Sim^{2}PIM: a Complete Simulation Framework for Processing-in-Memory

Bruno E. Forlin[a,*], Paulo C. Santos[a,*], Augusto E. Becker[a], Marco A. Z. Alves[b] and Luigi Carro[a]

[a] Federal University of Rio Grande do Sul, Porto Alegre, Brazil
[b] Federal University of Paraná, Curitiba, Brazil

ARTICLE INFO

Keywords:
Processing-in-Memory
Simulator
Performance
Cycle Accurate
Multi-Threaded

Abstract

With the help of modern memory integration technologies, Processing-in-Memory (PIM) has emerged as a practical approach to mitigate the memory wall while improving performance and energy efficiency in contemporary applications. Since these designs encompass accelerating and increasing the efficiency of critical specific and general-purposed applications, it is expected that these accelerators will be coupled to existing systems and consequently with systems capable of multi-thread computing. However, there is a lack of tools capable of quickly simulating different PIMs designs and their suitable integration with other hosts. This gap is even worse when considering simulations of multi-core systems. 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. The framework simulation achieves execution speeds and accuracy on par with the *perf* tool on host code, less than 10% run-time overhead, and around 2% difference in metrics. Additionally, by exploring the thread parallelism in the application and utilizing the host hardware, Sim^{2}PIM can achieve more than 8× simulation speedup compared to a sequential simulation and orders of magnitude compared to other simulators. Sim^{2}PIM is available to download at https://pim.computer/.

1. Introduction

Processing-in-Memory (PIM) and its variations have emerged as a prominent solution to the memory-wall problem, sharing a common approach: reducing data movement from main memory to processing units, aiming to increase performance and energy efficiency of computational systems. In the last decade, technological advancements have allowed the creation of a myriad of PIM designs in different flavors [1–7].

More recently, commercial PIM designs based on commodity Dynamic Random Access Memory (DRAM) devices have taken the spotlight [8, 9]. These designs are intended to be integrated with unmodified general-purpose processors to facilitate their adoption, expanding support for more applications, and reducing programming efforts. To seamlessly couple PIM and general-purpose systems, PIM must rely on the host resources (software or hardware) to provide necessary mechanisms, such as cache coherence, virtual memory support, data consistence, and communication. Thus, PIM performance is tied to its reliance on the host processor, which is increased when multiple units operate simultaneously in a Single Instruction Multiple Data (SIMD) or multi-threading fashion. Therefore, testing the interactions of PIM hardware, host integration and real applications is crucial.

This work was partially supported by FAPERGS, CAPES, CNPq and Serrapilheira Institute (grant number Serra-1709- 16621).

*Bruno E. Forlin and Paulo C. Santos are co-primary authors.

beforlin@inf.ufrgs.br (B.E. Forlin); pcssjunior@inf.ufrgs.br (P.C. Santos); aebecker@inf.ufrgs.br (A.E. Becker); mazalves@inf.ufpr.br (M.A.Z. Alves); carro@inf.ufrgs.br (L. Carro)

ORCID(s): 0000-0003-4822-1841 (B.E. Forlin); 0000-0001-8555-2637 (P.C. Santos); 0000-0002-2691-1481 (A.E. Becker); 0000-0003-2440-2664 (M.A.Z. Alves); 0000-0002-7402-4780 (L. Carro)

Although there have been considerable advances in the available tools to experiment on these designs, the field still lacks solutions capable of thoroughly simulating the interactions of PIM accelerators and different modern multi-core host processors. Even more critical is the lack of tools capable of doing so in a fast and simple manner, such that designers can quickly prototype hardware and algorithms.

In this work, Sim^{2}PIM [10] evolved from a single-thread PIM simulator, to a full-fledged multi-thread capable PIM simulation and instrumentation framework. The Sim^{2}PIM framework presents a high accuracy, low overhead, low execution time, and quick to implement simulation. These qualities place the Sim^{2}PIM framework in stark contrast to other current simulation methodologies like full-system [11, 12] and trace-based simulators [12–15].

With Sim^{2}PIM, the application is integrated and controlled by the framework, in an inversion of control, to become a single executable that can run natively on the host terminal. This allows Sim^{2}PIM to use host hardware for executing host instructions and the PIM-simulator for PIM instructions. Moreover, the framework is designed with multi-threaded code execution in mind, allowing for parallel applications to actually access multi-core hardware and software resources. This support is built around the *pthread* library, allowing for the library's complete functionality, including synchronization capabilities. Therefore, Sim^{2}PIM does not have to replicate or emulate this functionality.

Other current simulators presented in the literature must simulate performance metrics, while Sim^{2}PIM delivers the host's Hardware Performance Counter (HPC) as the most accurate baseline possible for real hardware. The simulator makes smart use of the host HPC to integrate code instrumentation directly with application code. This allows

Sim^{2}PIM to add functionality to the application environment and control even fine-grained PIM interaction with host hardware. Furthermore, by running natively, Sim^{2}PIM makes use of the Operating System (OS), with its libraries, kernel calls and any other native element. This integration between native application and the PIM can be added automatically by the instrumentation tool, or manually by invoking Sim^{2}PIM as an Application Programming Interface (API). The Sim^{2}PIM framework provides:

- **Hardware Prototyping Flexibility** - The PIM-simulator modularity allows the developer to deal with PIM hardware and its design independently from the application and instrumentation. The framework allows the designer to experiment different PIM designs and their interaction with different host resources, including multiple cores and multiple processors.
- **Fast PIM Prototyping** - The framework provides a flexible abstraction level, allowing the developer to decide the simulation level of detail during development, including hardware description language, PIM technology, and architecture. Since host integration is guaranteed, this leads to a fast implementation time.
- **Fast Application Prototyping** - Sim^{2}PIM also allows the developer to quickly experiment with multiple software-side techniques to improve PIM performance (e.g., number of threads, thread scheduling, data organization). This possibility is easily supported as the framework integrates natively with *C code* and its libraries.
- **Fast Execution** - Since the framework runs native host code on the host processor, and simulates only the PIM side, its performance is directly dependent on the level of detail and complexity of the PIM design.
- **Host Independence** - PIM designs are expected to be coupled with different hosts (e.g., Intel, AMD, ARM). This work allows experimenting PIM adoption in any system by running native code on the native host processor.
- **Host Metrics** - Sim^{2}PIM allows access to real metrics provided by the host's HPC. Hence, it is possible to evaluate the impact of the PIM design on the entire system.

