IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 27 September 2022, accepted 11 October 2022, date of publication 26 October 2022, date of current version 7 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3217484

RESEARCH ARTICLE

Pinning Page Structure Entries to Last-Level Cache for Fast Address Translation

OSANG KWON, (Graduate Student Member, IEEE), YONGHO LEE, AND SEOKIN HONG, (Member, IEEE)

Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, Republic of Korea

Corresponding author: Seokin Hong (seokin@skku.edu)

This work was supported in part by the Institute of Information Communications Technology Planning Evaluation (IITP) funded by the Korean Government (MSIT) through the Intelligent In-Memory Error-Correction Device for High-Reliability Memory (30%) under Grant 2021-0-00863, in part by the National Research Foundation of Korea (NRF) funded by the Korean Government (MSIT) (30%) under Grant 2022R1C1C1012154, in part by IITP funded by the Korean Government (MSIT) through the PIM Semiconductor Design Research Center (40%) under Grant 2022-0-01170, and in part by the BK21 FOUR Project.

ABSTRACT As the memory footprint of emerging applications continues to increase, the address translation becomes a critical performance bottleneck owing to frequent misses on the Translation Lookaside Buffer (TLB). In addition, the TLB miss penalty becomes more critical in modern computer systems because the levels of the hierarchical page table (a.k.a. radix page table) are increasing to extend the address space. To reduce TLB misses, modern high-performance processors employ a multi-level TLB structure using a large last-level TLB. Employing a large last-level TLB may reduce TLB misses. However, its capacity is still limited, and it can incur a chip area overhead. In this paper, we propose a Page Structure Entry (PSE) pinning mechanism that provides a large PSE store by dedicating some space to the last-level cache to store only the page structure entries. The PSE Pinning is based on three key observations. First, memory-intensive applications suffer from frequent misses in the last-level cache. Thus, most of the space in the last-level cache is not utilized well. Second, most PSEs are fetched from the main memory during the page table walk process, meaning that the cache lines for the PSEs are frequently evicted from on-chip caches. Finally, a small number of PSEs are frequently accessed while others are not. By exploiting these three observations, PSE Pinning pins the frequently accessed page structure entries to the last-level caches so that they can reside on the cache. Experimental results show that PSE Pinning improves the performance of memory-intensive workloads suffering from frequent L2 TLB misses by 7.8% on average.

INDEX TERMS Address translation, page walk, translation lookaside buffer, virtual memory.

I. INTRODUCTION

For decades, physical memory capacity has been increasing dramatically to accommodate the ever-growing memory footprint of modern applications. Recently, this trend has been rapid because of the emerging big-data applications consuming large amounts of data on physical memory [1], [2], [3]. The prevalence of cloud services also increases the demand for large memory capacity to provide more virtualized compute instances with a physical compute node [4].

The rapid scale-up of physical memory capacity reveals a fundamental limitation in the current memory subsystem. In most computing systems, from smartphones to data-center-scale servers, memory is orchestrated by the paging-based virtual memory that provides a virtual address space with the support of virtual-to-physical address translation [5]. As virtual memory provides an isolated large memory space for each process irrespective of the physical memory capacity, it is one of the common building blocks of modern computers. However, virtual memory has been ineffective in large-scale memory systems due to the address translation becoming a significant performance bottleneck. [6], [7], [8], [9], [10].

For address translation, the virtual memory typically uses a *radix page table*, which is a type of radix-tree data structure [11]. In the radix page table, each tree level is implemented with a table called Page Structure Table (PST).

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

An intermediate PST entry, called Page Structure Entry (PSE), provides an address of the next-level tree nodes, while the actual address translation information is stored in the leaf PSTs of the radix page table.

Address translation with the radix page table requires a tree traversal process called a *page walk*, which traverses each level of the radix tree from the root to the leaf. Because the PSTs are stored in memory, a page walk requires multiple memory references to provide all required PSEs to obtain the address translation information. Therefore, when the page walk process involves frequent access to the main memory, it significantly degrades system performance. Furthermore, the height of the radix tree increases as the address space is expanded [12], making virtual memory systems even less efficient. A page walk can be more critical in virtualized environments where second-level address translation (SLAT) is required for address translation. Intel and AMD processors employ an extended page table (EPT) [13] or rapid virtualization index (RVI) [14] to reduce the overhead of SLAT. However, even with these architectural supports, enormous memory accesses can be generated because of the page walks for irregular workloads because a page walk is required for each level of the guest page table [15], [16].

To mitigate page walk overheads, modern microprocessors employ a Translation Lookaside Buffer (TLB), which is a small cache that holds translation information for recently and frequently accessed pages. Even if the TLB can efficiently reduce the page walk overhead, its capacity is small (it holds the translation information for a few pages) because the TLB is on the critical path of the processor pipeline. Thus, the TLB is also implemented in a multi-level hierarchical structure, where a large L2 TLB is employed along with a small L1 TLB. Unfortunately, however, an L2 TLB suffers from frequent misses because L2 TLB capacity is not sufficient to meet the high capacity demand of many emerging big data applications with a large memory footprint and irregular memory access patterns [6], [17], [18], [19], [20].

There has been a large corpus of prior work on mitigating the overhead of page walks by reducing TLB misses [15], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30] or by reducing the TLB miss penalty [13], [16], [18], [31], [32], [33], [34], [35]. This study focuses on the second approach. Specifically, we propose a technique that avoids expensive off-chip main memory (i.e., DRAM) accesses by holding as many PSEs in the on-chip caches as possible. Because the main memory accesses contribute a large portion of the page walk latency as shown in Figure 1, eliminating the main memory accesses in the page walk process can lead to significant performance gain in the address translation.

Recently, Marathe et al. [35] proposed a similar approach, called CSALT, to reduce page walk latency. In their work, on-chip caches were dynamically partitioned into a TLB entry region and a normal data region. By holding frequently and recently referenced TLB entries in the TLB entry region, CSALT reduces the expensive off-chip memory accesses for reading the TLB entries from POM-TLB (proposed in [34])

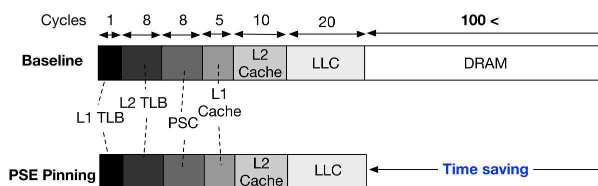

FIGURE 1. Page walk latency reduction with PSE Pinning. Our goal is to minimize the off-chip memory accesses by pinning frequently re-fetched PSEs to the last-level cache.

which is a large TLB located at the off-chip memory. Even though CSALT is an efficient method that provides a dedicated on-chip storage for holding the address translation information, it relies on their specific memory system architecture (i.e., POM-TLB). More recently, Park et al. [18] proposed a method called page table prioritization (PTP) to prioritize cache blocks such that cache blocks containing PSEs (PSE blocks) reside in the on-chip cache. The basic idea of PTP is similar to ours. However, PTP does not provide a method to determine the PSE blocks that must reside in the on-chip caches. Therefore, it can suffer from frequent conflict misses when many PSE blocks are referenced with a low temporal or spatial locality.

This paper proposes ***PSE Pinning*** to reduce page walk latency by pinning frequently accessed PSEs (referred to as *hot PSE*) to the last-level cache (LLC). The PSE Pinning is motivated by three key observations. First, the LLC is not effective for workloads with a large memory footprint and irregular memory access patterns, and thus the performance of those workloads is not sensitive to the LLC capacity. Second, many PSEs have the temporal/spatial locality, even in memory-intensive workloads. Finally, in the most page walk process, PSEs are fetched from the main memory, meaning they are repeatedly evicted from the LLC because of conflict misses on the LLC and are brought again from the main memory.

Based on these observations, PSE Pinning detects the PSE blocks (a cache block containing the PSEs) repeatedly brought from the main memory and then pins them to the LLC to make them reside in the LLC. By pinning the PSE blocks, the page walk process can find the required PSEs in the LLC without accessing the slow main memory (i.e., DRAM), significantly reducing the page walk latency as shown in Figure 1. PSE Pinning relies on two mechanisms: PSE selection and dynamic pinning. The PSE selection mechanism selects the PSE blocks that have to be pinned in the LLC by maintaining per-block access counters. When a PSE block is brought from the main memory, the access counter for that block is incremented. By referring to the access counter, PSE blocks repeatedly fetched from the main memory can be detected. A access counters can be maintained without additional storage overhead by storing them in unused bits of the PSE. The dynamic pinning mechanism detects program phases in which performance is significantly degraded by frequent TLB and LLC misses. After detecting the program phase, the PSE Pinning mechanism configures the LLC to

designate the LLC space specifically for the PSEs and then pin the selected PSEs there.

