Operand-oriented Virtual Memory Support for Near-Memory Processing

Duheon Choi, Taeyang Jeong, Joonhyeok Yeom, and Eui-Young Chung, *Member, IEEE,*

Abstract—Virtual memory support is one of the major challenges of near-memory processing (NMP). Many previous works focused on this issue, but there are practical limitations that conventional CPU hardware or memory allocation schemes should be modified. Another technique uses a specialized page table for NMP to avoid such limitations. However, the previous work proposed NMP-specific page table that has static page table walk latency regardless of data size. This causes unnecessarily long address translation time for relatively small data. In this paper, we propose an operand-oriented technique for virtual memory support. Our scheme does not pre-determine the size of shared space; rather, it allocates shared space depending on the size of operands data for NMP. Then, we significantly reduce page table walk latency by using our flexible page table, which adapts the page table hierarchy to the size of shared spaces. To prove our concept, we implement our scheme in a full-system simulator and an FPGA-based verification platform. We then compared it with CPU's page table and the previous NMP-specific page table. The experimental results show that our technique outperforms page table walk latency by 69.3 percent and 43.8 percent compared to the CPU's page table and the comparison, respectively.

Index Terms—Address translation, near-memory processing, page table, processing-in-memory, virtual memory support

1 Introduction

Near memory processing (NMP) is a practical memory-centric computing technique [1], [2]. NMP-based systems place accelerators near the memory, such as the logic layer of 3D-stacked DRAM [3]–[10] and the buffer chip in DIMM module [11]–[15]. These near-memory accelerators (NMACCs) benefit when operating data-intensive applications with high memory bandwidth and reduced data transaction paths. In addition, this memory-centric computing technique can overcome the limitations of conventional processor-centric acceleration by eliminating overhead due to data movement between main memory and accelerator's local memory, power consumption due to the accelerator's memory, and the limited memory footprint of accelerators.

However, when applying NMACCs to traditional computing systems, there is a challenge in supporting the virtual memory of a host process. In modern systems, CPU processes run on a virtual memory system, and operating systems (OSs) map these virtual memories to the physical address space (PAS) of the main memory into page units and store page mapping metadata in a page table. This means that the data are scattered in main memory regardless of their virtual address (VA). However, because the data to be operated by NMACCs are indexed on the virtual address space (VAS), they must resolve the CPU's virtual-to-physical page mapping.

Thus, many NMACCs prototyped in real systems only partially support the virtual memory [6], [11], [15], [16]. They ensure the dedicated PAS by reserving the memory zone for NMP in the booting sequence or registering their local memory in the system memory map. Then, the user allocates virtual memory to NMACC-dedicated spaces through a specific application programming interface (API). This method enables NMACCs to use their dedicated space as their local memory without address translation. However, it cannot be considered to be true memory sharing because the CPU cannot employ the NMACC-dedicated space for other uses. In addition, dedicating address spaces means that NMACCs still have limited memory footprint and memory resource inefficiency, because they can only access these dedicated spaces, which conventional accelerators do. Furthermore, because the workloads accelerated by NMACCs often require large memory footprints, these limitations will obstruct NMACCs from fully exploiting their capabilities.

To overcome these limitations, an NMACC needs to access main memory as the CPU does by resolving the memory mapping of the OS. However, the largest hurdle in this process is the infrastructure of the address translation based on the page table. Duplicating the page table walk in NMACCs has some challenges, such as page table walking latency and compatibility for page table structures across the CPU architectures [2], [3]. Therefore, some previous works avoid operating page table walk in NMACCs by using a contiguous range memory allocation method [17]–[20] or adding hardware modules to the host machine that performs address translation for NMACC [7], [8], [10]. However, these methods are difficult to apply in practice because they require modification of the CPU hardware, such as a translation look-aside buffer (TLB) and page table walker, or adding modules.

Meanwhile, IMPICA [3] proposed the region-based page table (RPT), which enables access to main memory by performing the page table walk in NMACCs. The RPT is an NMP-specific page table only referenced by NMACCs. It only stores page mapping metadata for a pre-determined

• *D. Choi, T. Jeong, J. Yeom, and E.-Y. Chung are with the School of Electrical and Electronic Engineering, Yonsei University, Seoul 03722, Korea.*
E-mail: {cdh0527, drthvbfg, skhds, eychung}@yonsei.ac.kr

Manuscript received September 15, 2022.

part of the virtual memory shared with NMACCs. Therefore, these metadata are stored in both the CPU's page table and the RPT. Then, the CPU and NMACC can access NMACC-shared regions through their respective address translation engines. The RPT solves the compatibility issue and improves the page table walking latency. In addition, an NMP-specific page table enables NMACC to support the virtual memory without modifying the CPU hardware.

However, there remain some issues regarding scalability and page table walk performance. Because the RPT has a structure with reduced address space coverage for improving page table walk performance, there is a limit to the amount of virtual memory that can be shared. In addition, the RPT has a static page table walk latency for its maximum coverage, regardless of the NMACC's memory footprint sizes. However, the memory footprints of accelerators have different shapes depending on the target kernels or different sizes with different input arguments. Although the RPT has improved page table walk latency through reduced hierarchies, the memory footprint diversity of accelerators remains an opportunity to provide improved page table walk latency.

In this paper, we propose a virtual memory support scheme that performs different page table walks according to the memory footprint shapes of NMACC. To address the variety of the accelerator's memory footprints, our scheme divides the NMACC's memory footprint by the unit of an operand. The operand is the data that the accelerator kernel uses during operation, and NMACC and the CPU should share the memory in which operands are allocated. Our scheme uses on-demand shared space declarations per each operand of NMACC. We then propose a flexible page table structure that configures different hierarchies corresponding to the size of shared spaces. We also propose an address translation hardware module for walking our page table. As a result, NMACCs can translate the virtual address with reduced page table walk latency which is different for each operand.

To evaluate our proposal, we first implemented a simple NMACC prototype to which our scheme was applied. We also built a field-programmable gate array (FPGA)-based NMACC verification platform and implemented an NMACC prototype that accelerates the kernel of matrix-vector multiplication to demonstrate that our framework can provide virtual memory support and performance improvements in address translation. In addition, we evaluated our proposal with a full system simulator to see its effects on the other data-intensive kernels.

The key contributions of this study can be summarized as follows:

- We propose an operand-oriented virtual memory support scheme. This enables precise management of the variety of shapes and sizes of NMACC's memory footprint in the virtual memory of the host process.
- We propose a novel NMP-specific page table, which has a flexible page table structure. This provides optimal page table walk latency according to the size of the address space to be covered.
- We verify the effectiveness and functionality of our proposal on real hardware through an FPGA-based verification platform. And we evaluate the performance of our proposal on four kernels in a simulation.

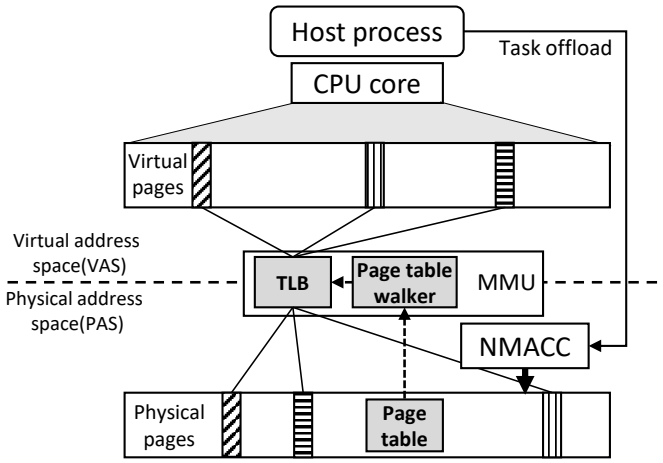

Fig. 1. Virtual memory system of computing system

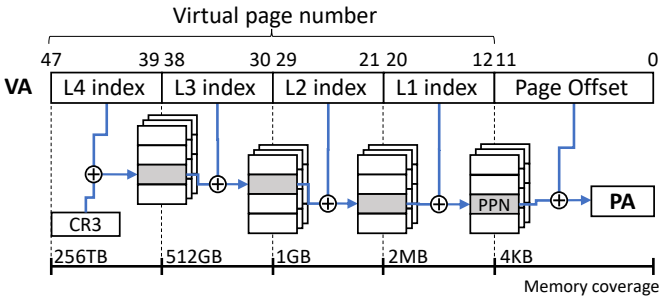

Fig. 2. Page table structure of traditional 64-bit architecture

2 Background and Motivation

2.1 Virtual Memory Support

A virtual memory system is an essential part of modern computing systems. The OS allocates physical pages only to virtual pages that are actually used, and stores mapping metadata in the page table. This system enables the CPU to run multiple processes efficiently on limited memory resources. Fig. 1 shows how a virtual memory system operates on the CPU. When a CPU core accesses memory with a VA, the memory management unit (MMU) converts the VA to a physical address (PA) through the TLB, which stores the page mapping metadata. However, when a TLB miss occurs, the page table walker searches the page table mapping metadata by accessing the page table in the main memory.

