PIMSim: A Flexible and Detailed Processing-in-Memory Simulator

Sheng Xu, Xiaoming Chen, Ying Wang, Yinhe Han, Xuehai Qian, and Xiaowei Li

Abstract—With the advent of big data applications and new process technologies, Process-in-Memory (PIM) attracts much attention in memory research as the architecture studies gradually shift from processors to heterogeneous aspects. How to achieve reliable and efficient PIM architecture modeling becomes increasingly urgent for the researchers, who want to experiment on critical issues from detailed implementations of their proposed PIM designs. This paper proposes PIMSim, a full-system and highly-configurable PIM simulator to facilitate circuit-, architecture- and system-level researches. PIMSim enables architectural simulation of PIM and implements three simulation modes to provide a wide range of speed/accuracy tradeoffs. It offers detailed performance and energy models to simulate PIM-enabled instructions, compiler, in-memory processing logic, various memory devices, and PIM coherence. PIMSim is open source and available at *https://github.com/vineodd/PIMSim*.

Index Terms—Processing-in-memory, simulator, heterogeneous computing, memory system

1 Introduction

THE technological evolution in the past decade is widening the performance gap between memory and processor, which is known as the "memory wall" bottleneck [1]. A revolutionary way for such dilemma is to move part of the computation to memory and let memory process data, known as Process-In-Memory (PIM) [2], [3], [4], [5], [11], [13], [14], [16], [17], [22]. Fig. 1 shows a conventional computer architecture and a PIM architecture. The primary innovation of PIM is the integration of lightweight computing logics into the memory. As such, some of the computation is moved from processors to memory, bringing two benefits. First, the in-memory computing logic has access to a much higher internal bandwidth than an off-chip bandwidth. Second, some data movements between the memory and processors are eliminated, which partially alleviates the memory wall problem.

In recent years, there reaches a pinnacle of PIM research in both academia and industry. To rapidly investigate new PIM techniques for research purpose, most of the PIM-related researches are based on simulation tools. As shown in Fig. 2, since 2013, the verification of PIM has started shifting from hardware tape-out for specified operations to software simulation for specified applications. The use of simulation tools has greatly promoted PIM-related researches.

However, there are many challenges and difficulties in PIM simulation. First, existing PIM simulation environments are usually composed of several different models including memory, processors, compiler, coherence, etc., lacking a consolidated architectural organization and abstraction. Some of the existing PIM experiments involve multiple separated simulators so that a completely automatic simulation process is almost impossible [13]. For example, one needs to generate trace files by a processor simulator and then feed them into a memory simulator. Second, most of existing PIM simulation environments are highly customized, i.e., a simulation environment is targeted at a certain kind of Instruction Set Architectures (ISAs) and designs, lacking versatility and flexibility in both configuration and modeling. For example, Smart Memory Cube (SMC) [10] specifies Hybrid Memory Cube (HMC) as the memory model and Advanced RISC Machine (ARM) as the ISA, which is not applicable to other memory models, ISAs or architectures. Furthermore, no flexible instruction-level PIM kernel (i.e., the codes that are executed by the in-memory logic) selection methods are provided, resulting in low flexibility and versatility. Third, existing PIM simulations ignore data coherence between processor and memory, new memory characteristics or co-simulation of different memory types.

Although the concept of PIM has been studied for years, the above issues call for new tool designs in the PIM simulation area. To overcome the difficulties in PIM modeling and simulation, we present PIMSim, a highly configurable PIM simulator. It integrates three accurate memory simulators, DRAMSim2 [20], HMCSim [6] and NVMain [7], to support hybrid memory simulations with new memory characteristics. To trade off speed and accuracy, we implement three different simulation modes, a full-system mode, an instrumentation-driven mode, and a fast mode. Experiments show that PIMSim achieves believable performance and energy reports in any mode. To guarantee the correctness of memory accesses, PIMSim provides a unified PIM coherence framework, which is expected to enable data coherence related research in PIM.

2 PIMSim: Architecture

In PIM, memory means not only a storage component but also a computing end. However, the processors cannot be eliminated because they have to provide necessary PIM instruction decoding and distribution. Thus, PIMSim relies on current architectural simulation with added components to support PIM simulation. Fig. 3 shows the top-level diagram of PIMSim. The added components are marked with the dotted line frame. The supported models and features of PIMSim are also listed in Table 1.

