Virtual Memory Support for PIM with Table-based Management

Seung Jae Yong and Eui-Young Chung
Dept. of Electrical and Electronic Engineering
Yonsei University
Seoul, Republic of Korea
rad2minimal@yonsei.ac.kr, eychung@yonsei.ac.kr

Abstract

Processing-in-Memory (PIM) is a technology to alleviate the memory wall. In the PIM architecture, there are processing units for data operations in the memory. Therefore, since data is processed directly in the memory, there is no need to transfer data between the CPU and memory, which can reduce energy consumption and latency associated with data movement. However, the current operating system (OS) lacks virtual memory support for the PIM architecture. Therefore, there is a significant delay in accessing PIM due to the overhead of the existing multi-level page table walking every time. In this paper, we propose a technique for efficiently mapping virtual addresses to physical addresses in PIM using table-based management. Our technique has the advantage of reducing unnecessary delays and maximizing the use of PIM without any hardware modifications or support. The proposed technique is evaluated using a full system simulator, and the results show that the PIM access time can be improved by approximately 15.04 times compared to the existing system.

Keywords: Processing-in-memory, virtual memory support, operating system, page table, device driver

1. Introduction

Processing-in-Memory (PIM) is a promising technology [1], [2] that can alleviate the memory wall problem and significantly enhance the performance of memory-bound applications. With PIM, data processing elements are placed in the memory, which eliminates the need to transfer data between the CPU and memory since data can be processed directly in the memory. This leads to faster and more energy-efficient computation by reducing energy consumption and latency associated with data movement.

However, current operating systems (OS) lack virtual memory support for PIM, and research on this topic is also inadequate. This is because virtual memory support for PIM architecture is challenging due to fundamental differences from traditional architectures. [2] Developing an OS that efficiently supports virtual address translation when programming for PIM architecture is a key issue.

Therefore, virtual memory support that fully leverages the capabilities of PIM is imperative. Additionally, it should be made compatible with existing address translation systems. One simple solution is to send all address requests for PIM operation logic to the virtual address space on the CPU-side [2]. However, every address translation results in a significant delay when communicating with the existing CPU-based address translation system, particularly in multi-level paging, where overhead is caused by page table walks as the CPU has to make multiple memory requests. This limits the advantages of using PIM.

In this paper, we propose a table-based virtual memory management technique for PIM architecture. To handle the page table walking overhead that occurs in the existing address translation process, we employ a technique that translates PIM virtual addresses to physical addresses. This is done with a lookup table within the PIM device driver each time the PIM application programming interface (API) is called in the user space. With this technique, PIM can be fully utilized without any hardware modifications or support, and unnecessary delays can be reduced by minimizing the page table walking overhead. We evaluate this technique using a full-system simulator and demonstrate an approximately 15.04 times improvement in PIM access time compared to existing systems.

2. Background

2.1 Operating system

Virtual Memory. Virtual memory is a technology that provides each program with its own memory space by using virtual addresses that are mapped to physical addresses. The address translation required for this mapping is performed by both the hardware and software of the CPU. The hardware employs a Memory Management Unit (MMU) to translate virtual addresses to physical memory addresses, while the software uses the virtual memory support of the OS to perform address translation. However, the current OS lacks virtual memory support and research for PIM architecture.

Page Table. A page table is a data structure used in computer systems that employ virtual memory. The OS stores mapping metadata for virtual addresses used by processes in the page table to manage them. The page table is organized as a tree structure, with each node having a data structure called a page table entry (PTE). The PTE contains mapping information between virtual and physical addresses, including page numbers and page frame numbers.

Typically, the OS uses a 4-level page table to map a process's virtual address space to a physical address space. Figure 1 shows the operation of a typical 4-level page table. Page table walking is performed to access the physical address space, which incurs overhead by accessing memory four or more times for walking a 4-level page table. The problem is that using CPU-side virtual memory to manage PIM operations or memory incurs the same overhead. Therefore, virtual memory support for PIM is needed to avoid page table walking overhead.

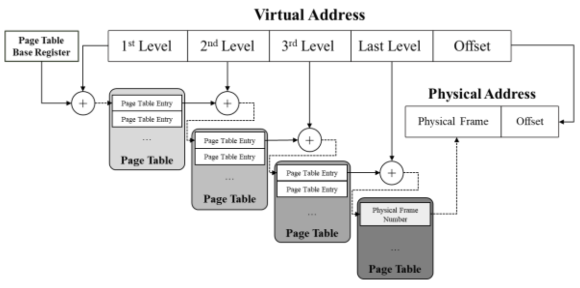

Figure 1. Conventional 4-level page table

Device Driver. A device driver is a software that provides an interface between the OS and a device. Among them, a PIM device driver is a device driver that controls PIM within the kernel space. This device driver communicates with the memory controller and sends commands to the PIM for data processing and computation, separate from the CPU. It then receives data and computation results processed inside the PIM and sends them to the CPU.