This system poses a major challenge for NMACCs to access main memory. Because NMACCs receive a task from the host process, addresses of the operand data are based on the VAS. However, the main memory stores the data on PAS, and the VA-to-PA mapping information is only obtained through the page table. Thus, for NMACCs to access data in the main memory, they must be able to support the virtual memory by resolving the VA-to-PA mapping.

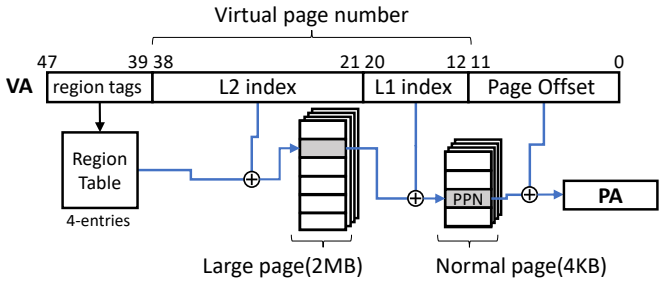

Fig. 3. Page table structure of the RPT

TABLE 1
VMA size ratio in each kernel's memory footprint

Kernel	Percentage of memory footprint per operands					
	Opd 1	Opd 2	Opd 3	Opd 4	Opd 5	Opd 6
GEMV	99.98	**0.01**	0.01			
BS	96.33	3.67				
MLP	33.33	33.33	33.33	**0.01**	0.01	
BFS	20.51	2.56	**2.56**	**2.56**	61.54	**10.26**
B+tree	**19.85**	80.13	0.01	0.01	0.01	0.01

TABLE 2
Page table walk count ratio of each operand

Kernel	Percentage of page table walks per operands					
	Opd 1	Opd 2	Opd 3	Opd 4	Opd 5	Opd 6
GEMV	66.49	**33.25**	0.26			
BS	99.25	0.75				
MLP	22.16	22.16	22.16	**33.25**	0.28	
BFS	2.07	0.61	**18.20**	**55.96**	2.57	**20.59**
B+tree	**17.81**	75.51	2.23	2.22	1.11	1.11

2.2 Challenges for Page Table Walking in NMACCs

A naive solution for virtual memory support is to duplicate the CPU's MMU into the NMACC. However, there are some issues for implementing CPU's page table walker in NMACC [2], [3]. The first is compatibility. The structure of the page table slightly differs among virtual memory infrastructures of CPU architectures. If the system has multiple CPU architectures as hosts, compatibility with the page table structures of these CPU architectures must be ensured. However, the near-memory environment does not have sufficient area to implement them all.

The second issue is address translation performance. Page table walk latency is one of the important factors in address translation time. Fig. 2 shows the traditional page table structure of 64-bit architectures. It stores mapping information for a 4KB page in an 8-byte entry. Since the page table also be managed in units of pages, it has a tree-type hierarchical structure. The 64-bit architecture has a coverage of 256TB through a 4-level hierarchical page table structure. It is indexed by a virtual page number (VPN) extracted from a VA, and the lowest level entries store mapping metadata including a physical page number (PPN) mapped to the virtual page number. Thus, the page table walk in the CPU requires as many pointer-chased memory reads as the levels of the page table hierarchy. However, data-intensive kernels accelerated by NMACCs result in many page table walks due to their large memory footprint, which incurs a large performance overhead due to page table walk latency [17], [21]–[24].

2.3 NMP-specific Page Table

To overcome issues for page table walking in NMACC, a virtual memory support method of using a page table specially constructed for NMP has been proposed in IMPICA [3] under the name of RPT. The RPT coexists with CPU's page tables and only stores the page mapping metadata for a shared region which is a pre-determined part of the virtual memory shared with the NMACC. When the OS updates a page table due to a page fault in these regions, it stores the page mapping metadata on both the CPU's page table and RPT. The CPU and NMACC then access the PAS of main memory mapped to the shared region through their respective address translation units. This method can solve the compatibility issue for CPU architectures by constructing a page table independent of the CPU.

Moreover, the RPT improves page table walk latency by reducing the hierarchy of page table structure. Fig. 3 shows the structure of the RPT. It has a two-level hierarchy by covering only the shared region which has an address space size of 512GB, and merging the upper hierarchy using large pages of 2MB size. The RPT then successfully reduces the page table walk latency with half page table walk procedure, compared to the 4-level page table of the CPU.

However, the RPT has some issues regarding scalability and page table walk latency. By reducing address space coverage, there is a limit to the amount of virtual memory that the NMACC can share with the host process. In addition, since the RPT has a single structure and uses predetermined shared space, it serves a static page table walk latency regardless of the memory footprint of NMACC. However, the memory footprints of NMACC can have variable sizes and shapes according to the target kernels. Since the page table walk latency depends on the number of layers of the page table configured corresponding to the address space size, there is an opportunity to further reduce page table walk latency by constructing a page table of an adequate hierarchy for the NMACC's memory footprint.

2.4 Splitting Memory Footprint of NMACC

The memory footprints of accelerators show different sizes and shapes in the target kernels. Considering that the accelerator's memory accesses consist of accesses to the kernel's operand data, in the aspect of virtual memory, NMACC's memory footprint can be split into virtual memory areas (VMAs) where each operand is allocated. Table 1 lists the VMA size ratios for each operand in the total memory footprint when they operate with a 1GB memory footprint. Through this table, it can be seen that the shape of the memory footprint appears according to the operand configuration of the kernel and the characteristics of each operand.

Specifically, since operand's VMAs have contiguous VAs corresponding to their size, their page mapping metadata can be managed in individual page tables. However, if they are managed in a single structured page table like RPT, the page table's hierarchy must be constructed according to the largest operand without considering the various sizes of

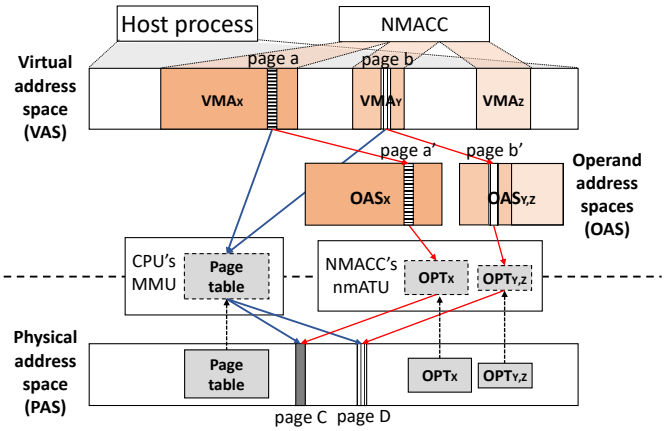

Fig. 4. Overview of operand-oriented virtual memory support

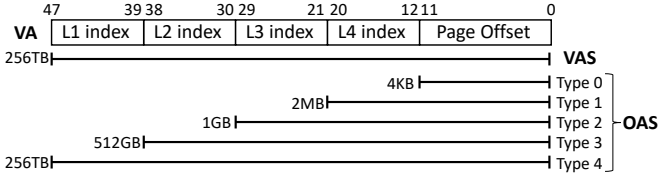

Fig. 5. Configuration of type of OAS

other operands. Table 2 lists page table walk count ratios that occur in the VMA for each operand in the same environment as Table 1. Considering that page table walk latency consumes most of the address translation time, address translation for the operand that generates many page table walks takes a large portion of the total address translation time. Comparing the underlined numbers between Table 1 and Table 2, even relatively small operands can cause a high rate of page table walks. Thus, reducing page table latency with a suitable page table for small operands can further improve the address translation performance.

3 Proposed Scheme

3.1 Overview

In this paper, we propose a virtual memory support scheme based on our novel NMP-specific page table. Our scheme aims to reduce page table walk latency by constructing suitable page table structures for NMACC's memory footprint. To adopt variable sizes and shapes of memory footprint, our scheme uses on-demand shard area determination and variable page table structures which have different coverage.

As mentioned in Section 2.4, the memory footprint of NMACC can be divided into operands allocated on each VMA. These areas are previously allocated and initialized by the host process through memory allocation APIs such as `malloc` or `mmap`. If the user defines the VMAs of these operands as shared areas through our API, then our software driver copies the page mapping metadata for the VMA of each operand into our NMP-specific page table called operand page table (OPT), which has a flexible structure according to the size of operand data. We also propose a near-memory address translation unit (nmATU), which translates the address based on our page table. By using

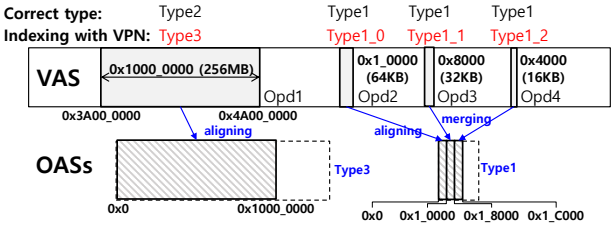

Fig. 6. Examples for OAS generation

our scheme, NMACCs can share various amounts of virtual memory based on reduced page table walk latency.

