Distributed Page Table: Harnessing Physical Memory as An Unbounded Hashed Page Table

Osang Kwon*§, Yongho Lee*§, Junhyeok Park*, Sungbin Jang*, Byungchul Tak†, and Seokin Hong*
*Department of Electrical and Computer Engineering, Sungkyunkwan University
†School of Computer Science and Engineering, Kyungpook National University
*{osang915, jhyn205, vzx00770, sunbi3361, seokin}@skku.edu †bctak@knu.ac.kr

***Abstract*—Virtual memory systems rely on the page table, a crucial component that maps virtual addresses to physical addresses (i.e., address translation). While the Radix Page Table (RPT) has traditionally been used for this task, its limitations have become more apparent with the rise of memory-intensive applications. Recently, Hashed Page Tables (HPTs) have been explored as an alternative page table structure to offer faster address translation. However, the HPT introduces its own set of challenges particularly in resizing the page table and allocating contiguous physical memory space for storing the table.**

To tackle the fundamental problem of the existing HPT designs, this paper introduces *Distributed Page Table (DPT)*, a novel approach that utilizes the physical memory as a huge hashed page table. DPT distributes Page Table Entries (PTEs) across the entire physical memory space, significantly reducing the hash collisions while avoiding the table resizing overheads. When distributing the PTEs across the physical memory, they can be mapped to memory locations already allocated to data pages. This new type of collision, referred to as address collision, may reduce the effectiveness of the DPT. This paper showcases that the DPT can effectively resolve the address collision with three simple yet efficient techniques: *Strided Open Addressing (SOA)*, *Collision-Aware Virtual Address Allocation (CVA)* and *Collided Page Displacement (CPD)*. Our experimental results demonstrate that DPT achieves average performance improvements of 12.6%, 11.6%, and 8.7% compared to traditional RPT, the latest large-coverage TLB design, and state-of-the-art HPTs, respectively.

***Index Terms*—Virtual Memory, Page Table, Hashed Page Table**

I. Introduction

Virtual memory is an essential abstraction in modern computer systems as it provides vital functionalities including memory virtualization and process isolation [1], [2], [4], [7], [26], [37], [51], [53], [64]. The page table, which is responsible for organizing address translations from virtual to physical memory, plays a central role in implementing the virtual memory. However, with the ever-increasing computational demands and the rise of memory-intensive applications, conventional page table structures are now facing fundamental limitations which require more advanced and efficient structures [28], [29], [35], [36], [55], [65].

For many years, the radix-tree structure has been the cornerstone of modern page tables [2], [27]. In the Radix tree-based Page Table (RPT), the Page Table Entries (PTEs) are organized in a hierarchical tree structure and are accessed through a sequential traversal process known as a page table walk. To enhance this process, various caching strategies have been introduced [6], [8], [9], [25]. However, the inherent structure of the radix tree often becomes a bottleneck, primarily because it requires sequential memory accesses in the address translation, especially for the irregular memory-intensive workloads [10], [32], [33], [52], [65].

Hashed Page Tables (HPTs) have been introduced as a promising alternative to the traditional RPTs [23], [24], [26], [30], [31], [55], [65]. In this approach, the virtual page number is hashed to quickly index the tables, simplifying the translation process to a single memory access. However, HPTs come with their own set of challenges. First, managing hash collisions effectively remains a significant hurdle, often leading to sequential memory lookups or the costly process of dynamically resizing the table [55], [56], [65]. Second, maintaining a single global HPT that contains PTEs for all active processes is impractical [17], [55], [59], [65]. While using a HPT for each process can address this issue, determining the appropriate size for each per-process HPT is still challenging [55], [56]. Third, the HPTs require contiguous physical memory space to store the page table, which can lead to significant inefficiencies in memory management. In the worst-case scenario, where the HPT is significantly large, the chances of finding a sufficiently large contiguous memory space would be minute.

Recent studies have made significant progress in addressing the challenges associated with HPTs. Skarlatos et al. [55] proposed Elastic Cuckoo Page Table (ECPT), which utilizes cuckoo hashing to reduce the overhead caused by hash collisions in HPTs. ECPT employs per-process HPTs that are dynamically resized according to occupancy. Although the ECPT efficiently handles the hash collisions by adjusting HPT sizes to meet application demands, it still requires contiguous memory space for HPT storage. To overcome this limitation, Stojkovic et al. [56] proposed an alternative approach that partitions the HPT into discontiguous chunks of physical memory and then uses a hash function to select a specific chunk. While this approach reduces the need for contiguous memory allocation, it still necessitates chunk resizing, potentially leading to frequent data migrations, especially when the initial chunk size is significantly smaller than the application's memory footprint.

The fundamental limitations of the HPTs primarily stem from their constrained size. Thus, a straightforward approach

§Both authors contributed equally to this work.

to overcome these limitations is to use a sufficiently large HPT so that all PTEs can be accommodated in the table without hash collisions. However, employing such a large HPT is impractical because it requires a contiguous physical memory space for allocation.

In this paper, we present ***Distributed Page Table (DPT)***, a novel page table structure designed to overcome the limitations of HPTs. DPT provides an illusion of a huge contiguous page table without requiring the allocation of contiguous physical memory space to hold the table. To achieve this, DPT utilizes the entire physical memory as a huge page table and distributes PTE pages (i.e., pages that hold PTEs) across the physical memory space by determining their locations (i.e., physical address) using a hash function. This approach eliminates the need for a large contiguous memory space and significantly reduces the frequency of hash collisions by treating the entire memory space as the hashing target.

Even though DPT can significantly reduce the hash collision between PTE pages, it may introduce a new type of collision. Since DPT determines the physical address of the PTE pages using a hash function, these pages might be mapped to physical memory locations already allocated to regular data pages. We refer to this situation as *address collision*. Address collisions can reduce the effectiveness of DPT. However, compared to hash collisions, they are much easier to resolve due to the greater flexibility in allocating the physical and virtual addresses for data pages.

To resolve address collisions, we introduce three efficient techniques: *Strided Open Addressing (SOA)*, *Collision-Aware Virtual Address Allocation (CVA)*, and *Collided Page Displacement (CPD)*. SOA sequentially searches for a free physical frame using a probing method designed for fast searches in a virtual memory system that supports multiple page sizes. CVA, on the other hand, searches for non-collision virtual memory regions instead of non-collision physical frames. While SOA and CVA help DPT avoid address collisions, shortages of free physical frames or non-collision virtual memory regions can still result in address collisions. To address this, CPD migrates collided data pages to free physical frames. Additionally, we introduce two techniques for further optimization: *Fragmentation-Aware PTE Allocation (FAP)* and *PTE Pooling (PTP)*. FAP aims to minimize memory fragmentation by allocating PTE pages within already fragmented memory areas, while PTP mitigates address collisions by preemptively allocating PTE pages in the contiguous virtual memory space of the Heap area. These techniques are orthogonal and can be combined to enhance system performance.

We evaluate DPT using a cycle-level multi-core simulator extended to support detailed address translation mechanisms, including page table structures and the page walk process. Experimental results demonstrate that DPT outperforms RPT and the state-of-the-art HPT. Compared to RPT, the latest large-coverage TLB design, and the state-of-the-art HPT, DPT achieves average performance improvements of 12.6%, 11.6%, and 8.7%, respectively.

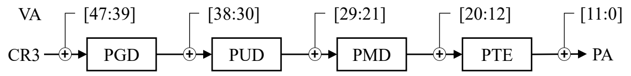


Fig. 1: Virtual to physical address translation for using a Radix Page Table in x86-64 architecture.

II. Background

A. Radix Page Table: Radical Factor of Costly Translation

In modern computer systems, a Radix Page Table (RPT) is commonly employed to manage virtual-to-physical address mapping. The RPT organizes these mappings using a multi-level tree data structure. When an address translation request misses the Translation Lookaside Buffer (TLB), the hardware performs a Page Table Walk (PTW) to find the required mapping information by traversing multiple levels of the in-memory page table.

Figure 1 illustrates an example of a PTW for a 4-level page table with a 4KB base page in the x86-64 architecture [2], [27]. The hardware page table walker retrieves the base address of the page table from the CR3 register and sequentially accesses each level (PGD, PUD, PMD, and PTE) of the page table. This sequential access entails significant performance overhead.

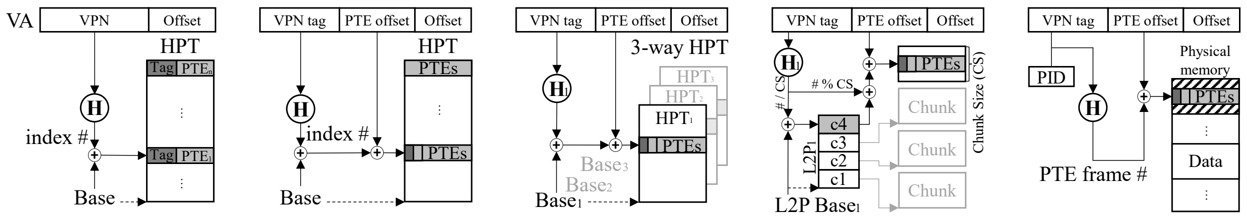
To reduce this overhead, a caching mechanism known as Page Walk Cache (PWC) [8] has been introduced. The PWC stores recently accessed entries of the intermediate-level page table (from PGD to PMD). However, the PWC is less effective for irregular memory-intensive workloads, especially in a system with a deeper page table (e.g., 5-level page table) [28], [48], [55].