Experimental results with a cycle-level simulator show that PSE Pinning reduces the main memory accesses for fetching the PSE blocks by 65.1% on average, leading to an average reduction of 28.4% in page walk latency. With the dramatic improvement in the page walk latency, PSE Pinning achieves a speedup of 7.8% on average for memory-intensive workloads suffering from frequent page walks.

II. BACKGROUND

A. VIRTUAL MEMORY SYSTEM

Virtual memory (VM) is a fundamental building block in modern computer systems. It simplifies application programming by providing a separate (virtual) address space for each process. Furthermore, virtual memory enables applications to use more (virtual) memory space than physical memory capacity. In virtual memory with a paging scheme, address translation is performed in page size granularity (e.g., typically 4KB).

A page table is commonly used for address translation to maintain the virtual to physical address mapping information, but with different structures across computer systems. The x86-64 architecture, for example, uses a radix page table implemented with a multi-level radix-tree structure. In modern computer architectures, the radix page table level has increased from four to five to accommodate the rapidly increasing memory footprint of memory-intensive applications. With the expansion of the page table level, the virtual address (VA) has increased from 48 to 57 bits, expanding the virtual memory space from 256 TB to 128 PB [12].

Figure 2 shows a 5-level radix page table in the x86-64 architecture. Each level of the page table is denoted as Page Map Level-5 (PML5), Page Map Level-4 (PML4), Page Directory Pointer (PDP), Page Directory (PD), and Page Table (PT), respectively [12]. In this paper, we refer to all these tables as *Page Structure Tables (PSTs)* and the entries of the tables as *Page Structure Entries (PSEs)*. The PSEs of each intermediate PST holds the address pointing to the next-level PST, while the PSEs of the last-level PST (i.e., a leaf of the tree) hold the address translation information. The entry of the last-level PST is also known as Page Table Entry (PTE).

To translate a given virtual address to a physical address, the radix page table is traversed from the PML5 table (root PST) to the page table (leaf PST). This traversal process is called a *page walk*, which incurs multiple memory accesses for address translation. In the x86-64 architecture, the CR3 register contains the base address of the PML5 table. The base address is added to bits [56:48] of the virtual address to get the address of the corresponding PSE that holds the base address of the PML4 table. The objective of the page walk is to find a page table entry in the page table (leaf PST) by repeatedly carrying out this search process. Since consecutive searches are required for the 5-level radix page table, address translation incurs frequent memory accesses. Although PSEs are cacheable in the memory hierarchy, the page walk process involves five main memory accesses in the worst case.

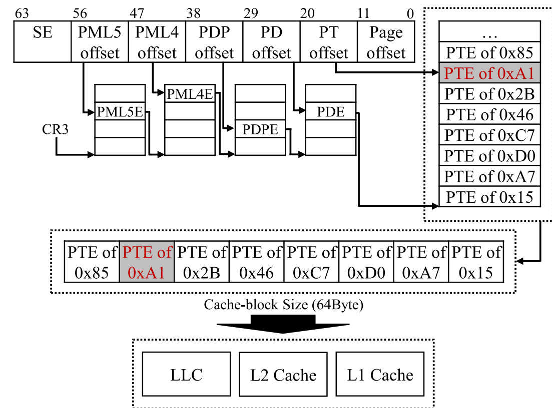

FIGURE 2. Virtual to physical address translation. The hierarchical structure of radix page table makes the address translation less efficient due to frequent main memory accesses.

Owing to consecutive memory accesses, page walk may introduce a significant amount of latency into the address translation. In order to reduce the address translation latency, Translation Lookaside Buffer (TLB) is utilized by the Memory Management Unit (MMU). TLB maintains a cache of the most recently and frequently accessed address translation information. When performing address translation, the CPU first checks the TLB, and if the necessary address translation information is present, a page walk is bypassed. Even if the TLB reduces the number of page walks, it cannot reduce the latency of the page walk caused by a TLB miss. Therefore, if the TLB miss rate is high, the system's performance will be degraded dramatically as a result of the long page walk latency.

B. PAGE STRUCTURE CACHE

To minimize the page walk latency, the Intel x86 architecture includes Page Structure Caches (PSCs) to hold the PSEs of the intermediate PST (from PML5 to PD). By holding recently and frequently used PSEs, the PSC allows the page walk to skip some levels of the radix page table, reducing the memory accesses involved during the page walk and consequently reducing the page walk latency [17], [19], [36]. For example, PML4 can be accessed directly while skipping the PML5 table access if high-order 9 bits of the virtual address (VA[56:48]) required for the PML5 index are matched on a PML5-PSC (PSC holding the entries of the PML5 table). In the best case, if VA[56:21] is matched on the PD-PSC (PSC holding the entries of the PD table), the 5-step page walk is reduced to just a single step. On average, PSCs enable skipping steps 2.2 $\sim$ 2.9 in the 4-level page walk [18].

Unfortunately, as pointed out in a previous study [18], the efficiency of PSCs varies according to the locality of the memory references, and it rarely performs effectively for workloads with irregular access patterns. When a PSC miss occurs, the page walk process traverses the memory hierarchy, from the L1 data cache to the off-chip main memory,

TABLE 1. Workload specification.

Workload	Description	Footprint
lbm	Computational Fluid Dynamics benchmark from SPEC 2006	450MB
leslie3d	Computational Fluid Dynamics benchmark from SPEC 2006	123MB
soplex	Simplex Linear Program Solver benchmark from SPEC 2006	126MB
omnetpp	Discrete Event Simulation benchmark from SPEC 2006	170MB
xsbench	Monte Carlo neutron transport algorithm benchmark	3,885MB
tc	Triangle Counting benchmark from GAPBS	17.0 ∼ 37.5GB
cc	Connected Components benchmark from GAPBS	17.2 ∼ 17.5GB
sssp	Single-Source Shortest Path benchmark from GAPBS	33.7 ∼ 35.0GB
bc	Betweenness Centrality benchmark from GAPBS	19.5 ∼ 20.0GB
bfs	Breadth-First Search benchmark from GAPBS	17.6GB
gups	Giga Updates Per Second benchmark from HPCC	2,050MB

looking for the required PSEs. As a result, page walk latency varies dramatically depending on the incidence of PSC misses and the memory hierarchy level at which the PSEs are found.

C. EMERGING WORKLOAD CHALLENGES

Table 1 shows the memory footprints of the workloads used in this study, which comprises memory-intensive workloads from the SPEC CPU 2006 suites [37], GAP benchmark suites [38], XSBench [39] and HPCC [40]. As shown in the table, the emerging graph analytic workloads have a considerably larger memory footprint than the memory-intensive SPEC CPU2006 benchmarks. Their memory footprints range from a few to several gigabytes.

Graph analytic workloads often have poor locality in memory reference because of their irregular memory access patterns and large memory footprints [41]. When the workload lacks locality in the memory reference, caches such as the TLB and PSC operate less efficiently, resulting in frequent misses. For these workloads, page walks are required to locate entries in the lower memory hierarchy, resulting in frequent off-chip memory accesses.

Due to the frequent off-chip memory accesses, address translation is faced with a drastic increase in latency. Memory accesses triggered by page walks can account for up to 20∼40% of the total memory accesses [20]. Recent studies have pointed out that address translations requiring many memory accesses are the primary performance bottlenecks [18], [29], [30], [33], [34], [42], [43].

The address translation latency may be reduced by increasing the size of TLB and PSC. Unfortunately, this can produce chip area overheads because the TLB and PSC must be kept close to the processor core and small in size to enable fast address translation. To make matters worse, memory footprints have grown larger, and the height of the radix page table has been expanded from four to five. As the height of the page table increases, the page walk overheads also increases [32]. Thus, a new mechanism is required to reduce address translation overheads with minimal hardware modification by mitigating this challenge of virtual memory on the emerging workloads.