2.1 Application Partitioner

The frontend of PIMSim is an application partitioner, which recognizes and distributes PIM instructions. The instructions that are executed in memory are called PIM kernels. The selection of PIM kernels greatly determines the system performance. Existing PIM simulators do not provide instruction-level PIM kernel selection, which makes them tough and incapable. For example, SMC [10] makes the entire program as a single PIM kernel, prohibiting users from selecting finer-grained PIM kernels, which has great limitations for applications that can be beneficial from instruction-level offloading. To address this limitation, PIMSim provides four flexible PIM annotation styles and a corresponding compiler to let users define PIM kernels in the source code, with negligible impact on the programming model.

Besides the provided application partitioner, PIMSim also supports dynamic feedback to decide whether a PIM instruction should be executed in memory, which takes advantages of tracking execution factors such as cache hit rate or locality like PEI [19]. During execution, users can insert a self-defined monitor to watch the specified metrics such as Instructions Per Cycle (IPC) or memory access intensity and use them as feedback to dynamically adjust their own application partitioning strategy.

- *S. Xu and X. Li are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China and also with the University of Chinese Academy of Sciences, Beijing 101408, China. E-mail: {xusheng02, lxw}@ict.ac.cn.*
- *X. Chen, Y. Wang, Y. Han are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. E-mail: {chenxiaoming, wangying2009, yinhes}@ict.ac.cn.*
- *X. Qian is with the University of Southern California, Los Angeles, CA 90007. E-mail: xuehai.qian@usc.edu.*

Manuscript received 10 July 2018; revised 30 Nov. 2018; accepted 3 Dec. 2018. Date of publication 6 Dec. 2018; date of current version 14 Jan. 2019.
(Corresponding author: Sheng Xu.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2018.2885752

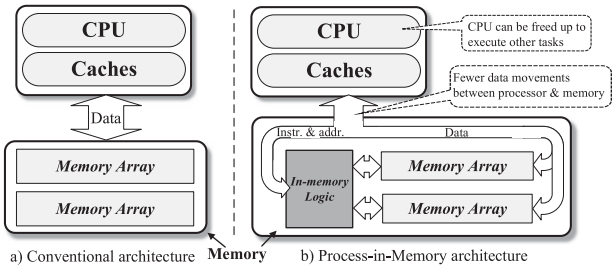

Fig. 1. Conventional computer architecture and PIM.

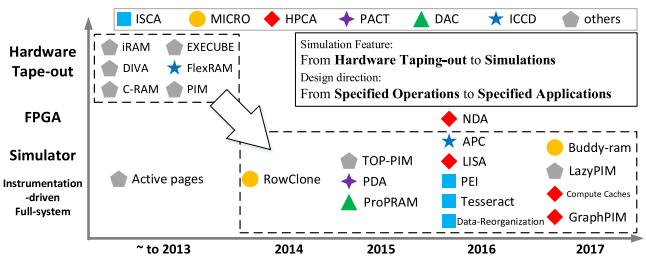

Fig. 2. Evaluation methods of recent PIM designs.

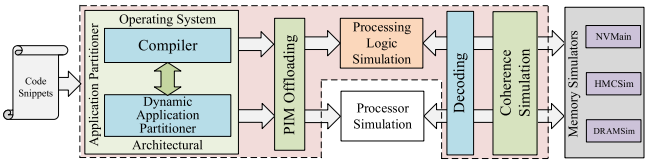

Fig. 3. Top-level architectural diagram of PIMSim.

2.2 PIM Logic Modeling

The in-memory logic is called PIM Logic. Its design is a key factor which has great impact on system performance. However, user-specified PIM Logic designs can vary from logic gates to processor cores, which increases complexity in simulator design. In PIMSim, PIM Logic can be easily configured as either processors or specified accelerators. In the full-system mode of PIMSim, all of the processor models provided by GEM5 [8] can be configured as PIM Logic. Besides, PIMSim also provides an interface that makes it co-work with a processor simulator to mimic the target PIM cores. For example, when one wants to integrate GPGPU-Sim [18] into PIMSim and treat a graphics processing unit (GPU) as the PIM Logic, he/she just needs to implement all the interfaces and PIMSim can invoke GPGPU-Sim automatically. In the fast simulation mode, PIMSim provides a fast but accurate core and a pipeline model to conduct PIM Logic simulations.

