# Sim$^2$PIM: A Fast Method for Simulating Host Independent & PIM Agnostic Designs

Paulo C. Santos*, Bruno E. Forlin*, Luigi Carro
Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
Email: {pcssjunior, beforlin, carro}@inf.ufrgs.br

***Abstract*—Processing-in-Memory (PIM), with the help of modern memory integration technologies, has emerged as a practical approach to mitigate the memory wall and improve performance and energy efficiency in contemporary applications. However, there is a need for tools capable of quickly simulating different PIMs designs and their suitable integration with different hosts. This work presents *Sim$^2$PIM*, a Simple Simulator for PIM devices that seamlessly integrates any PIM architecture with the host processor and memory hierarchy. Sim$^2$PIM's simulation environment allows the user to describe a PIM architecture in different user-defined abstraction levels. The application code runs natively on the Host, with minimal overhead from the simulator integration, allowing Sim$^2$PIM to collect precise metrics from the Hardware Performance Counters (HPCs). Our simulator is available to download at https://pim.computer/.**

***Keywords*—*Processing-in-Memory, Simulator, Performance, Cycle Accurate***

## I. Introduction

Processing-in-Memory (PIM), Near-Data Accelerator (NDA), and Computing-In-Memory (CIM) have emerged to mitigate memory wall, general-purpose processors limitations, and improve modern systems' performance and energy efficiency. PIM and its variations have been proposed in different flavors, from full processing cores implementations to small Functional Units (FUs), and whole new technologies such as Memristor crossbars. However, the field lacks a generic set of tools and techniques for these novel systems, as often researchers have to spend a significant amount of time and effort in building the needed simulation environment [1].

As these PIM designs rapidly develop, some of the most prominent ones adopt minimal hardware (simpler FUs [2], [3]) or take advantage of inherent analog processing capability, such as *RERAM* and *MEMRISTOR* [4], [5]. However, unlike typical processors, these designs cannot rely on traditional solutions for code offloading (e.g., OpenMP, MPI), cache coherence (e.g., MOESI protocol), and virtual memory management (e.g., duplicating Translation Look-aside Buffer (TLB)), since PIMs are commonly displaced from the Host's processor reach, thus requiring innovative approaches. For example, designs like [6], [7], [3], [5], [8] assume Instruction Set Architecture (ISA) extensions at the host processor's side, which are capable of handling the PIM instructions, while some designs choose to rely on additional hardware responsible for identifying and offloading code [9].

Currently, there exists no single framework that can seamlessly implement different approaches for code offloading, cache coherence, or virtual memory support mechanisms. Nonetheless, these mechanisms are essential for the PIM adoption, and although they are explored in the literature [10], [9], [11], [5], [6], most PIM designers do not spend much time developing them due to the lack of tools, choosing to focus on the computing aspect [1]. The broad array of PIM simulators presented in the literature miss performance and real metrics, restricting the developer to exploit different integration between Host and PIM devices, while also hindering productivity. Furthermore, most of these simulators are trace-based [12], [13], [14], [15], [16], [17], requiring that the developer spends more time designing traces for each application. Therefore, a *toolset* that can deliver high simulation performance (since a PIM simulation is a system simulation), while still being exact on the behavior of the overall system (since using a PIM requires a detailed analysis of memory hierarchy and OS contribution) is necessary.

Sim$^2$PIM is based on compile-time instrumentation to provide high accuracy, low non-recurring costs, fast prototyping, and simulation capabilities. Sim$^2$PIM allows:

- **Host Independence** - PIM designs are expected to be coupled with different hosts (e.g., Intel, AMD, ARM). This work allows PIM adoption in any processor by running native code on the native host processor.
- **PIM Agnostic** - There exists a myriad of PIM designs in the literature. They either modify Dynamic Random Access Memory (DRAM) modules, take advantage of 3D-stacked memories, or adopt new memory cell technologies. The provided simulator technique allows experimentation with PIM technology in any memory hierarchy level, only requiring that the developer models the PIM architecture and its interoperability.
- **Fast Prototyping** - This work allows the developer to deal only with PIM hardware and its design, without the need to get involved with the Host's details and HOST-PIM hardware integration. Moreover, with many possibilities in the literature, collecting information from designs under test at different detail levels is essential. Thus, the simulator provides a flexible abstraction level, allowing the developer to decide the design's level of detail during development.
- **Fast Execution** - Since the simulator runs native host code on the Host, and simulates only the PIM side, its performance is directly dependent on the level of detail and complexity of the PIM design.
- **Host Metrics** - Sim$^2$PIM allows access to real metrics

*Both authors contributed equally to this research.
We acknowledge the support of FAPERGS, CNPq, CAPES-Finance Code 001

provided by the Host's Hardware Performance Counter (HPC). Hence, it is possible to evaluate the impact of the PIM design on the entire system.