B. Hashed Page Table: Hashing instead of Walking

Hashed Page Table (HPT) mitigates address translation overhead in the RPT by employing a hash function instead of the repetitive and serial page table walk [18], [22], [24], [26], [31], [55], [57], [65]. Figure 2a illustrates the address translation process in a conventional HPT system. The hash function (denoted as H) uses a Virtual Page Number (VPN) as input and generates a hashed value, serving as an index for the table. This approach enables address translation with a single memory access when no hash collisions occur. HPT has been employed in several real-world systems [23], [24], [26].

C. Hashing, Is It a Panacea?

Although HPT is useful in mitigating the address translation overhead, it has limitations due to its reliance on “hashing” instead of “walking”. Firstly, if a hash collision occurs, additional memory access is required to locate the necessary Page Table Entries (PTEs) using a collision handling scheme (such as chaining and open addressing [65]). Unfortunately, it is impossible to achieve zero collisions since the virtual address space is typically much larger than the physical address space. Secondly, the entry size of HPT is larger than that of RPT because the VPN is stored in the entry as a hash tag, as shown in Figure 2a. The larger PTE size increases the overall



(a) Conventional HPT (b) PTE clustering (c) ECPT (d) ME-HPT (e) **DPT (Our proposal)**

Fig. 2: Hashing-based page tables: (a) Conventional hashed page table (HPT) (b) PTE-clustered HPT, (c) Elastic cuckoo page table (ECPT) [55], (d) Memory-efficient HPT (ME-HPT) [56], and (e) Distributed Page Table (DPT). Conventional HPT, ECPT, and ME-HPT allocate and manage page tables within single or multiple continuous memory spaces. In contrast, DPT can allocate PTE pages anywhere in physical memory using hashing, eliminating the need for contiguous memory spaces.

page table size and reduces the entry density in a cache line, leading to lower hit rates for the PTEs in cache hierarchy [65]. Finally, the hashing-based indexing scheme reduces the spatial locality of PTEs in memory. Since the hash function uses the VPN as a hash key (i.e., hash input), PTEs of contiguous virtual addresses are scattered across the HPT. This low spatial locality leads to inefficient fetching of PTEs from the in-memory page table, particularly for workloads with high page-access locality [6].

D. Prior Work - Advancements in Hashing Approaches

Yaniv et al. [65] proposed a PTE clustering scheme to improve the spatial locality of the PTEs by storing consecutive PTEs that share the same VPN tag in a hash slot, as illustrated in Figure 2b. Additionally, the VPN tag is embedded into unused bits of the PTE to reduce the HPT entry size, allowing the hash slot to fit within a single cache line.

Skarlatos et al. proposed the Elastic Cuckoo Page Table (ECPT) [55], adopting cuckoo hashing [43] to handle the hash collisions in HPT. ECPT utilizes an n-way set-associative structure with a distinct hash function for each way, as illustrated in Figure 2c. When allocating PTEs, ECPT chooses a specific way (i.e., page table) and calculates an index using the associated hash function. If the target index in the selected page-table way is already occupied, the new PTE displaces an existing entry in that way. The displaced entry is then reassigned to an alternate page-table way. This displacement and reassignment process continues iteratively until all PTEs are placed in the page table without causing further evictions.

While ECPT enables fast table lookups by simultaneously probing all ways, it necessitates the allocation of multiple page-table ways in contiguous physical memory regions. Such allocations, often requiring multiple attempts, may result in failures in highly fragmented systems [56].

To address the need for contiguous memory regions in ECPT, Stojkovic et al. proposed the Memory-Efficient Hashed Page Table (ME-HPT) [56]. In ME-HPT, a page table consists of multiple small chunks (typically 8KB) whose base addresses are maintained in a dedicated table called L2P table, as illustrated in Figure 2d. ME-HPT can expand the page table size with low overhead by simply allocating additional chunks. However, if all entries in the L2P table are occupied, the chunk size must be enlarged, resulting in significant overhead due to costly data migrations.

A recent study utilizes the hashing to develop a TLB compression scheme called Mosaic, which aims to expand TLB coverage [21]. The Mosaic scheme employs Compressed Physical Frame Numbers (CPFNs) in each TLB entry to hold more address translation information. It introduces a hash-based method to convert CPFNs to their physical addresses.

III. Motivation

A. Key Challenge of Prior HPTs: PTE Migrations

Recent studies have proposed new forms of HPT designs to handle hash collisions better and to enable efficient page table resizing. However, these designs may result in frequent PTE migrations. There are two main reasons for this.

First, new HPT designs use cuckoo hashing, which relocates a PTE from a collided entry to another table (i.e., way). If yet another collision occurs, the PTE needs to be migrated again. Second, dynamic resizing of the page table, a key operation of HPT designs, causes PTEs to migrate from the old table to a new one. For instance, ME-HPT starts with small chunks (e.g., 8KB) and increases their size as needed. During resizing, PTEs from multiple small chunks are consolidated into a larger one, which can lead to repeated migrations. This issue becomes more prominent if the initial chunk size is significantly smaller than the application's memory footprint in which case the resizing will be more frequent. Although ME-HPT uses in-place resizing to minimize the cost of migrations, the rehashing process can still cause cache pollution by loading unnecessary data [54], [62], [63].

B. Performance Impact of PTE Migrations

Kernel Perspective: We measure the PTE migrations per page allocation (PMPA) in the ME-HPT using our simulation environment. As shown in Figure 3 (left), the PMPA values for GRAPH, GEN, RND, and XS workloads are 7.5, 7.7, 2.8, and 13.9, respectively. To assess the performance impact of the PTE migrations, we emulate them on a real Linux system by conducting the migration at the measured PMPA rates during the memory allocation process (See Section VIII-A for detailed experimental methodology).

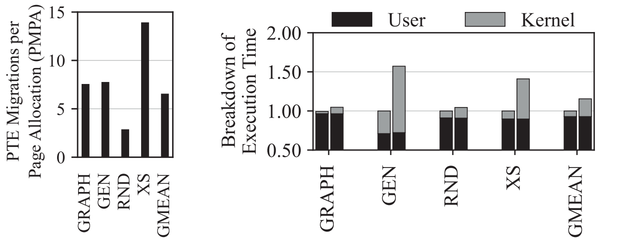


Fig. 3: Frequency of PTE migrations (left) and their performance impact (right) in a conventional system. The workloads used in this experiment are detailed in Table II; 'GRAPH' represents the average of experimental results across all GraphBIG workloads.

Figure 3 (right) demonstrates the effect of PMPA by comparing the execution time of workloads in the baseline system (left bar) with that in the system using the ME-HPT (right bar). As shown in the figure, PTE migrations lead to execution time increases of 5.3%, 57.2%, 4.4%, and 41% for the GRAPH, GEN, RND, and XS workloads, respectively (15.5% on average). The performance impact is particularly severe for workloads such as GEN and XS, where kernel time significantly contributes to the total execution time.

This performance degradation stems from two key issues during the PTE migrations. First, user threads requesting page allocation experience delays because they must wait for the migration to complete [60]. While page allocation can be performed in parallel with the execution of other threads, the delay in the execution of requesting user threads is unavoidable. Second, the page allocation time is extended due to the PTE migrations, intensifying the contention among threads accessing and updating the page table. Page allocation is managed through the page fault handling process, which consumes thousands of CPU cycles. During this process, a locking mechanism is essential to prevent other threads from accessing the page table while it is being updated. When multiple threads generate page faults simultaneously, contention for the lock can lead to severe performance degradation. This problem is exacerbated when a thread holds the lock for an extended period during page allocation [3], [15].

Coherency Perspective: ME-HPT employs a metadata cache called Cuckoo Walk Cache (CWC) to hold the PTE metadata necessary for calculating the physical address of PTEs. As PTE metadata is updated due to migrations, maintaining coherence between the CWCs of different processor cores becomes challenging. In particular, frequent changes in PTE locations due to cuckoo hashing exacerbate this issue, leading to potential conflicts where different CWCs may contain inconsistent information.

Two main approaches can be used to maintain coherence between CWCs. First, a costly CWC shootdown technique, similar to the TLB shootdown, may be necessary. Second, error-handling techniques can address inconsistencies when accessing PTEs from CWCs. ME-HPT uses the second approach to manage this challenge. It initially reads the PTE from the physical address calculated using metadata from the CWC. If the read PTE is incorrect, ME-HPT recalculates the actual physical address and re-accesses the memory to retrieve the correct PTE. This method allows ME-HPT to avoid the costly CWC shootdown process.

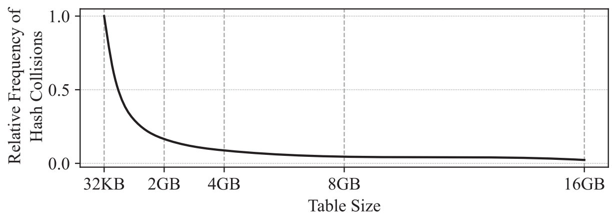


Fig. 4: Frequency of hash collisions for various HPT sizes (normalized to a 32KB HPT).