Our proposal minimizes overheads when not simulating PIM instructions. Thus it achieves execution speeds similar to performance profiling tools such as *perf* on host code, with as little as 10% run-time overhead and less than 2% metrics difference for most applications. Additionally, utilizing the host hardware and OS resources allows Sim^{2}PIM to simulate multiple PIM threads concurrently, exploring natural parallelism for the tested applications, achieving more than 8× simulation speedup compared to a sequential simulation and orders of magnitude compared to other simulators.

This paper is organized as follows: In Section 2, we show the wide range of PIM architectures and some integration strategies to unmodified host processors. Section 3 we introduce the gap that other available simulators leave for single and multi-thread applications and show how Sim^{2}PIM is well suited to this environment. Section 4 presents a comprehensive view of the complete framework architecture. In Section 5 we evaluate qualitatively Sim^{2}PIM in relation to other PIM simulators, and measure its simulation speed and accuracy overheads in relation to the host baseline. Finally, we conclude the paper on Section 6 and present the next steps of our research on Section 7.

2. Background

Processing-in-Memory (PIM) aims to mitigate data movement and bring processing capabilities closer to memory at different levels. Several memory technologies and modern integration techniques are candidates for PIM implementations, such as by exploring and modifying ordinary DRAMs memories [6, 16], by examining inherent capabilities on modern *Memristor* devices [17], or supported by 3D-Stacked solutions [18, 19]. Figure 1 displays common types of PIM devices and their possible placement in a 3D-stacked memory model. Each approach has its own specific set of characteristics, and such differences imply different architectural designs. These solutions can be characterized as:

- **Near-Data Processing Core** approaches propose memory chips with complete processing cores, which rely on their cache hierarchy and proven methods for multi-task processing [2, 9, 20].
- **Specialized Processing Units** adopts application-specific hardware units capable of directly accessing the PIM memory to improve computation time [21].
- **Near-Data Functional Units** bring simpler Functional Units (FUs) to the memory, taking full advantage of the available area and power for compute capability. This approach requires the host to orchestrate code emission and communication [1, 4, 8, 22].

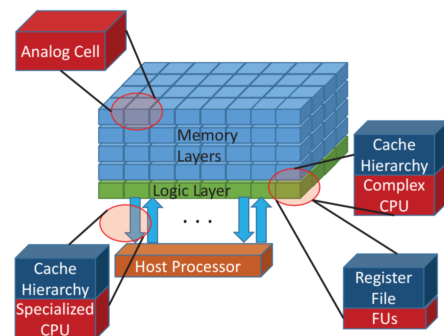


Figure 1: Common Types of Processing-in-Memory Devices

Table 1
Comparing features of PIM Simulators and Sim^{2}PIM.

	PIMSim[13]	SiNUCA [12]	CLAPPS [14]	HMC-SIM 2.0 [30]	MNSIM [31]	CIM-SIM [15]	gem5 [11]
Host Independent	N	N	Y	Y	Y	Y	implementable
PIM Agnostic	Y	Y	N (HMC)	N (HMC)	N (Memrsitor)	N (Memrsitor)	Y
Fast Prototyping	N	N	Y	N	N	N	N
Fast Execution	N	N	Y	Y	Y	Y	N
Host Metrics	profiling	N	N	N	N (behavior-level)	N (functional simulation)	implementable
Flexible Abstraction Level	3 modes	N	N	N	N	N	Y
Native System Calls	Y	Y	Y	N	-	-	(gem5-libs)
Thread-isolated Simulation	N	N	N	N	-	N	N
System-level support for PIM	N	N	N	N	-	N	Y

- **Analog Cells** compute data *in situ* using DRAMs cells [6, 16] or Memristive devices [5]. As they usually present more restricted areas and power boundaries, these devices also rely on the host processor.

PIM designs that adopt full processors require no modifications on the host side to provide cache coherence, data consistency, and virtual memory support since they can rely on well-established multi-processing methods (i.e., OpenMP, MPI) [23–25].

Some of the PIM solutions described above do not have the means to integrate with the host by themselves. Thus, to couple with a general-purpose environment, these PIMs require a series of novel solutions for code offloading, cache coherence, and virtual memory support [26]. Code offloading can happen in different granularities. The fine-grain approaches rely on individual instructions being offloaded to computing units [21, 27, 28]. Coarser grained implementations dispatch more complex commands to PIM units, such as MPI, OpenMP [23], CUDA-like functions [25, 29], and kernel functions[18].

Cache coherence is dealt with distinct methods in PIM designs. GraphPIM[21] uses a reserved uncacheable memory space for PIM memory, and to guarantee cache coherence, all data allocated to this region bypasses the cache hierarchy. DNN-PIM[18] proposes a modification to openCL, inserting an explicit method for host-PIM synchronization. Some designs use flush calls to guarantee cache coherence via high-level API [25] or compiler inserted flush instructions [28].

Virtual memory support allows PIM to integrate easily with current programming methods and practices, including the abstraction of memory addresses and ensuring multi-process isolation. This can be accomplished if the PIM replicates the host Translation Look-aside Buffer (TLB) and the Memory Management Unit (MMU) hardware [2, 20], or PIM must be able to share or access the host's TLB [21, 26, 28]. As seen by recent commercial solutions [8, 9], integration with an unmodified host processor is preferred for a rapid PIM adoption. Few solutions provide seamless integration with the host processor [18, 20, 25], and even fewer tackle these three requirements [28].

3. Related Works

Not all PIM designs are created equally, and most simulators available can handle only a tiny subset of these [14, 30, 31]. The simulation must be aware of the OS and the underlying hardware to handle a multi-thread application in a multi-core environment accurately. Some simulators include the hardware and a virtualized operating system, while others simulate only the PIM device and use the complete host system.