The rest of this paper is divided as follows: in Section II we present a brief overview of PIM technologies. Section III shows other simulation environments for PIM, their features, and what they lack. We present the simulator architecture in Section IV. We evaluate Sim$^2$PIM in comparison to other simulators in Section V. Finally, we conclude in Section VI.

## II. Background

Processing-in-Memory (PIM), Near-Data Accelerator (NDA), and Computing-In-Memory (CIM) are all terms related to the concept of reducing data movements, and computing closer to, or inside, the memory device. Several technologies appear as familiar candidates for PIM devices, such as logic layers for 3D-Stacked memories [18], [19], modified DRAM rows capable of computing [20], and ReRAM or Memristor devices [8].

Figure 1 depicts the most common types of in-memory computing approaches. Each technology has radically different characteristics, like throughput, energy consumption, reliability, and programmability, directly impacting architecture. As a result, there have been several proposals for different PIM architectures, centered around different technologies:

- **Near Data Processing Core** designs attempt to insert full processing cores in the memory chip, facilitating integration by relying on proven methods for multi-core systems.
- **Near Data Functional Units** PIMs rely on Simple FUs are prominent solutions since they consider the inherent embedded nature of PIM that demands restrictions like area and power dissipation.
- **Specialized Processing Units** aim at accelerate applications by accessing the main memory directly, usually adopting Application Specific Integrated Circuit (ASIC) designs.
- **Analog Cells** to compute data *in situ*, using Memristor, ReRAM or DRAM modifications, requiring low-overhead in terms of hardware for processing.

There has been a significant effort in exploring new technologies and architectures to provide PIM. However, most available solutions put considerably less effort into developing host interconnection solutions, with offloading mechanisms, virtual memory mapping, and cache coherence mechanisms. Nevertheless, the lack of studies and solutions to these questions hinders adopting these architectures at a larger scale [1].

## III. Related Works

Simulation-Based modeling allows one to achieve more accurate performance numbers. While architects often resort to modeling the entire micro-architecture precisely, this approach can be relatively slow compared to analytic techniques [1]. The trade-off between simulation speed, accuracy, and development time can be the key to facilitate broader research and to reduce time to market.

System-level simulators like gem5 [21] and SiNUCA [12], while able to accurately simulate PIM devices, often introduce

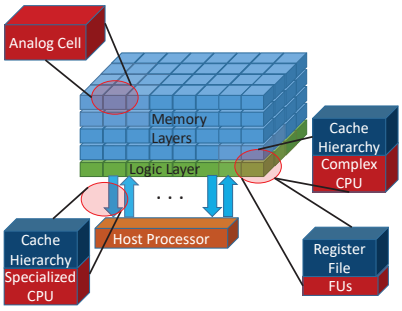


Fig. 1: Common Types of Processing-in-Memory Devices

significant runtime overheads and development costs. gem5 is able to simulate system-level architecture as well as the processor microarchitecture, however each component must be developed and integrated into the system, requiring a high development cost and very slow simulation time.

Developed in SystemC, CLAPPS [14] aims to fill the gap between the ease of implementation and accurate modeling of the Hybrid Memory Cube (HMC) specification. It provides support for customized PIM instructions, dependent on the user's logic layer design. HMC-SIM 2.0 [17] is built as a stand-alone C library with an accurate HMC 2.0 specification. It supports expansions to the logic layer capabilities using reserved opcodes from the HMC ISA. Although they provide simulation, they do not provide any integration between the host and the simulated HMC device, adopting a memory trace file as input.

PIMSim [13] proposes an all-in-one simulator for PIM architectures, using three accuracy and simulation speed modes. The authors claim that existing PIM simulation environments are composed of several different models, from memory to compilers and that the lack of a consolidated architecture for PIM abstraction makes a completely automatic simulation process almost impossible. The three simulation modes are: a full system gem5-based simulation, a PinTool [22] enabled mode for instruction instrumentation, and a fast-simulation mode using memory traces. This trace-driven simulator has a deep focus on providing multiple options and parameters for a PIM designer. However, we leverage that the trade-off between the author's simulation speed and accuracy is far more significant than presented. In the case of gem5's simulation speed overhead, for a complete simulation of the whole environment one can reach 1,000 times depending on the operation mode, while the PinTool instrumentation overhead in simulation speed can reach up to 100 times for a given application. Furthermore, by providing a fully integrated environment, the authors have locked newer memory and PIM architectures from being incorporated, without significant rework of the simulator itself.