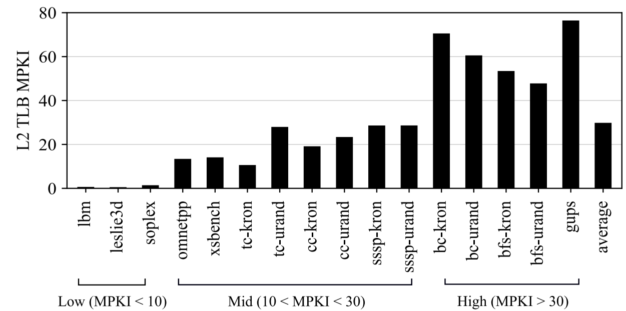

FIGURE 3. L2 TLB misses per kilo-instruction (MPKI). Memory-intensive workloads with irregular memory access patterns suffer from frequent misses on L2 TLB.

III. MOTIVATION

A. TLB MISSES

Because the memory footprint of new memory-intensive applications can exceed several terabytes, the TLB is not enough to handle address translations for such massive data, resulting in frequent TLB misses. Figure 3 shows Miss Per Kilo Instructions (MPKI) in the L2 TLB for various memory-intensive workloads. As shown in the figure, the workloads with relatively small memory footprint (lbm, leslie3d, and soplex) experience negligible L2 TLB misses. However, L2 TLB does not perform well on workloads with large memory footprints and irregular memory access patterns (bc-kron, bc-urand, bfs-kron, bfs-urand, and gups), and hence these workloads suffer from frequent L2 TLB misses. Because of the frequent misses on the L2 TLB, the fraction of page walks in total memory access latency becomes considerably high when these workloads are executed, resulting in a substantial decrease in system performance [15], [43], [44].

B. PAGE WALK LATENCY

As described in Section II-B, when both the TLB and PSC fail to translate an address, a page walk is performed to traverse the memory hierarchy from the L1 cache to the off-chip main memory. Figure 4 shows the contribution of each level of the memory hierarchy on the page walk latency. As shown in the figure, a large portion of the latency encountered during page walks is due to the main memory accesses. On average, 61% of the page walk latency is contributed by accessing the main memory (denoted by MEM). In the worst case, the main memory accounts for up to 91% of the page walk latency. Applications with a large memory footprint and irregular memory access patterns involve severe cache contention [45]. Consequently, the on-chip cache has a relatively low contribution to page walk latency compared with the main memory. *This observation implies that address translation overhead can be significantly reduced by avoiding access to the main memory during page walks.*

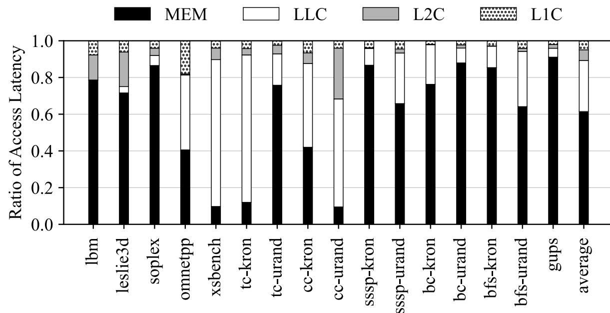

FIGURE 4. Distribution of page walk latency. Off-chip main memory contributes up to 91% of the page walk latency.

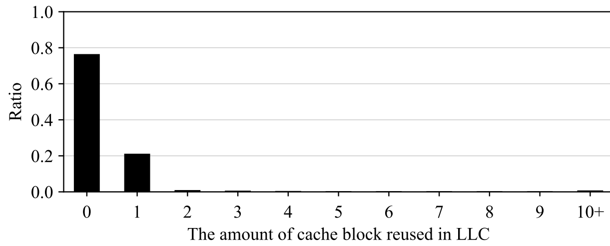

FIGURE 5. Distribution of cache block reused in LLC. Overall, 90% of the cache blocks are reused no more than once before eviction.

C. LLC'S INCOMPETENCE

The LLC is a large on-chip cache that is employed to reduce expensive main memory accesses. If a PSE is found in the LLC, the number of accesses to the main memory can be significantly reduced. Unfortunately, however, the LLC does not work well for memory-intensive workloads with a large memory footprint and irregular memory access patterns. Figure 5 shows the reuse pattern of cache blocks on the LLC for the workloads listed in Table 1. The X-axis represents the number of times a cache block has been reused (i.e., hit count) on the LLC until it is evicted. For example, a reuse count of zero indicates that a cache block is never reused within the LLC; such a block is referred to as a dead block [30]. As shown in the figure, 77% of cache blocks are dead, and 90% of cache blocks are reused no more than just once.

The prevalence of dead blocks makes the LLC ineffective, especially for memory-intensive workloads, leading to the wastage of LLC capacity. To demonstrate the performance impact of the LLC capacity allocated to *data blocks* (i.e., cache blocks containing normal data), we measured the execution time while varying the number of ways allotted to the normal data in the LLC. Conventional LLC stores both normal data and PSEs. However, in this experiment, we employed a separate virtual buffer to store all *PSE blocks* (i.e., cache blocks containing PSEs) to eliminate any performance impact caused by the PSE blocks. As shown in Figure 6, most workloads experience a slight increase in the execution time with the reduced cache ways allocated to the data blocks. Compared with the baseline (16 ways), the execution time increases by an average of only 1.9% and 2.9% when the number of cache ways allocated to the data blocks is reduced to 10 and 8, respectively. In the workloads with a modest memory footprint (such as leslie3d and soplex), reducing the LLC capacity significantly impacts performance, as their working set is mostly accommodated in the LLC. However, in memory-intensive workloads with a large working set (such as lbm, tc-urand, sssp-urand, and gups), reducing the LLC capacity rarely affects the performance because LLC is ineffective due to the frequent capacity misses. *This observation implies that the performance impact of the cache capacity allocated to the normal data blocks is small, motivating to store more PSE blocks in the LLC instead of normal data blocks.*

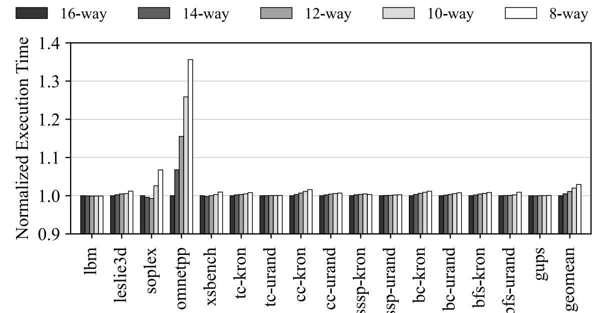

FIGURE 6. Performance impact of LLC ways allocated to data blocks. Most memory-intensive workloads are less sensitive to the LLC capacity.

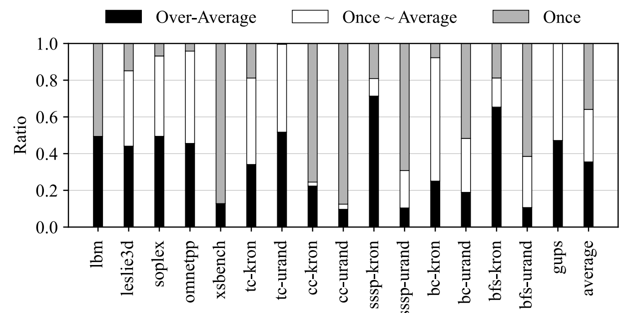

FIGURE 7. Access pattern of PSE blocks.[1] On average, 37.5% of PSE blocks are frequently re-fetched from the main memory, while 38% of PSE blocks are fetched only once.

D. ACCESS PATTERN OF PSE BLOCKS

As described in Section III-B, the main memory accesses account for more than half of the page walk latency. Therefore, we need to minimize the main memory accesses during the page walk by keeping PSEs in the LLC as much as possible. While pursuing this goal, it is necessary to balance the number of data and PSE blocks in the limited LLC space. If we reduce the space allocated for data blocks too much in order to accommodate more PSE blocks, the performance may suffer significantly.

To determine the PSE blocks that must be stored in the LLC, we analyze the access patterns in fetching the PSE blocks from the main memory and classify PSE blocks into three groups based on their access pattern. As shown in

[1]The "Average" in the legend is the average number of PSE blocks re-fetched from the main memory for each workloads.