2.2 Related works

There are several studies [4], [5], [6] on virtual memory support for partially enabling PIM in actual systems. During the booting process, they specify certain physical addresses to enable access to PIM. Then, they allocate the designated PIM address space to virtual memory using provided APIs.

However, these studies have the limitation that the CPU cannot use this memory space for other purposes, and there are limitations in the memory footprint and resource efficiency of PIM due to the limited address space [3]. Additionally, some studies suggest that processing elements inside PIM can directly access physical addresses without any virtual memory support [4], or an additional SRAM buffer is required to access PIM [5]. These points demonstrate the limitations of requiring hardware support and modifications to use PIM.

Therefore, this paper proposes virtual memory support for PIM that is designed to operate based on existing hardware using flexible address space without requiring additional hardware support or modifications.

3. Proposed scheme

Table-based Management for PIM. In this paper, we propose a table-based management technique within the PIM device driver to enable efficient usage of PIM in user space. Our approach is aimed at reducing the overhead of page table walking in the existing CPU-side virtual memory support. As discussed in Section 2.1, the existing 4-level page table requires accessing memory more than four times during the page table walk, which can cause significant overhead. To address this issue, our technique quickly translates PIM virtual addresses to physical addresses, significantly improving access time to PIM.

When the user accesses PIM or offloads PIM tasks, our PIM device driver translates the virtual address of PIM into its physical address. To achieve this, the PIM device driver refers to a lookup table within the PIM device driver, separately from the existing page table. As shown in Figure 2, this lookup table contains mapping metadata for PIM physical addresses, enabling access to the desired PIM physical address by referencing the lookup table without incurring the overhead of the existing page table walking.

Figure 3 illustrates how the lookup table operates. ① During the kernel boot process, the device driver initializes to generate the lookup table and allocates space for it. The device driver then uses *memremap()* to map PIM physical addresses to kernel virtual addresses. The device driver writes the mapping metadata which contains PIM physical address and their corresponding 1:1 mapped kernel virtual address to the lookup table. Then, ② when the *PIM_enable* API is called from the user space, the device driver uses *mmap()* to map each kernel virtual address to its corresponding user virtual address, updating metadata for each mapping in the lookup table. Once all mapping processes are successful, the device driver sends a *ready* signal to the kernel, indicating that PIM is ready for use. At this point, ③ when the user offloads a PIM task using the *PIM_execute* API, the device driver supports the

PIM accelerator to access the required physical address by referencing the lookup table. It then performs the offloaded PIM task and sends the computed result back to the CPU via an interrupt signal. Since the mapped user virtual address space should not be changed by the CPU, it needs to be protected. Therefore, the system call *mlock* is used to prevent the user virtual address space from being mapped differently. ④ When the user is done using PIM, the *PIM_free* API is called, which frees the user virtual address space by using *munmap()*, ensuring the availability and flexibility of the user virtual address spaces.