The primary goal of HPT is to enable the retrieval of PTEs with a single memory access. However, resolving inconsistencies in CWCs often requires two sequential memory accesses, which offsets the benefits of CWCs even when hit rates are high. Additionally, since CWCs are core-specific, a single PTE migration can trigger the error-handling process across multiple cores, leading to significant performance degradation. Given that computing systems are expected to scale up to as many as 640-core CPUs by 2037, as projected by the International Roadmap for Devices and Systems (IRDS) [13], the impact of metadata consistency issues in HPT designs is likely to become more severe. This suggests the need for further optimizations or alternative approaches to maintain performance efficiency in massively parallel systems.

C. Reducing Hash Collisions with Large HPT: A Simple Yet Challenging Approach

Effectively handling hash collisions is crucial for HPTs as they rely on hash functions. While several methods exist for mitigating hash collisions, avoiding them entirely remains to be challenging. Collision resolution techniques often initiate the page table resizing when the overhead of collision handling becomes excessive due to too frequent collisions. ME-HPT addresses collisions through PTE migration [56], but frequent collisions lead to many migrations inducing high overheads.

We conduct an experiment to observe the effect of page table size on hash collisions. In this experiment, we allocate the page table in a contiguous memory space and measure the frequency of hash collisions during data access to 100 million random addresses. Figure 4 shows a rapid decrease of hash collision counts as table sizes increase from 32KB table size although the gain diminishes at the size beyond 4GB. The number of hash collisions at 16GB table size is about 2.3% of the hash collisions at 32KB table size.

This experimental result demonstrates that larger-sized HPTs effectively manage hash collisions by significantly reducing their frequency. Despite these promising results, the challenge remains in securing contiguous memory space for these large tables (e.g., 4GB). *Therefore, we aim to develop an HPT structure capable of handling substantial sizes without requiring contiguous memory allocation.*

IV. Distributed Page Table

In this section, we introduce a novel page table structure called a ***Distributed Page Table (DPT)*** that can overcome the fundamental limitation of prior HPT designs. The **key idea** of DPT is to distribute page table entries across the entire memory space. Instead of allocating small contiguous regions of the physical memory to store the page table, DPT uses the entire physical memory space as a huge HPT. Prior HPTs use the hash value as a table index. In contrast, our DPT uses the hash value as the physical memory address of the PTEs, as illustrated in Figure 2e. This approach eliminates the need to allocate the page table in a contiguous space. This section describes how the DPT allocates the page table entries (PTEs) and uses them for the address translation.

A. Address Translation Flow

In conventional HPT, the virtual page number (VPN) is used as a hash key to determine the table index for the virtual address (VA). However, this does not consider the reference locality of the PTEs. To tackle this, a prior study [65] proposed a locality-aware addressing scheme that divides a VA into three fields: VPN tag (33 bits), PTE offset (3 bits), and page offset (12 bits). In this way, PTEs for eight contiguous virtual pages are stored in a cache line, increasing the locality of the PTEs.

DPT uses a similar approach to cluster contiguous PTEs in the address translation process. However, unlike conventional PTE clustering, DPT clusters 512 PTEs for a 2MB virtual memory region and stores them in a 4KB page. This method is similar to the typical RPT, which stores 512 PTEs in a PTE page that is a leaf node of the radix tree. To this end, DPT uses 9 bits of a virtual address for the PTE offset.

Figure 5 illustrates how our DPT generates a physical PTE address for a given VA. It computes the physical frame number (PFN) of the PTE page using a hash function that takes a VPN tag and a process ID (PID) as inputs. The PID is used as a unique seed for the hash function since the VPN is not unique across processes. To specify the location of a PTE within the 4KB PTE page, a 9-bit PTE offset is appended to the PFN.

B. Allocating Page Table Entries

Figure 6 illustrates the DPT's PTE allocation process. Initially, DPT checks a frame bitmap to determine if the target physical frame for the hashed PFN is free (❶). The bitmap indicates whether each physical frame (4KB in size) is free or allocated to store PTEs. Each bitmap entry contains a Free bit and a PTE flag bit (Free, PTE). An entry of (1, 0) indicates the frame is free, while (0, 0) or (0, 1) indicates that it is

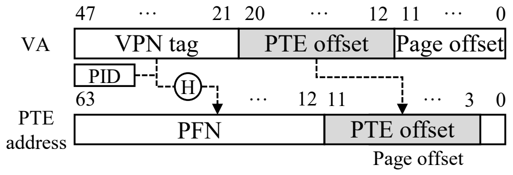


Fig. 5: PTE address generation of DPT.

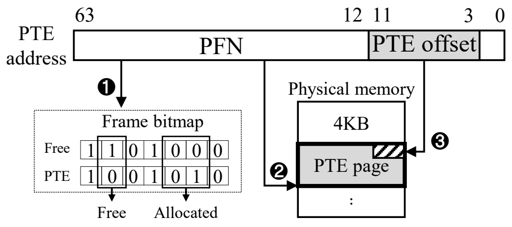


Fig. 6: PTE allocation of DPT.

allocated. If the target frame is free, DPT assigns it to a PTE page for the given VA (❷). Then, DPT uses the PTE offset to place the PTE at a specific location within the page (❸). If another PTE page or data page already occupies the target frame, DPT employs collision resolution techniques, which will be described in the following section.

V. Resolving Address Collision

DPT expands the range of hash functions by utilizing the entire physical memory space, significantly reducing the likelihood of hash collisions. However, as memory usage increases, there is a possibility that DPT inadvertently assigns a PTE page to a physical address already occupied by another page (either a PTE page or a regular data page). We refer to this situation as an *address collision*. This section introduces three novel techniques to resolve address collisions: Strided Open Addressing (SOA), Collision-Aware Virtual Address Allocation (CVA), and Collided Page Displacement (CPD). These techniques are orthogonal and can be combined to enhance overall system performance. During runtime, DPT applies these three techniques sequentially in the order of SOA, CVA, and CPD.

A. Strided Open Addressing

Motivated by the hash table's open-addressing method, we propose the *Strided Open Addressing (SOA)* technique that sequentially seeks a free physical frame with a *strided searching* method. Modern operating systems support multiple page sizes: usually 4KB (base page), 2MB (large page), and 1GB (huge page). Therefore, a collided address can be located on a base, large, or huge page. By taking into account the multiple page size support, the strided searching method generates a sequence of PFNs with two parameters (*stride* and *step*) along with a hash function to find a free physical frame quickly.

In this study, we use three *stride* values (1, 512, and 256K) with the assumption that the operating system supports the above three page sizes. When a collided address is located on a base page, the strided search uses a *stride* of 1. Conversely, it uses 512 and 256K *stride* for the collision on a large page and a huge page, respectively. The *step* parameter can reach MAX_SOA_STEP, which is a design parameter.

When a PFN calculated with a hash function is already allocated, SOA initially determines the page size allocated to that PFN by checking the free bits of the frame bitmap. Then, it calculates the next PFN to be probed by using the equation 1 with an appropriate *stride* and a *step* of 1.

$$PFN_{PTE} = Hash(VPN\ tag) + Stride \times Step \quad (1)$$

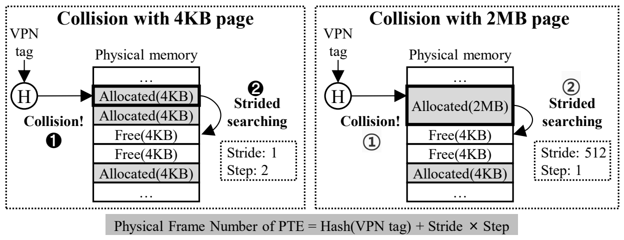


Fig. 7: Strided open addressing for a collision with 4KB page (left) and for a collision with 2MB (right).

If the next PFN is also already allocated, the *step* value is incremented by 1, and the PFN is recalculated. If SOA fails to find a free physical frame with the current *stride* until the *step* value reaches MAX_SOA_STEP, it then uses the next larger *stride* value. However, the SOA could fail even after a full search. Then, the DPT uses another collision resolution method, which will be described in the next subsections.

Figure 7 illustrates how SOA resolves the address collision for two scenarios. In the first scenario (left), an address collision occurs when the PFN calculated with a hash function for a given VPN tag is already allocated to a 4KB base page (❶). To resolve this collision, SOA searches for a free frame in the subsequent frames. In this example scenario, SOA finds a free frame in two steps (❷). Therefore, the values for the two SOA parameters (*stride* and *step*) are set to 1 and 2, respectively, in a *metadata table* (will be described in the section VII), for the corresponding VA region. These parameters are then used for PFN calculation when accessing the VA region.

In the second scenario illustrated in Figure 7 (right), the collided physical frame is already allocated to a 2MB large page (①). Therefore, with a short *stride* (e.g., a *stride* of 1), it may be difficult to find a free frame within a limited number of steps. In such cases, SOA uses a *stride* of 512 to search for free frames beyond the 2MB range. In the example shown in Figure 7, SOA successfully finds a free frame by using a *stride* of 512 and a step of 1 (②).