Fig. 4 shows how our scheme supports virtual memory for an NMACC that accelerates a kernel with three operands. In the situation where the host process initializes operands X, Y, and Z for acceleration, the user defines their VMAs as shared areas through our API. Then, our software driver generates OASs by aligning (X and Y) and merging (Z) these VMAs. When the shared area is fully defined, our driver constructs an OPT for each OAS and fills the OPT entries by copying page mapping metadata from the CPU's page table. After OPT construction, OPTs store OAS-to-PAS mapping metadata, and VAS-to-OAS can be converted by simple address shifting between OA and VA. In NMACC, nmATU translates the PE's VA to PA based on OPT. First, it simply converts VA to OA and translates OA to PA by performing the OPT walk. In Fig. 4, when CPU accesses the virtual pages a and b, MMU translates VAs of a and b to PAs of A and B. When NMACC access the same pages, nmATU first converts VAs of a and b to OAs of a' and b', then it translates OAs of a' and b' to PAs of A and B. In these procedures, since the $OPT_{Y,Z}$ has fewer hierarchies than OPT_X, nmATU's address translation time for b to B is reduced than a to A.

The remainder of this section describes the elements of our scheme. Section 3.2 describes the OAS, which is our logical address space, and Section 3.3 describes the OPT, which has a flexible structure according to the size of address space coverage. Section 3.4 describes the nmATU, the address translation unit attached to the memory interface of the NMACC. At last, Section 3.5 describes the software stacks and design considerations.

3.2 Operand Address Space

In this work, we construct variable page table structures with different address space coverage sizes as shown in Fig. 5. By default, in a conventional computing system, the page table stores page mapping metadata in its entries indexed by VPN. However, as shown in Fig. 6, if our page table with reduced address space coverage is indexed with VPNs, VMAs can not be able to allocate a suitable type for the VMA size due to biased VPN values. In the case of Opd1 of Fig. 6, it has a size of 256 MB and has to be declared in type 2, but when indexing with their VPN, it exceeds the address space size of type 2. This causes allocating Opd1 in type 3, resulting in an unnecessarily large page table being constructed. In addition, in the case of Opd2, Opd3, and Opd4, generating individual OPT for each smaller operands incurs resource overhead.

Algorithm 1 OAS generation

```
1:  procedure OAS_GENERATION(VA, Size)
2:      mlock(VA, Size)
3:      N_page ← calculate_number_of_page(VA, Size)
4:      Type ← classify_OAStype(N_page)
5:      for i = 0 → N_OAS do
6:          if OAS[i].type== Type && remain pages of OAS[i]>N_page then
7:              create_Operand(VA, Size, i,OAS[i].lastOPN)
8:              Add N_page to OAS[i].number of pages
9:              Done ← true
10:             break
11:     if Done != true then
12:         create_OAS(Type,0)
13:         create_Operand(VA, Size,N_OAS, 0)
14:         N_OAS ← N_OAS + 1
```

To address this issue, we use a logical address space called the OAS. The OAS is generated by shifting the operand's virtual addresses, and operand page numbers (OPNs) of the OAS are used for indexing our page table. For indexing our page table built to the size of the operand's VAS, we divide the OAS into five types of different sizes, as shown in Fig. 5. These sizes are the same as the coverage of each level of the CPU's page table because it has the most granular level configuration in the paging system of the OS. Our driver then either generates an OAS with a suitable type according to the size of operands or merges it into existing OAS. In Fig 5, for Opd1 and Opd2, the driver generates a suitable type of OAS and their VAs are shifted to zero-base OA. For Opd2 and Opd3, the driver shifts their VAs behind of OAS generated for Opd2. This enables the driver to construct suitable page tables related to the sizes of the operand's VMA and reduces memory resource usage by reducing the number of page tables generated.

3.2.1 OAS Generation

Generating the OAS is the same as defining the VMA to share with the NMACC. When the user calls the OAS generation API, our driver manages a list of OASs and shared VMAs using two types of metadata: OAS and operand metadata. The OAS metadata consists of type and valid page numbers, and it is used in the OPT construction. And the operand metadata consists of a base VA and size of the operand's VMA, ID of the allocated OAS, and a base OPN. It is used to copy the context of the CPU's page table to fill the OPT. Algorithm 1 describes our driver's OAS generation procedure. Our driver first classifies the type of OAS with the number of pages of the VMA. Then, if that number of pages can be inserted into the remaining space of the OAS of the same type generated earlier (line 6), the driver creates operand metadata so that the operand's virtual pages are inserted into the remaining space of the matched OAS, and it also adds the number of operand's pages to the matched OAS metadata (lines 7-8). Meanwhile, if the existing OAS has insufficient space for the input operand's pages, the driver creates OAS metadata and operand metadata so that operands are assigned to the new OAS (lines 12-14).

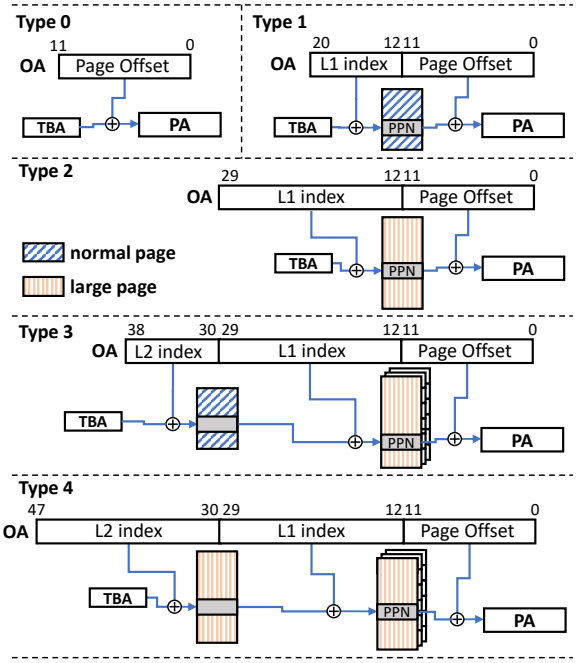

Fig. 7. OPT structures for each OAS type

3.3 Operand Page Table

The page table walk latency is directly related to the number of levels in the page table hierarchy, and the page table hierarchy is related to the size of the address space covered. Because a CPU page table manages page mapping metadata of numerous processes, it is constructed with only normal pages (4KB) for resource efficiency. However, for NMACC, the page table only needs to store page mapping metadata of the operand's VMA to be shared. In Section 3.2, our driver generated the OAS by classifying, aligning, and merging page numbers depending on the operand's VMAs sizes. Then, our driver constructs an OPT with a reduced hierarchical structure for each type of these OASs and copies and stores their page mapping metadata from the host's page table.

Fig. 7 describes the OPT structures for each OAS type. It has four structures (except type 0, which does not require the page table) according to the coverage of each type. We further reduced the hierarchy of page table by combining normal pages and large pages (2MB) for each type of OAS. In an example of Fig. 6, the page mapping metadata for Opd1, Opd2, Opd3, and Opd4 is stored in OPT type 1 and type 2, and page table walk of OPT for these operands requires only one memory read. In fact, the modern OS provides huge pages with 1GB, and it is possible to further reduce the hierarchy. However, securing huge pages requires a specific setup, and there is a limit on the number of huge pages that can be secured. Therefore, we design the OPT using only normal pages and large pages.