Simulators such as gem5 [11] and SiNUCA [12] can simulate entire micro-architectures with an elevated level of accuracy. SiNUCA [12] is a trace-based simulator, which uses traces generated on a real machine. The simulator has accurate descriptions of the hardware components down to the processor's pipeline. However, it can not simulate the interactions with the operating system and other processes. The simulator also suffers from the flaws of other trace-based simulators in that the benchmark can not interact with simulated hardware, as the traces have already been collected.

Researchers have used the gem5 [11] simulator to evaluate a set of applications under different hardware configurations, making this setup perfect for hardware-software co-design. The simulator is divided into several independent modules, coupled and decoupled to test different combinations. However, this modularity and broad configuration options create a notoriously steep learning curve for using the gem5 environment. While the code maintainers strive to improve usability, testing disruptive new hardware such as PIM units on the simulator can prove a hurdle, even for simplistic experiments.

A faster alternative is to use PinTools [32]. Utilizing trace files containing cycles, memory access, and data as input for basic processor and memory hierarchy models,

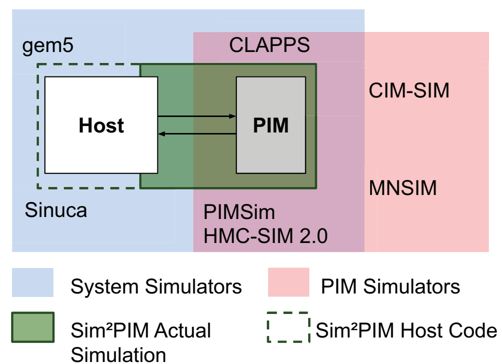


Figure 2: Simulators scope when considering system integration. gem5 [11], Sinuca [12], CLAPPS [14], PIMSim [13], HMC-SIM 2.0 [30], CIM-SIM [15], and MNSIM [31].

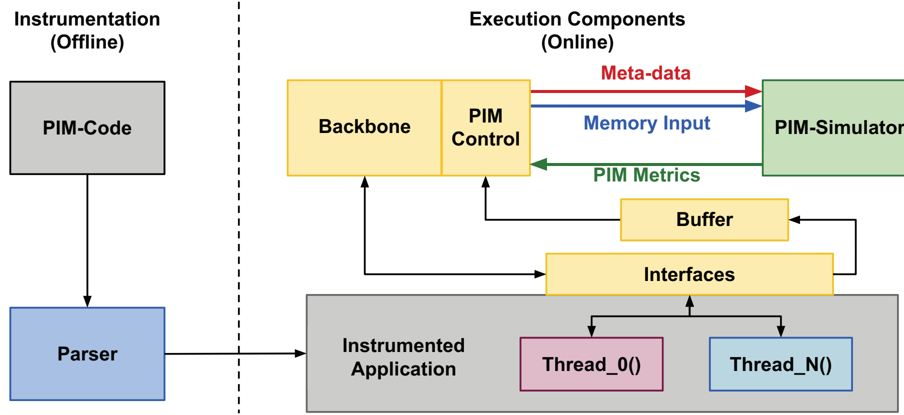


Figure 3: Overview of Sim^{2}PIM modular components and execution phases.

acting much like SiNUCA, the tool can interpret each issued instruction. PinTools's huge instrumentation overheads prohibit direct code measurements and gem5's extensive simulation times and barrier of entry. None of these simulators can handle threaded applications with native system calls. Baremetal simulators as [12] can not simulate OS-level thread scheduling and system calls, while gem5 based simulators still face long simulation times and added virtualization overheads.

However, even with limited support from simulators, multi-thread applications are a majority in high-performance computing applications. Thus, the need arose for simulators capable of handling multiple memory stacks at the host and PIM sides. Developed explicitly for this purpose, MultiPIM [33], based on two other simulators [34, 35], can simulate a multi-stacked-memory PIM device. The simulator offloads *POSIX* and *OpenMP* threads, mapping them to the PIM hardware, maintaining coherence between cores, and a PIM-side task scheduler. It can therefore handle multi-thread applications on side. However, the simulator utilizes Intel's PinTool [32] based instruction feeding mechanism, which interprets each instruction in the virtual environment at runtime. The program must then simulate all the metrics in a virtual environment, bearing a long simulation time.

Figure 2 summarizes the current academic simulation ecosystem [11, 12]. Also, current PIM architecture simulators are either architecture-specific [30] 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 [31], and again rely on tools presenting a heavy overhead [15]. Table 1 shows the most important features present in several available simulators for comparison. As Table 1 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.

4. Framework Architecture

The Sim^{2}PIM framework operation is composed of two different phases, offline instrumentation and online execution. The offline phase contains the instrumentation parser, responsible for inserting instructions in the PIM's application assembly code. This is accomplished with a low, and more importantly, known overhead, as is shown in Section 4.1. The online phase is composed of several functionally separate modules. It integrates the offline phase output, the backbone, and the PIM-simulator interface. An overview of the modules is shown in Figure 3.

Sim^{2}PIM strives in software-hardware co-design by not hampering software development and providing a clean interface for any level of PIM simulation. For example, if the PIM-simulator is implemented as a timing-aware functional simulation written in *C* language, the use flow of Sim^{2}PIM is as follows:

1. Generate PIM code assembly (e.g., using a PIM compiler [27]).
2. Parse the assembly with the instrumentation tool (in blue Figure 3).
3. Compile the Sim^{2}PIM backbone (in yellow Figure 3).
4. Compile the PIM-simulator (in green Figure 3).
5. Link the instrumented assembly with the backbone and PIM-simulator.
6. Execute the binary.

If the application software needs to be rewritten, only the instrumented assembly must be generated again. When a hardware detail must be changed, the PIM-simulator can be modified and re-linked to the rest of the framework. For the scenario described, Sim^{2}PIM usage does not differ from compiling a program with an ordinary compiler to run on the terminal. Sim^{2}PIM supports *pthread*, as it is the lowest level API for multi-threading. It could also easily be extended to other multi-processing paradigms.

4.1. Instrumentation

Sim²PIM leverages a static instrumentation tool, namely a parser. The role of the parser is to replace any code offloading method, such as explicit PIM instruction (e.g., generated by a compiler [27]), code annotated section (e.g., OpenMP, MPI [23]), or function/block call (e.g., CUDA [36] by an instrumented code that represents the original offloading method behavior, and provides the necessary information to the simulator.