According to Xia et al., standard system simulators cannot support memristor devices and memristor-based computing structures [16]. Therefore, they propose MNSIM, which simulates a memristor-based neuromorphic accelerator's performance, providing the user with metrics to estimate the average and worst-case speed. This simulator uses configuration files to generate circuit modules based on a predefined hierarchical structure. Similarly CIM-SIM [15], a SystemC functional

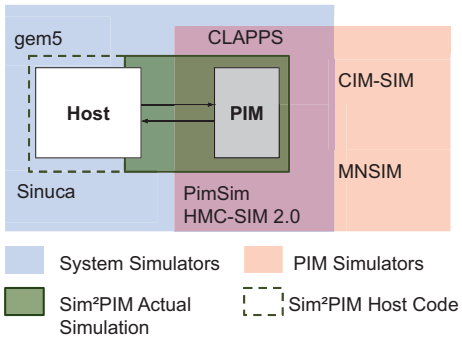


Fig. 2: Simulators scope when considering system integration. gem5 [21], Sinuca [12], CLAPPS [14], PimSim [13], HMC-SIM 2.0 [17], CIM-SIM [15], and MNSIM [16].

simulator, focuses on providing an architectural backbone with a concise ISA and peripheral digital modules. In this manner, designers can test their PIMs with different technologies, compilers, and topologies, narrowing the development gap between other, more established technologies and memristors. However, neither simulator provides integration with a host CPU, relying on the integration with other simulators like gem5, to provide metrics for hardware/software co-design.

The current simulation ecosystem is shown in Figure 2, where system architecture simulators have a near-ideal accuracy at the cost of significant simulation and development times [21], [12]. Current PIM architecture simulators are either architecture-specific [17] or rely on system simulators [14], or other tools with a heavy overhead [13]. Finally, simulators for newer technologies are incomplete, as they do not try to simulate a fully connected system [16], and again rely on tools presenting a heavy overhead [15]. As this review clarifies, PIM's current simulation ecosystem lacks a low-overhead solution capable of providing system integration with coherence and code offloading mechanisms, together with virtual memory capabilities, which does not rely on full-fledged system simulators.

## IV. Simulator - Architecture

This work's main objective is to provide a simulation environment that allows the experimentation of different designs of PIM, regardless of memory technology or their insertion into memory hierarchy (from the cache memory to main memory). Moreover, the proposed simulator aims to support access to the native host's Hardware Performance Counter (HPC), which gives reliable metrics on the host side, only requiring proper simulation on the PIM side. The general simulation flow comprises three phases: instrumentation of application code, the compilation of simulation environment, and propper simulation execution. The input to this flow is a PIM application code, which the simulator uses to produce detailed metrics for the application, directly executing in the host machine and the designed PIM simulator.

### A. Instrumenting the application

The first stage comprises a post-compilation assembly code parser, responsible for instrumenting the user application at

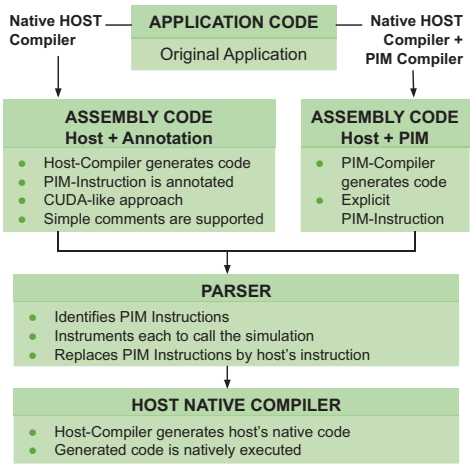


Fig. 3: Overview of the Instrumentation

instruction level. It searches for user-defined PIM instructions, annotated code, or other means of representation for accelerable kernels. The parser replaces each of these instructions with a set of native host's instructions, including a *store* instruction containing the PIM instruction as data, and a function call to the simulator. Figure 3 depicts the flow of this module. Since the parser instruments the user application code after its compilation, all optimizations are left untouched, although the insertion of a simulation function call may require adjustments to save registers from the *caller* side, following the *calling conventions* of each architecture (e.g.,[23]).

