vPIM: Efficient Virtual Address Translation for Scalable Processing-in-Memory Architectures

Amel Fatima*, Sihang Liu†, Korakit Seemakhupt*, Rachata Ausavarungnirun‡, and Samira Khan*
*University of Virginia †University of Waterloo ‡King Mongkut's University of Technology North Bangkok

***Abstract*—3D-stacked memory technologies make it possible to integrate computation logic into the memory stack to reduce data movement between CPU and memory, enabling processing-in-memory (PIM). PIM systems scale in capacity and bandwidth by connecting multiple PIM stacks through a memory network. They also need to be programmable, where having virtual memory support is critical. Existing address translation schemes, however, are not optimized for a scalable PIM system. In this work, we propose VPIM, a virtual address translation scheme for scalable, multi-stack PIM systems. VPIM optimizes contention of the memory network in a PIM system and reduces translation time with pre-translation. Our evaluation shows a speedup of $4.4\times$ and $1.7\times$ compared to conventional radix and cuckoo hash page tables in eight memory-intensive workloads.**

I. Introduction

Data-intensive applications spend significant time and energy in moving data between CPU and memory [2, 3]. The 3D-stacked memory technologies, such as High Bandwidth Memory (HBM) [9], enable the integration of computation inside memory to reduce this data movement overhead. E.g., the logic layer in memory vaults can have computation units [2, 3]. This paradigm shift is called processing-in-memory (PIM).

A PIM system needs to be scalable, as a single memory stack has limited capacity and bandwidth. For scalability, multiple PIM stacks are connected over a memory network [3, 21]. We refer to accesses to other stacks as *cross-stack* accesses, which have much longer memory access latency. Another practical requirement for PIM systems is memory management. A PIM system can either use a separate physical address space [7, 17] or a unified virtual memory [16] between the CPU and PIM cores. Maintaining a separate physical address space introduces programming difficulties due to explicit data copying from/to the host processing. In comparison, a unified virtual memory is easier to program, and therefore, architecting virtual memory is desirable [15].

Support for a unified virtual memory comes at the cost of virtual-to-physical address translation. One approach is to forward the address translation from PIM to the CPU side [6, 19]. Such a scheme can introduce high communication overhead, overshadowing PIM's benefit of reduced data movement. An alternative approach is to add a Page Table Walker (PTWer) and a TLB near the PIM cores [6, 21]. This approach reduces the communication overhead and, is the approach that we will follow in this paper. However, the area constraint in PIM does not allow for a large TLB to mitigate Page Table Walks (PTWs). In a conventional radix page table, a PTW requires memory accesses in a pointer-chasing pattern to access the multi-level radix page table [25], which cannot benefit from the abundant memory parallelism in PIM. Even worse, this scheme introduces slower cross-stack access.

Hash page table [8] is an alternative approach. Recent proposals [25] integrate an efficient collision resolution technique, cuckoo hashing [20], that enables sending parallel accesses to the hash table, thus hiding the latency due to collisions. Unfortunately, direct integration of the cuckoo hash page table into PIM leads to parallel accesses toward different PIM stacks, causing increased network contention.

The goal of this work is to redesign the address translation scheme for scalable multi-stack PIM systems. We propose VPIM, a virtual address translation scheme designed for multi-stack PIM systems that leverages the parallelism within each PIM stack without incurring excessive cost due to cross-stack accesses, by adopting two key ideas.

Network-contention-aware hash: In Cuckoo hashing, parallel hash functions map an address to random hash table locations. We can limit the number of cross-stack access in a PTW if we map all indexes associated with a virtual address to the same stack. Based on this key insight, we propose a *network-contention-aware* scheme that hashes all the indexes of a virtual address to the same stack. This approach exploits the abundant memory bandwidth within a stack and minimizes cross-stack hash table accesses during a PTW.

Pre-translation: Besides memory parallelism, PIM also has abundant parallel processors. PIM-friendly programs usually feature high parallelism—often manifest as independent loops that parallelly execute on PIM processors. Thus, it is possible to execute future parallel iterations ahead of time to enable *pre-translation*. However, the limited vault area does not permit extra logic. Past research has shown that triggering memory accesses that lead to address translation only requires a small fraction of instructions as compared to the original procedure [11]. Thus, we propose to repurpose a few PIM cores as pre-translation cores to trigger future address translations, moving PTWs off the critical path.

We summarize the contributions as the following:

- This work proposes VPIM, an efficient address-translation scheme for scalable, multi-stack PIM systems.
- VPIM minimizes cross-stack hash page table accesses by hashing all indexes associated with an address to the same stack, and mitigates translation time with pre-translation.
- To evaluate VPIM, we model a scalable, multi-stack PIM system using ZSim [24] which shows a speedup of $4.4\times$ and $1.7\times$ over radix and Cuckoo hash table schemes.