The strided probing method can be easily implemented by checking the free bits in the frame bitmap. The search function begins at a specific free bit indexed by a hash function for a given VA. During the strided search, subsequent bits are checked within a search range (i.e., stride x MAX_SOA_STEP) until a free frame is found. If no free frame is detected, it uses a larger stride (i.e., 512 or 256K) and continues searching.

B. Collision-Aware Virtual Address Allocation

Although SOA can effectively resolve address collisions by searching for an available physical frame, its capability is limited to the range of the search window defined by its parameters (*stride* and *step*). Since these parameters are stored in memory for each virtual memory region and loaded during address translation, using a large number of bits for these parameters is impractical due to hardware overheads.

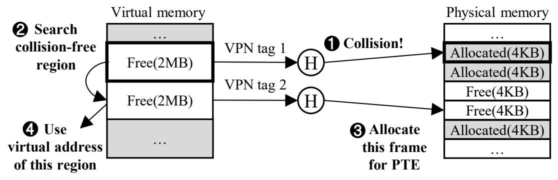


Fig. 8: Collision-aware virtual address allocation.

As a hardware-efficient approach, we propose the *Collision-Aware Virtual Address Allocation (CVA)* technique. Rather than selecting a non-collision PFN for a given VA, CVA chooses a non-collision VA to resolve the address collisions. Fortunately, the operating system has significant flexibility in choosing a virtual memory region for allocation. Moreover, the virtual address space is often much larger than the physical memory, providing the operating system with a wide range of options for allocating a virtual memory region in the memory allocation process.

CVA leverages the flexibility in selecting a virtual memory region during the memory allocation. Since DPT allocates a single PTE page for every 2MB virtual memory region as described in Section IV-A, CVA searches 2MB regions where the collisions do not occur. Figure 8 illustrates how CVA finds a collision-free virtual memory region. When a memory allocation request is made, DPT identifies an available virtual memory region and computes its PFN using the region's VA. As shown in the figure, the PFN derived from VPN tag 1 leads to an address collision (❶), as that PFN is already in use. Consequently, CVA recalculates the PFN for another available virtual memory region (❷) and then checks potential collisions with the PFN. Since the physical frame corresponding to this new PFN is free, it is assigned to store the PTE page for the virtual memory region (❸). Finally, the memory allocator provides the application with the virtual address of this newly allocated region to fulfill the memory allocation request (❹).

As described above, CVA searches the collision-free region in a 2MB unit. Therefore, finding a collision-free region larger than 2MB can be challenging because all 2MB chunks of the large region must be free of collisions. Due to this constraint, CVA limits the memory allocation size to MAX_CVA_SIZE, which serves as the upper boundary on the memory size that can be allocated using CVA. Additionally, CVA limits the maximum number of searches to MAX_CVA_CNT. These two parameters ensure the memory allocation process remains efficient and does not become a performance bottleneck due to extensive searching for collision-free regions.

In demand paging, a PFN can be lazily assigned to an allocated VA only when the VA is accessed [20], potentially limiting the applicability of CVA. This limitation can be addressed by proactively pre-allocating PFNs for allocated VA or by resolving collisions on the lazily assigned PFNs using other collision-resolving techniques (SOA and CPD).

C. Collided Page Displacement

The use of SOA and CVA techniques can help DPT avoid address collisions. However, even with these techniques,

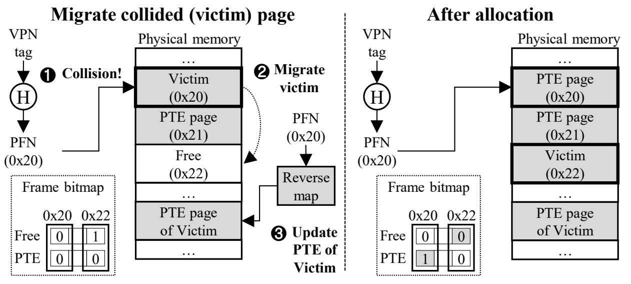


Fig. 9: Collided page displacement.

DPT may still encounter address collisions, particularly when physical memory usage is high and there are insufficient free physical frames (or available non-collision VA regions).

To address this challenge, we propose Collided Page Displacement (CPD) that migrates a data page already allocated to the collided physical frame to a new frame. Figure 9 shows how CPD resolves the address collision. CPD selects the page residing in the collided page as a victim. If the victim page is a data page (❶), CPD migrates it to a free frame (❷).

When a page is migrated to a different physical frame, the victim page's PTE must be updated to accurately redirect the VA associated with the victim to a new physical frame. To this end, CPD leverages the reverse map, a mechanism frequently used in modern operating systems for efficiently handling page swapping, migration, and compaction [11], [40], [46]. The reverse map maintains the physical address for the PTE corresponding to each physical frame, allowing us to directly obtain the physical address of the victim page's PTE (❸).

CPD does not migrate the PTE pages because the page tables have unmovable characteristics that ensure stability in modern operating systems [44], [66]. Additionally, DPT cannot migrate specific pages marked as non-migratable by I/O drivers or kernel components [20]. To resolve the address collisions on non-migratable or PTE pages, CPD collaborates with SOA to identify potential victim pages among migratable ones. With SOA, the PFN is calculated with the equation 1, allowing CPD to select a suitable victim from a set of candidates reachable by SOA.

VI. Additional Optimizations

A. *Fragmentation-Aware PTE Allocation*

External fragmentation occurs when there is sufficient total free memory, but the non-contiguous arrangement of free blocks prevents the allocation of larger continuous spaces. With DPT, this fragmentation can worsen because it distributes PTEs across the entire physical memory. Since the operating system prevents the migration of PTE pages, combining small free memory blocks into a large contiguous block during memory compaction [14] becomes challenging [47], [66].

To minimize external fragmentation caused by PTE allocations, DPT adopts a Fragmentation-Aware PTE Allocation (FAP). When allocating a PTE page, FAP opportunistically selects a physical frame within a fragmented 2MB region already broken into non-contiguous 4KB pages. To this end,

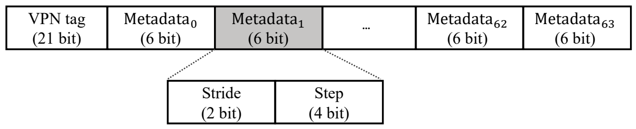


Fig. 10: DPT metadata table entry.

FAP uses the frame bitmap to check if a free 4KB physical frame, which is selected with a hash function, resides within a fragmented 2MB region. On the address collision, FAP allocates one of the free physical frames selected by the collision resolution techniques (i.e., SOA, CVA, or CPD).

B. *PTE Pooling*

DPT can employ the PTE Pooling (PTP) to reduce address collisions by pre-allocating multiple PTE pages in physical memory. During application initialization, PTP creates a reserved region by preemptively allocating PTE pages for the contiguous virtual memory space of the heap area. At this stage, physical frames are not allocated to each PTE to avoid unnecessary memory usage. Thus, both the *PFN* and *Present* fields of these PTEs are set to zero, indicating that physical frames have not yet been allocated. When a memory allocation request is made, the virtual address space in this reserved region is dynamically allocated to the application at runtime. This approach ensures efficient memory usage and reduces the likelihood of address collisions.

VII. Implementation

In this section, we describe a new data structure required to implement DPT and minor modifications to the Memory Management Unit (MMU). We also describe how the DPT can be implemented in the Linux kernel.

A. *DPT Metadata Table*

During the address translation, DPT needs to obtain the SOA parameters (*stride* and *step*) to quickly calculate the PTE page address for a requested virtual address. To achieve this, DPT stores these parameters in a *DPT Metadata Table (DMT)* when allocating the PTE pages. As illustrated in Figure 10, each entry in the DMT contains multiple metadata, each of which comprises *stride* and *step* for a 2MB virtual memory region. The DMT is indexed using a hash function combined with an open-addressing method, which requires including a VPN tag in each entry.

The metadata size (i.e., the number of bits used for *stride* and *step*) is a design parameter. Figure 10 shows a DMT entry comprising 64 metadata (each with a 2-bit stride and a 4-bit step) and a 21-bit VPN tag. The validity of metadata is indicated using an unused stride value in the 2-bit stride field. In the current design, DPT supports only three strides (i.e., 2'b00: stride of 1, 2'b01: stride of 512, and 2'b10: stride of 256K). Consequently, the unused stride (i.e., 2'b11: invalid) can be used to indicate the metadata invalidity.

The DMT is a process-specific structure allocated in a designated memory region for each process. The base address of each DMT is accessed through a specific processor register,

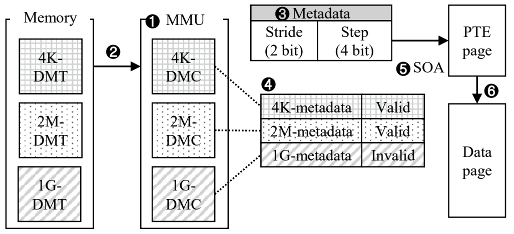


Fig. 11: Address translation of DPT with DMC and DMT.

enabling the Memory Management Unit (MMU) to locate the DMT for address translations. During a context switch, the operating system updates this register with the base address of the new active process's DMT.