To illustrate the instrumentation process, Listing 1 highlights a PIM instruction (*PIM_LOAD*) generated by the XXX compiler[omitted for blind review], and the Listing 2 shows the result after the parser instruments the code. It is possible to observe in line 3 of the Listing 2, the *PIM_INST_STRING* represents the aforementioned PIM instruction (*PIM_LOAD* and its arguments) that is placed as an ordinary string in the user's application *.data* section (not shown in Listing). Hence, the instruction is emitted to the simulator via global variable memory address provided by the simulation environment (*GLOBAL_VAR_PIM_INST*). This variable is part of the shared memory space provided by the simulator to allow data exchange between application and simulation. In case of PIM instructions that access memory (e.g., *PIM_LOAD*), the simulator provides resources that allows it to access the user application memory space. For this, the variable *GLOBAL_VAR_PIM_INST_ADDR* can receive the address of the PIM instruction, previously computed, as shown in lines 5 and 6 of the Listing 2.

Listing 1:
Original x86+PIM Code Snippet - Annotated or Compiled

```
1 movq %rax , %r14
2 . . . . .
3 PIM_LOAD 32512(%rbx,%rax), %PIM_REG_0 ;PIM
      instruction
4 . . . . .
5 addq $2048 , %rax
```

Listing 2: Parsed x86+PIM Code Snippet

```
movq %rax , %r14
. . . . .
movq (PIM_INST_STRING), %rcx        ;read PIM instruction as a string
movq %rcx , (GLOBAL_VAR_PIM_INST) ; PIM Instruction is emitted to the simulator
leaq 32512(%rbx,%rax), %rcx        ;PIM memory access calculation in case of LOAD/STORE
movq %rcx , (GLOBAL_VAR_PIM_INST_ADDR) ; PIM memory access address is emitted to the simulator
. . . . .
addq $2048 , %rax
```

This approach provides unlimited resources for the developer to explore different PIM designs and techniques to couple host and PIM, such as code-offloading [9], cache coherence [11], [5], [6], and virtual memory management. For instance, if the designer targets a PIM device placed within the host's cache memory, similar to [24], it is important that the simulator is able to reproduce the timing and behavior of this design, which means providing reliable metrics and behavior accordingly. In this case, the simulator must keep alive, in the cache memory, the input and output ports/variables that represent the communication between host and PIM. This way the metrics will be representative accordingly. On the other way, if the PIM being designed is placed within the main memory module, *flush* operations over PIM variables can ensure the communication between PIM and host will be done to/from main memory, hence reproducing the intended behavior. In these examples, the code-offloading would consider the real timings expected for each PIM design. In similar way, cache coherence and virtual memory management can be done by inserting the proper instructions via parser and detailing the simulator.

### B. Simulator Flow and Integration

The instrumented application code now needs to be integrated with the simulator. We have designed the simulator program to accommodate the application function call between HPC collection calls. Therefore, for an unmodified host application code, the metrics collected would correspond to the application's performance costs being executed in the host (e.g., core cycles, retired instructions). Figure 4 presents the overview of the proposed simulator and highlights its main modules. The role of each module is as follows:

**SETUP COUNTERS:** To allow access to the native host's HPC, the simulator identifies the host processor in use and selects the proper Model-Specific Register (MSR) addresses to configure the hardware counters. Sim$^2$PIM identifies the processor by checking the *CPUID* register's content, which provides information about the current host processor. It is also possible to manually configure the counters.

**WARMUP:** Considering the relevance of precise metrics, and the fact that different hosts and compilers may result in different measurements, this module executes several HPC calls, back-to-back, as to collect the call overhead. These overheads include the serialization instructions inserted on each HPC call and the instruction overhead required to read the HPC.