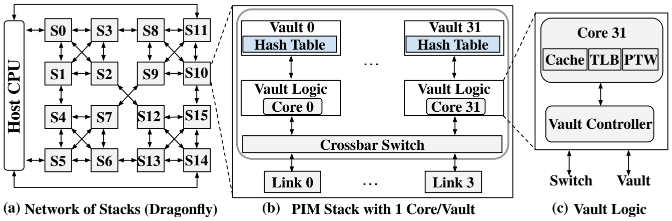


Fig. 1. Overview of PIM Stacks and the Architecture

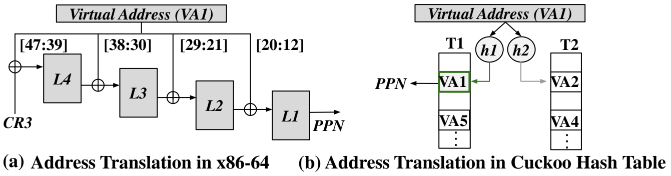


Fig. 2. Address Translation in (a) radix page table and (b) cuckoo hash table.

II. Background and Motivation

A. Processing-in-Memory (PIM) Architecture

Figure 1 shows the organization of the PIM system that is followed in the paper. It consists of 16 memory stacks connected over a memory network, following dragonfly topology [3, 21]. Each stack consists of 32 vertical DRAM partitions, called vaults, each with its vault controller and a low-profile processor in the logic layer, enabling vaults to operate in parallel. Vaults are interconnected with a crossbar switch.

B. Virtual Memory

Virtual memory aims to provide a view of a large memory address space by *translating* virtual addresses to physical addresses available in the system during each memory access. To enable virtual memory, the OS maintains page tables for the address mapping at page granularity, and TLBs in the CPU cache these mappings near the processor. Next, we will talk about different page table schemes.

Radix Page Table is a commonly used approach [1] which uses a radix tree to maintain the address mappings. Figure 2a demonstrates a 48-bit virtual address, wherein the 36 higher-order bits correspond to the virtual page number (VPN) and are divided into four 9-bit indexes; each index corresponds to one level in the radix tree (i.e., L4—L1). A translation is referred to as a page table walk (PTW), accessing four levels in the radix tree to obtain the physical page number (PPN), following a pointer-chasing memory access pattern.

Hash page table is an alternative design that uses a hash function to map VPNs to PPNs. Compared to the radix page table, the hash page table only takes one memory access. However, it suffers from a major overhead of hash collisions that can degrade its performance. Recent works [25] have proposed a resurgence of this page table design by adopting Cuckoo hashing [20] to mitigate hash collisions.

Cuckoo hashing [20] is a collision resolution algorithm for hash page tables, that allows an element to have *n* hash locations, i.e., n-ary. An entry can reside in one of these locations, which are looked up in parallel. Figure 2b shows lookup operation for the virtual address VA1 in a 2-ary cuckoo hash table, consisting of two hash functions (h1 and h2) that correspond to two tables (T1 and T2). Function h1 and h2 hash VA1 and generate two indexes that look up T1 and T2 in parallel, finally obtaining the target entry in T1.

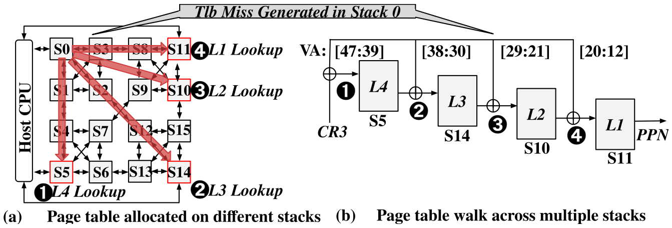


Fig. 3. Radix page table walk in a multi-stack PIM system.

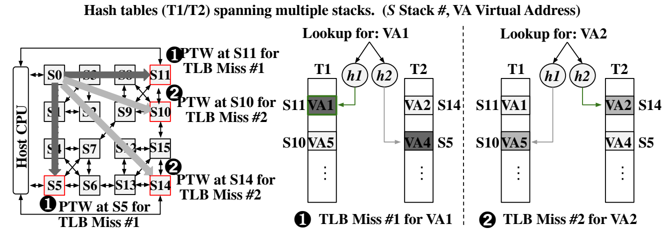

Hash tables (T1/T2) spanning multiple stacks. (S Stack #, VA Virtual Address)
Host CPU
S0
S3
S8
S11
S1
S2
S9
S10
S4
S7
S12
S15
S5
S6
S13
S14
1 PTW at S11 for TLB Miss #1
2 PTW at S10 for TLB Miss #2
2 PTW at S14 for TLB Miss #2
1 PTW at S5 for TLB Miss #1
Lookup for: VA1
T1
h1
h2
T2
S11
VA1
VA2
S14
S10
VA5
VA4
S5
1 TLB Miss #1 for VA1
Lookup for: VA2
T1
h1
h2
T2
S11
VA1
VA2
S14
S10
VA5
VA4
S5
2 TLB Miss #2 for VA2

Fig. 4. Cuckoo hash lookup in PIM.