PIMSim also provides a high-level black-box model for rapidly verifying architecture designs. Users just need to supply the computing progress, input and output registers, etc. for PIM kernels, with performance parameters such as computing latency, energy consumption per operation, etc. PIMSim estimates performance based on the supplied black-box information without conducting detailed instruction-level simulations.

2.3 Memory Organization

PIMSim integrates some frequently used memory simulators and their features. We have surveyed the requirements by PIM researchers and industrial products, showing that the most common memory models are Dynamic Random-Access Memory (DRAM), Hybrid Memory Cube (HMC) [12], and a series of Non-Volatile Memory (NVM). To support such requirements, we integrate DRAMSim2 [20], HMCSim [6] and NVMain [7] into the PIMSim, so that PIMSim provides the ability of co-simulation and cross-simulation of them. To address issues caused by co-simulation of different memory simulators, we modified and provided uniform interfaces for all the integrated memory simulator.

TABLE 1
Supported Models and Features of PIMSim

Components	Supported models by default	Customizable
Programming Model/Compiler	Four basic styles (PIMSim)	√
Application Partitioning	Static and Dynamic (PIMSim)	√
PIM Logic	Processors (PIMSim, GEM5) and Black-Box Model (PIMSim)	√
Coherence	Fine- and Coarse-Grained Coherence (PIMSim)	√
Interconnection	Bus, Crossbar (GEM5)	√
Data Mapping Policy	Under development	√

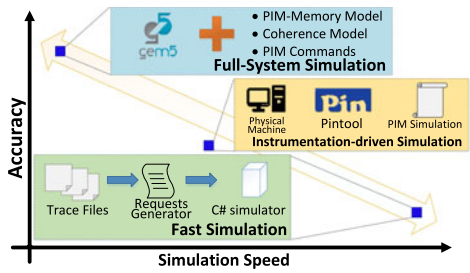

Fig. 4. Three simulations modes considering speed/accuracy tradeoff in PIMSim.

3 PIMSim: Design Features

In this section, we describe the design features of PIMSim that are different from existing PIM simulation environments. We first introduce three simulation modes of PIMSim and how they trade off speed and accuracy. We then describe PIM-enabled instructions and the implementation. Finally, we provide a closer look to the coherent hierarchy design in PIMSim.

3.1 Three Simulation Modes with a Trade-Off Between Speed and Accuracy

To provide flexible PIM simulations and meet different simulation requirements, PIMSim delivers a wide variety of capabilities and components in simulating PIM architectures, which vary in multiple dimensions and cover a wide range of speed and accuracy trade-offs. PIMSim provides three simulation modes: fast, instrumentation-driven and full-system simulations, as shown in Fig. 4.

Fast Simulation. In this mode, PIMSim avoids modeling unnecessary devices and operating systems to give a quick overview of PIM architectural designs. Only primary models, such as processors, memory, PIM Logic, and simple lock-based PIM coherent methods [19], are simulated. The input files are memory traces that include memory access operations, cycles, and data, which can be generated by other simulators, making it swift and flexible to migrate previous work to PIM architectures.

Instrumentation-driven Simulation. In this mode, PIMSim provides an efficient tool that uses Pin [9] as a frontend to feed real-time traces to the simulator by instrumenting. Thus, PIMSim is able to simulate PIM simultaneous during the application runtime. Upon the Pin tool found the instruction sequences that match the user-defined instruction sequences to be offloaded, it refers to architecture- and system- level characteristics and dispatches them as PIM operations to the simulator.

Full-System Simulation. In the full-system mode, PIMSim executes both user- and kernel-level instructions and models a

```
1:  #PIM_START
2:  #ID = 0          //define the execution logic ID
3:  #INPUT = a
4:  #INPUT = b
5:  #OUTPUT = c
6:      c = a + b;
7:  #PIM_END
```

```
1:  #PIM_START
2:  #ID = 0  //define the execution logic ID
3:      c = a + b;
4:  #PIM_END
```

```
1:  #PIM_CODE_ADD
2:  #ID = 0     //define the execution logic ID
3:  #DEFINE_INPUT Input1
4:  #DEFINE_INPUT Input2
5:  #DEFINE_OUTPUT Output1
6:          Output1 = Input1 + Input2;
7:  #PIM_END
8:      PIM_ADD(c, a, b);
```

```
1:  PIMKernel(c, a, b, 0); //the execution logic ID is 0
2:  //The function "PIMKernel" is preset in PIMSim
      to perform predefined simple kernerls
```

Fig. 5. Four examples of annotating PIM kernel codes in PIMSim to calculate $"c = a + b"$.