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
```

For example, in the case of a PIM that adopts an exclusive Instruction Set Architecture (ISA), a compiler can mix both host and PIM instructions [27, 28], as illustrates in Listing 1. Hence, the parser will scan the code and replace each PIM instruction with a predetermined code block to meet the target code offloading method behavior [28]. Additionally, this block is responsible for feeding the instruction and data address to the simulator with a minimal and known overhead.

Listing 2:
Parsed x86+PIM Code Snippet

```
1  movq %rax, %r14
2  . . . . .
3  subq $120, %rsp ;Adjusting Stack Pointer
4  pushq %rax ;Saving registers
5  pushq %rbx ;Saving registers
6  pushq %rcx ;Saving registers
7  pushq %rdx ;Saving registers
8  pushq %rdi ;Saving registers
9  pushq %rsi ;Saving registers
10 . . . . .
11 movq PIM_LOAD_OPCODE, %rcx ;read PIM instruction as a
       string
12 movq %rcx, (GLOBAL_VAR_PIM_INST) ;PIM Instruction is
       emitted to the simulator
13 leaq 32512(%rbx, %rax ), %rcx ;PIM memory access
       calculation in case of LOAD/STORE
14 movq %rcx, (GLOBAL_VAR_PIM_INST_ADDR) ;PIM memory
       access address is emitted to the simulator
15 callq PIM_interface ;PIM interface call (Section 4.2)
16 . . . . .
17 popq %rsi;Recovering registers
18 popq %rdi;Recovering registers
19 popq %rdx;Recovering registers
20 popq %rcx;Recovering registers
21 popq %rbx;Recovering registers
22 popq %rax;Recovering registers
23 addq $120, %rsp ;Adjusting Stack Pointer
24 . . . . .
25 addq $2048, %rax
```

In Listing 2, lines 3 through 9 and lines 17 through 23 maintain the architecture calling convention and adjusts stack pointers to avoid overlapping data addresses in the stack, much like Pin [32]. Lines 11 through 14 in Listing 2, the PIM instruction is fed to the simulator as data. This method allows a user-defined instruction encoding, easily customizable and accessible by the simulator on a fixed address (as shown in the code comments). This also allows the user to experiment with any instruction, even iteratively explore different PIM ISAs. Finally, line 15 adds the *PIM_interface* function call to the code, which will effectively call the simulator, as described in Section 4.2.

Contrary to Pin's dynamic instrumentation [32], the instrumentation is static, so it can be done beforehand, avoiding Pin's severe Just In Time (JIT) execution overheads [13, 32]. Due to the modular nature of the framework and the application being instrumented separately, Sim²PIM avoids linkage errors. The interface API calls can be inserted automatically by the parser or manually by the programmer in the original *C* code. It can replace *pthread* calls with the Sim²PIM interface API to allow dynamic simulation support for multi-threaded applications. For basic tests inserting the interface API directly might prove faster. However, as the application code grows larger, it becomes easier and more reliable to use the parser. The currently available interfaces represent the most basic functionality of Sim²PIM.

4.2. Interfaces

The role of the interfaces is to add functionality to the application code with the least amount of interference as possible. This is accomplished in two ways: first, code and memory accesses inside the interfaces are kept to a minimum, avoiding too much interference with the caches. Second, as will be discussed in Section 4.3.1, the interfaces efficiently use the HPC to avoid measuring their overhead.

PIM_interface: This interface is inserted right after PIM instruction offloads or annotated PIM blocks. The PIM_interface serves as the output of the application environment. It contains the logic required to retrieve the PIM instructions and memory access addresses. This information is offloaded from the application to the backbone through a non-blocking software FIFO buffer, discussed in Section 4.3.3. The only scenario where this interface presents blocking behavior happens when the host and PIM need to synchronize, discussed in Section 4.5.

create_interface: We encapsulate the original *pthread_create* calls in application code by directly changing the function call, with the same inputs. As will be shown in Figure 6, this wrapper allocates a new physical core for the thread and then measures the *pthread_create* with the performance counters. It substitutes the thread function with a dedicated thread launcher function, which is responsible for executing the performance counter setup for the new thread before it is launched (Setup_Thread), as shown in Figure 4.

join_interface: Much like the *create_interface*, this function encapsulates calls to *pthread_join* function. The *pthread_join* function is a blocking interface that awaits the end of the issued thread. Due to its blocking behavior, measuring the performance counters when the thread stays blocked is pointless, as simulated and executed metrics are

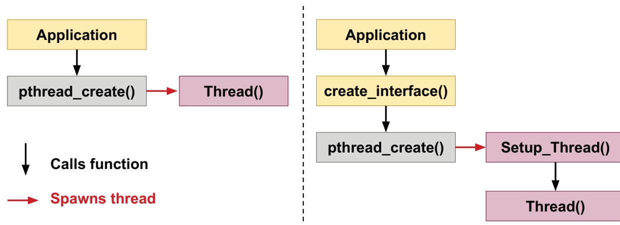


Figure 4: Creation of threads before instrumentation (left) and after (right).

distinct. Thus, this wrapper's role is to avoid measuring the blocking behavior, as will be shown in Figure 6.

4.3. Backbone

Sim²PIM online phase tries to isolate the application, backbone, and PIM-simulation in different physical cores for higher simulation speed and precision. Not only does this provide more accurate metrics for the application isolated from the simulation environment, but it also allows for any application threads to actually execute in parallel. The backbone contains Sim²PIM entry-point. It is responsible for generating the multi-thread infrastructure in which the entire simulation will execute. There are inherent advantages to using multiple threads in the simulation environment. Since the application, the backbone, and the PIM-simulator are all threads in the same program, they can easily share the same memory space. Thus it is trivial for the PIM to operate over the target application data as if both PIM and data were physically on the same memory device. This allows simulating PIMs that reside on the system's main memory or accelerators in a specific memory device (by adjusting the data movement overheads).