C. Rethinking Page Table Structure Design

While virtual memory is desirable in PIM systems, a follow-up question is what address translation mechanism a multi-stack PIM system should adopt. A radix PTW makes pointer-chasing accesses across levels, and thus, cannot leverage the high memory bandwidth of PIM. On the other hand, the limited area in a vault cannot fit a large TLB to mitigate PTWs. Even worse, different levels of the page table access can span different PIM stacks. For example, Figure 3 demonstrates a TLB miss in stack S0 that leads to accesses to multiple stacks when walking through L4–L1. Such cross-stack accesses can increase access latency and contention in the memory network. In comparison, hash tables eliminate pointer-chasing accesses. Especially with Cuckoo hashing, the parallel accesses not only mitigate hash collisions but also exploit the memory parallelism in PIM systems. Unfortunately, direct integration of Cuckoo hashing into a PIM system leads to performance degradation due to slower cross-stack accesses. Therefore, there is a need to redesign the page table in PIM systems.

III. High-Level Ideas

Our goal is to enable efficient address translation in scalable, multi-stacked PIM systems. In this section, we discuss the challenges in integrating the Cuckoo hash page table into such a PIM system and present our key ideas.

A. Memory Network Overheads

Figure 4 demonstrates PTWs in a 2-ary Cuckoo-hash-based page table in PIM. Upon a TLB miss in stack S0 (step ①) for virtual address VA1, the hash functions h1 and h2 direct to two potential locations of this entry that are held by stacks S11 and S5, leading to two parallel accesses to these stacks.

These parallel lookups reduce the PTW latency compared to the radix page table that performs serialized lookups.

Challenge: Parallel lookups in cuckoo hash tables can lead to contention in the network. We study the impact of parallel page table lookups on cross-stack memory accesses. We observe that the *vault service time* remains largely the same but the *network traversal time* for a memory request increases with more parallel accesses, as concurrently issued hash table lookups contend with other memory accesses in the memory network, leading to performance degradation.

Our Approach: Network-contention-aware hash: The concurrent cross-stack PTWs are the primary source of the overhead in a Cuckoo hash design. We observe that the placement of hash table entries is determined by the hash functions. If the hash functions map the output indexes to the same stack, there needs only one cross-stack access during the PTW, which can trigger multiple parallel page table accesses, significantly reducing the number of cross-stack accesses. Therefore, we propose a *network-contention-aware* hashing scheme. Figure 5 shows a high-level view of our approach. The core components of our approach are two types of hash functions: The *stack hash function*, hS (S stands for stack), generates an index that directs to a stack where other hash table lookup operations will be performed. The *vault hash functions*, hV's (V stands for vault), generate the remaining indexes to vaults within the same stack selected by hS (Details of index generation in section IV-A2). Figure 6b demonstrates the execution timeline in Figure 5. Figure 6a and 6b compare a naive Cuckoo hash table with our network-contention-aware scheme and show that our approach significantly reduces the network traversal time of memory requests.

B. Cross-Stack PTW Overheads

Challenge: A system without PTW overhead is on average 1.4× faster than the system with our network-contention-aware hashing scheme, leaving optimization opportunities on the table. Ideally, one would like to have TLB misses identified ahead of time, to overlap PTWs with program execution.

Our Approach: Pre-translation: A PIM system has many low-profile cores, which as a whole, provide massive parallelism. Thus, programs suitable for PIM offloading also feature high parallelism and manifest as independent loops that can be assigned to different cores and executed in parallel [2]. Given such an execution pattern, it is possible to execute future work (i.e., the next parallelizable iteration) ahead of time, making it possible to *pre-translate* cross-stack PTWs before a future iteration happens. However, the area for processing logic in memory stacks is limited, so the goal is to pre-translate without extra logic. We found that, on average, the work needed for a normal iteration is 4.22× compared to the minimum code required to trigger TLB misses. This implies that a few cores dedicated to pre-translation work can assist multiple main cores. We also measured the execution time with the number of main cores reduced from 512 to 480, only leading to a 6% degradation. Therefore, we propose to use *a small fraction* of PIM cores as pre-translation cores to assist the main cores. Figure 6c shows a high-level view of the pre-translation scheme. The pre-translation cores execute simplified code of a future code section ahead of the progress of main cores. In step ①, the pre-translation core triggers a TLB miss ahead of time and stores the translation result to a *pre-translation buffer (PB)* which is located in the pre-translation core's local vault as shown in Figure 8. In step ②, the main core, after encountering a TLB miss, access the *PB* within the stack, to read the pre-translated result. Upon another TLB miss in step ③, the main core misses in the *PB* as the pre-translation result is not available and falls back to the normal cross-stack PTW. We can get significant performance benefit by only reducing a fraction of the cross-stack PTWs.