Figure 7, the number of accesses to the main memory varies significantly across PSE blocks. On average, 37.5% of PSE blocks are fetched frequently from the main memory, while 38% of the PSE blocks are fetched only once. *This observation on the skewed access counts of the PSE blocks motivates the techniques for identifying the PSE blocks that are frequently re-fetched from the main memory and pinning those PSE blocks to the LLC.*

IV. PINNING THE PSE TO LLC

A. PSE PINNING

This study focuses on mitigating page walk latency to reduce address translation overhead. To achieve this goal, we propose a novel mechanism called PSE Pinning (PSP). This mechanism sticks *hot PSE blocks* (i.e., PSE blocks frequently re-fetched from the main memory) in the LLC, allowing the address translation to be completed without accessing the main memory. PSP operates in three basic steps. First, it detects program phases where the TLB misses frequently occur, and the performance is not sensitive to the LLC capacity. Second, the hot PSE blocks that cause frequent main memory accesses are identified. Since the LLC capacity is limited, holding all PSE blocks in the LLC is impractical. Therefore, the PSP carefully identifies the most frequently reused PSE blocks inside the LLC based on the observed PSE access patterns. Finally, the LLC is configured to allocate a portion of its storage capacity dedicated only to the PSE blocks. Depending on the frequency of the LLC and TLB misses, the ratio of the LLC space dedicated to the PSE blocks is dynamically adjusted to appropriately balance the cache spaces between the data block and PSE blocks.

Figure 8 shows the overall operating flow of PSP. If a page walk misses on the PSC, the page walker looks for PSEs in the L1 and L2 caches. If it fails to find the requested PSEs in those caches, it searches them in the cache space dedicated to the PSEs in the LLC (❶). In the baseline system, it is highly likely that the LLC does not have the desired PSEs. However, if frequently reused PSE blocks are pinned to the LLC, most address translations can be completed without accessing the main memory (❷).

To effectively utilize the limited LLC capacity, it is necessary to identify the PSE blocks that cause frequent memory accesses. To this end, the number of main memory accesses per PSE block is tracked. When installing a PSE block to the LLC, the block is pinned to the dedicated LLC space if it has already been brought from memory several times (❸). In the conventional LLC replacement policy, a victim block is selected without distinguishing between the data and PSE blocks. However, our proposed PSP excludes the pinned PSE blocks in the victim selection because they are expected to be reused soon.

Adjusting the dedicated LLC space to the PSEs is essential for efficient pinning with low-performance impact. To this end, the PSP determines the appropriate program phase for pinning the PSEs by profiling the LLC and L2 TLB

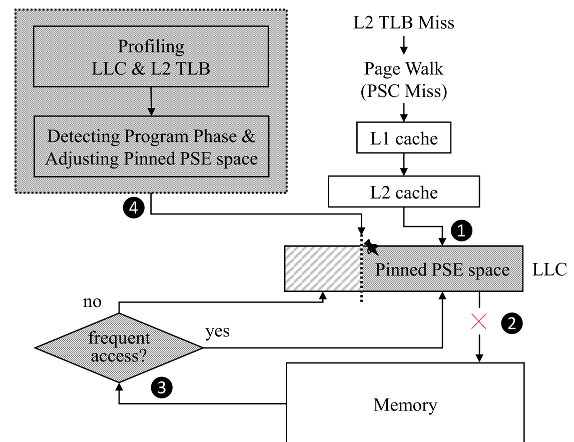

FIGURE 8. Overview of PSP.

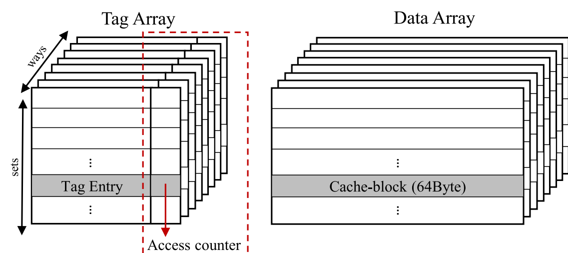

FIGURE 9. LLC structure with access counters. PSE Pinning uses an access counter per a cache block to select PSE blocks to pin.

misses. Subsequently, it dynamically adjusts the LLC space dedicated to the pinned PSE blocks per cache set (❹). For the program phase where both TLB and LLC are not effective (i.e., frequent misses on both LLC and TLB), PSP allocates more space for the PSE blocks. In contrast, for the phase of the program where TLB and LLC operate effectively, PSP reduces the LLC space allocated to the PSE blocks.

B. SELECTING PSE TO PIN

As described in Section III-D, on average, only 37.5% of PSE blocks are frequently fetched from the main memory. If these blocks are frequently evicted from the LLC due to the conflict misses and fetched again from the expensive off-chip main memory, the system performance will be degraded due to a slow address translation. This subsection describes how the PSP mechanism detects the PSE blocks that have to be pinned in the LLC.

PSP employs an *8-bit access counter* per a PTE block to identify the PSE blocks that are frequently read from the main memory. A simple way to maintain the access counter per a PTE block is to extend the tag array of the LLC to include an access counter per cache block, as shown in Figure 9. The access counter tracks the number of main memory accesses for the corresponding cache block in the LLC. When a cache block is read from the main memory and installed in the LLC, the access counter is incremented. A cache block with a counter value of 1 indicates that it has been read from the main memory only once, whereas a cache block with a

FIGURE 10. PSE formats of Intel's 5-Level paging scheme [46].

FIGURE 11. Storing an access counter in PSEs. An access counter is interleaved across eight PSEs within a PSE block.

large counter value indicates that it has frequently been read from the main memory. When hot PTE blocks (frequently read from the main memory) are successfully identified and pinned to the LLC, the page walk latency can be reduced significantly because address translation can avoid excessive main memory accesses.

Adding an 8-bit access counter per cache block in the LLC incurs significant hardware overheads. To avoid hardware costs, we store the access counter in the PSE's existing unused fields. Figure 10 depicts the format of five different types of page structure entries (i.e., PML5E, PML4E, PDPTE, PDE, and PTE) in the five-level paging scheme [46]. The length of all PSE types is 64 bits, and each field in the entry specifies specific information used in the address translation. In the figure, we can observe that all PSE type has "Ignored" and "Reserved" fields that are not used in the current virtual memory systems. Even without using these unused bits, we can store a 48-bit base address of the PST in a PSE, resulting in a maximum memory capacity of 1024 PB, which is significantly larger than the typical memory capacity of modern server systems.

By utilizing the "Ignored" bits, it is possible to secure up to 11 bits. Furthermore, when Reserved bits are combined with Ignored bits, the maximum number of bits that can be utilized is further increased. In addition, since a PSE block contains eight PSEs, each bit of the 8-bit access counter for a PSE block can be distributed across the eight PSEs within the block, as shown in figure 11. Therefore, only one unused bit of a PSE is used to accommodate the 8-bit access counter for a PSE block.

FIGURE 12. Program phase of bfs-kron.

C. DYNAMIC PINNING

PSP dynamically adjusts the LLC space through program phase recognition. To determine the program phase, PSP profiles LLC miss rate and L2 TLB MPKI during the program execution and utilizes them as reference factors. The LLC miss rate can be used to determine the locality of memory references, and the L2 TLB MPKI is closely related to page walk frequency. Figure 12 shows the L2 TLB MKPI and LLC miss rate during program execution for the bfs-kron workload. Both TLB miss counts and LLC utilization exhibit various patterns during program execution and can be utilized as indicators of the program phase.In this subsection, we present a process of analyzing TLB and LLC misses to find the program phases where the LLC is ineffective, and the L2 TLB suffers from frequent misses. PSP classifies the program phases into four classes, as shown in Table 2. If the LLC miss rate is high and there are many L2 TLB misses during a phase, the PSE blocks can be safely pinned in the LLC without any impact on the performance. This phase is referred to as *Strong Pinning* where PSE blocks need to be pinned aggressively in the LLC. Conversely, if there are a few misses on LLC and L2 TLB during a phase, it is not required to pin the PSEs in the LLC, and sometimes it is necessary to remove the pinned PSEs from the LLC. This phase is referred to as *Pinning Out*.

As described above, the program phase is classified according to the level (high or low) of reference factors (L2 TLB MPKI and LLC miss rates). Thus, PSP collects the LLC miss rate and L2 TLB MPKI at regular intervals and determines the level of reference factors as high or low. The reference factors' level (i.e., high or low) can be determined only when the corresponding comparison target is available. The simplest and most precise way to obtain the comparison target is to accumulate the events throughout the entire execution. However, continuous collection of the LLC miss rate and L2 TLB MPKI during the entire execution of a program can incur high overheads in the storage space.

To minimize the overheads required to detect the program phase, we define a *local value (LV)* storing the reference factor for the current interval of program execution and a *global value (GV)* that is updated with the *LVs* during the entire program execution. The program phase is identified by evaluating the changes of *LV* and *GV*.

$$GV_{n+1} = \frac{LV + GV_n}{2} \tag{1}$$

TABLE 2. Program phase classification.