Additionally, the host's cache hierarchy guarantees cache coherence for the simulation data. The process described in [28] can be used to implement the coherence and virtual memory support in unmodified hardware, or the simulator could be coupled with other methods of cache coherence [37]. The backbone is responsible for several housekeeping tasks and interfaces, including environment setup, application thread management, instruction and data buffers, and PIM-simulator and application interfaces. It also contains the low-overhead assembly functions responsible for calling the HPC, and the output functions responsible for delivering the final metrics.

4.3.1. *Precise measurements*

The HPC are overflow counters. This means one must acquire the difference between two consecutive measures instead of an absolute value. Sim²PIM makes use of inline functions to call the *rdpmc* instructions directly. These instructions take as input the configured HPC register and an output register. Multiple threads can share a core, so the return values of the *rdpmc* instructions are stored in a per-thread data structure. The usage of the counters is straightforward. At the beginning of a backend code segment, the counter values are collected with a **STOP COUNTERS** call. When the backend code ends, the counter values are collected with a **RESUME COUNTERS** call. The values subtracted from subsequent resume-stop calls represent the measured code, the *application space*. Backend code effectively runs in a blind spot of the measurement functions, which we call the *Sim²PIM space*.

There are multiple HPC available for any given host, such as *retired instructions*, *unhalted cycles*, *cache misses/hits*, among others. Sim²PIM avoids over-engineering a solution to this multitude of possibilities by providing a clean, uniform interface through a user-defined configuration file. The counters themselves can be configured beforehand for any host architecture.

4.3.2. *Environment Setup*

Sim²PIM flow for a single-threaded application occurs as in Figure 5. We can see that the *application space* (dashed lines) contains the original application code, while the rest of the framework resides in *Sim²PIM space*. The following steps are executed in this part of execution:

CONFIGURE COUNTERS: At the start of the simulation, the configuration file is read, and the counters are selected for the simulation (run time defined as command-line inputs). There is no need to recompile the simulator for using different active counters, and more than one counter can be used simultaneously. However, they will be triggered sequentially to access multiple counters at once, which reduces the precision due to compounding overheads. This feature is helpful for when an extended test run is required, and a trade-off between losing accuracy in some metrics and a smaller number of total executions is acceptable.

WARM-UP: Different hosts and compilers may result in different measurements and generated code. Considering the relevance of precise metrics, this module executes several

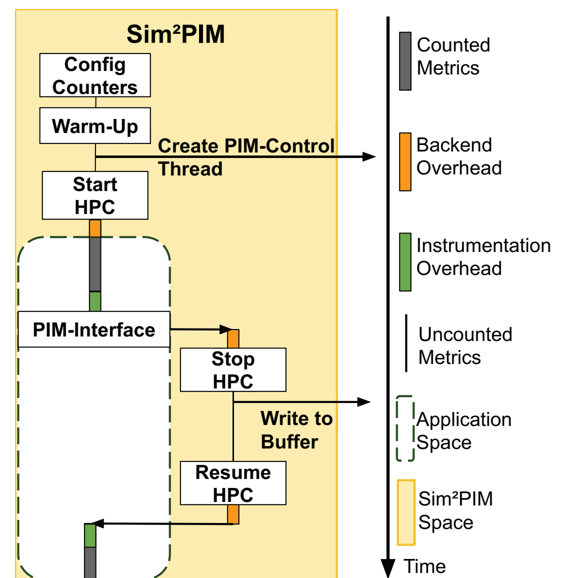


Figure 5: Interfaces and overheads of offloading data from the application to PIM-simulation.

back-to-back HPC calls to collect the instrumentation overhead. These overheads include the serialization instructions inserted on each HPC call (as is shown in Section 4.1) and the instruction overhead required to read the HPCs. In Figure 5 the measured overheads for instrumentation are represented before and after the PIM-interface (green in Figure 5), and the HPC calls overheads before each counter invocation (orange in Figure 5). The values extracted here are subtracted after each call in *Sim²PIM space*.

CREATE ENVIRONMENT: Sim²PIM tries to isolate the simulation from the user application to provide more accuracy for the hardware counters by reserving a physical core for the PIM-Control and another one for the PIM-simulator. 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.

4.3.3. Communication Buffer

PIM devices that connect with unmodified host processors through the memory channel do not have 2-way communication; hence the PIM can not directly write on processor cache memory and registers. This induces PIM designs to operate asynchronously from the host akin to the main memory device, which means the host does not block on PIM instructions.

When a single-thread application interacts with the PIM-simulator, the instruction offload throughput from the host might not be enough to keep the PIM-simulator fully occupied. However, as more threads offload to the PIM-simulator, there will be contention on the input-side and a bottleneck on the output. This can happen because one core is responsible for simulating PIM instructions delivered from many cores. Although this is an exclusive simulation bottleneck, it may cause contention on the host side, artificially reducing the PIM instruction offload throughput and the host's native instruction execution.

To mitigate this effect in the multi-threaded architecture of Sim²PIM, the PIM-interface writes a data structure to a non-blocking FIFO buffer (i.e., a controlled shared memory space). Each thread has its row in the buffer, so they do not contend between themselves. The data structure written contains instructions, data addresses, and any other meta-data Sim²PIM has access to, including current performance counter values. The non-blocking buffer is implemented with atomic primitives to avoid costly system calls. On the other side of the buffer, the PIM-Control retrieves the data structure from all the threads and offloads them to the PIM-simulator as needed. As shown in Figure 5, when the application code calls the PIM-interface, the HPCs are sampled, and the code is now in *Sim²PIM space*. Thus, it can interact with the instruction buffer without the risk of interfering with host metrics. When the PIM-interface is done with the buffer, the counters are resumed, and the code returns to *application space*.