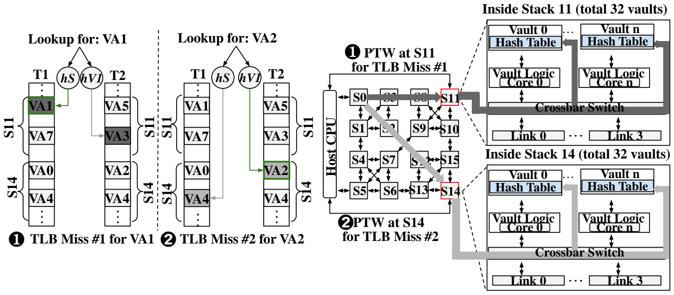
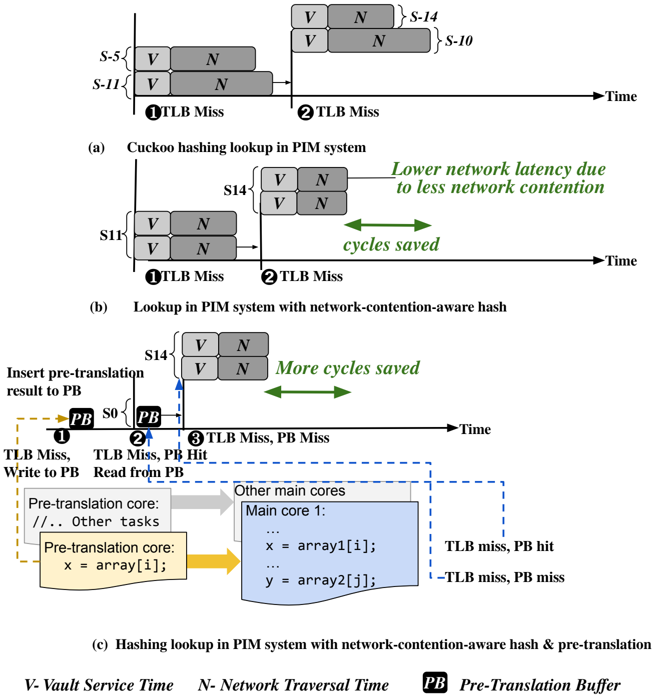


Fig. 5. Page table based on the network-contention-aware Cuckoo hash.



```
Pre-translation core:
//.. Other tasks
```

```
Pre-translation core:
  x = array[i];
```

```
Main core 1:
  ...
  x = array1[i];
  ...
  y = array2[j];
```

(a) Cuckoo hashing lookup in PIM system

(b) Lookup in PIM system with network-contention-aware hash

(c) Hashing lookup in PIM system with network-contention-aware hash & pre-translation

Fig. 6. An overview of pre-translation for cross-stack PTWs.

IV. Mechanism of vPIM

In this section, we describe the mechanisms of vPIM and demonstrate them with walk-through examples.

A. Network-Contention-Aware Hashing

We describe the page table allocation and the hashing scheme that enable network-contention-aware hashing.

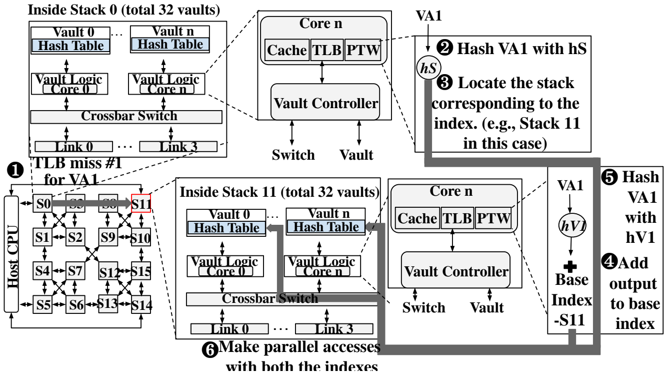
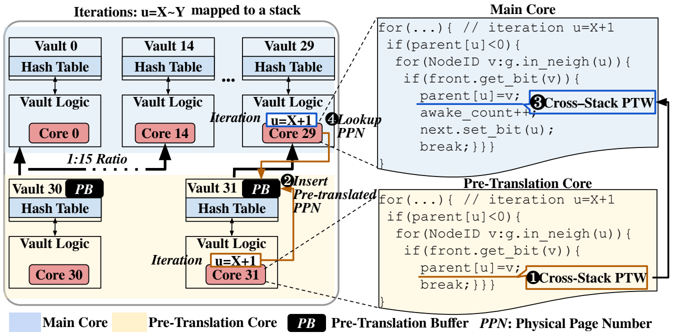