Program Phase	LLC Miss Rate	L2 TLB MPKI
Strong Pinning	H	H
Weak Pinning	H	L
Keeping	L	H
Pinning Out	L	L

Equation 1 is used to calculate the *GV*. The *LV* and GV_n of the current interval determines the next GV_{n+1}. To determine the level of a reference factor, GV_{n+1} is compared with *LV*. If *LV* is greater than GV_{n+1}, the level of the corresponding reference factor is considered "high". Otherwise, the level is deemed to be "low". With the determined level of the reference factors (i.e., L2 TLB MPKI and LLC miss rates), a program phase is classified.

According to the identified program phase, PSP adjusts *pinning threshold* to control the maximum number of pinnable PSE blocks in a cache set. PSP allocates more LLC space to the PSE blocks when the pinning threshold is less than the number of currently pinnable PSE blocks for a set. To this end, when evicting a victim from a physical cache line, the line is deactivated so that it is not selected to accommodate a new data block. The deactivated line is then only utilized to store the PSE blocks.

D. IMPLEMENTATION

This section describes hardware extensions and algorithms for PSP. To determine the program phase, we need a process that collects the LLC miss rate and L2 TLB MPKI) at a specific interval (defined as a number of committed instructions). To this end, we use three 32-bit counters, each of which counts LLC misses, LLC accesses, and L2 TLB misses, respectively. When a miss occurs on the LLC and the L2 TLB, the corresponding counter is incremented by 1. For the LLC, the number of accesses should be counted as well as the number of misses to calculate the miss rates. Each counter value is initialized at every interval. By default, we set the interval to 10 million instructions to avoid too frequent initialization.

Modern processors typically employ performance measurement facilities that profile the various hardware event [47], [48]. Consequently, the continuous profiling of the LLC and L2 TLB events can be implemented with minimal hardware overhead by utilizing those performance measurement facilities (e.g., Performance counters on Intel processors [47]), which are already incorporated into the modern processors.

Algorithm 1 describes a mechanism that classifies the program phase and adjusts *Pinning threshold*. For the phase classification, the local value of the LLC miss rate (*lx*) and that of the L2 TLB MPKI (*ly*) are collected for the current interval, and then used to update their global value (*gx* and *gy*). The phase classification is basically conducted by checking whether the local value is relatively greater than the global one. If both *lx* and *ly* are greater than their global value

Algorithm 1 Updating Pinning Threshold

Input : profiling result for specific interval
lx ← LLC miss rate, ly ← L2 TLB MPKI
Output : pinning_threshold

function compare (local, global)
  if local ≥ 1.05 x global **then**
    return HIGH
  else if local ≤ 0.95 x global **then**
    return LOW
  end if
  return IGNORE
end function

Initialization :
  gx ← compute_gv (lx), gy ← compute_gv (ly)
  /∗ compute global value with eq.(1) ∗/
  x ← compare(lx,gx), y ← compare(ly,gy)

1: **if** lx > standard_mr **AND** ly > standard_mpki **then**
2:   **if** x is HIGH **AND** y is HIGH **then**
3:     increase pinning_threshold by 2 /∗ Strong Pinning ∗/
4:   **else if** x is HIGH **then**
5:     increase pinning_threshold by 1 /∗ Weak Pinning ∗/
6:   **else if** x is LOW **AND** y is LOW **then**
7:     decrease pinning_threshold by 1 /∗ Pinning Out ∗/
8:   **end if**
9: **else if** lx < standard_mr **OR** ly < standard_mpki **then**
10:   decrease pinning_threshold by 1
11: **end if**

(*gx* and *gy*), the program phase is determined as *Strong Pinning*. If only *lx* is greater than *gx*, the program phase is determined as *Weak Pinning*. If both *lx* and *ly* are smaller than their global value (*gx* and *gy*, respectively), the program phase is determined as *Pinning Out*. In this process, the local value is considered to be greater (or smaller) than the global one only when their gap is higher than a specific rate (i.e., 5% in our default configuration). Algorithm 1 also compares the *lx* and *ly* with a standard miss rate (denoted by *standard_mr*) and a standard L2 TLB MPKI (denoted by *standard_mpki*), respectively, as shown in line 1. By referring to these standard values, the algorithm can prevent unnecessarily pinning PTE blocks to the LLC in the program phase where the LLC miss rate or L2 TLB MPKI is low. The *standard_mr* and *standard_mpki* are obtained by offline profiling that collects the LLC miss rates and L2 TLB MPKIs for all target workloads and calculates the averages of the collected statistics. In this paper, we obtained the *standard_mr* and *standard_mpki* for all workloads listed in Table 1. Since the standard values can be different depending on the target workloads, it is required to provide a facility to update them. This

Algorithm 2 Replacing PSE Blocks

Input : *blk_access_cnt* is the number of accesses for a PSE block *pin_cnt* is the number of pinned blocks in current set
Initialization : pin ← false

1: **if** *blk_access_cnt* > hot_threshold **then**
2: pin ← true
3: increase *pin_cnt* by 1
4: **end if**
5: **if** *pin_cnt* > pinning_threshold **then**
6: remove pin for pinned blocks by replacement policy
7: **end if**
8: find victim for no-pinned blocks by replacement policy
9: update cache block using pin state (true or false)

can be easily implemented, as will be discussed later in this section.

After phase classification, the pinning threshold is adjusted to control the maximum number of PSE blocks pinned to the LLC according to the program phase. For example, in the "Strong Pinning" phase, the pinning threshold is set to a high value to allocate more space for the PSEs. With a high pinning threshold, PSP aggressively pins more PSEs in the LLC. In the opposite case (i.e., "Pinning Out" phase), where both LLC miss rate and L2 TLB MPKI are low, the pinning threshold is decremented to reduce the maximum number of PSEs that can be pinned in the LLC space. We empirically determine the upper bound of the pinning threshold to 14 blocks. By limiting the maximum pinning threshold, we can prevent all LLC spaces are occupied by the PSE blocks.

Algorithm 2 shows an LLC replacement mechanism that selects a victim while considering the pinned PSEs. In the baseline LLC, the replacement mechanism selects a victim among all cache blocks in a corresponding set. However, in the LLC with PSP, the replacement mechanism excludes the pinned PTE blocks from the victim candidates.

When installing a PSE block in the LLC, PSP determines whether or not the block is a hot PSE block that needs to be pinned in the LLC. For detecting the hot PSE blocks, we define a *hot-PSE threshold* (denoted by *hot_threshold*) which is compared to the access counter of the PSE block fetched from the main memory. Suppose the access counter of the PSE block (denoted by *block_access_cnt*) is greater than a hot-PSE threshold. In that case, the block is determined to require pinning in the LLC.[2]

In the case where the newly installed PSE block is determined to be pinned, the number of pinned blocks in the corresponding cache set can exceed its upper bound. To avoid this situation, the total number of pinned blocks is compared to the pinning threshold whenever a PSE block is pinned. If the total number of pinned PSE blocks is greater than the pinning threshold, one of the PSE blocks in the set is selected by using the cache replacement policy and converted to a normal PSE block. And then, a victim block is selected among data blocks or PSE blocks that are not pinned.

To implement PSP, two additional bits are added to each tag entry. One is used to distinguish PSE blocks and normal data blocks, while another is used to determine whether a cache block is pinned or not. These additional bits may occupy a negligible amount of LLC area (less than 1%). In addition, we need a facility to update the standard values for adapting the algorithm to various target workloads. This requirement can be easily met by storing the standard values in the processor's registers. The processor contains a variety of control registers that can be used to control the behavior of its various components. To store the standard values in the processor's registers, we can either add new control registers to the processor or utilize the free bits of the existing control registers (e.g., the EFER register [11]).

[2]Since approximately 40% of the PSE blocks are fetched from the main memory only once (shown in Figure 7), it is sufficient to set the hot PSE threshold to a small value for selecting the PSE blocks to be pinned.

TABLE 3. Simulated system configuration.