In particular, in the case of type 3, we can choose to merge the upper levels or the lower levels of the page table structure. Merging the upper levels configures the lowest levels with normal pages, so the memory resource used when the page table allocates an additional page to store new metadata is minimized. In the case of RPT,

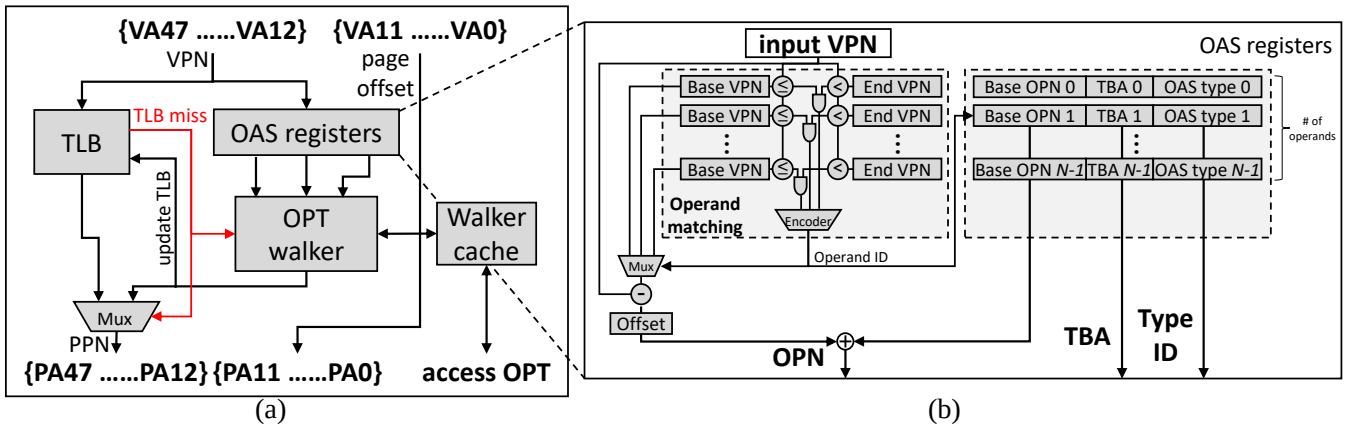

Fig. 8. Procedure of (a) address translation in nmATU and (b) converting VPN to input values of OPT walker in OAS registers

TABLE 3
MMR configuration

Name	Bits	Description
Base VPN	36	Base virtual page number of operand's VMA
End VPN	36	Last virtual page number of operand's VMA
Base OPN	36	Base operand page number of operand
TBA	64	Base physical address of the OPT
Type ID	3	OAS type for operand

which pre-determines the shared area, it merges the upper levels because it continually stores additional metadata. On the other hand, the page table which merges the lower levels requires large pages when adding metadata, but it has higher data reusability during the page table walk. Therefore, it can benefit page table walk performance with a cache. In this work, our scheme takes resource efficiency by on-demand shared area definition and page mapping metadata concatenation in OAS generation. For this reason, we merge the lower levels of the page table hierarchy to enhance the synergy with the walker cache.

As a result, the OPT can cover all address space with a hierarchical structure of zero to two levels, which requires less than half the number of memory accesses than the CPU's 4-level page table in the page table walk. Compared to RPT, OPT requires the same or fewer memory accesses than RPT with two levels. In addition, the OPT can cover the address space of 256TB by type 4, which is the maximum size used in the modern system, so it can overcome the limitation on the amount of virtual memory the RPT can cover.

3.3.1 Page Table Walk for OPT

As shown in Fig. 7, each type of OPT has different address space coverage and hierarchy configuration. Thus, it is different to index the OPT by the OPN's valid index bit length and index bit configuration for each level. To index page table entries of 8 bytes, a normal page requires 9 index bits, and a large page requires 18 bits. Therefore, for a correct page table walk according to OAS type, the OPT walker needs an additional input called OAS type to know the OAS type in addition to the input OPN and table base address (TBA).

3.3.2 OPT Construction

Through OAS generation, the driver has metadata about both the operand and OAS. When the OPT construction API is called, the driver constructs the OPT through the OAS metadata and fills it by copying the host's page table entries based on the Operand metadata. After OPT construction, the driver stores the information for OPT walking in the NMACC's memory-mapped register (MMR) as described in Table 3. A set of MMR consists of five registers and has a total size of 175 bits. To apply our scheme, our address translation unit embedded in NMACCs needs to have as many MMR sets as there are operands, and this number is determined when designing the NMACC.

3.4 Near-memory Address Translation Unit

Since OPT is a customized page table for NMP, we propose the nmATU for address translation using OPT. NMACCs can access the virtual memory of the host process regardless of CPU architecture through embedding our nmATU. Fig. 8a shows the design of our nmATU. Similar to the CPU's MMU, it translates a VA into a PA by converting the VPN to the PPN. If a TLB miss occurs, the nmATU obtains PPN through OPT walking. However, the OPT walker requires three inputs as mentioned in Section 3.3: OPN, TBA, and OAS type. Therefore, the nmATU has OAS registers module that outputs the information needed by the OPT walker.

The OAS registers outputs the input values of the OPT walker based on VPN. The OAS registers holds the values stored as MMRs in the driver. This module identifies the operand to which Input VPN belongs and outputs the OAS information. Fig. 8b shows the procedure of OAS registers. This process takes one cycle and is hidden because it runs in parallel with TLB searching. First, it compares Input VPN to Base VPN and End VPN, looking for the operand to which the Input VPN belongs. Then, it outputs Base OPN, TBA, and OAS type based on this operand ID. It simultaneously calculates the offset page number of operand's VMA by subtracting Base VPN from Input VPN. Finally, the OPN is calculated by adding the offset page number and Base OPN.

```
int *A, *B;
A = (int *)malloc(size_A);
B = (int *)malloc(size_B);
...
Initialize A and B
...
OAS_generate(A, size_A);
OAS_generate(B, size_B);
OPT_construct();
...
run_NMP();
...
OPT_free();
```

Fig. 9. Example of user code for constructing OPT

3.4.1 Walker cache

Since the page table has a tree-type structure with high data reusability, it can benefit from cache memory. Therefore, many existing CPU architectures use a walker cache (WC) [25]. The upper layer page table entries cover a wide address space, and in the page table walk that occurs by accessing this space, these entries are reused. Therefore, the WC can reduce the page table walk latency by a large number of cache hits for the entries in the upper level of the page table. Since the OPT has fewer levels than the CPU's page table, the effect of reducing page table walk latency due to WC is less than that for the CPU. However, because the OPT uses much fewer entries in the WC during the page table walk procedure, it has high capacity efficiency. Therefore, the OPT walker can be expected to have a good effect on the WC with an even smaller capacity than the CPU's page table or RPT.

3.5 Software Stack and Design Consideration

3.5.1 Software Driver and API

The software stack of our scheme must be run on the OS-level. Therefore, we implement our driver as a device driver in Linux 4.9.0. There are three APIs for employing our scheme: `OAS_generate`, `OPT_construct`, and `OPT_free`. `OAS_generate` and `OPT_construct` are used for constructing the OPT and transferring information of the OPT to nmATU in the NMACC. `OPT_free` clears all metadata of OAS and operand and deletes OPTs.

However, there are conditions when calling APIs to successfully construct an OPT. Fig. 9 shows an example of using our API when sharing operands A and B with NMACC. First, since the driver creates the OAS using the operand's VMA, `OAS_generate` must be called after the operand is allocated in virtual memory through memory allocation APIs such as `malloc` or `mmap`. Second, `OPT_construct` must be called after the data initialization. The driver copies the page mapping metadata in the CPU's page table when filling in the OPT, but the page table entries for A and B are empty before data initialization because the OS does not map the physical page until the data is actually written.

In the case of `run_NMP`, when offloading the task to NMACC, a host process only needs to send the base VA of operand data to NMACC for memory sharing because NMACC can operate on the same VAS as the host process through the nmATU.

3.5.2 Coherency for Page Mapping Metadata

When the host constructs OPTs, our driver copies page mapping metadata from the CPU's page table, so the OPT stores the latest data. However, there is a coherency issue in two situations for page mapping metadata during the acceleration. The first situation is data coherency between the CPU's and NMACC's TLBs. NMACCs do not update TLB data because it only performs tasks offloaded from the CPU, but the CPU can update TLB data. The second situation is metadata coherency between the CPU's page table and the OPT. The OPT stores copies of the CPU's page table entries. However, if the CPU updates the page table entries that are copied to the OPT, then the OPT has outdated page mapping metadata. These issues occur when the CPU modifies the page mapping of the operand's virtual pages, such as selecting the victim page during the swap operation. Therefore, we use the system call `mlock` for memory protection. This system call prevents the mapping metadata of pages from being updated in a specific VMA. We insert this into the `OAS_generate` API to protect the page mapping of VMAs of operands shared with the NMACC.

4 Evaluation

In this section, we present the evaluation methodology and experimental results of the proposed address translation scheme. To evaluate our scheme, we first designed a simple NMACC prototype that applied our scheme. We verified our scheme's functionality and performance effectiveness on a real system by demonstrating it on an FPGA-based NMACC verification platform. Then, to evaluate our scheme in more kernels and detailed architectural studies, we implemented it in a simulation environment.

4.1 Architecture of NMACC Prototype

Fig. 10a shows the structure of the NMACC prototype. It consists of simple external interface modules and processing logic. The host interface module communicates with our driver of the CPU through memory-mapped I/O. It receives scalar arguments of the kernel and OPT information described in Section 3.3. The memory interface module accesses the memory through address translation by our nmATU. For comparison with other page table structures, we also implemented both the CPU's 4-level page table (L4PT) walker and RPT walker as page table walkers in the nmATU.