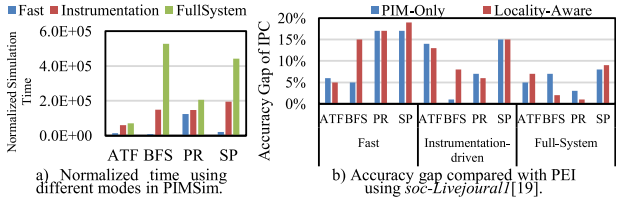

Fig. 6. Evaluation results of PIMSim.

complete system including the operating system, memory, and peripherals. PIMSim also implements detailed PIM-enabled instructions to support user-defined PIM instructions. Detailed coherence mechanisms are implemented to support both fine-grained (e.g., MESI [15]) and coarse-grained (e.g., page-grained [4], [11] and LazyPIM [17]). PIMSim can also be configured in different ISAs including but not limited to x86, ARM, and SPARC. Several PIM micro-benchmarks are integrated into PIMSim, which give users a shortcut to simulate PIM.

3.2 PIM-Enabled Instructions, Offloading, and Flexible PIM Code Annotations

PIMSim implements basic PIM instructions to coordinate PIM system controlling, such as PIM load/store, register read/write, calculation and cache flushing operations. Flexible methods are also provided to define users' own PIM instructions. In the full-system mode, PIMSim uses pseudo instructions to transmit PIM operands without hurting the ISAs, which accommodates users with various types of ISAs. Note that we implement the added pseudo instructions and let them occupy the pipeline (such as decoding and virtual address transfer) like normal instructions, which avoids hurting the pipeline stage of the host-side processors.

Typically, computing partitioning in PIM is of two aspects: 1) partitioning between the host and memory and 2) partitioning among PIM Logics. These two partitioning aspects are both supported in PIMSim by simplified PIM code annotations, which are listed in Fig. 5 (in blue color). The compiler in PIMSim automatically transfers such annotations into PIM-enabled instructions and dispatches them to destiny based on the PIM Logic's index.

PIMSim provides two offloading methods: instruction bypass and in-memory binary, to satisfy different requirements. Instruction bypass is to give instructions to PIM Logic through off-chip connections, which is suitable for simple PIM kernels and specified in-memory accelerators. In-memory binary writes the executable binaries in the memory and informs PIM Logic with only start address provided, which minimizes the consumption of off-chip bandwidth when using complex PIM kernels.

3.3 Unified Framework for PIM Coherence

Coherence in PIM is another important factor that affects system performance. Experiments show that different coherence approaches make the performance vary from -30 to $+80$ percent [19]. Such a considerable performance difference motivates us to implement different coherence protocols in PIMSim. Currently, invalidation-based fine-grain coherence protocols, such as MEI and MESI [15], enable data coherence in cache-based architectures. However, existing PIM simulation environments provide little support for such fields. PIMSim supports three PIM coherences of different granularities to provide comparisons with newly designed PIM coherence methods.

Coherence designs in PIM generally lead to inconvenience in using solutions raised by other researchers. For example, if one wants to implement a coherence scheme like LazyPIM [17], one cannot simply use preprocessor directives like *#include* to call it directly. Instead, one has to do plenty of modifications to the processors, PIM Logic, coherence bus, etc. To avoid such dirty work, PIMSim implements a unified framework to provide a non-conflict including methods to call other PIM schemes. Users need only set the coherence name in the configure file, which saves much work in simulating PIM coherence.

4 Validation and Benchmark Evaluation

First, we evaluate the simulation time of PIMSim. We choose the applications used in [19], Average Teenage Follower (ATF), Breadth-First Search (BFS), Pagerank (PR), and Shortest Path (SP), as its applications are representative in PIM. Each application is implemented by C++. PIMSim runs on a host system with an Intel (R) Xeon(R) E7-8830 processor running at 2.13 GHz, with 32 GB of DRAM. We use OProfile [21] to select PIM kernels. The workload is a real-world graph called *soc-Livejoural1* [19], with 4.8M nodes and 69M edges. The simulation time is normalized to the real execution time on the host. As shown in Fig. 6a, the fast simulation mode performs 4.2×10^4 X slower than real execution, while the instrumentation-driven mode is 1.4×10^5 X slower and the full-system mode is 3.1×10^5 X slower.