4.3.4. PIM-Control Interface

The PIM-Control Interface is responsible for reading the communication buffer (Section 4.3.3), invoking the PIM-simulator, and collecting the PIM simulation metrics. It is worth noting that any PIM-simulator that fulfills the interface requirements can be coupled to this interface, especially those that take trace-files as inputs [12–15, 30, 31]. As shown in Figure 3, the PIM-Control Interface feeds three types of input to the PIM-simulator: meta-data, PIM instruction, and data addresses. Meta-data outputs consist of those originating from the host code instrumentation. These can include the thread that originated the instruction and any available metrics, even the cycle in which the PIM instruction was issued and the number of cache misses. The PIM instruction and the address are the outputs from the host processor to the PIM.

Table 2 shows the execution traces collected from the Communication Buffer by the PIM-Control Interface in an example application. With it, the PIM-simulation is aware of the thread that sent the instruction, the PIM instruction opcode, data addresses in case of memory instructions, and any host metrics that the simulation might need. Thus, the PIM-Control Interface can operate as a dynamic trace generator for any simulation that is couple with it, providing run-time metrics. If the PIM-simulator is not integrated within the application address space (if the simulator is in another process), this control thread can access the data addresses directly and transmit the data back and forth to the simulator. As discussed in Section 4.3.3, the buffer consists of a dedicated memory space, accessible by each thread, this is seen in the first column of Table 2, where PIM instructions from different cores are received simultaneously.

4.4. Application Thread Management

As shown in Figure 6, the application itself is a function to be called in the backbone between instrumentation sections. For the simulation of a multi-thread application, each thread will perform its backend calls. As each core has its performance counters, we must ensure that the threads do not migrate cores, which would result in a wrong calculation of the metrics.

If the number of threads is greater than the number of available cores, Sim²PIM can offer two distinct strategies: **1)** it can allow for all the threads to be launched by the main thread and dispute core time with each other. This approach would enable the OS to optimize thread context switches and simultaneously keep a more significant number of threads alive. Alternatively, **2)** it can allow the execution of only one thread per physical core at a time. Thus, this solution provides the best accuracy for individual threads, even allowing for better profiling of the PIM instructions.

As shown in Section 4.2 *pthread_create()* and *pthread_join()* functions are replaced by the interface functions, *create_interface* and *join_interface* respectively. These interfaces wrap around the *pthread* call functionality, adding the capability of setting the core affinity, marking the physical core as in-use (if using the strategy mentioned above

Table 2
Run-Time traces collected from the host execution. The first 6 PIM instructions received by the PIM-simulation from a multi-threaded vecsum application with Reconfigurable Vector Unit (RVU)-based ISA [4].

Thread	PIM Instruction	Opcode	Memory Address	Cycle	Host Instructions	L1 Hit	L1 Miss	L2 Hit	L2 Miss
0	PIM_LOAD	0x005200000000000	0x00a14180	817	147	16	1	1	0
1	PIM_LOAD	0x005200000000000	0x00c14180	517	147	16	0	0	0
0	PIM_ADD	0x095200002000040	-	898	258	48	1	1	0
1	PIM_ADD	0x095200002000040	-	597	258	49	0	0	0
0	PIM_STORE	0x015200002000000	0x00e14180	1917	389	80	1	1	0
1	PIM_STORE	0x015200002000000	0x01014180	2425	389	82	0	0	0

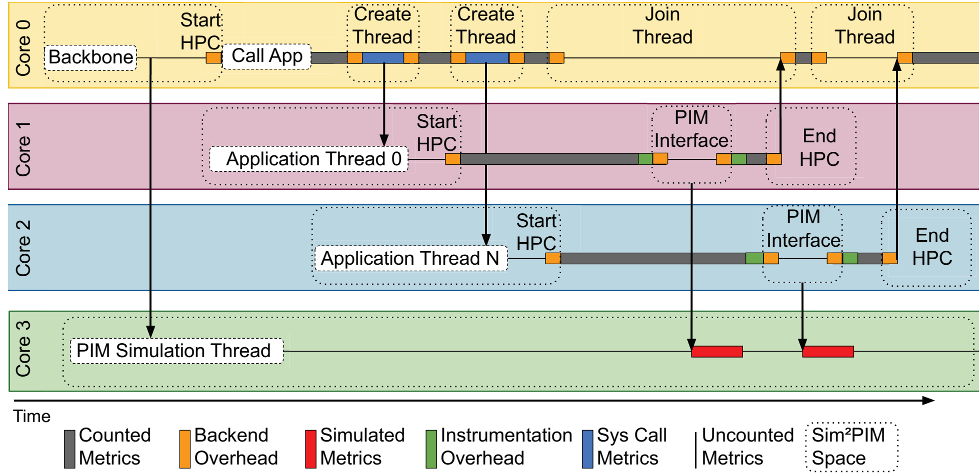


Figure 6: Overhead diagram for a multi-thread application on the Sim²PIM with the Hardware Performance Counters (HPC).

2)), and acquiring metrics before the start of the thread function itself. Inside the *create_interface* function, the only measurement made is around the original *pthread_create()* call. We make sure to measure the thread's launch, as this can pose a significant overhead in multi-thread applications, shown in Figure 6 as *Sys Call Metrics*.

If Sim²PIM multi-thread application executes on a Non-Uniform Memory Access (NUMA) system, it will face the same challenges as a real PIM implementation on this scenario, which means different memory access latencies for each processor and PIM devices distribution on the memory systems. On a system with multiple main memory modules (regardless of the number of memory controllers), the thread mapping and affinities set will affect the performance of the PIM application as much as in a real system. This happens because the host metrics are precise and represent the host's behavior found on the actual system. Thus it is possible for the user to try different configurations on the thread mapping to achieve better performance.

Each thread counts its metrics, and we assume all the threads are alive simultaneously. For all metrics, core/thread-wise metric counts are available. Only the largest value is considered to the final count for the total program *elapsed-cycles* and *elapsed-time* metrics, as it is the bottleneck for program conclusion. The interface function around *pthread_-join()* guarantees the waiting time for the simulations to end is not counted on the benchmark's main thread metrics, as shown in Figure 6.

4.5. Thread Synchronization