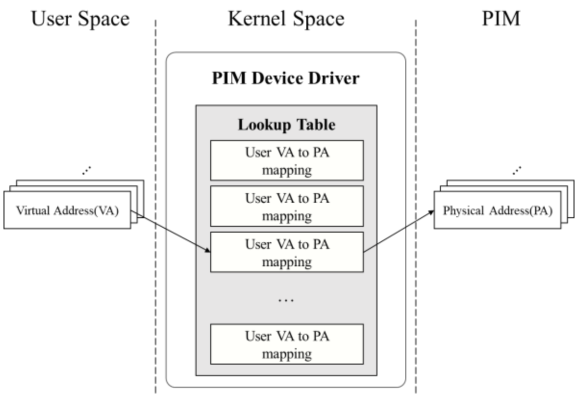

Figure 2. Proposed table-based management

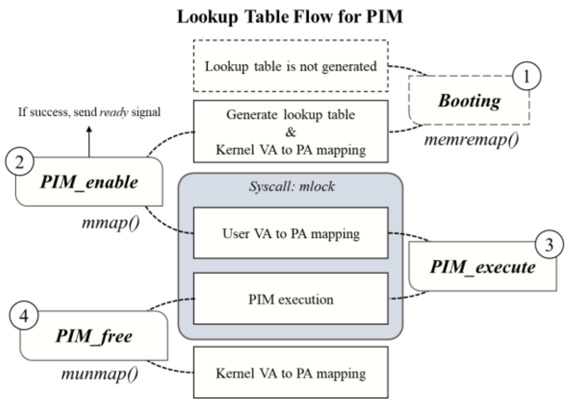

Figure 3. Lookup table flow within the PIM device driver

4. Evaluation

To evaluate the effectiveness of our proposed technique, we need a simulation environment that operates on the OS level. Therefore, we implemented our technique using the gem5 full-system simulator [7] based on Linux kernel 5.4.49. We constructed the system using an X86TimingCPU that uses a 4-level page table. We also implemented a functional PIM connected to the CPU via the PCI bus. We assumed that the physical address of the PIM was known and mapped it to the kernel's virtual address during the boot process. Additionally, we configured the kernel to manage the PIM as conventional CPU-side virtual memory during the boot process. To prevent memory swap operations, we conducted simulations with memory footprints of 500 MB, 1 GB, 2 GB, and 4 GB, using a memory size of 8 GB. The simulation configuration is shown in Table 1, and an overview of the gem5 full-system simulator we constructed is provided in Figure 5.

Table 1: Simulation configuration

CPU	X86TimingCPU, 1GHz, 4-level page table
OS	Linux kernel 5.4.49
Memory	DDR4_2400_8x8, 8GB
PIM	DDR4_2400_8x8, 8GB, PCI-based PE, PCI-based memory with device driver

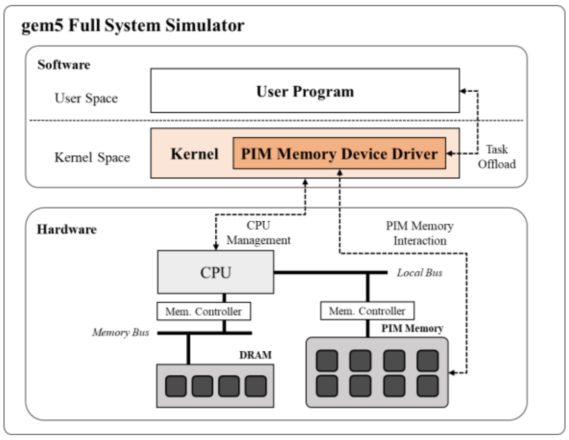

Figure 4. gem5 full system simulator platform

We measured the memory access time of the existing virtual memory support technique and our proposed table-based management technique with memory footprints of 500 MB, 1 GB, 2 GB, and 4 GB using simulation. We changed the kernel configuration to enable both techniques and conducted 10 simulations for each memory footprint. We then calculated the average simulation time for both techniques and compared the results. Our technique outperformed the existing method by reducing the delay caused by the large page table walking overhead. We also calculated the performance gain of our technique for each memory footprint, which showed an average improvement of approximately 15.04x compared to the existing method. Figure 5 summarizes our simulation results, including the simulation time and the performance gain of our proposed technique. These results demonstrate the effectiveness and efficiency of our table-based management technique for PIM.

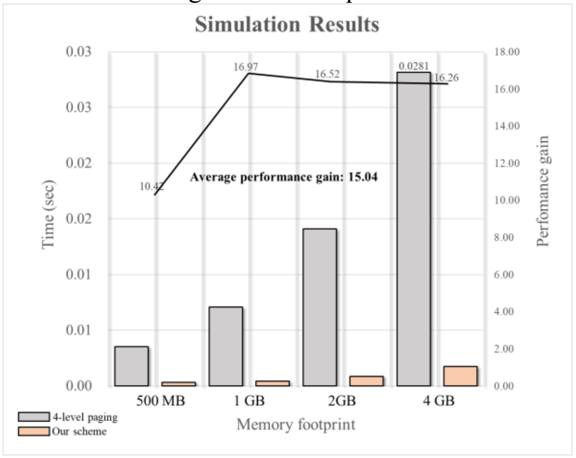

Figure 5. Simulation results

5. Conclusion

In conclusion, this paper proposed a novel table-based management technique for PIM virtual memory support, which significantly reduces the overhead of page table walks. By using a lookup table within the device driver, we achieved an average performance gain of approximately 15.04x compared to the existing method, without requiring any hardware modifications or support. Our approach was evaluated through simulations using the gem5 full-system simulator, and the results demonstrate the effectiveness of our proposed technique. In future work, we plan to investigate further system software improvements for PIM to enhance its performance and efficiency.

6. Acknowledgment

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT). (No.2021-0-00754, Software Systems for AI Semiconductor Design)

References

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

[2] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu, "Processing-in-memory: A workload-driven perspective," IBM Journal of Research and Development, 2019.

[3] D. Choi, T. Jeong, J. Yeom, and E.-Y. Chung, "Operand-oriented Virtual Memory Support for Near-Memory," IEEE Transactions on Computers, Early Access.

[4] 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.

[5] 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.

[6] 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.

[7] 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.