We also evaluate PIMSim with the same parameters as PEI [19] to test the accuracy of PIMSim. Fig. 6b shows a comparison of the IPC result between [19] and PIMSim. *PIM-Only* represents the architecture that offloads all instructions to the memory. *Locality-Aware* represents the same architecture as PEI. Compared with PEI, the three modes of PIMSim, fast, instrumentation-driven, and full-system simulation, produce 18.75, 14.63 and 7.88 percent differences in the performance result, respectively. Combined with the simulation time results shown in Fig. 6b, the three simulation modes of PIMSim fully consider the tradeoff between the speed and the accuracy.

5 Conclusion

In this paper, we present PIMSim, a detailed, flexible and efficient PIM system simulator to overcome the difficulties in PIM modeling and simulation. PIMSim provides a configurable PIM Logic module which can be flexibly defined by users. It also provides host-side and memory-side co-simulation, with the compatibility to existing memory simulators. Experiments show that the results are relatively accurate compared with state-of-the-art PIM designs. The corresponding authors are Yinhe Han and Xiaowei Li.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under grants 61522406, 61834006, and 61521092, Beijing Municipal Science & Technology Commission (Z171100000117019, Z181100008918006), Strategic Priority Research Program of the Chinese Academy of Sciences (XDPB12), and an Innovative Project of Institute of Computing Technology, CAS, under Grant 5120186140.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.

[2] Y. Kang, et al., "FlexRAM: Toward an advanced intelligent memory system," *Proc. IEEE 30th Int. Conf. Comput. Des.*, 2012, pp. 5–14.
[3] D. Patterson, et al., "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar./Apr. 1997.
[4] S. Xu, et al., "PIMCH: Cooperative memory prefetching in processing-in-memory architecture," in *Proc. 23rd Asia South Pacific Des. Autom. Conf.*, 2018, pp. 209–214.
[5] S. Li, et al., "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proc. 53nd ACM/EDAC/IEEE Des. Autom. Conf.*, 2016, pp. 1–6.
[6] J. D. Leidel and Y. Chen, "Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 621–630.
[7] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems," *IEEE Comput. Archit. Lett.*, vol. 14, no. 2, pp. 140–143, Jul.-Dec. 1, 2015.
[8] N. Binkert, et al., "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
[9] C.-K. Luk, et al., "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*. vol. 40, no. 6, pp. 190–200, 2005.
[10] E. Azarkhish, et al., "Design and evaluation of a processing-in-memory architecture for the smart memory cube," in *Proc. Int. Conf. Archit. Comput. Syst.*, 2016, pp. 19–31.
[11] K. Hsieh, et al., "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation," in *Proc. IEEE 34th Int. Conf. Comput. Des.*, 2016, pp. 25–32.
[12] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Proc. Hot Chips 23 Symp.*, 2011, pp. 1–24.
[13] Y. Wang, et al., "ProPRAM: Exploiting the transparent logic resources in non-volatile memory for near data computing," in *Proc. 52nd Annu. Des. Autom. Conf.*, 2015, pp. 1–6.
[14] K. K. Chang, et al., "Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 568–580.
[15] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the IEEE futurebus," *ACM SIGARCH Comput. Archit. News*, vol. 14. no. 2, pp. 414–423, 1986.
[16] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. Int. Conf. Parallel Archit. Compilation*, 2015, pp. 113–124.
[17] A. Boroumand, et al., "LazyPIM: An efficient cache coherence mechanism for processing-in-memory," *IEEE Comput. Archit. Lett.*, vol. 16, no. 1, pp. 46–50, Jan.-Jun. 2017.
[18] T. M. Aamodt, et al., "GPGPU-Sim 3. x manual," 2012, [2013-08-08]. [Online]. Available: http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual.
[19] J. Ahn, et al., "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 336–348.
[20] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan.-Jun. 2011.
[21] J. Levon and P. Elie, "Oprofile—a system profiler for linux, 2004." [Online]. Available: http://oprofile. sourceforge. net (2007).
[22] R. Nair, et al., "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM J. Res. Develop.*, vol. 59, no. 2/3, pp. 17–21, 2015.