The DMT is highly efficient in managing address translations due to its compact structure, where each entry can cover a substantial portion of physical memory. A single DMT entry containing 64 metadata can cover up to 128MB of physical memory space. In our default setup, the DMT is initialized with 8192 entries, with a total size of 512KB. Even with this small size, the DMT can cover 1TB of physical memory space.

B. *DPT Metadata Cache in MMU*

The DPT metadata can be stored in the cache hierarchy to enhance address translation. However, a cache miss on the metadata requires additional off-chip memory access, which can be particularly challenging for irregular and memory-intensive workloads. To decouple the DPT from cache behavior, DPT employs a DPT Metadata Cache (DMC) in the MMU. In the current system with RPT, the MMU employs Page Walk Caches (PWCs) [6], [39], [48]. We repurpose the PWC as a DMC to store frequently accessed DMT entries.

The DMC and PWC both employ a set-associative cache structure, but they differ significantly in their operational focus and coverage. The main distinction between the DMC and PWC lies in their address space coverage. The DMC can handle approximately eight times the memory space of the PWC with the same cache size. While the PWC is designed to store physical addresses of the next-level page table, the DMC is designed to store a set of DMT metadata. Since the DMT metadata consists of small bits, DMC can store multiple metadata, each covering a contiguous 2MB memory region, within a single DMT entry. This capability provides high efficiency and scalability for managing larger memory spaces with the DMC.

C. *Supporting Multiple Page Sizes*

To support multiple page sizes (4KB, 2MB, and 1GB), DPT employs the corresponding DMT and the DMC for each page size as shown in Figure 11. The DMTs are referred to as 4K-DMT, 2M-DMT, and 1G-DMT, corresponding to the 4KB, 2MB, and 1GB pages, respectively. Similarly, the DMCs are named 4K-DMC, 2M-DMC, and 1G-DMC. For instance, a 4K-DMT stores metadata for calculating PTE page addresses for a 4KB page, while a 2M-DMT stores metadata for calculating PTE page addresses for a 2MB page.

As shown in Figure 11, for translating a requested virtual address (VA), DPT accesses all DMCs for different page sizes simultaneously (❶). Each DMC is indexed differently based on its page size. For example, to access the 4K-DMC, the upper 24 bits of the VA are used to calculate the set and tag bits. The indexing method for DMCs corresponding to larger page sizes follows a similar approach, except for excluding an additional 9 bits from the VA. If a translation request results in a miss in the DMCs, the MMU reads the necessary metadata from a DMT stored in the cache hierarchy (❷).

With the three metadata read from the DMCs (❸), the page size for a requested VA is determined. For example, as shown in Figure 11, both 4K-metadata (i.e., metadata for 4K page) and 2M-metadata (i.e., metadata for 2MB page) are valid (❹), indicating that metadata for both 4KB and 2MB pages exists for the VA. In this case, since the 2MB virtual memory space covered by the 2M-metadata already encompasses 512 4KB pages covered by the 4K-metadata, only the 4K-metadata is accessed to check for the presence of the page table entry (PTE). If all metadata is invalid, DPT allocates PTE pages (using SOA, CVA, and CPD in the event of an address collision) and updates the DPT metadata (❺). Once a PTE page is allocated, it is used to store the translation information for the data page (❻).

D. *Compatibility with Linux Memory Management Subsystem*

To integrate DPT seamlessly with the Linux kernel, we ensure compatibility with the Linux memory zone framework. Memory zones in Linux are used to handle memory allocation across different types of memory (e.g., DMA, normal, and high memory). Both the conventional RPT and our proposed DPT implementation use ZONE_NORMAL so that PTEs are allocated within the appropriate memory zone. This compatibility is essential for maintaining system stability and efficiency of memory allocations across different zones. By aligning DPT's memory allocation strategies with ZONE_NORMAL boundaries, we can avoid potential conflicts within the conventional Linux memory management infrastructure.

E. *DPT Implementation in Linux Kernel*

In this section, we describe an implementation example of DPT on Linux (version 5.11.6) to demonstrate the feasibility of DPT in a real operating system.

Frame Bitmap: The DPT system uses a frame bitmap to check physical frame usage and the presence of the PTE page for efficient memory management. When implementing DPT in Linux, we can leverage existing kernel functions rather than using a separate data structure (i.e., frame bitmap). For example, *PageBuddy()* can be used to check if a physical frame is free. To check whether a physical frame is allocated to PTEs, we can use *PageTable()* function. Similarly, we can use *PageLRU()* and *compound_order()* functions for the PTE allocation. *PageLRU()* can be used to determine whether a page is migratable, while *compound_order()* helps identify fragmented pages and determine the size of an already allocated page.

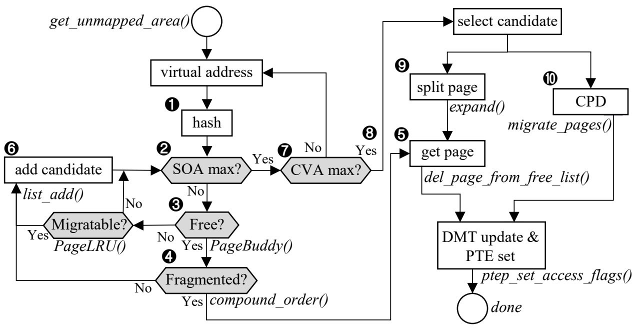


Fig. 12: PTE page allocation flow in Linux kernel with DPT.

We describe how these functions are used during the PTE allocation below.

PTE Allocation: Figure 12 shows the PTE page allocation flow of the Linux kernel using DPT. In conventional Linux, PTE pages are not allocated when creating a new Virtual Memory Area (VMA) as it uses the demand allocation scheme [20]. However, DPT needs to allocate the PTE pages when creating a VMA to use the CVA technique. To this end, we implement the PTE allocation in the *get_unmapped_area()* that creates a new VMA and returns the VMA address.

To allocate a PTE page, DPT calculates a base PFN for the given virtual address by using a hash function (❶). Then, DPT applies the SOA with the base PFN to find a physical frame that is free (❸) and resides within a fragmented region (❹). During this process, the *PageBuddy()* and *compound_order()* functions are used. Upon success in finding the physical frame, DPT allocates it to a new PTE page (❺). The SOA parameters (*step* and *stride*) corresponding to the physical frame are stored in the DMT, and the PTE is initialized using *ptep_set_access_flags()*. After that, the allocated physical frame is deleted from the free list of the buddy allocator by using *del_page_from_free_list()* (❺).

When SOA fails to find a suitable physical frame with the allowed step and stride values (❷), it adds physical frames to a candidate list using *list_add()* (❻). At this stage, a physical frame allocated to a migratable page and a free frame residing within the unfragmented region can be considered as a candidate frame. If SOA fails, the CVA technique is employed to find a collision-free VMA (❼). When the CVA operation reaches its limit (i.e., MAX_CVA_CNT), a victim frame is selected from the candidate list (❽). The choice of the victim frame depends on system priorities. If performance is prioritized over memory fragmentation, a free frame within an unfragmented region is selected to avoid data migration. In this case, a large physical memory region is split into smaller regions using the *expand()* function (❾). Otherwise, if memory fragmentation is a critical concern, a frame already allocated to a page will be selected as the victim. Then, the CPD technique migrates the page to another free frame using the *migrate_pages()* function (❿).

F. Supporting Virtualization

In a virtualized environment, address translation involves two steps: first, translating the guest virtual address (gVA) to a guest physical address (gPA) and then translating the gPA to the host physical address (hPA). With a 4-level page table, this process requires up to 24 memory accesses [48], [57]. DPT can significantly reduce the address translation overhead as it only requires two memory accesses for the translation (one for gVA-to-gPA mapping and another for gPA-to-hPA). Consequently, DPT can access the data page with up to three memory accesses in the virtualized environment.

The DPT can be implemented in both a guest OS and a host OS. In a virtualized environment, the guest OS assigns a unique guest PID (gPID) to each process running within the guest. DPT utilizes this gPID to perform address translation within the guest OS, converting a gVA to a gPA. Since the gPA is still a virtual address from the host's perspective, additional translation is needed to map the gPA to a hPA. This second translation stage is managed by the host OS, which assigns a unique host PID (hPID) to each virtual machine. DPT uses the hPID to perform the host-level translation, ensuring that the gPA is accurately mapped to the hPA.

VIII. Experimental Evaluations

A. Experimental Methodology

Simulated System: We evaluate our proposed DPT using the Sniper multi-core simulator [12] extended to include page table walker, page walk cache, memory allocator, and large page support. To simulate the system overheads, we incorporate data migrations and page fault handling latencies, which are obtained from a real system, into the simulator. We compare DPT against several techniques, including the conventional RPT, the state-of-the-art HPT called ME-HPT [56], and the latest large-coverage TLB called MOSAIC [21]. Table I describes the simulated system configuration used in our evaluation. We conduct trace-based simulations for multi-programmed workloads on a 4-core configuration. In the default simulation setup, the memory allocator is configured with an initial capacity utilization of 70%, meaning that 70% of the memory space is pre-allocated.

Real System: We demonstrate the feasibility of DPT by implementing its page table allocation mechanisms in the Linux kernel 5.11 running on a real server equipped with an Intel i9-11900 processor at 2.5GHz and 128GB DDR4-3200 DRAM. Using this prototype, we analyze the frequency of address collisions in DPT and measure the memory fragmentation rates, page fault overheads, and data migration overheads.