CPU	4-wide Out of Order, 4.0GHz
L1 ITLB	64-entry, 4-way, 1-cycle, 4-entry MSHR
L1 DTLB	64-entry, 4-way, 1-cycle, 4-entry MSHR
L2 TLB	1536-entry, 12-way, 8-cycle, 4-entry MSHR, 1 page walk / cycle
Page Structure Caches	4-level Split PSC, 2-cycle PML5: 2-entry, fully; PML4: 4-entry, fully, PDP: 8-entry, 4-way; PD: 32-entry, 8-way
L1 I-cache	32KB, 8-way, LRU, 4-cycle, 8-entry MSHR
L1 D-cache	32KB, 8-way, LRU, 5-cycle, 8-entry MSHR
L2 Cache	256KB, 8-way, LRU, 10-cycle, 16-entry MSHR
LLC	2MB, 16-way, DRRIP [49], 20-cycle, 64-entry MSHR
DRAM	3200MHz, tRP=tRCD=tCAS=13.75ns

V. EXPERIMENTAL METHODOLOGY

A. WORKLOADS

We use 16 memory-intensive workloads from SPEC CPU 2006, XSBench, GAP benchmark suites, and HPCC for our evaluations. Table 1 describes the workloads and their respective memory footprint. Among various graph inputs in the GAP benchmark suite, we conduct experiments with kron and urand inputs because they are relatively large and cause graph algorithms to have irregular memory accesses. Additionally, the workloads are classified into three groups according to the L2 TLB misses per kilo instructions (L2 TLB MPKI): Low (L-group), Mid (M-group), and High (H-group). Because the workloads belonging to M-group and H-group cause frequent L2 TLB misses, they are suitable for evaluating the performance improvement with PSP. Even though L-group is not sensitive to the page walk latency since they accomplish most address translations in the TLB, we use them to verify whether PSP does not degrade the system performance for the workloads with low TLB misses.

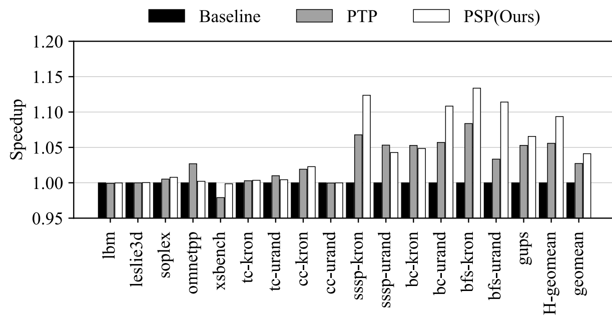

FIGURE 13. Performance.

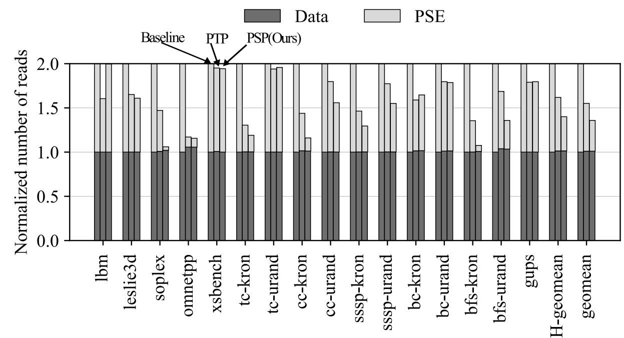

FIGURE 14. Normalized number of read requests to the main memory for fetching normal data and PSEs.

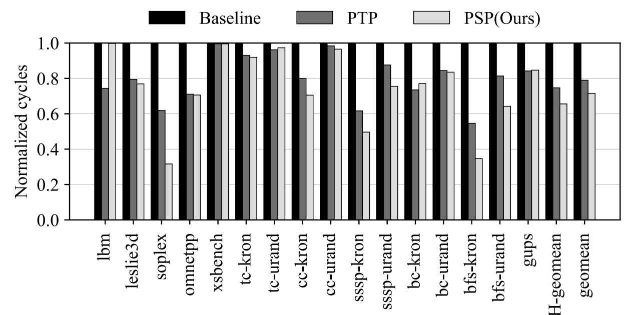

FIGURE 15. Page walk latency.

B. SIMULATION

We use ChampSim [50], a trace-based simulator modeling the 4-wide out-of-order processor, for our evaluation. ChampSim includes PSCs as well as L1 and L2 TLBs, and it models the page walk on the 5-level radix page table. We modified ChampSim to implement the proposed PSP and a competitive technique called Page Table Prioritization (PTP) [18]. PTP prioritizes the PTE blocks to keep them in the cache when the TLB miss rate is high. With a high probability (e.g., 99%), PTP selects a victim among data blocks, which allows more PTE blocks can reside in the cache. Although PTP has a similar goal to PSP, PTP uses an empirical probability with a simple phase detection, and it does not provide a mechanism to distinguish hot PSE blocks. Workload trace files are generated using the SimPoint methodology [51]. For every workload, we warm up the simulated system for 250 million instructions and execute 200 million instructions in detailed mode. Table 3 summarizes the system configurations used in our evaluations.

VI. SIMULATION RESULTS

A. PERFORMANCE

Figure 13 shows the performance improvement of the proposed PSP mechanism when compared to the baseline and PTP (cache prioritization) [18]. Since we focus on reducing the page walk overhead, we compare the performance gains of the PSP and PTP for H-group workloads separately. In Figure 13, H-geomean is the geometric median for H-group workloads, while geomean is the geometric median for all workloads.

Our PSP mechanism achieves an average 4% performance improvement for all 16 workloads when compared to the baseline. For five H-group workloads (bc-kron, bc-urand, bfs-kron, bfs-urand, and gups), which are the target workloads of the PSP mechanism, the performance improvement with PSP is 7.8% on average. Even for M-group workloads, PSP achieves a performance improvement of more than 2.6% on average. However, because the L-group workloads have high hit rates on the L1 and L2 TLBs, there is trivial performance improvement when employing either PSP or PTP mechanisms. Our PSP outperforms PTP for most workloads. In particular, it provides much higher performance gains than PTP for the H-group workloads. This higher performance gain of PSP compared with that of PTP is mainly due to its efficient pinning mechanism. The goal of PTP is similar to that of PSP, but PTP only provides a static prioritization method that gives a higher priority to all PTE blocks.

B. MEMORY ACCESS

PSP reduces the number of main memory accesses required for fetching PSEs. Figure 14 shows the normalized number of read requests to the main memory for fetching PSEs and normal data. Overall, both PTP and PSP increase the memory accesses for fetching normal data by only 1% on average. While the PSP slightly increases the memory access for the normal data, it significantly reduces the memory access for the PSEs. With PSP, the memory access for the PSEs is reduced by 65.1% on average. PTP, our competitive technique, reduces the memory access for the PSEs by 45% on average.

By avoiding the memory accesses involved in the page walk, PSP significantly reduces the page walk latency as shown in Figure 15. PSP reduces the page walk latency by 28.4% on average. For bfs-kron, PSP reduces page walk latency by 65.3%, which is 19.9% greater than the reduction achieved with PTP. Compared to PTP, PSP provides a 7.4% higher reduction in the page walk latency on average.

C. ENERGY CONSUMPTION

Figure 16 shows the normalized energy consumption of the memory system. To evaluate the impact of the PSP on energy consumption, we extracted the parameters regarding

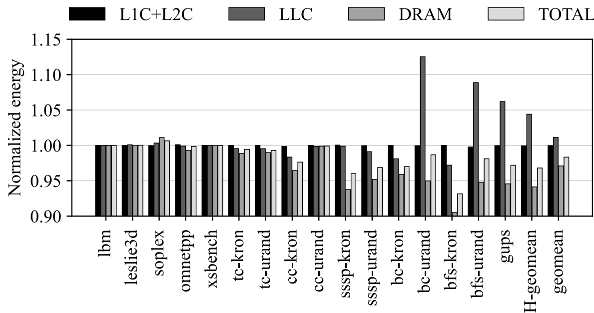

FIGURE 16. Dynamic energy consumption of caches and main memory (DRAM). Results are normalized to the baseline.

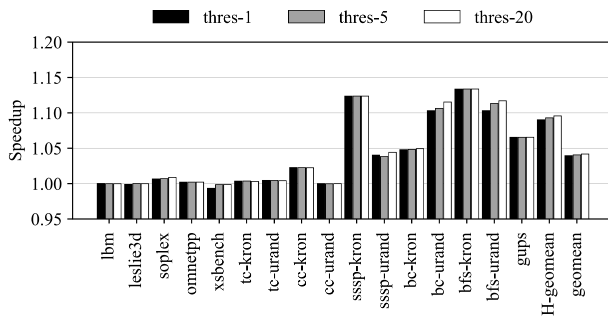

FIGURE 17. Impact of hot-PSE threshold.