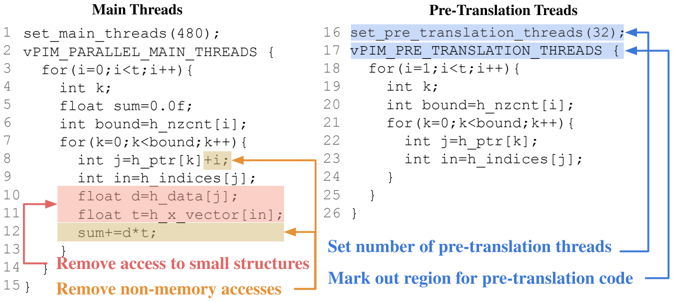
Fig. 7. Mechanism of the network contention-aware hashing scheme.



```
for(...){ // iteration u=X+1
 if(parent[u]<0){
  for(NodeID v:g.in_neigh(u)){
   if(front.get_bit(v)){
     parent[u]=v;
     awake_count++;
     next.set_bit(u);
     break;}}}
}
```

```
for(...){ // iteration u=X+1
 if(parent[u]<0){
  for(NodeID v:g.in_neigh(u)){
   if(front.get_bit(v)){
     parent[u]=v;
     break;}}}
}
```

Fig. 8. The pre-translation mechanism.


Main Threads
Pre-Translation Treads
Remove access to small structures
Remove non-memory accesses
Set number of pre-translation threads
Mark out region for pre-translation code

```
1  set_main_threads(480);
2  vPIM_PARALLEL_MAIN_THREADS {
3    for(i=0;i<t;i++){
4      int k;
5      float sum=0.0f;
6      int bound=h_nzcnt[i];
7      for(k=0;k<bound;k++){
8        int j=h_ptr[k]+i;
9        int in=h_indices[j];
10       float d=h_data[j];
11       float t=h_x_vector[in];
12       sum+=d*t;
13     }
14   }
15 }
```

```
16 set_pre_translation_threads(32);
17 vPIM_PRE_TRANSLATION_THREADS {
18   for(i=1;i<t;i++){
19     int k;
20     int bound=h_nzcnt[i];
21     for(k=0;k<bound;k++){
22       int j=h_ptr[k];
23       int in=h_indices[j];
24     }
25   }
26 }
```

Fig. 9. Code filtering for the pre-translation cores.

1) Allocation of Hash Page Table in vPIM*:* In the baseline cuckoo hash table, allocation of the hash page table is done, irrespective of the memory stack. Therefore, a PTW may access the hash tables located on different memory stacks, leading to increased contention in the memory network. To mitigate the performance interference caused by the cross-stack PTWs, vPIM allocates an equal fraction of the hash page table to every stack. Each fraction belonging to a specific stack has a fixed base index, *Base Index* (BI). This allocation scheme enables directing all hash indexes of a virtual address to the same stack. Next, we describe the hashing scheme.

2) Generation of Indexes: vPIM uses two types of hash functions. The first type is the *stack hash function*, hS, which generates indexes ranging from the first to the last entry in the hash table. It indicates the index to the first hash table (hS_Index) and the target stack ID for remaining parallel accesses. Specifically, hS takes the following steps:

$$hS\_Index = hS(VPN) = Mask(SHA1(VPN), [0, n-1]),$$

where n is the total number of entries and$Mask$ converts the hash value (from SHA-1 algorithm) to a range between 0 and $n-1$. The stack ID is obtained from the mapping according to the Base Indexes set during the page table allocation time. The second type of hash function is the *vault hash function*, hV. It generates an index to the same stack as hS but probably different vaults inside that stack. In a Cuckoo hash scheme with two parallel hash tables, i.e., 2-ary, there is one vault hash hV1 that indexes to the second hash table. With more hash tables, say n-ary, there will be $n-1$ value hash functions: hV1 ... hV(n-1), each corresponding to one hash table. Back to our assumed 2-ary system, hV1 takes the following steps:

$$\begin{aligned} hV1\_Index =& hV1((VPN) \\ &= Mask(SHA1(VPN), [0, (n/s)-1]) + BI(StackID), \end{aligned}$$

where n is the total number of hash entries and s is the number of stacks. The final index from hV1 is the original hash value converted to a range between *0* and *(n/s)-1* using *Mask*, plus the Base Index (*BI*) associated with the selected stack ID.