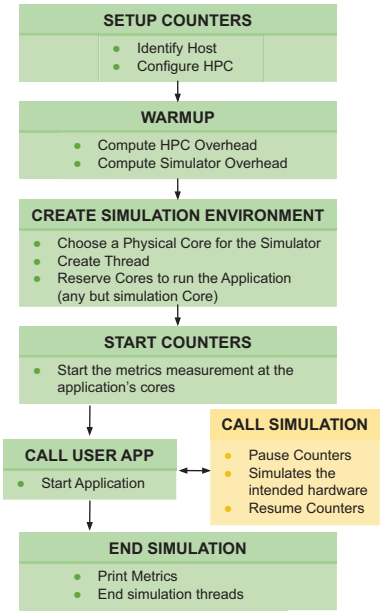


Fig. 4: Overview of the Simulator

**CREATE ENVIRONMENT:** Sim$^2$PIM isolates the simulation from the user application to provide more accuracy for the hardware counters by reserving a physical core for the simulator and the metrics management. The rest of the host's cores remain free for the user's application. This approach reduces the interference in the data and instruction caches, hence allowing more precise measurements.

**START COUNTERS:** After setting up the simulation environment, Sim$^2$PIM reads the counters before calling the application. The HPC are overflow counters; this means one must acquire the difference between two consecutive measures instead of an absolute value.

**CALL USER APPLICATION:** The simulator calls the user application as an ordinary function. This way, it can be implemented regardless of the simulation environment, allowing for several liberties on the application's development, such as choice of libraries, independent compiler optimizations, and single or multi-threading experimentation. As presented in Section IV-A, the parser instruments the user application in order to support the simulation calls.

**CALL SIMULATION:** The simulator is called as a simple function, directly from the *USER APPLICATION*. This function call follows the pattern shown in Figure 5, in which Sim$^2$PIM collects and accumulates the HPC discounting the overheads found in the *WARMUP* phase. Then, Sim$^2$PIM launches the simulation environment as a new thread, where the actual hardware simulation happens. Sim$^2$PIM adopts *global address variables* to share resources between the hardware-under-simulation and the application-under-test, granting unrestricted low-overhead global access by the user's application. This method guarantees that the application and the simulated PIM device achieve seamless data coherence and consistence, providing low overhead and fast communication between both threads with no need for third part libraries or

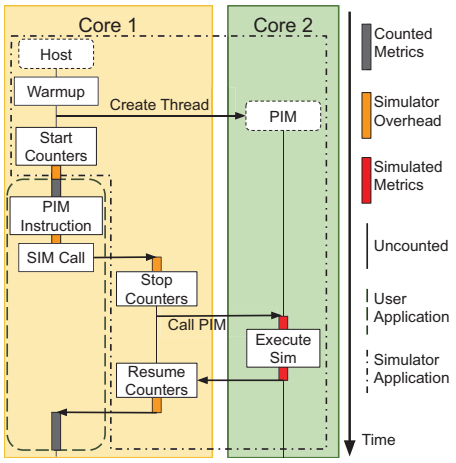


Fig. 5: Interfaces and overheads between user application and PIM simulation

complex communication protocols. Also, since the simulator and application execute under the same process, both operate in the same virtual address space. After the simulation thread ends, Sim$^2$PIM collects the HPC once more, resuming the counter count.

**END SIMULATION:** The simulation concludes at the end of the application. The accumulated metrics are correctly summarized, taking into account the overheads mentioned above.

It is essential to observe that the simulation does not require particular compilers or tools, such as PIM compilers or third party libraries. Since the parser replaces PIM instructions by the native host's function call, we use an ordinary compiler that supports the host. However, it is also important to mention that Sim$^2$PIM does have drawbacks. In multi-core applications, it can use at most $(MAX\_CORES - 1)$ cores, since the simulator reserves one core exclusively to improve measurements. Also, since Sim$^2$PIM does not simulates host, and therefore it can only consider the host in which it is currently executing, it is limited by the availability of HPCs on the processor (which are fairly common nowadays in Intel, AMD and ARM processors).

### C. Simulation Overheads

As aforementioned, to improve the accuracy of the measurements the proposed tool considers the inherent overheads of HPC routines, as well as the simulator caller. These overheads are directly dependent on the host processor, and on the compiler that is responsible for generating code that can contain instructions in different numbers and types. To illustrate this impact, Table I summarizes these overheads for two hosts, considering the flow presented in Figure 5.