The processing element is an application-specific processing logic for the target kernel. It operates based on the host's VA and has a data buffer of 64 bytes for each operand. The design of the memory interface module is not closely related to the design of the processing logic because it sends memory transactions to the memory with the same interface as other external IPs. Therefore, the performance of the processing element can be applied to various acceleration methodologies, such as parallelization by many processing elements (PEs) and data buffer optimization. In this work, we implemented NMACCs with a single PE optimized for memory operation and minimal data buffer.

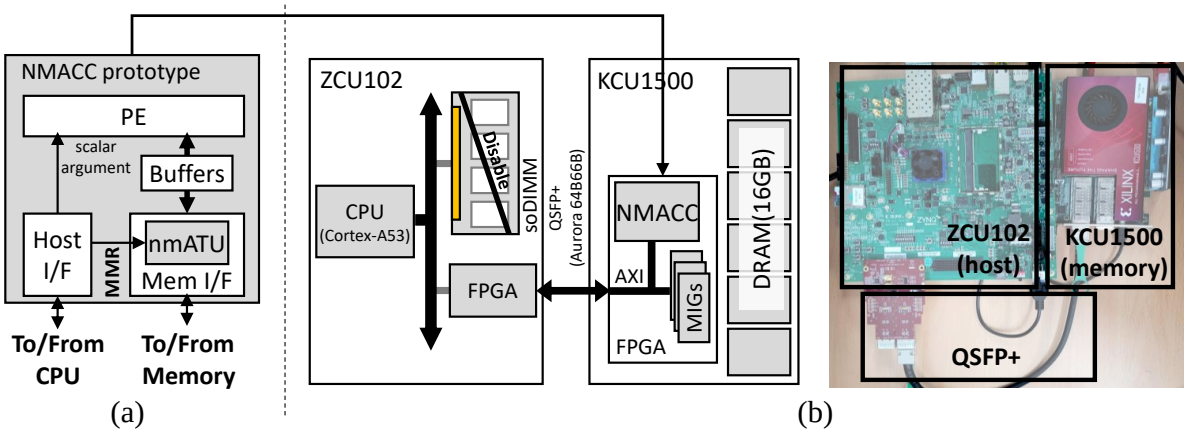

Fig. 10. (a) Architecture of NMACC prototype (b) FPGA-based verification platform

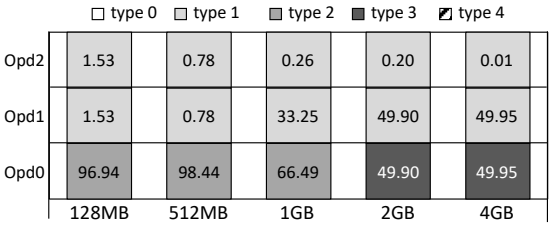

Fig. 11. OAS type allocation and TLB miss count ratio of each operand in NMP with GEMV kernel

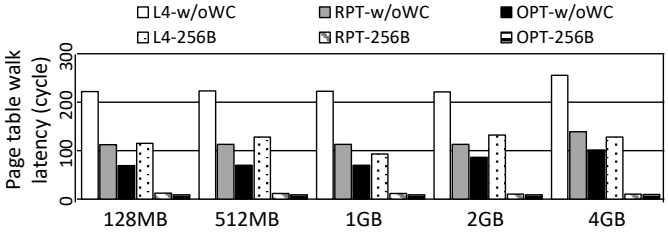

Fig. 12. Average page table walk latency of NMACCs

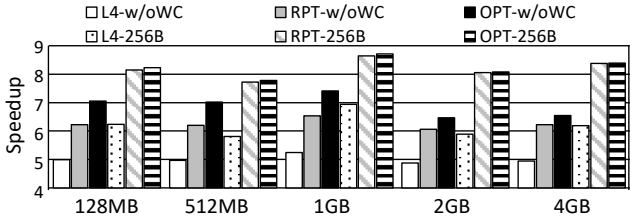

Fig. 13. NMACC Speedup with different page table, normalized to CPU-only

4.2 Demonstration on the FPGA-based System

We built an FPGA-based NMACC verification platform consisting of a host board (Xilinx Zynq ZCU102) and memory board (Xilinx Kintex KCU1500). Many 3D-stacked DRAM-based NMACC systems have the CPU, NMACC's logic area, and DRAM on the same board. However, due to the limitations of the size of memory connected to the FPGA which implements the NMACC logic, we built an environment in which the CPU and memory are different boards similar to the DIMM-based NMACC system. Fig. 10b shows our platform. We disabled the SODIMM slots of the host board and set the address space of its FPGA as the main memory space by modifying the first and second bootloaders and the device tree. Then, we connected FPGAs between the host board and memory board through a Xilinx Aurora 64B66B link to pass the CPU's main memory accesses. As a result, the host board's CPU recognized the memory board's DRAM as the main memory.

In this platform, we loaded our software stack into Linux kernel 4.9.0 and implemented a processing logic for the kernel of GEMV [26] as the PE of the NMACC prototype, which was programmed in the memory board's FPGA with a clock frequency of 200Mhz. As shown in Table 5, GEMV is a matrix-vector multiplication application that has three operands and a sequential access pattern. To see the effect of page table structure clearly, we implemented the NMACCs both without the WC and with a 256B size WC in the nmATU. We then experimented by increasing the memory footprint of NMACCs with 128MB, 512MB, 1GB, 2GB, and 4GB by scaling the size of the input matrix and vectors.

As shown in Fig. 11, in most experiments, more than half of TLB misses occur in OAS type 1 and type 2, and they perform page table walk with only a quarter of the memory reads compared to L4PT and half compared to RPT. Fig. 12 shows average page table walk latency during NMP. In the NMACC without WC, the OPT reduced page table walk latency by 65.1 percent and 32.6 percent compared to 4LPT and RPT, respectively. In NMACC with WC, page table walk performance for all page tables was significantly improved due to WC. The OPT reduced page table walk latency by 92.1 percent and 18.7 percent compared to 4LPT and RPT, respectively.

Fig. 13 shows the performance improvement of NMACC when compared to CPU-only computation. This performance is measured in terms of overall execution time, which includes both computation time and address translation time. By improving computation time through effective memory bandwidth utilization using hardware acceleration, all NMACCs achieved at least 4.89x speedup over CPU-only computation. Meanwhile, the variations seen in the results are attributed to the address translation time, specifically the page table walk latency, which is the main focus of this pa-

TABLE 4
Simulation platform configuration

CPU	4 Out-of-Order core, 2GHz, ARMv8 ISA, L4 page table
Memory	DDR4_2400_16x4, 8 GB, 1 channel, 2 ranks, 16 bank each rank, FR-FCFS scheduling
OS	Linux kernel 4.9.0
NMACC	trace-driven model for PE, 500Mhz, 32-entry TLB, 256B WC with LRU, OPT walker, L4PT walker, RPT walker

TABLE 5
Kernels used in our evaluation

Benchmark	# of operand	Domain	Access pattern
GEMV [26]	3	Linear algebra	seq
BS [26]	2	Data analytics	seq, rand
MLP [26]	5	Neural network	seq
BFS [27]	6	Graph algorithms	seq, rand
B+tree [27]	6	Searching	seq, rand

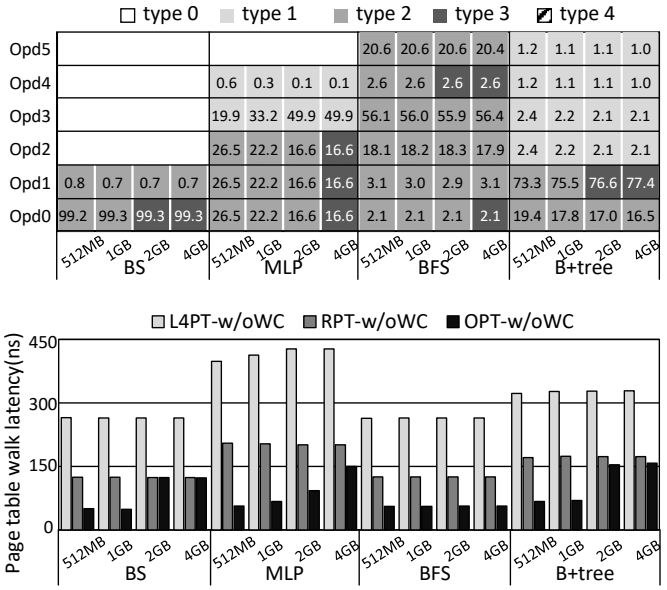

Fig. 14. (a) OAS type allocation and TLB miss count ratio for each operand, (b) average page table walk latency without WC

per. In NMACCs without the WC, the OPT improved overall performance by 37.8 percent and 10.3 percent compared to 4LPT and RPT, respectively. When WC was applied, the overhead of the page table walk was almost eliminated in overall performance when using the RPT and the OPT. The OPT improved overall performance by 32.6 percent and 0.6 percent compared to 4LPT and RPT, respectively.