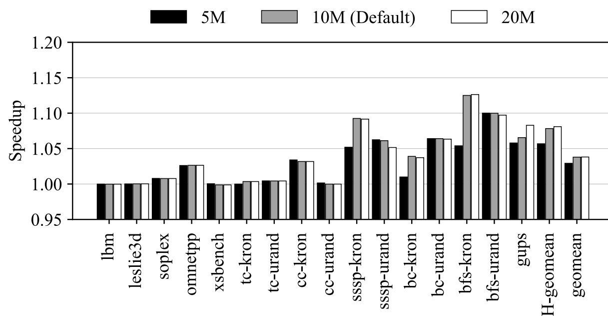

FIGURE 18. Impact of profiling interval.

the energy consumption of the memory system, including on-chip caches and the main memory, by using CACTI [52]. Then, we use the extracted parameters to estimate the energy consumption of the memory system in the system-level simulation with the Champsim. As shown in the figure, PSP reduces the total energy consumption of the memory system by 1.6% on average. This energy-saving is mainly achieved by reduced access to the main memory. Even if the PSP increases the energy consumption of the LLC due to the increased hits when accessing the PSE blocks, it significantly reduces the energy consumption of the main memory, which is a major contributor to the energy consumption.

D. SENSITIVITY ANALYSIS

In this section, we analyze the performance impact of PSP according to various system configurations.

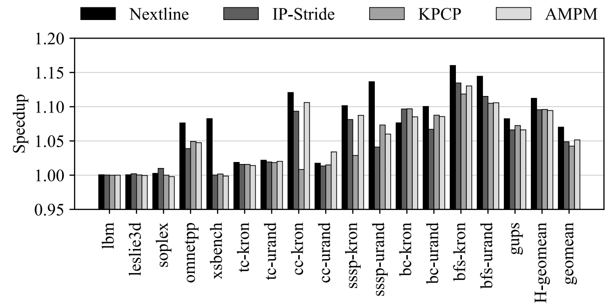

FIGURE 19. Impact of LLC prefetcher.

1) IMPACT OF HOT PSE THRESHOLD

First, we compare the performance impact of PSP according to the *hot-PSE threshold* used in Algorithm 1. The hot-PSE threshold controls the aggressiveness of. With a low hot-PSE threshold, PSP pins more PSE blocks in the cache, which reduces the percentage of the data blocks in the cache. Increasing the hot-PSE threshold reduces candidate PTE blocks for pinning. Thus, with a large hot-PSE threshold, PSP conservatively selects the PSE blocks to be pinned in the cache.

As shown in Figure 17, the performance difference across various hot-PSE thresholds is very small. For some workloads, PSP with a hot-PSE threshold of 1 can deliver almost optimal performance. This is because PSP pins the PSE blocks in the program phase, where the LLC does not effectively provide the data blocks because most of the data blocks are not reused in the cache. In addition, a large number of the PSE blocks (62% on average) are fetched from the main memory more than once as shown in Figure 7. For some benchmarks, such as bc-urand and bfs-urand, that have a relatively small number of hot PSE blocks, PSP achieves better performance when conservatively pinning the PSE blocks in the cache by using a high hot-PSE threshold.

2) IMPACT OF PROFILING INTERVAL

Second, we evaluate the performance impact of profiling intervals in PSP. Figure 18 shows the performance improvement of PSP for various profiling intervals: 5M (Million instructions), 10M, and 20M. Overall, PSP with long profiling intervals delivers better performance for the H-group workloads. In the case of sssp-kron, however, the performance of PSP is sensitive to the profiling interval. This result implies that it is required to determine an appropriate profiling interval for each workload or dynamically adjust the profiling interval during the program execution.

3) IMPACT OF CACHE PREFETCHER

Third, we evaluate the performance improvement with PSP for various LLC prefetchers: Next-line prefetcher, Instruction Pointer-based stride prefetcher (IP-stride) [53], Kill the Program Counter Prefetcher (KPCP) [54], and Access Map Pattern Matching prefetcher (AMPM) [55]. When employing a prefetcher, PSP delivers better performance than the

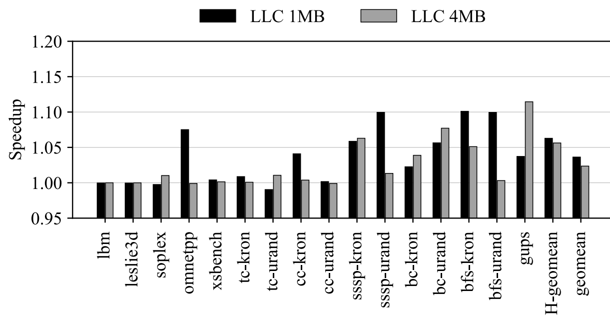

FIGURE 20. Impact of LLC size.

configuration without the prefetcher. As shown in Figure 19, PSP improves the performance by 7%/4.8%/4.2%/5.1% with Next-line/IP-stride/KPCP/AMPM, respectively, on average. For the H-group workloads, PSP with a prefetcher achieves more than 9% performance improvement on average.

4) IMPACT OF LLC SIZE

Finally, we evaluate the impact of LLC size on performance. Figure 20 shows the performance gain of PSP according to the LLC size. Overall, PSP achieves higher performance when the LLC size is relatively small because it pins the PSE blocks in the LLC for the program phase where the LLC miss rate is high. For example, in the case of omnetpp and sssp-kron, the LLC miss rate increases by more than double when the LLC size is reduced by half (i.e., 1MB), meaning that the small LLC does not operate effectively. By aggressively pinning the PSE block for those workloads, PSP achieves significant performance gain. In contrast, for gups, the PSP provides better performances for the configuration with a larger LLC. This is because gups have a completely random access pattern, and as can be seen in Figure 4, most of the PSE access latency is contributed by the main memory accesses. For the workloads with random memory access patterns, increasing the LLC size does not help to improve the performance, meaning most LLC space is not effectively utilized. However, when PSP is applied, more PSE blocks will be pinned in the enlarged LLC, leading to a significant reduction in the page walk latency.

VII. RELATED WORK

In this section, we summarized prior works about reducing the address translation overhead in virtual memory systems.

A. HASHED PAGE TABLE

In order to avoid the page walk latency, we can use a hashed page table [31] that store the PTEs in a hash table. Even if the hashed page table can provide a PTE with a single memory access, it can suffer from the hash collision. Skarlatos et al. [33] proposed Elastic Cuckoo Hashing, an algorithm for gradual resizing, to reduce hash collision overhead.

B. MULTIPLE PAGE SIZE

Several recent studies have attempted effective address translation through using various page sizes: small pages (4KB page) and large pages (e.g., 2MB page) [22], [23], [24], [25], [26]. Using a large page is an efficient way to increase the TLB reach, reducing the TLB misses. However, the efficiency of this approach is highly dependent on the memory access pattern of the workloads. If there is an insufficient locality in memory references, using large pages may result in fragmentation, page fault latency, and paging traffic overhead.

C. TLB PREFETCHING

TLB prefetching mechanisms have been studied to reduce TLB misses [15], [28], [29]. If the prefetching decision is correct, we can reduce the TLB misses by prefetching the TLB entries in advance. Otherwise, these mechanisms can significantly reduce performance because they pollute the TLB. The TLB prefetch mechanism is orthogonal to our PSP and is thus used synergistically.

D. USING MEMORY AS LARGE TLB

A recent study proposes a large virtual TLB called POM-TLB that uses a memory region to store TLB entries [34]. POM-TLB converts frequent memory accesses involved in the page walks into one memory access by maintaining TLB entries in the memory. POM-TLB can place most address translation information by using a 16MB space of the main memory. However, POM-TLB still has difficulties with expensive off-chip memory access. To reduce the off-chip memory accesses in the POM-TLB, CSALT [35], a caching mechanism for POM-TLB, was proposed. CSALT provides a dedicated on-chip cache space to store the frequently and recently accessed POM-TLB entries.