## V. Evaluation

To compare the available simulators, Table II shows the most interesting features present in several available simulators. Sim$^2$PIM targets each highlighted feature. The proposed simulator allows its execution in any host, and therefore integrates this host with the under-test PIM. Since the simulator

TABLE I: Overheads for two different HPCs, unhalted cycles and retired instructions. Measured with 10,000 repetitions in the warmup phase of two different processors.

| Overheads | | Intel Core i5-7600@GCC7.5 | AMD R5-1600@GCC9 |
|---|---|---|---|
| # Cycles | Simulation Call | 174 | 184 |
| | Stop and Resume Counters | 168 | 180 |
| # Instructions | Simulation Call | 11 | 10 |
| | Stop and Resume Counters | 9 | 8 |

can be written as an simple program, it allows the implementation of any PIM designs, and its detailing will dictate how fast it is to prototype different designs as well its experimentation performance. This characteristic makes it flexible by accepting different languages and models (e.g., C++, Python, SystemC, etc), or even full-fledged simulators [15]. Sim$^2$PIM considers real metrics provided by the host, which increases the overall measurements, perfectly coupling host and PIM metrics, requiring only the PIM side to be specified. Also, to provide even more precision, the simulator isolates simulation through a separated thread. Moreover, since the application is natively executed in the host, native system calls and Operating System (OS) libraries are available. Additionally, we provide seamless interoperability with standard system features, such as virtual address translation, data coherence and code offloading.

### A. Experiment

To evaluate our design, we choose as case study a PIM that implements simple FUs in memory [3]. We choose a Nehalem-based processor (Intel i7-860) as host, since it is present in the gem5 simulator, increasing the comparison fairness. We have simulated several applications in our environment, but here we show only the Convolutional Neural Network (CNN) application [25], since we want to compare our solution to others available.

Figure 6 shows a comparison with the actual hardware between a PinTool-based simulator, gem5, and our approach. Although gem5 can simulate the complete system, it can not achieve the same accuracy results for the Host as the HPC. One can question that it is possible to implement 100% accuracy in gem5, however, this will further increase the simulation time. Sim$^2$PIM is faster and more accurate since it can extract the metrics directly from the processor's HPCs, with no dependency on approximate models. A PinTool-based solution can monitor the application; however, the Just In Time (JIT) compiler nature of the tool inserts several delays in the execution, making Sim$^2$PIM faster and more accurate, as it does not depend on trace analysis.

Sim$^2$PIM can not reach 100% accuracy, since we can not fully eliminate the overheads. During the parser phase, some of the inserted instructions will have variable cycle measurements due to the interference of register saving instructions in the application's computational data. It becomes clear that the trade-off of simulating the Host and application in these architectures is not worth the accuracy gains. This trade-off becomes even more disadvantageous if we consider the massive development overhead of modifying these tools, compared to only selecting the correct HPC and applying light modifications to the application code.

TABLE II: Comparing features of PIM Simulators and Sim$^2$PIM.

| | PIMSim[13] | SiNUCA [12] | CLAPPS [14] | HMC-SIM 2.0 [17] | MNSIM [16] | CIM-SIM [15] | gem5 [21] | Sim$^2$PIM |
|---|---|---|---|---|---|---|---|---|
| **Host Independent** | N | N | Y | Y | Y | Y | implementable | Y |
| **PIM Agnostic** | Y | Y | N (HMC) | N (HMC) | N (Memrsitor) | N (Memrsitor) | Y | Y |
| **Fast Prototyping** | N | N | Y | N | N | N | N | Y |
| **Fast Execution** | N | N | Y | Y | Y | Y | N | Y |
| **Host Metrics** | profiling | N | N | N | N (behavior-level) | N (functional simulation) | implementable | Y |
| **Flexible Abstraction Level** | 3 modes | N | N | N | N | N | Y | Y |
| **Native System Calls** | Y | Y | Y | N | - | - | (gem5-libs) | Y |
| **Thread-isolated Simulation** | N | N | N | N | - | N | N | Y |
| **System-level support for PIM** | N | N | N | N | - | N | Y | Y |

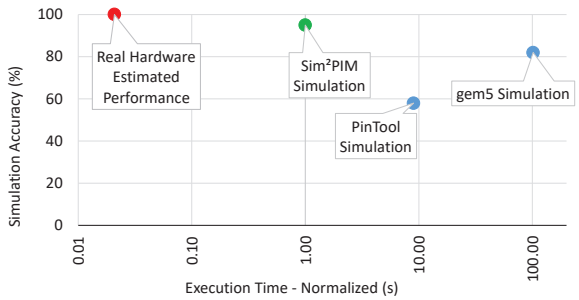


Fig. 6: Execution Time for Convolutions of Yolo3 CNN application [25] on tested PIM

## VI. Conclusions and Future Work

In this work, we presented Sim$^2$PIM, a fast simulator for PIM architectures, which, by design, is independent of the Host and the simulated PIM. The simulator presents a fast simulation speed, dependent only on the PIM description's complexity level, avoiding to simulate hardware that is already available at the Host. Moreover, the presented simulator adopts the native Hardware Performance Counters information to extract several metrics. We believe that providing a fast and precise prototyping environment will accelerate the development of new PIM architectures and concepts. As future work, we plan to expand our design to thoroughly simulate multithreading-capable PIM devices. We also aim to narrow down further the overheads experienced by the simulator interconnection.