Workloads: Table II shows the workloads used in our experiments. Consistent with prior works [32], [33], [55], [56], we select memory-intensive workloads with irregular memory access patterns and a high L2 TLB MPKI (greater than 6.5). These workloads are chosen from diverse benchmark suites, including GraphBIG [41], GenomicsBench [58], HPCC [38], XSBench [61], and Sparse Length Sum from DLRM [42].

TABLE I: Simulated System Configuration

Component	Parameter
CPU	4-way Out of Order, 2.66GHz
L1 ITLB	64-entry, 8-way, 1-cycle
L1 DTLB	4KB page: 64-entry, 4-way, 1-cycle 2MB page: 32-entry, 4-way, 1-cycle
L2 TLB	1,536-entry, 12-way, 8-cycle
Page Walk Cache	3-level split PWC, 4-way; PGD: 16-entry; PUD: 16-entry; PMD: 32-entry; 2-cycle
L1 I/D-cache	32KB, 4-way, LRU, 4-cycle
L2 Cache	512KB, 16-way, LRU, 16-cycle
LLC	8MB, 16-way, LRU, 36-cycle
DRAM	16GB, DDR4-3200, 45ns latency; Capacity utilization: 70%
MOSAIC [21]	MOSAIC-8 TLB model
ME-HPT [56]	Supported page size: 4KB, 2MB 128-entries/way, 4-way for each page size Cuckoo Walk Cache: 16-entry each, 2-cycle Cuckoo Walk Table : 128 entries × 2 ways each Occupancy: 0.6; Chunk sizes: 8KB, 1MB L2P table size: 32-entry, 4-way Hash function: CITY [19], 2-cycle
DPT	Supported page size: 4KB, 2MB Hash function: CITY, 2-cycle MAX_SOA_STEP: 4; MAX_CVA_CNT: 4 2M-DMT: 4096 entries; 4K-DMT: 8192 entries 2M-DMC: 16-entry, 2-cycle, 4-way 4K-DMC: 32-entry, 2-cycle, 4-way Migration latency: 1100-cycle ≈ 2x(tRCD+tCL/tCWL+tBURSTx64+tRTP+tRP) TLB shootdown latency: 6.6 μs [34]

TABLE II: Workloads

Suite	Workloads	Input size
GraphBIG [41]	Betweeness Centrality (BC), Breadth-first search (BFS), Connected components (CC), Graph coloring (GC), PageRank (PR), Triangle counting (TC), Shortest-path (SSSP)	8GB
GenomicsBench [58]	K-length substring of DNA sequence counting (GEN)	33GB
HPCC [38]	Giga updates per second (RND)	10GB
XSBench [61]	Monte Carlo neutron transport (XS)	9GB
DLRM [42]	Sparse-length sum (DLRM)	10.3GB

B. Overall Performance

Speedup: Figure 13 compares the performance improvement of DPT with RPT, ME-HPT, and MOSAIC. On average, DPT achieves a speedup of 12.6% compared to the conventional RPT. Against the state-of-the-art techniques, MOSAIC and ME-HPT, DPT delivers average performance improvements of 11.6% and 8.7%, respectively.

As the TLB size increases, the advantage of using DPT diminishes because TLB misses become less frequent. However, even with a large TLB, workloads with irregular memory access patterns and large working sets can still experience frequent page walks. In Figure 13, MOSAIC shows negligible speedup compared to RPT and ME-HPT. This is attributed to the irregular memory access patterns and high TLB MPKI of the selected workloads. While increasing TLB coverage can be an easy and effective approach, this experimental result highlights the need for fundamental solutions to address page walks. Additionally, since DPT does not modify the TLB architecture, it is complementary to the techniques aimed at enhancing TLB coverage.

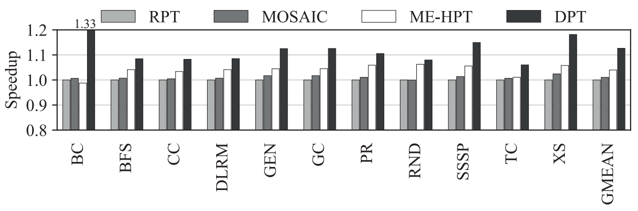


Fig. 13: Performance of RPT, ME-HPT, MOSAIC, DPT and DPT+MOSAIC (normalized to the RPT).

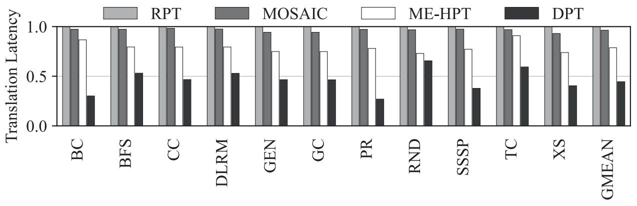


Fig. 14: Address translation latency (normalized to the RPT).

Translation Latency: In Figure 14, we compare the address translation latency across competitive mechanisms. DPT shows the lowest address translation latency. Compared to RPT, MOSAIC, and ME-HPT, DPT reduces the latency by 55.4%, 51.8%, and 34.2%, respectively. These improvements are achieved by reducing page walks compared to RPT-based multi-level memory accesses and by avoiding multi-way memory accesses involved in ME-HPT.

C. Memory Subsystem Characterization

Cache MPKI: Figure 15 shows the normalized Misses Per Kilo Instruction (MPKI) of the level-2 cache (L2C) and the last-level cache (LLC) across the competitive mechanisms. Compared to RPT, DPT reduces the MPKI by 14.7% for L2C and 14.8% for LLC. RPT and MOSAIC experience higher cache MPKI because page walks access the caches at each level of the radix-tree page table, causing more cache block replacements. In contrast, DPT and ME-HPT typically require only a single memory access for the address translation, resulting in fewer cache misses than RPT. However, ME-HPT's need for parallel accesses across multiple HPT ways, along with PTE migration through cuckoo hashing, can pollute the caches by loading unnecessary PTEs. The PTE migrations also lead to invalid PTEs being accessed due to coherence issues in the MMU cache (i.e., CWC), resulting in additional memory accesses. Since DPT resolves the address collisions without PTE migrations by utilizing SOA and CVA, it has a relatively small impact on the MPKI of the caches.

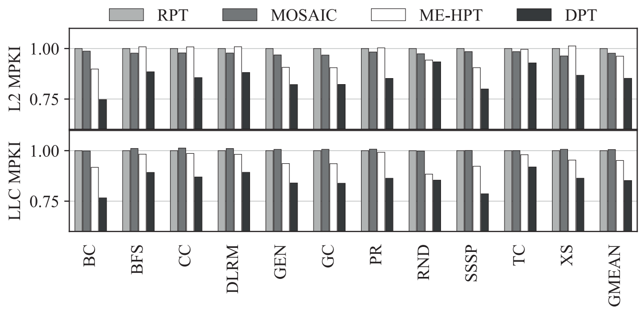


Fig. 15: MPKI of caches (normalized to the RPT).

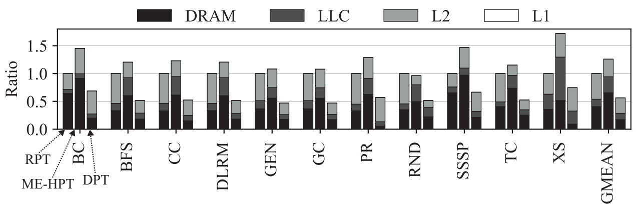


Fig. 16: Breakdown of the servicing location of memory requests that fetch translation information (normalized to the RPT).

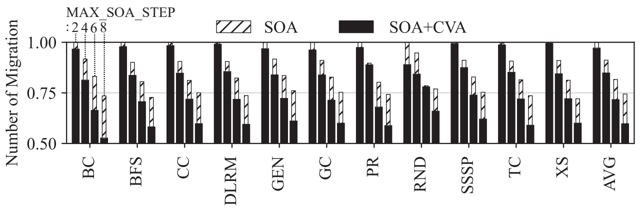


Fig. 18: The number of data migrations in DPT using SOA and using SOA+CVA (normalized to the DPT using SOA with a *MAX_SOA_STEP* of 2).

Breakdown of Memory Requests: Figure 16 shows the breakdown of memory requests involved in the address translation for different page table structures. This figure demonstrates that ME-HPT increases memory requests involved in the address translation, whereas DPT reduces them. On average, ME-HPT results in a 25.8% increase in memory requests compared to RPT, whereas DPT achieves a 43.5% reduction. ME-HPT's higher request count is due to parallel requests and cuckoo hashing. Additionally, the repetitive PTE migrations pollute the caches as unnecessary data is loaded into them. Consequently, approximately 65% of ME-HPT's requests are loaded from the main memory (DRAM).

In contrast, DPT reduces memory requests by eliminating both sequential access for multi-level translation (involved in RPT) and parallel accesses for multi-way lookups (involved in HPT). Furthermore, DPT maintains valid address translations in the DMC, as PTE pages remain unmovable. DPT predominantly retrieves translation information from the L2 cache, with relatively lower retrieval rates from the main memory (DRAM) and LLC. This result highlights that DPT effectively stores translation information without causing cache pollution.