3) Example: Figure 7 shows a step-by-step example to demonstrate our network-contention-aware hashing design. Step ①: execution in stack S0 encounters a TLB miss when accessing VA1. Step ②: in stack S0, the virtual address VA1 is first hashed by *hS*. Step ③: hS directs the translation to vault 0 of stack S11. Step ④: in stack 11, *VA1* is further hashed with the vault hash function *hV1*. Step ⑤: the value generated from *hV1* is added to the emphBase Index for stack S11, and generates an index for the second page table in stack S11 (located in vault n of stack S11 in this example). Step ⑥: the two hash indexes generated by hS and hV2 are then accessed in parallel within stack S11. One of them returns the final translation result (the other one can be a collision).

B. Pre-translation

This mechanism repurposes a small number of cores for pre-translation in each stack, which runs ahead of the remaining main cores within the stack. The pre-translation results are stored in a *pre-translation buffer* (PB), which cache the virtual-physical address mappings and is located in the pre-translation core's vault as shown in Figure 8.

1) Programming Interface: Threads for the main computation work are referred to as *main threads* and those for pre-translation as *pre-translation threads*. Figure 9 demonstrates an example that uses the vPIM library to define main threads and pre-translation threads. First, functions `set_main_threads` (line 1) and `set_pre_translation_threads` (line 16) define the number of cores allocated to main threads and pre-translation threads, respectively. Code wrapped by `vPIM_PARALLEL_MAIN_THREADS` is for main threads, which are automatically generated from loops (line 3). This approach is derived from the OpenMP wrapper [18]. The vPIM library pins these main threads to the main cores using Linux's scheduler affinity interface. Likewise, code wrapped by `vPIM_PRE_TRANSLATION_THREADS` is

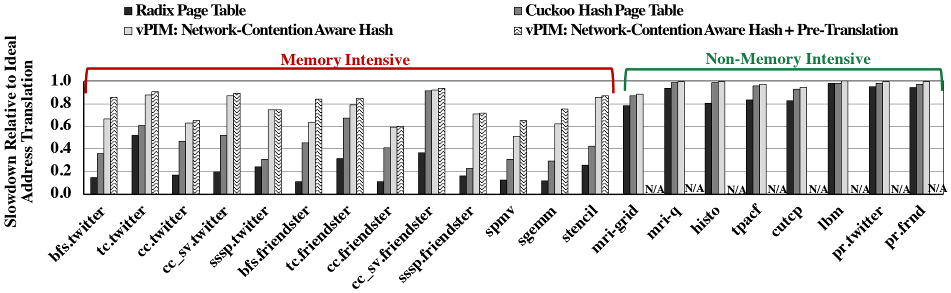


Fig. 10. Performance of vPIM over the baselines.

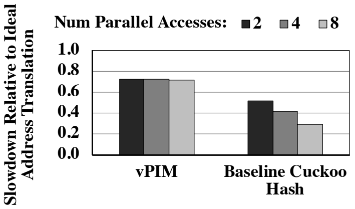


Fig. 11. Performance with varying parallel accesses to hash table.

for pre-translation threads which are also generated from loops. However, the vPIM library, assigns the loop iterations to these threads in a way that each pre-translation thread encounters the future iterations for all its associated main threads in a round-robin manner. We take an approach similar to past works on helper threads for prefetching [11], to minimize our code. Instructions that access smaller data structures (less likely to trigger TLB misses), non-memory instructions, and instructions that change the memory state, are all removed.

2) Coarse-Grained Synchronization between Main and Pre-Translation Cores: Pre-translation cores assist a specific set of main cores within a stack. Thus, they have to ensure that they remain ahead of those cores in order to trigger useful PTWs for future use. The parallelization wrappers, `vPIM_PARALLEL_MAIN_THREADS`, and `vPIM_PRE_TRANSLATION_THREADS` take the loop variable, e.g., `i` in Figure 9, to track the progress. Since the pre-translation core starts the execution an iteration ahead of the main cores, vPIM library checks if at least the distance (i.e., +1) is maintained between the main and pre-translation threads after the completion of a set of iterations (This ensures coarse-grained synchronization to minimize synchronization overhead). If the checking fails, the vPIM library moves the pre-translation thread ahead of the main threads, i.e., increment the distance to be `+1` of the main thread.