## References


[1] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans *et al.*, "Near-memory computing: Past, present, and future," *Microprocessors and Microsystems*, 2019.

[2] Hybrid Memory Cube Consortium, "Hybrid memory cube specification rev. 2.0," 2013, http://www.hybridmemorycube.org/.

[3] P. C. Santos, G. F. Oliveira, D. G. Tomé, M. A. Alves, E. C. Almeida, and L. Carro, "Operand size reconfiguration for big data processing in memory," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017.

[4] H. A. D. Nguyen, J. Yu, M. A. Lebdeh, M. Taouil, S. Hamdioui, and F. Catthoor, "A classification of memory-centric computing," *J. Emerg. Technol. Comput. Syst.*, 2020.

[5] A. Drebes, L. Chelini, O. Zinenko, A. Cohen, H. Corporaal, T. Grosser *et al.*, "Tc-cim: Empowering tensor comprehensions for computing-in-memory," in *IMPACT 2020 workshop*, 2020.

[6] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2018.

[7] V. T. Lee, A. Mazumdar, C. C. del Mundo, A. Alaghi, L. Ceze, and M. Oskin, "Application codesign of near-data processing for similarity search," in *2018 IEEE Int. Parallel and Distributed Processing Symp (IPDPS)*, 2018.

[8] L. Xie, H. Cai, and J. Yang, "Real: Logic and arithmetic operations embedded in rram for general-purpose computing," in *2019 IEEE/ACM Int. Symp on Nanoscale Architectures (NANOARCH)*, 2019.

[9] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *Int. Symp. on High Performance Computer Architecture (HPCA)*. IEEE, 2017.

[10] P. C. Santos, J. P. Lima, R. F. Moura, M. A. Alves, L. Carro, and A. C. S. Beck, "Solving datapath issues on near-data accelerators," in *IFIP WG10.2 Working Conference: Int Embedded Systems Symp (IESS)*.

[11] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE Computer Architecture Letters*, 2016.

[12] M. A. Z. Alves, C. Villavieja, M. Diener, F. B. Moreira, and P. O. A. Navaux, "Sinuca: A validated micro-architecture simulator," in *2015, 17th Int. Conf. on High Performance Computing and Communications*, 2015.

[13] S. Xu, X. Chen, Y. Wang, Y. Han, X. Qian, and X. Li, "Pimsim: A flexible and detailed processing-in-memory simulator," *IEEE Computer Architecture Letters*, 2018.

[14] G. F. Oliveira, P. C. Santos, M. A. Z. Alves, and L. Carro, "A generic processing in memory cycle accurate simulator under hybrid memory cube architecture," in *Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2017.

[15] A. BanaGozar, K. Vadivel, S. Stuijk, H. Corporaal, S. Wong, M. A. Lebdeh *et al.*, "Cim-sim: Computation in memory simulator," in *22nd Int. Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '19, 2019.

[16] L. Xia, B. Li, T. Tang, P. Gu, P. Chen, S. Yu *et al.*, "Mnsim: Simulation platform for memristor-based neuromorphic computing system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[17] J. D. Leidel and Y. Chen, "Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations," in *2016 IEEE Int Parallel and Distributed Processing Symp Workshops (IPDPSW)*, 2016.

[18] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *2018 51st Annual IEEE/ACM Int. Symp on Microarchitecture (MICRO)*, 2018.

[19] P. C. Santos, G. F. Oliveira, J. P. Lima, M. A. Alves, L. Carro, and A. C. Beck, "Processing in 3d memories to speed up operations on complex data structures," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018.

[20] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," in *Proceedings of the 52nd Annual IEEE/ACM Int Symp on Microarchitecture*, ser. MICRO '52, 2019.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation." Association for Computing Machinery, 2005.

[23] Intel. (2013) System v application binary interface. [Online]. Available: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

[24] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *Int. Symp. on Computer Architecture (ISCA)*. IEEE, 2015.

[25] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.