Through verification in real hardware, we confirmed that our scheme successfully provided the ability to support the virtual memory of the host process to the NMACC. These results show that NMP-specific page tables can provide high performance to kernels with sequential access patterns by adding WCs because these patterns had a high hit rate in the TLB and the walker cache. However, since it was difficult to see the effect of page table walk latency on sequential patterns, we further evaluated our scheme on kernels with random access patterns through simulation.

4.3 Simulation Methodology

4.3.1 Simulation Setup

To evaluate our scheme in simulation, we implemented the NMACC prototype in the gem5 full-system simulator [28] and loaded our software stack in the Linux kernel 4.9.0. Table 4 describes the configuration of simulation platform. We built a system with an ARMv8-based CPU that uses a 4-level page table (L4PT). We implemented the NMACC's PE as a trace-driven model. We extracted virtual memory traces on VMAs of the target kernel's operands using Pin [29]. Then, during simulation, we converted this virtual memory trace to gem5's VAS by shifting the trace's addresses into each operand's VMA of the benchmarks run on the gem5 system. In our trace-driven model, we assumed that the NMACC's PEs computing operations are hidden by memory operations. We can make such an assumption because the kernels we implemented in PE are data-intensive kernels with memory-bound performance. To evaluate the performance of our page table, we implemented the L4PT walker and RPT walker in nmATU as a comparison group. The L4PT walker received the table base register (TTBR) value from the CPU and directly accessed the CPU's page table directly from the main memory. We used Cacti [30] to estimate the energy consumption of WC for the 90nm process technology.

4.3.2 Kernels of NMACC

To evaluate our proposal, we implemented four kernels of data-intensive benchmarks from the PrIM [26] and Rodinia [27] benchmark suites. Table 5 describes the features of these kernels. MLP, which operates a three-layered multi-layer perception, has a strong sequential access pattern similar to GEMV. On the contrary, BS, which operates a binary search algorithm, and BFS, which operates a breadth-first search algorithm, have a strong random access pattern. B+tree has mixed access patterns. We scaled up the memory footprint of kernels by using input argument adjustments and input data generators provided by each benchmark, and we evaluated our proposal on these kernels with memory footprints of 512MB, 1GB, 2GB, and 4GB.

4.4 Impact of the OPT

Our scheme aims to construct appropriate page tables for each operand to reduce page table walk latency of the NMACC. Fig. 14a shows which OAS types our driver has assigned to each operand. Even in cases where the total memory footprint exceeds 1GB, our scheme allocates type 1 and type 2 to most operands, reducing page table hierarchies. In addition, the number in the box of Fig. 14a shows the percentage of TLB miss counts for each operand. In the case of BS, the largest operand caused most TLB misses. Most of the page table walk time in this kernel was spent by the page table structure of the largest operand. However, in BFS, large operands caused only small TLB misses. The proportion of page table walk operations due to large operands in the total page table walk time was

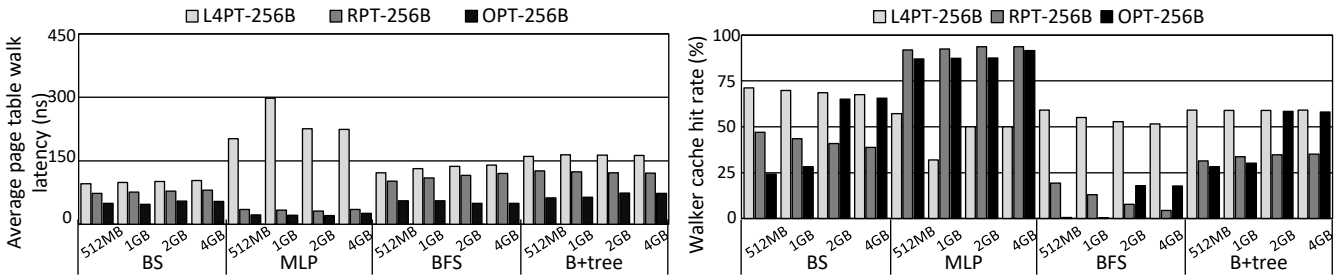

Fig. 15. (a) Page table walk latency with 256 bytes WC, (b) walker cache hit rate

small. From this figure, it can be seen that the correlation between operand size and TLB miss count is weak, and their relationship is determined by the operational characteristics of the kernel.

To see the effect of page table hierarchy, we implemented the nmATU, which had a 64 bytes buffer instead of the WC. Fig. 14b shows average page table walk latency without the WC. In the memory footprint of 1GB or less, OPTs were built as a single level, type 1 and type 2, and it significantly reduced page table walk latency. The largest improvement was in 512MB of MLP, which reduced page table walk latency by 89.1 percent compared to L4PT and 72.5 percent compared to RPT. In this case, the OPT improved the page table walk latency over the memory access ratio decreased by hierarchy reduction. This is because the page mapping metadata was more densely stored in OPT due to OAS merging, and a buffer hit occurred in the 64-byte buffer due to the increased spatial locality. However, in the memory footprint of 2GB or more, the latency improvement decreased as OPT allocates type 3 with two hierarchies to some operands. In the case of BS, most of the page table walks occurred in type 3 OPT, and the page table walk latency was similar to that of RPT which had 2 levels hierarchy. In contrast, in the case of BFS, since most page table walks occurred due to access to small operands, OPT provided significantly reduced latency on all of the memory footprints by constructing small page tables for the small operands. As a result, our OPT reduces page table walk latency by 72.8 percent compared to the L4PT and 44.5 percent compared to the RPT in overall kernels.

4.5 Synergy with Walker Cache

4.5.1 Page Table Walk Latency

Since accessing the page table that has a hierarchical structure has high data reusability, WC can significantly improve the page table walk performance. Fig. 15a shows average page table walk latency with WC of 256B. And Fig. 15b shows the hit rate of WC. The WC improved page table walk latency by 50.2 percent in L4PT, 44.6 percent in RPT, and 43.8 percent in OPT than the case without the WC.

In the cases of kernel BS, BFS, and B+tree, WC reduced L4PT's page table walk latency by more than half. This is because the WC filters memory accesses to the upper-level entries of the page table. A single WC block that stores upper-level entries can cover 4TB (L4) and 8GB (L3) of address space, respectively. Therefore, in Fig. 15b, the WC hit rate for L4PT is shown mostly over 50 percent in BS, BFS, and B+tree. However, the RPT already eliminates upper levels, it cannot take the effect of WC as in L4PT. It improved latency by only 21 percent compared to L4PT in BS, BFS, and B+tree. On the other hand, our OPT reduced page table walk latency by eliminating more levels by using type 1 and type 2 of OPT. Furthermore, in 2GB and 4GB of BS and B+tree, WC significantly reduces OPT's page table walk latency. This is because, unlike RPT, the 2-level hierarchical structures of OPT (type 3 and type 4) merge the lower layers to increase data reusability in the page table walk. The L2 entry of RPT has coverage of 2MB, but the L2 entry of OPT has coverage of 1GB. This can also be seen in the WC hit rate of Fig. 15b, and in these cases OPT had a cache hit rate as high as L4PT.

On the contrary, MLP shows a different aspect. The OPT and the RPT significantly reduced page table walk latency while exhibiting high WC hit rates. This is because memory accesses of page table walks had high data reusability due to the sequential access pattern of MLP. However, L4PT has a lower WC hit rate in MLP than RPT and OPT. Since the L4PT had to use four cache blocks for every page table walk, WC suffered from cache pollution due to a page table walk through each operand. Besides, MLP operated on multiple operands at the same time, and page table walk operations for multiple operands occurred in a cross. Thus, these page table walks evicted the WC data which was upper-level entries of other operand's page table walks.

In summary, the OPT used WC efficiently through a reduced hierarchy, and even in a situation where a two-level hierarchy had to be used, it took more effects of WC than RPT through a WC-friendly structure. As a result, the OPT reduced page table walk latency by 69.3 percent compared to L4PT and 43.8 percent compared to RPT.

4.5.2 Walker Cache Size Efficiency

Reducing the levels of the page table structure decreased not only the page table walk latency but also the number of WC blocks required during a page table walk. Therefore, we evaluated the WC size efficiency of our scheme. Fig. 16 shows average page table walk latency at different WC sizes in NMP with a memory footprint of 2GB. In this figure, WC with OPT shows the saturated effect at a smaller size than those of RPT and L4PT. Other page tables must be cached at least as many as their number of hierarchies because the cache was polluted by memory reads for higher levels, but single-level hierarchy page tables enabled more efficient use of WC. Even in the case of BFS, it can be seen that the latency improvement is maintained even with a 64-byte WC.