When using *pthreads*, the programmer primarily handles synchronization between threads, with mutexes, semaphores, and other atomic operations. This behavior does not change on Sim²PIM, as the host memory caches are still used for shared memory space between host threads. Synchronization between PIM-threads still happens at the host side, as the PIM depends on the host for memory accesses and instruction offloading.

If shared data is used concurrently by multiple threads (e.g., a shared vector), the typical approach would be to use *pthread* barriers or other atomic operations to avoid using outdated values in other threads, maintaining synchronization. This approach remains valid for most situations in the simulation environment, as all synchronization mechanisms still execute natively. However, in cases where the last issued PIM instruction before the barrier was a store, there can be a race condition between the PIM store instruction and the next host load instruction. A simple solution is for the host to consider PIM store instructions synchronization barriers in the execution.

4.6. PIM-Simulator

As described in Section 4.3.4, the PIM-simulator module is fed from the PIM-Control interface. PIM instruction and data addresses are used to trigger specific operations (e.g., PIM arithmetic, PIM memory access), while the meta-data involves metrics that might be useful to the simulation, such

as the cycle in which the PIM instruction was emitted. This module can take the complexity level the designer needs, from a cycle-accurate to an instruction-level look-up table. Moreover, due to the modular nature of the Sim²PIM, for this module, any language can be adopted, as well as connected to different tools (e.g., specialized memory or PIM simulators [15]). Therefore, the designer can use hardware description languages (e.g., SystemVerilog, SystemC, VHDL) or high-level abstract languages (e.g., C, C++, Python).

5. Framework Evaluation

As shown in Table 1, most simulators focus on PIM architecture experimentation. However, they lack connections between actual hosts and simulated architectures. Thus, their metrics need to be *virtualized*, as no real hardware is in play. Furthermore, this behavior prevents the utilization of existing host resources, such as multiple cores, integration with the memory system, OS support, and the HPC. While these features can be simulated, they make implementation more complex, more costly, and less accurate if not done carefully.

This Section evaluates the Sim²PIM framework, showing the benefits of simulating only the design of interest. To assess the framework, we implemented three different PIM approaches, as illustrated in Figure 7. The first is a PIM based on analog computation on DRAM cells [16] that also serves as the host main memory. This approach can be seen in Figure 7a, it violates the DRAM command timings to force cells to share charges and accomplish analog bit-wise computation on DRAM rows. This approach is the simplest to map to our simulation system, as it uses standard DRAM data arrangement. The second is a *memristor* based on the technology used in the ISAAC pipeline [38]. The array acts as an accelerator to the system. This approach does not use the PIM as a main memory unit. Thus, we separate a range of addresses to act as the memristor data space, as shown in Figure 7b. The communication between main memory and memristor memory is done with Direct Memory Access (DMA) commands. Finally, the last approach implements FUs within a 3D-stacked memory [1, 4, 8]. This case study is based on the RVU architecture [4] and [28]. We map groups of vaults in the RVU to a different DRAM chip, thus they operate on their own data addresses, as illustrated in Figure 7c. For the PIMs that use different memory technologies than the host system, the data access latency is adjusted according to estimates provided by the original authors. Table 3 summarizes the hosts' systems and PIMs parameters.

5.1. Overhead Evaluation

Two main types of overheads are inserted in the application code by Sim²PIM: *PIM_interface* insertion and the *create_interface*. The former interface is inserted before each PIM instruction, while the last one replaced the original *ptrhead_create()* in case of a multi-threaded application. As mentioned in Section 4.3.2, the warm-up phase is required to remove the overheads from the HPC and PIM_interface calls. To exemplify this, we collected these overheads for two different host processors for the instructions and cycles metrics. These overheads are directly dependent on the host processor, as illustrated in Table 4.

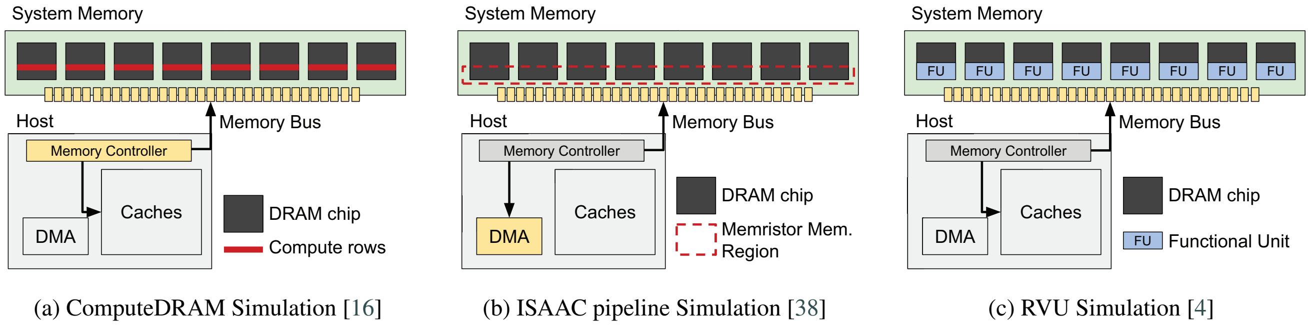
In the case of multi-threaded applications, the multi-thread overheads happen in the *create_interface* function, as aforementioned in Section 4.4. To evaluate the impact of these overheads on PIM multi-thread simulation, we compared the same set of applications executing with and without the simulator. We selected the well-established *perf* as an easy to deploy, low-overhead, and high-accuracy performance profiling tool for the comparison. We used some of the algorithms on the PolyBench benchmark suite [39]. The results are shown in Table 5.

Table 3
Baselines and Case Study PIM Parameters

Baseline/Host Intel i5-7600 @ 3.5GHz; Cache per Core L1 = 32kB; L2 = 256kB; Last Level Cache = 6MB; Main Memory DDR4 1x16GB 2400MHz CL18;
Baseline/Host Intel Xeon Silver-4214 @ 2.2GHz; Cache per Core L1 = 32kB; L2 = 1024kB; Last Level Cache = 16MB; Main Memory DDR4 2x32GB 2400MHz CL16;
Baseline/Host AMD R5-1600 @ 3.2GHz; Cache per Core L1 = 32kB; L2 = 512kB; Last Level Cache = 8MB; Main Memory DDR4 2x8GB 2666MHz CL16;
ComputeDRAM [16] Bus Frequency: 400 MHz Configurable bit-width, up to 8 kB bit-wise Vector Operations Operations in bus cycles: Row Copy 18 cycles; SHIFT 36 cycles; AND 172 cycles; OR 172 cycles; XOR 444 cycles; ADD 1332 cycles