D. Effectiveness of DPT Components

DPT Metadata Cache: As described in Section VII, DPT stores the SOA parameters (i.e., stride and step) in the DPT Metadata Table (DMT). Since the DMT is stored in memory, DPT employs the DPT Metadata Cache (DMC) to hold recently used metadata, reducing the need for frequent memory accesses to the DMT. During program execution, DPT can retrieve the metadata from the DMC, on-chip caches (L1/L2/LLC), or off-chip memory (DRAM). Figure 17 illustrates the latency associated with fetching DPT metadata from these storage components. In the figure, ALLOC represents the cases where the metadata is not available in the DMT.

As described in Section VII-A, one PTE page can be represented by 6-bit SOA parameter. A cache line in the DMC can hold multiple SOA parameters, thus providing extensive virtual address coverage. Consequently, as shown in Figure 17, approximately 75.8% of the metadata fetch latency originates from the DMC, with the remainder primarily coming from the L2 cache. Even if a metadata fetch request misses in the DMC, the latency remains relatively low because the required metadata can still be retrieved from the cache hierarchy.

SOA and CVA: Figure 18 shows how SOA and CVA techniques impact the number of data migrations. The effects of these techniques are compared for various SOA parameters. All results are normalized to the SOA with a *MAX_SOA_STEP* of 2. The hatched bars in the figure represent the number of data migrations when using only SOA in the DPT. For all workloads, the number of data migrations decreases as the *MAX_SOA_STEP* increases. Specifically, compared to SOA with the *MAX_SOA_STEP* of 2, the data migrations are reduced by 8%, 19%, and 26% with *MAX_SOA_STEP* of 4, 6, and 8, respectively.

The black bars represent the number of data migrations when DPT uses both SOA and CVA. The addition of CVA significantly reduces the data migrations. Specifically, compared to using only SOA, the use of SOA+CVA reduces the number of data migrations by 25% on average when *MAX_SOA_STEP* is configured to 8.

DPT's SOA effectively avoids address collisions but requires repeated checks of the frame bitmap (*struct page* in Linux) to find a free physical frame. Figure 19 shows the overhead of SOA in a real system. As shown in the figure, the performance impact of repetitive search of the SOA is negligible. This is because this process involves accessing a small structure that is typically cached.

E. Additional Analysis

Memory Fragmentation: Figure 20 shows the impact of DPT with FAP on memory fragmentation, measured by

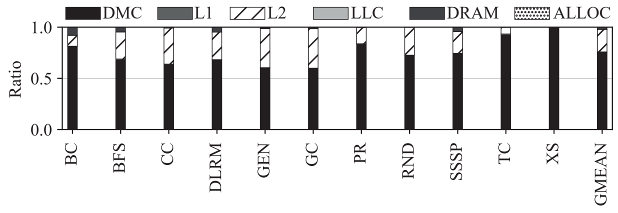


Fig. 17: Latency breakdown by hit location for DPT Metadata.

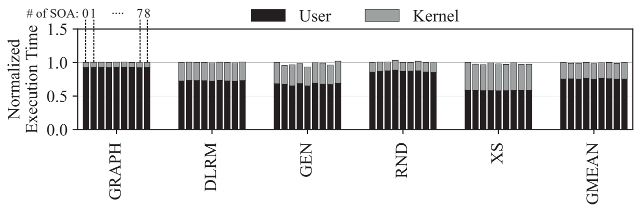


Fig. 19: Performance overhead of the strided search of SOA in Linux system. 'GRAPH' represents the average of experimental results across all GraphBIG workloads.

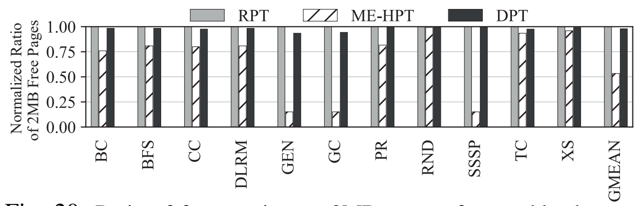


Fig. 20: Ratio of free contiguous 2MB pages after workload completion (normalized to the RPT).

the number of free contiguous 2MB pages after workload completion. The figure shows that DPT maintains a similar number of free 2MB pages compared to RPT, whereas ME-HPT experiences a significant reduction in the free 2MB pages. On average, DPT reduces free 2MB pages by 3% compared to RPT, whereas ME-HPT experiences a 47% reduction.

Data Migration: Figure 21 compares the frequency of data migrations. In this experiment, we vary the initial memory capacity utilization to accelerate data migrations caused by address collisions. "Sniper" refers to the simulated system, while "Linux" denotes the real system. As memory utilization increases, the data migrations per page allocation slightly increase. Even in the worst-case scenario (90% memory utilization), the migration rate of DPT remains below 0.0007 in both simulated and real systems, which is negligible compared to ME-HPT (shown in Figure 3). Applying FAP, which significantly reduces the memory fragmentation, leads to a slight increase in the migration rate due to more conservative PTE page allocation. Nonetheless, even at 90% memory utilization, FAP increases the migration rate by only 0.0002 in both the simulated and real systems.

Performance Impact of Memory Utilization: We conduct experiments to evaluate the performance improvement of DPT over RPT for various memory utilization. Figure 22 shows the experimental results for five different utilization levels. As shown in the figure, the performance gain remains stable up to around 80% utilization, with a slight drop at 90%. At an extreme memory utilization level (98%), the performance declines by approximately 3.3%. As memory utilization increases, the frequency of address collisions also rises. However, this increase in collisions does not lead to allocation failures because PTE allocation can still be successful through SOA and CVA mechanisms. If the PTE allocation fails within the allowed iterations for SOA and CVA, CPD can be triggered to resolve the collisions by migrating a data page. At 98% utilization, CPD occurs 13.4× more frequently than at 70% utilization, resulting in performance degradation due to TLB shootdown during the page migration. However, this overhead occurs only during page faults, specifically when allocating PTEs, and not during every PTE lookup.

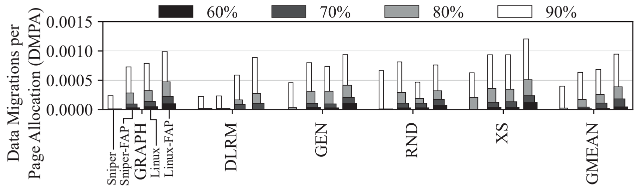


Fig. 21: Data migrations per page allocation for the DPT in the simulated system and the real system across various memory utilization levels. 'GRAPH' represents the average of experimental results across all GraphBIG workloads.

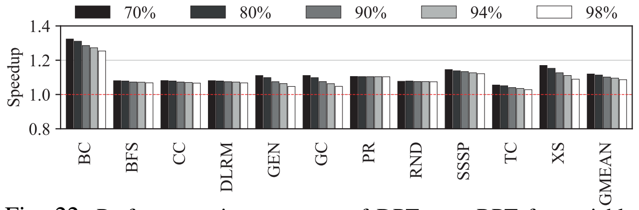


Fig. 22: Performance improvement of DPT over RPT for variable memory utilization levels.

Although high memory utilization can reduce the performance improvement of DPT, this does not significantly limit the effectiveness of DPT in practice. In conventional operating systems, searching or compacting contiguous memory space is often necessary when allocating memory. If the memory usage is significantly high, however, it is challenging to perform these operations. Therefore, memory utilization needs to be maintained at an appropriate level. Recent studies indicate that memory capacity utilization in HPC systems is generally low (less than 35% for 90% of the system running time [16], [45], [49], [50]). Additionally, the Linux system defines *vm.swappiness* parameter for the memory swap mechanisms which controls the tendency of the kernel to swap memory pages. A *vm.swappiness* value of 60 (the default) balances between swapping and caching.

Performance for System with THP: Figure 23 compares the performance of MOSAIC, ME-HPT, and DPT with RPT for the system using the Transparent Huge Page (THP) [5]. MOSAIC achieves a minor performance improvement as the baseline TLB also performs well with the large page support, while ME-HPT and DPT improve the performance by 3.3% and 11.2%, respectively. ME-HPT performs similarly to DPT in some workloads but underperforms in others due to its maintenance of different page tables for varying page sizes, which affects the total page table access count. DPT experiences a slight reduction in performance improvement compared to using only base pages, which is attributed to increased TLB coverage with the large page support. Nevertheless, DPT remains effective for the system with THP as it efficiently handles various page sizes.

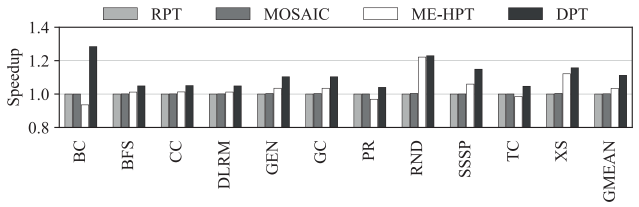


Fig. 23: Performance comparison for the system with THP [5] (normalized to the RPT).