E. PRIORITIZING PTE IN CACHE

Park et al. [18] proposed the flattened page table (FPT) and page table prioritization (PTP). FPT flattens the page table level to reduce the number of memory accesses during the page walk. PTP provides a high priority to the PSE blocks so that they can reside in the cache. PSP differs from the PTP in three folds. First, PSP keeps the PSE blocks within the cache if they are frequently re-fetched from the main memory. Therefore, PSP can store the frequently accessed PSE blocks with a long reuse distance, even if they are not recently and frequently accessed on the cache. Second, PSP maintains the access history (i.e., access count) for all PSE blocks. By utilizing the history, PSP can correctly determine the PSE blocks that should be pinned to the cache. However, PTP relies only on the cache replacement policy to determine the PSE blocks to be stored in the cache. Third, PSP dynamically adjusts the ratio of pinned PSE blocks and normal blocks. By contrast, PTP uses an empirically determined static ratio. Therefore, the PTP's decision could be sub-optimal because it can fail to take into account the dynamic behavior of workloads.

VIII. CONCLUSION

This study proposed a PSP mechanism that pins the page structure entry in the LLC to reduce the page walk overheads.

By repurposing a part of LLC space as a large PSE storage in the program phase, where both LLC and TLB suffer from frequent misses, PSP reduces the page walk latency by up to 65.3% and achieves an average speedup of 7.8% for the memory-intensive workloads with irregular access patterns. PSP mechanism can be implemented with trivial hardware overheads by storing metadata in the unused bits of the PSE. As emerging applications are expected to have a larger memory footprint and irregular memory access patterns, PSP is likely to be an essential solution for virtual memory systems.

REFERENCES

[1] R. Casado and M. Younas, "Emerging trends and technologies in big data processing," *Concurrency Comput., Pract. Exper.*, vol. 27, no. 8, pp. 2078–2091, Jun. 2015.

[2] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from internet services," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 488–499.

[3] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, *Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 316–331, doi: 10.1145/3173162.3173177.

[4] J. Barr. (2017). *Ec2 In-Memory Processing Update: Instances With 4 to 16 TB of Memory + Scale-Out Sap Hana to 34 TB*. [Online]. Available: https://aws.amazon.com/blogs/aws/ec2-in-memory-processing-update-instances-with-4-to-16-tb-of-memory-scale-out-sap-hana-to-34-tb

[5] A. Bhattacharjee and D. Lustig, "Architectural and operating system support for virtual memory," *Synth. Lectures Comput. Archit.*, vol. 12, no. 5, pp. 1–175, 2017.

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

[7] A. Bhattacharjee, "Preserving virtual memory by mitigating the address translation wall," *IEEE Micro*, vol. 37, no. 5, pp. 6–10, Sep. 2017.

[8] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3S, pp. 158–169, Jan. 2016.

[9] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 37–48, 2012.

[10] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 643–656.

[11] Intel. (2022). *Intel® 64 and IA-32 Architectures Software Developer Manuals*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[12] Intel, "5-level paging and 5-level EPT," White Paper 335252-001, May 2017. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html

[13] P. Guide, "Intel® 64 and IA-32 architectures software developer's manual," *Volume 3B Syst. Program. Guide*, vol. 2, no. 11, pp. 152–165, 2011.

[14] AMD, "AMD64 architecture programmer's manual volume 2: System programming," AMD Pub, 2022, vol. 24593.

[15] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 1023–1036.

[16] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proc. 13th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS XIII)*, 2008, pp. 26–35.

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

[18] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, "Every walk's a hit: Making page walks single-access cache hits," in *Proc. 27th ACM Int. Conf. Architectural Support for Program. Lang. Operating Syst.*, Feb. 2022, pp. 128–141.

[19] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO-46)*, 2013, pp. 383–394.

[20] A. Bhattacharjee, "Translation-triggered prefetching," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2017, pp. 63–76.

[21] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 558–567.

[22] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 435–448, May 2017.

[23] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on *NUMA* systems," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2014, pp. 231–242.

[24] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Ingens: Huge page support for the OS and hypervisor," *ACM SIGOPS Operating Syst. Rev.*, vol. 51, no. 1, pp. 83–93, Sep. 2017.

[25] T. Michailidis, A. Delis, and M. Roussopoulos, "MEGA: Overcoming traditional problems with OS huge page management," in *Proc. 12th ACM Int. Conf. Syst. Storage*, May 2019, pp. 121–131.

[26] A. Panwar, S. Bansal, and K. Gopinath, "HawkEye: Efficient fine-grained OS support for huge pages," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 347–360.

[27] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, Jun. 2015, pp. 66–78, doi: 10.1145/2749469.2749471.

[28] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative TLB for chip multiprocessors," *ACM SIGPLAN Notices*, vol. 45, no. 3, pp. 359–370, Mar. 2010.

[29] G. Vavouliotis, L. Alvarez, V. Karakostas, K. Nikas, N. Koziris, D. A. Jimenez, and M. Casas, "Exploiting page table locality for agile TLB prefetching," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 85–98.

[30] C. Mazumdar, P. Mitra, and A. Basu, "Dead page and dead block predictors: Cleaning TLBs and caches together," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2021, pp. 507–519.

[31] B. L. Jacob and T. N. Mudge, "A look at several memory management units, TLB-refill mechanisms, and page table organizations," *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 295–306, Nov. 1998.

[32] I. Yaniv and D. Tsafrir, "Hash, don't cache (the page Table)," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 1, pp. 337–350, Jun. 2016.

[33] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2020, pp. 1093–1108.

[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 Comput. Archit. News*, vol. 45, no. 2, pp. 469–480, 2017.

[35] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, "CSALT: Context switch aware large TLB," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2017, pp. 449–462.

[36] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *Proc. 39th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2012, pp. 476–487.

[37] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[38] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," 2015, *arXiv:1508.03619*.

[39] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "Xsbench-the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *Proc. The Role React. Phys. Toward Sustain. Future (PHYSOR)*, 2014, pp. 1–10.

[40] 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 *Proc. ACM/IEEE Conf. Supercomput.*, vol. 213, pp. 118877–1188455, Dec. 2006.

[41] S. Eyerman, W. Heirman, K. Du Bois, J. B. Fryman, and I. Hur, "Many-core graph workload analysis," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2018, pp. 282–292.

[42] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, "Enhancing and exploiting contiguity for fast memory virtualization," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 515–528.

[43] S. Kumar, A. Prasad, S. R. Sarangi, and S. Subramoney, "Radiant: Efficient page table management for tiered memory systems," in *Proc. ACM SIGPLAN Int. Symp. Memory Manag.*, Jun. 2021, pp. 66–79.

[44] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, "Mitosis: Transparently self-replicating page-tables for large-memory machines," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2020, pp. 283–300.

[45] M. Dimitrov, K. Kumar, P. Lu, V. Viswanathan, and T. Willhalm, "Memory system characterization of big data workloads," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2013, pp. 15–22.

[46] Intel®. (2018). *Intel® 64 and IA-32 Architectures Software Developers Manual Combined Volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[47] R. D. T. Willhalm. (2010). *Intel Performance Counter Monitor—A Better Way to Measure CPU Utilization*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/performance-counter-monitor.html

[48] (2015). *Amd μprof*. [Online]. Available: https://developer.amd.com/amd-uprof/

[49] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, 2010.

[50] (2022). *The Champsim Simulator*. [Online]. Available: https://github.com/ChampSim/ChampSim

[51] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for accurate and efficient simulation," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 318–319, Jun. 2003.

[52] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 1–25, Jul. 2017.

[53] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surveys*, vol. 32, no. 2, pp. 174–199, Jun. 2000.

[54] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 737–749, May 2017.

[55] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," *J. Instruct.-Level Parallelism*, vol. 13, no. 2011, pp. 1–24, 2011.

OSANG KWON (Graduate Student Member, IEEE) is currently pursuing the master's degree with the Department of Electrical and Computer Engineering, Sungkyunkwan University, South Korea. His current research interests include memory systems, computer architecture, and virtual memory.

YONGHO LEE is currently pursuing the master's degree with the Department of Electrical and Computer Engineering, Sungkyunkwan University, South Korea. His current research interests include heterogeneous memory systems, nonvolatile memory, and computer architecture.

SEOKIN HONG (Member, IEEE) received the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2015. From 2015 to 2017, he was a Senior Engineer at Samsung Electronics. During his two years there, he was involved in a project that developed the 3D-stacked memory. In 2017, he moved to the IBM T. J. Watson Research Center, where he worked on secure processor architectures and emerging memory/storage systems. He is currently an Assistant Professor at Sungkyunkwan University, South Korea. His current research interests include the design of low power, reliable, and high-performance processor architectures and memory systems. He received best paper awards from International Conference on Computer Design (ICCD), in 2010, and Design Automation and Test in Europe (DATE), in 2013.