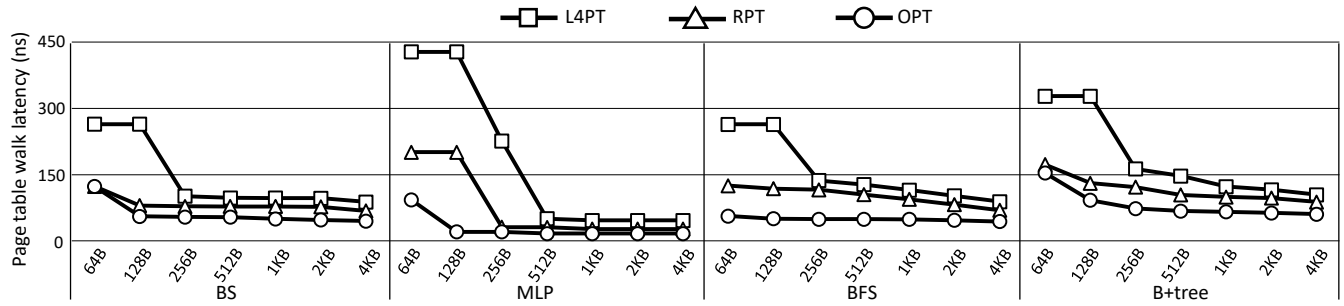

Fig. 16. Page table walk latency with different WC size

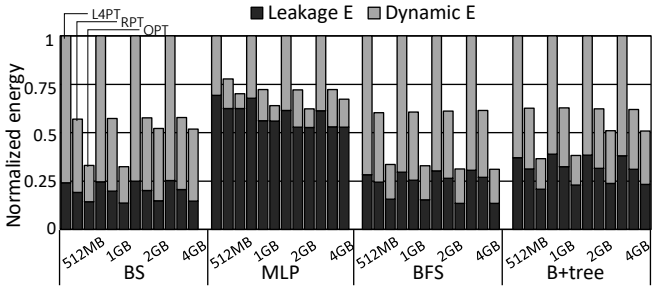

Fig. 17. Normalized energy consumption of the WC

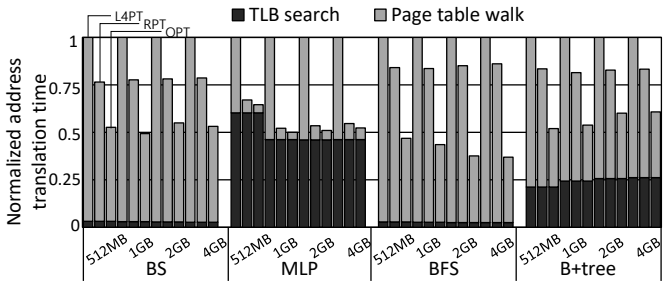

Fig. 18. Normalized address translation time

This is because most TLB misses of BFS were incurred by accessing small operands that were managed by single-level page tables, which required only one memory read during the page table walk process.

In summary, the OPT reduced WC size requirements by more than half compared to L4PT. This means that NMACC can have virtual memory support ability with less hardware resources by using our OPT. This can provide many opportunities for NMACCs using many PEs in near-memory environments with small area constraints.

4.5.3 Walker Cache Energy Efficiency

Fig. 17 shows the energy consumption of WC. The energy of WC is divided into dynamic energy and leakage energy. The dynamic energy of WC is related to the number referenced to WC. Because the OPT has a reduced hierarchy than other page tables, it does a page table walk with fewer memory accesses that reference the WC. As a result, the OPT reduced dynamic energy consumption of the WC by 68.6 percent and 37.2 percent compared to the L4PT and RPT, respectively. The leakage energy is related to computation time, and OPT improved NMACC execution time by improving page table walk latency. Therefore, leakage energy of the WC was improved by 36.8 percent and 24.9 percent compared to L4PT and RPT, respectively. In the case of MLP with a sequential access pattern, since the TLB miss rate was low, the number of WC references was relatively small, and the WC energy consumption was dominant in leakage energy. However, in other kernels with random access patterns, the consumption rate of WC's dynamic energy was high in WC's energy. As a result, OPT reduced energy consumption of the WC by 41 percent and 25.1 percent compared to L4PT and RPT, respectively.

Through Sections 4.5.2 and 4.5.3, reducing the levels of a hierarchy of the page table structure decreased the WC access count of the page table walker, which improved the efficiency of WC size as well as energy efficiency. Our OPT effectively reduced the number of WC accesses, thus indicating high synergy with WC.

4.6 Address Translation Time

Address translation time consists of TLB searching and page table walk latency. Since the TLB searching time is the same regardless of the page table structure, our scheme's address translation performance improvement is affected by the occurrence rate of page table walks caused by TLB misses. Considering the page table walk latency is significantly longer than TLB searching, even a small percentage of TLB misses can have a significant impact on address translation time. Fig. 18 shows a breakdown of address translation time normalized by L4PT. In the case of MLP and B+tree, the TLB hit rate was more than 95 percent, and most address translation was completed only with the TLB of nmATU. Therefore, the ratio of TLB search in address translation time was high. However, in the case of BS and BFS with strong random access patterns, most of the address translation time was consumed by page table walk latency.

In summary, the OPT effectively improved address translation performance for these kernels with strong random access patterns. The OPT reduced address translation time by 49.1 percent and 32.3 percent compared to the L4PT and RPT, respectively.

4.7 Total Performance

4.7.1 NMACC Performance

We evaluate the impact of our scheme on NMACC's total execution time. Fig. 19 shows improvement of NMACC ex-

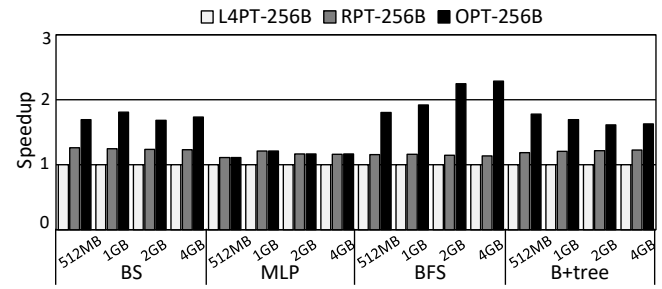

Fig. 19. NMACC performance improvement by page table structure, normalized by L4PT

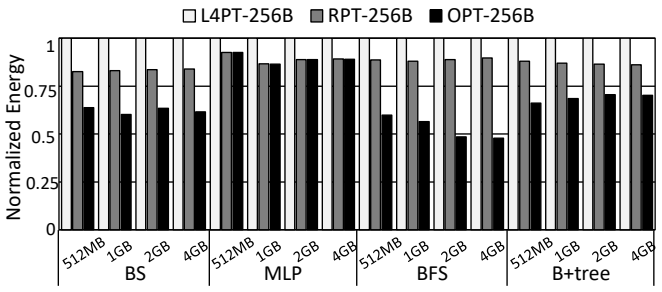

Fig. 20. Normalized energy consumption of main memory

ecution time normalized on L4PT-based NMACC. OPT significantly improved performance for kernels with random access patterns that cause many TLB misses. The condition for OPT to work best is when a kernel has a random access pattern that incurs a large number of TLB misses and when these TLB misses are incurred by a relatively small operand. In our evaluation, the BFS kernel represented that case. For BFS, OPT improved NMACC performance by 2.06x compared to L4PT and 1.79x compared to OPT. However, in kernels with sequential access patterns, address translation time did not account for much of the total execution time because it had a high TLB hit rate and WC hit rate. In MLP, OPT improves NMACC performance by 16 percent compared to L4PT and 0.2 percent compared to RPT. As a result, the OPT improves NMACC performance by 1.65x compared to L4PT and 1.39x compared to RPT in overall kernels.

4.7.2 Energy Consumption of Main Memory

Fig. 20 shows the energy consumption of the memory system normalized by the case of using the L4PT. In terms of power, the OPT increased the power of main memory by 8.76 percent and 5.1 percent compared to the L4PT and RPT, respectively. This is because OPT reduced the address translation time, thereby increasing the throughput of the memory. However, the reduced execution time through this decreased the total energy consumption. Furthermore, OPT reduced the number of memory reads due to the page table walk, which also reduced the energy of the main memory. As a result, OPT reduced the energy consumption of main memory by 31.6 percent and 21.5 percent compared to the L4PT and RPT, respectively.