3) Walk-Through Example of Pre-Translation: Figure 8 shows a walk-through example of pre-translation. Step ①: pre-translation core in a stack executes iteration *X+1* and triggers a TLB miss, resulting in a PTW. Step ②: it performs the PTW and buffers the translation in its *pre-translation buffer* (PB). Step ③: the main core starts iteration *X+1* later and encounters a TLB miss when accessing the same address. Step ④: the main core looks up the PB in the same stack before performing a PTW, and hits PB. Thus, it no longer needs a PTW.

V. Evaluation Methodology

A. System Configuration and Design Points

We model our PIM cores in ZSim [24], memory system in Ramulator, [13] and memory network in BookSim2 [10]. Table I lists our system configuration. In vPIM, the only additional structure are the two PBs per stack (each has 1024 entries or 16kB), associated with the two pre-translation cores.

TABLE I
Simulated system configuration.

PIM Cores	
Processor	512 in-order cores; 1/vault, 1-issue, 2GHz
L1 Cache	16 kB I/D, 4-way, 64 B cache line, LRU
I&D TLB	64 entries, LRU (per core)
Page Table Walker	1 (per core)
Memory	
Number of Stacks	16
Capacity per Stack	4 GB, 4 layers, 8 banks per vault
Vaults per Stack	32, 32 TSV per vault
Maximum Bandwidth	10GB/s per Vault, 120GB/s per Link

We compare the performance of four designs: (1) a conventional Radix page table, (2) a Cuckoo hash page table without optimizations for PIM, (3) a Cuckoo hash with our network-contention-aware hashing from vPIM (4) a Cuckoo hash with network-contention-aware hashing and pre-translation from vPIM. Baseline Cuckoo hash and vPIM use 2-ary hash tables.

B. Workloads

We evaluate the workloads from Gapbs [5] and Parboil [26] benchmark suites. The Gapbs workloads take two real-world input graphs: Friendster [22] and Twitter [14]; the parboil workloads take the largest dataset in the benchmark suite. These workloads fall into two categories: memory-intensive and non-memory-intensive. We follow the evaluation strategy of prior works [23, 25], by performing evaluation of vPIM on both categories and then performing sensitivity studies on memory-intensive workloads only. The simulation starts from a warm-up stage, and then enters a region-of-interest of a total of 500 M instructions that perform the core tasks on PIM.

VI. Evaluation Results

A. Overall Performance and Performance Breakdown

Figure 10 compares the slowdown due to address translation of our design points over an ideal system without address translation overhead. Overall, vPIM, with both network-contention-aware hashing and pre-translation, has a speedup of 3.18× and 1.44× over the radix and Cuckoo hash page tables. For memory-intensive workloads, the speedup is 4.4× and 1.7×respectively. Pre-translation reduces the number of main cores and thus, only workloads consuming more than 20% of execution time on address translation (through profiling) will enable the pre-translation cores. Thus, non-memory intensive workloads do not enable pre-translation (N/A in Figure 10).

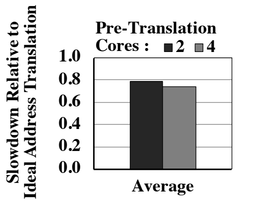


Fig. 12. Performance with varied pre-translation cores per stack.

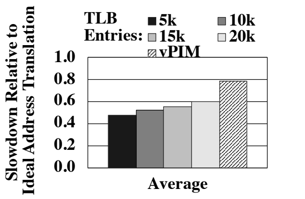


Fig. 13. Performance comparison with in-memory TLB.

Figure 10 further shows the speedup from each optimization in vPIM. We observe in Figure 10 that the network-contention-aware hashing outperforms both baselines and the performance trend from pre-translation falls into three categories. (1) Workloads with significant performance improvement, e.g., *spmv*, *sgemm*, and *bfs*, where pre-translation cores run fast enough to pre-translate a large number of iterations for the main cores. (2) Workloads with limited margin to improve, e.g., *tc*, *cc_sv*, and *stencil*, that already achieve close-to-ideal performance, with network-contention-aware hashing. (3) Workloads with limited improvement, e.g., *cc* and *sssp*, that have limited room to be simplified for pre-translation cores, resulting in long execution time for pre-translation.

B. Sensitivity Study-Parallel Access and Pre-Translation Core

Figure 11 shows that more parallel accesses degrade performance in the Cuckoo hash page table, due to increased network contention. In comparison, because of the contention awareness, vPIM does not have major performance degradation with more parallel hash table accesses. Figure 12 shows the average performance with varying numbers of pre-translation cores per stack. The benefit is a result of a trade-off between the address translation overhead and the reduced performance due to fewer cores for the workload. For our workloads, we see the best performance with two pre-translation cores.

