph.D. degree in Computer Science from New Jersey Institute of Technology. His research interests lie in the areas of computer systems, including operating systems, edge and cloud computing, virtualization, memory and storage systems, and systems architecture.



**Jiyuan Zhang** received a BS degree in computer science from the New Jersey Institute of Technology in Newark, New Jersey in 2022. He is currently pursuing a master's degree at the University of Illinois at Urbana-Champaign. His research interests include computer systems, memory and storage system, and computer architecture. He is a student member of IEEE and IEEE Computer Society.

**Jianchen Shan** is an Assistant Professor in the Computer Science Department at Hofstra University. His research interests include Cloud Systems, Parallel and Distributed Systems, and Operating Systems. He received his Ph.D. degree in Computer Science from the New Jersey Institute of Technology.

**Xiaoning Ding** is an Associate Professor at the New Jersey Institute of Technology. His interests are in the area of experimental computer systems, such as distributed systems, virtualization, operating systems, and storage systems. He earned his Ph.D. degree in computer science and engineering from the Ohio State University.