4.8 Overhead

Our scheme incurs overheads in two aspects on the host CPU. First, there is a memory resource overhead consumed by the OPT. The page table stores mapping metadata for a 4KB page in an 8-byte entry. In addition, since the OPT has a hierarchical structure of up to 2 levels, it consumes less than 1 percent of memory resources compared to the memory footprint of NMACCs. However, in the worst case, when an operand's size is the low boundary in coverage of type 1 and type 2 of OAS, the OPT consumes the memory resource almost equal to the size of the operand's size because page tables are allocated in page units. In this case, the memory resource overhead that occurs is 2MB or less, and this overhead is also minimized by merging address spaces in the OAS generation process.

The second overhead is OPT construction time. The most time required in the process of OPT generation is the time to fill the OPT by copying the page mapping metadata for the VMA of operands from the CPU's page table. However, this operation also takes a very short time compared to the total operation time of the NMACC because bulk page table walks for contiguous VAS have high data locality, and this operation is mostly processed in cache hits. In our evaluation, in kernels except for MLP, the OPT construction took less than 0.2 percent of the NMACC operation time. However, since MLP performed high-throughput processing, the OPT construction took an average of 2.2 percent of the NMACC execution time. Compared to the RPT, the OS updates both the page table and the RPT when a page fault occurs in the shared regions. It has been reported that this takes a small amount of time compared to page fault latency. And since OPT is initialized in bulk, it has a small overall latency compared to the RPT updated by a single entry.

5 Related Works

There are various methods for supporting virtual memory when sharing memory between the NMACC and CPU. Some prior NMACC has assumed that contiguous VASs are allocated to contiguous PASs with modification to the OS's memory allocator [31] [32]. This method enables the NMACC to perform address translation without a page table walk. However, contiguous PA allocation can worsen memory fragmentation, and if memory fragmentation is severe, contiguous range allocation can fail. These problems have been addressed through new virtual memory allocation methods that use segment allocation [18], [19], [20]. They replace the existing page-based memory allocation and allocate VMAs generated by the host process to a contiguous PAS. However, these methods are difficult to apply in existing computing systems because they require significant modifications to the host machines, such as TLB and page table walker. In addition, since these methods are not completely free from the memory fragmentation issue, they use segment-page hybrid allocation [19], [20]. In such an environment, NMACC should also need to be capable of page table walks.

Some NMACCs support virtual memory by adding hardware modules next to the MMU or cache of the host CPU [10], [14]. These hardware modules automatically translate the PA of the data they operate on when the

NMACC is running. This enables the NMACC to take more features, such as cache coherency, in addition to virtual memory support, but it is difficult to apply to the existing system because it requires modification of the CPU hardware. In addition, since they use the CPU's MMU, the operand data size of the NMACC command is limited to the page size. This means that in operating the NMP kernel, command streams must be continuously sent to the CPU to NMACC in page units. This requires additional off-chip traffic and CPU resource for managing the NMP command stream.

Some NMACCs perform address translation using their own TLB but do not perform a page table walk [5], [33], [34]. When an NMACC's TLB miss occurs, it sends an interrupt to the CPU so that the CPU fills the NMACC's TLB. This method can provide virtual memory support to NMACCs without modifying the page table or hardware of the CPU. However, this can cause significant off-chip traffic and latency to handle the NMACC's TLB misses.

6 Conclusion

In this paper, we proposed a virtual memory support scheme with OPT to reduce the page table hierarchy. The key idea is to separate each operand's VMA of the kernel and to construct an appropriate page table structure according to their sizes. For this, we proposed a software stack that separates VMA and constructs an OPT. We then proposed a hardware module that conducts page table walks based on the OPT in an NMACC. We modularized our framework to apply to NMACCs and support the virtual memory system. Then, we demonstrated our proposed scheme in an FPGA-based verification platform and evaluated it in a full-system simulator.

Through simulation, our scheme considerably reduced the page table hierarchies of operands, which is directly reflected in the page table walk latency. Our OPT improved page table walk latency by 69.3 percent and 43.8 percent compared to the CPU's page table and previous work, respectively. And our OPT also improved the size efficiency and energy efficiency of WC. As a result, our proposed scheme improved the NMACCs performance by 1.65x and 1.39x compared to the L4PT and RPT, respectively.

We expect our scheme to be a practical solution for memory sharing between NMACCs and CPU. In addition, we plan to expand our solution to cloud server systems that manage the memory with tiered page tables. We leave these components for future work.

Acknowledgments

This work was supported by Institute of Information communications Technology Planning Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2022-0-00050, Development of PIM Computing Architecture based on Data-Flow), Samsung Research Funding Incubation Center of Samsung Electronics under Project Number SRFC-IT2002-01, and the Samsung Electronics Company, Ltd., Hwaseong, Korea. The EDA tool was supported by the IC Design Education Center(IDEC), Korea.

References

[1] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "Near-memory computing: Past, present, and future," *Microprocessors and Microsystems*, vol. 71, p. 102868, 2019.

[2] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," *arXiv preprint arXiv:2012.03112*, 2020.

[3] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 2016, pp. 25–32.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[5] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. Costa, J. Doi, C. Evangelinos *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17–1, 2015.

[6] M. Gao and C. Kozyrakis, "HRL: Efficient and flexible reconfigurable logic for near-data processing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 2016, pp. 126–137.

[7] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "Lazypim: An efficient cache coherence mechanism for processing-in-memory," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 46–50, 2016.

[8] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng *et al.*, "Conda: Efficient cache coherence support for near-data accelerators," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 629–642.

[9] J. Jang, J. Heo, Y. Lee, J. Won, S. Kim, S. J. Jung, H. Jang, T. J. Ham, and J. W. Lee, "Charon: Specialized near-memory processing architecture for clearing dead objects in memory," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 726–739.

[10] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 336–348.

[11] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong *et al.*, "Application-transparent near-memory processing architecture with memory channel network," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 802–814.

[12] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 740–753.

[13] M. Alian and N. S. Kim, "Netdimm: Low-latency near-memory network interface architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 699–711.

[14] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee *et al.*, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 790–803.

[15] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon *et al.*, "Near-memory processing in action: Accelerating personalized recommendation with axdimm," *IEEE Micro*, 2021.

[16] C. H. Kim, W. J. Lee, Y. Paik, K. Kwon, S. Y. Kim, I. Park, and S. W. Kim, "Silent-pim: Realizing the processing-in-memory computing with standard memory requests," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 251–262, 2021.

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

[18] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 66–78, 2015.

[19] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal, "Range translations for fast virtual memory," *IEEE Micro*, vol. 36, no. 3, pp. 118–126, 2016.

[20] N. Hajinazar, P. Patel, M. Patel, K. Kanellopoulos, S. Ghose, R. Ausavarungnirun, G. F. Oliveira, J. Appavoo, V. Seshadri, and O. Mutlu, "The virtual block interface: A flexible alternative to the conventional virtual memory framework," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 1050–1063.

[21] S. Srikantaiah and M. Kandemir, "Synergistic tlbs for high performance address translation in chip multiprocessors," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 313–324.

[22] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 743–758, 2014.

[23] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: a gpu memory manager with application-transparent support for multiple page sizes," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 136–150.

[24] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 503–518, 2018.

[25] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 48–59.

[26] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture," *arXiv preprint arXiv:2105.03814*, 2021.

[27] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[29] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[30] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[31] J. Lee, J. Chung, J. H. Ahn, and K. Choi, "Excavating the hidden parallelism inside dram architectures with buffered compares," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 6, pp. 1793–1806, 2017.

[32] S. F. Yitbarek, T. Yang, R. Das, and T. Austin, "Exploring specialized near-memory processing for data intensive operations," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1449–1452.

[33] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien *et al.*, "Data access optimization in a processing-in-memory system," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015, pp. 1–8.

[34] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 113–124.

Duheon Choi received the B.S. degree in electrical and electronic engineering from Yonsei University, Seoul, Korea, in 2015, where he is currently working toward the Ph.D. degree in electrical and electronic engineering. His research interests include system software for processing-in-memory and system-level design.

Taeyang Jeong received the B.S. degree from Yonsei University, Seoul, South Korea, in 2017, where he is currently pursuing the Ph.D. degree in electrical and electronic engineering. His current research interests include hybrid memory systems and system software for processing-in-memory.

Joonhyeok Yeom received the M.S. degree in electrical and electronic engineering from Yonsei University in Seoul, Korea, in 2019. He is currently a Ph.D candidate in Yonsei University. His research interests include NAND flash applications, near-data processor design and system-level architecture.

Eui-Young Chung received the B.S. and M.S. degrees in electronics and computer engineering from Korea University, Seoul, South Korea, in 1988 and 1990, respectively, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2002. From 1990 to 2005, he was a Principal Engineer with SoC Research and Development Center, Samsung Electronics, Yongin, South Korea. He is currently a Professor with the School of Electrical and Electronic Engineering, Yonsei University, Seoul. His current research interests include system architecture and very large scale integration design, including all aspects of computer-aided design with the special emphasis on low-power applications and flash memory applications.