C. Comparison with in-Memory TLB

Figure 13 compares vPIM with an in-memory TLB [23] (5 k to 20 k entries). We observe that vPIM performs significantly better since memory-intensive workloads do not have a good locality and incur frequent TLB misses. In-memory TLB design aims to reduce the TLB miss rate, which is orthogonal to vPIM's goal of reducing the TLB miss penalty. Thus, the two designs can work collaboratively for better performance.

VII. Related Work

To reduce the overhead of address translation, prior works have proposed to optimize the TLB organizations by prefetching [12], large part-of-memory TLB [23], and speculative TLBs [4]. However, as PIM systems have a limited area for computation logic, these complicated hardware designs do not apply. Thus, this work takes an approach that reduces the TLB miss penalty. There have also been proposals on hash page tables [25] that reduce the PTW latency. However, they are designed for conventional, CPU-centric systems whereas this work targets scalable, multi-stack PIM systems.

VIII. Conclusions

We propose vPIM that optimizes address translation in (PIM) systems by introducing a network-contention-aware hashing scheme that efficiently integrates hash-table-based address translation in PIM. Moreover, vPIM repurposes some PIM cores for pre-translation to move page table walks off the critical path of execution. Our evaluation, done using ZSim, shows a speedup of 4.4× and 1.7× compared to the radix and cuckoo hash page table, in eight memory-intensive workloads.

IX. Acknowledgement

We thank the anonymous reviewers for their valuable feedback. This work is supported by the SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP), National Science Foundation (NSF), and Thailand MHESI Research Grant for New Scholars (RGNS 64-091).

References

[1] R. Agarwal *et al.*, "The intel 80386 architecture and implementation," *IEEE Micro*, 1985.

[2] J. Ahn *et al.*, "PIM-Enabled Instructions: A low-overhead, locality-aware processing-in-memory architecture," in *ISCA*, 2015.

[3] ——, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015.

[4] T. W. Barr *et al.*, "Spectlb: A mechanism for speculative address translation," *SIGARCH Comput. Archit. News*, 2011.

[5] S. Beamer *et al.*, "The GAP benchmark suite," *arXiv*, 2015.

[6] M. Gao *et al.*, "Practical near-data processing for in-memory analytics frameworks," in *PACT*, 2015.

[7] P. Gu *et al.*, "ipim: Programmable in-memory image processing accelerator using near-bank architecture," in *ISCA*, 2020, pp. 804–817.

[8] J. Huck *et al.*, "Architectural support for translation table management in large address space machines," in *ISCA*, 1993.

[9] JEDEC, "High Bandwidth Memory (HBM) DRAM," Standard No. JESD235, 2013.

[10] N. Jiang *et al.*, "A detailed and flexible cycle-accurate network-on-chip simulator," *ISPASS*, pp. 86–96, 2013.

[11] M. Kamruzzaman *et al.*, "Inter-core prefetching for multicore processors using migrating helper threads," in *ASPLOS*, 2011.

[12] G. Kandiraju *et al.*, "Going the distance for TLB prefetching: An application-driven study," in *ISCA*, 2002, pp. 195–206.

[13] Y. Kim *et al.*, "Ramulator: A fast and extensible dram simulator," *CAL*, 2016.

[14] H. Kwak *et al.*, "What is Twitter, a social network or a news media?" in *WWW*, 2010.

[15] J. Lowe-Power *et al.*, "Supporting x86-64 address translation for 100s of GPU lanes," in *HPCA*, 2014.

[16] Z. S. H. Michael J. K. Nielsen, "Unified memory computer architecture with dynamic graphics memory allocation," U.S. Patent 6,104,417, Aug. 15 2000.

[17] T. Nowatzki *et al.*, "Stream-dataflow acceleration," in *ISCA*, 2017.

[18] OpenMP, "The OpenMP API specification for parallel programming," https://www.openmp.org/, 2022.

[19] M. Oskin *et al.*, "Active pages: A computation model for intelligent memory," in *ISCA*, 1998.

[20] R. Pagh *et al.*, "Cuckoo hashing," *Journal of Algorithms*, 2004.

[21] J. Picorel *et al.*, "Near-memory address translation," in *PACT*, 2017.

[22] R. A. Rossi *et al.*, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[23] J. H. Ryoo *et al.*, "Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb," in *ISCA*, 2017.

[24] D. Sanchez *et al.*, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA*, 2013.

[25] D. Skarlatos *et al.*, "Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism," in *ASPLOS*, 2020.

[26] J. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.