IX. Conclusion

In this paper, we proposed a new page table structure called the Distributed Page Table (DPT) as a novel approach to overcome the limitations of HPTs. The main idea in DPT's approach is to utilize the entire physical memory space using a hash function to eliminate the need to prepare a large contiguous physical memory space, a limitation faced by state-of-the-art HPT techniques. To handle the issue of potential collisions in DPT, we introduced three novel techniques - SOA, CVA, and CPD. Additionally, PTE Pooling (PTP) was designed to reduce collisions, and Fragmentation-Aware PTE Allocation (FAP) was employed as an additional optimization technique to reduce fragmentation. We verified the effectiveness of DPT's approach by seamlessly integrating it into current virtual memory systems and conducting various performance measurements. DPT delivered performance improvements of 12.6%, 11.6%, and 8.7% over traditional RPT, the latest large-coverage TLB technique, and state-of-the-art HPTs, respectively.

Acknowledgment

The authors would like to thank Dr. Eun Kyung Lee from IBM and the anonymous reviewers for their valuable feedback in improving the quality of this paper. This work was partly supported by the National Research Foundation of Korea (NRF) grant (No.2022R1C1C1012154, 60% / No.2021R1A5A1021944, 20%), Institute of Information & communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (No.2022-0-01170, 10% / No.2021-0-00863, 10%). Seokin Hong is the corresponding author.

References

[1] E. Abrossimov, M. Rozier, and M. Shapiro, "Generic virtual memory management for operating system kernels," in *Proceedings of the twelfth ACM symposium on Operating systems principles*, 1989, pp. 123–136.

[2] AMD, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, AMD, 2023.

[3] B. S. An, M. H. Cha, S.-M. Lee, W. H. Yang, and H. Y. Kim, "Providing scalable single-operating-system numa abstraction of physically discrete resources," *ETRI Journal*, 2024.

[4] A. W. Appel and K. Li, "Virtual memory primitives for user programs," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, 1991, pp. 96–107.

[5] A. Arcangeli, "Transparent hugepage support," in *KVM forum*, vol. 9, 2010.

[6] 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.

[7] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 237–248. [Online]. Available: https://doi.org/10.1145/2485922.2485943

[8] 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*, ser. ASPLOS XIII. New York, NY, USA: Association for Computing Machinery, 2008, p. 26–35.

[9] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 383–394.

[10] ——, "Translation-triggered prefetching," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 63–76.

[11] M. J. Bligh and D. Hansen, "Linux memory management on larger machines," in *Proc. Linux Symposium*, 2003.

[12] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[13] I. T. Community, "Systems and architectures - international roadmap for devices and systems (irds)," 2022.

[14] J. Corbet. (2010) Memory compaction. [Online]. Available: https://lwn.net/Articles/368869/

[15] ——, "Concurrent page-fault handling with per-vma locks," 2022. [Online]. Available: https://lwn.net/Articles/906852/

[16] R. S. S. Dittakavi, "Deep learning-based prediction of cpu and memory consumption for cost-efficient cloud resource allocation," *Sage Science Review of Applied Machine Learning*, vol. 4, no. 1, pp. 45–58, 2021.

[17] C. Dougan, P. Mackerras, and V. Yodaiken, "Optimizing the idle task and other mmu tricks," in *OSDI*, 1999, pp. 229–237.

[18] S. Eranian and D. Mosberger, "The linux/ia64 project: kernel design and status update," *HP LABORATORIES TECHNICAL REPORT HPL*, vol. 85, 2000.

[19] Google, *CITY Hash*, Google, 2012.

[20] M. Gorman, *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[21] K. Gosakan, J. Han, W. Kuszmaul, I. N. Mubarek, N. Mukherjee, K. Sriram, G. Tagliavini, E. West, M. A. Bender, A. Bhattacharjee *et al.*, "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, Volume 3*, 2023, pp. 433–448.

[22] G. Hoang, C. Bae, J. Lange, L. Zhang, P. Dinda, and R. Joseph, "A case for alternative nested paging models for virtualized systems," *IEEE Computer Architecture Letters*, vol. 9, no. 1, pp. 17–20, 2010.

[23] J. Huck and J. Hays, "Architectural support for translation table management in large address space machines," in *Proceedings of the 20th annual international symposium on computer architecture*, 1993, pp. 39–50.

[24] IBM, *PowerPC Microprocessor Family: The Programming Environments Manual for 64 and 32-Bit Microprocessors*, IBM, 2003.

[25] Intel, *TLBs, Paging-Structure Caches, and Their Invalidation*, Intel, 2008.

[26] ——, *Intel Itanium Architecture Software Developer's Manual, Volume 2*, Intel, 2010.

[27] ——, *Intel 64 and IA-32 Architectures Developer's Manual, Volume 3*, Intel, 2016.

[28] ——, *5-Level Paging and 5-Level EPT*, Intel, 2017.

[29] ——, "Sunny cove microarchitecture," 2018. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

[30] S. Jang, J. Park, O. Kwon, Y. Lee, and S. Hong, "Rethinking page table structure for fast address translation in gpus: A fixed-size hashed page table," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2024.

[31] J. Jann, P. Mackerras, J. Ludden, M. Gschwind, W. Ouren, S. Jacobs, B. F. Veale, and D. Edelsohn, "Ibm power9 system software," *IBM Journal of Research and Development*, vol. 62, no. 4/5, pp. 6–1, 2018.

[32] K. Kanellopoulos, R. Bera, K. Stojiljkovic, F. N. Bostanci, C. Firtina, R. Ausavarungnirun, R. Kumar, N. Hajinazar, M. Sadrosadati, N. Vijaykumar *et al.*, "Utopia: Fast and efficient address translation via hybrid restrictive & flexible virtual-to-physical address mappings," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1196–1212.

[33] K. Kanellopoulos, H. C. Nam, N. Bostanci, R. Bera, M. Sadrosadati, R. Kumar, D. B. Bartolini, and O. Mutlu, "Victima: Drastically increasing address translation reach by leveraging underutilized cache resources," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1178–1195.

[34] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "Latr: Lazy translation coherence,"

in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 651–664.

[35] O. Kwon, Y. Lee, and S. Hong, "Pinning page structure entries to last-level cache for fast address translation," *IEEE Access*, vol. 10, pp. 114 552–114 565, 2022.

[36] ——, "Virtual pte storage: Repurposing last-level cache to accelerate address translation for big data workloads," in *2022 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. IEEE, 2022, pp. 1–5.

[37] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.

[38] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The hpc challenge (hpcc) benchmark suite," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, vol. 213, no. 10.1145, 2006, p. 1.

[39] Z. Ma, Y. Tan, H. Jiang, Z. Yan, D. Liu, X. Chen, Q. Zhuge, E. H.-M. Sha, and C. Wang, "Unified-tp: A unified tlb and page table cache structure for efficient address translation," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 255–262.

[40] D. McCracken, "Object-based reverse mapping," in *Linux Symposium*, 2004, p. 357.

[41] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: understanding graph computing in the context of industrial solutions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[42] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

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

[44] 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.

[45] G. Panwar, D. Zhang, Y. Pang, M. Dahshan, N. DeBardeleben, B. Ravindran, and X. Jian, "Quantifying memory underutilization in hpc systems and using it to improve performance via architecture support," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 821–835.

[46] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing memory-mapped {I/O} for fast storage devices," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 813–827.

[47] 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 *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 913–925.

[48] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, "Every walk'sa 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.

[49] I. Peng, I. Karlin, M. Gokhale, K. Shoga, M. Legendre, and T. Gamblin, "A holistic view of memory utilization on hpc systems: Current and future trends," in *The International Symposium on Memory Systems*, 2021, pp. 1–11.

[50] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 183–190.

[51] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew, "Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures," in *Proceedings of the second international conference on Architectual support for programming languages and operating systems*, 1987, pp. 31–39.

[52] 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.

[53] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler, "Lightweight recoverable virtual memory," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 1, pp. 33–57, 1994.

[54] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch *et al.*, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 185–197.

[55] 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.

[56] J. Stojkovic, N. Mantri, D. Skarlatos, T. Xu, and J. Torrellas, "Memory-efficient hashed page tables," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1221–1235.

[57] 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.

[58] A. Subramaniyan, Y. Gu, T. Dunn, S. Paul, M. Vasimuddin, S. Misra, D. Blaauw, S. Narayanasamy, and R. Das, "Genomicsbench: A benchmark suite for genomics," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 1–12.

[59] M. Talluri, M. D. Hill, and Y. A. Khalidi, "A new page table for 64-bit address spaces," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995, pp. 184–200.

[60] C. Tirumalasetty, C. C. Chou, N. Reddy, P. Gratz, and A. Abouelwafa, "Reducing minor page fault overheads through enhanced page walker," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 4, pp. 1–26, 2022.

[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] H. Wang, J. Zhang, S. Shridhar, G. Park, M. Jung, and N. S. Kim, "Duang: Fast and lightweight page migration in asymmetric memory systems," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 481–493.

[63] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 331–345.

[64] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Cramm: Virtual memory support for garbage-collected applications," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 103–116.

[65] 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.

[66] K. Zhao, K. Xue, Z. Wang, D. Schatzberg, L. Yang, A. Manousis, J. Weiner, R. Van Riel, B. Sharma, C. Tang *et al.*, "Contiguitas: The pursuit of physical memory contiguity in datacenters," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.