ISAAC pipeline - [38] Crossbar read: 100 ns (10 MHz) Crossbar - 128x128, 2-bit cell; *In-Situ Multiply Accumulate* (IMA) - 8 crossbars; *Tile* - 8 IMAs, capable of 8 concurrent operations; 64 kB eDRAM buffer per Tile; Vector-Matrix Multiplication: Size: 128 elements vector and 128x128 elements matrix (16-bit word); Latency: Pipeline of 22 cycles.
RVU Processing Logic [4, 28] Operation frequency: 1 GHz; Up to 32x 64 functional units (integer + floating-point); Vector sizes (bytes): 32x 256, 16x 512, 8x 1024, 4x 2048, 2x 4096, 1x 8192 Latency (cycles): 1-alu, 3-mul. and 20-div. int. units; Latency (cycles): 5-alu, 5-mul. and 20-div. fp. units; Register bank: 8 sets of 32 composable registers of 256 bytes each;

Table 4
Average overheads for two different HPCs, unhalted cycles and retired instructions. Measured with 10,000 repetitions in the warm-up phase of two different processors.

Overheads		Intel Core i5-7600@GCC7.5	AMD R5-1600@GCC9
# Cycles	Instrumentation	174	184
	Backend	168	180
# Instructions	Instrumentation	28	27
	Backend	9	8



(a) ComputeDRAM Simulation [16] (b) ISAAC pipeline Simulation [38] (c) RVU Simulation [4]

Figure 7: Sim²PIM address space organization for different types of PIMs in the evaluated system. The PIM can be part of the host memory space (7a,7c) or be on a separate address space (7b).

Table 5
Simulated Cycles vs. Simulation time for Sim²PIM and *perf* on the AMD processor.

Benchmark - data size	Perf cycles		Sim²PIM cycles		cycles % increase		Perf Time (s)		Sim²PIM Time (s)	
	1T	4T - Average	1T	4T - Average	1T	4T - Average	1T	4T - Average	1T	4T - Average
vecsum - 32MB	1.25E+07	3.06E+06	1.23E+07	3.05E+06	-1.799	-0.541	0.0165	0.0083	0.037	0.035
gemm - 1,5MB	2.89E+07	9.02E+06	2.84E+07	8.77E+06	-1.577	-2.771	0.0173	0.0089	0.029	0.033
2mm - 750kB	1.57E+08	3.93E+07	1.57E+08	3.92E+07	-0.018	-0.255	0.0524	0.0219	0.039	0.046
covariance - 16MB	1.58E+09	4.30E+08	1.61E+09	4.23E+08	1.898	-1.734	0.7163	0.4740	1.041	0.5
Floyd-Warshall - 8MB	1.18E+10	3.93E+09	1.18E+10	3.88E+09	-0.018	-1.261	3.2186	1.1955	3.232	1.22
Nussinov - 8MB	2.86E+10	7.23E+09	2.88E+10	7.21E+09	0.597	-0.319	7.7568	1.9769	7.825	1.998

For both the single thread and multi-thread results, we can see that for most benchmarks, Sim²PIM and *perf* show similar results, with a trend towards more minor results in Sim²PIM. We leverage this trend is due to the instrumentation provided by Sim²PIM to be more accurate due to the use of hard-coded HPC, not depending on slower system calls. The cycles metric is influenced by several factors, including the congestion of the memory subsystem and frequency fluctuations. Thus some applications may present more variation than others. Splitting the application between threads might also affect this behavior, increasing the traffic in the processor caches. We can also see the execution time collected with the *time* command for Sim²PIM and *perf* itself. Although Sim²PIM adds execution time, this effect is smaller as the execution gets longer. We can see that Sim²PIM's performance for host code is very competitive, especially if we consider the usual run-times of other simulators.

5.2. Single-Thread Simulation Time Evaluation

Sim²PIM merits lay on top of its high simulation speeds, high-accuracy host hardware metrics, and the backbone's structure high modularity. These characteristics make Sim²PIM especially suited to evaluate the interactions between host hardware and the PIM device, including the system's memory hierarchy and technology. Thus we evaluate the 3 selected PIM architectures in their interaction with the host system with the characteristics highlighted in Table 3. This is not an attempt at comparing the architectures, as this is beyond the scope of this paper, so for each test we execute a benchmark that is best suited for each architecture. In all tests, the data starts on the host main memory, we measure the cycles spent on the host, the simulated PIM metrics, and the actual runtime of the simulator.

For the ComputeDRAM PIM benchmark, we chose a database bitmap indices application [40]. It filters a database of 64 thousand identifiers (e.g., users of a platform) searching the presence of any of 100 characteristics. Mapping this application to the memory, each identifier occupies a column in memory, with each characteristic occupying a bit in the row. Thus, we must execute 100 *OR* operations between rows to filter the database. When the PIM operations are completed, the host accesses the resulting data to check for membership. The results can be seen in Figure 8. We can see that most of the simulated metrics happen on the PIM-side of the simulation, as the host has a merely auxiliary role in the PIM's operation. The application and simulation time are also proportional to this characteristic, meaning most of the time spent on Sim²PIM's execution is on the simulation side.

For the memristor architecture similar to the ISAAC pipeline [38], we chose a series of vector-matrix multiplications, representing the convolution operations of a Convolutional Neural Network (CNN). As this is not an attempt at benchmarking the architecture, we only execute a single layer with pipelined input data. This means we operate only one *Tile* of the architecture. As ISAAC's pipeline is arbitrarily long to accommodate the appropriate number of layers in a Deep Neural Network (DNN), one *Tile* is enough to present the behavior of interaction with the host. Assuming an application layer that performs a dot-product operation on a $4 \times 4 \times 64$ matrix, and there are 64 kB of input data to be processed. The host will have to fill the accelerator eDRAM memory and collect the results back to main memory. The results of this operation can be seen in Figure 8. As the PIM and host do not share memory space, the host has a role of moving data back and forth from the accelerator, this impact is visible as for this small application, the data movement

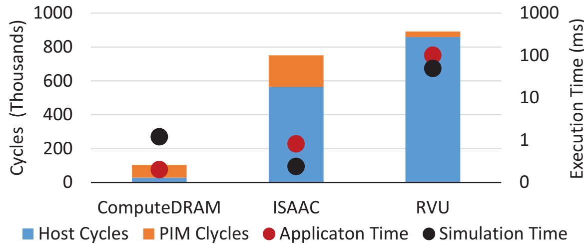


Figure 8: Simulated Cycles and Simulation Time for each PIM architecture executing different benchmarks.

impact on the host is large. Of course, this impact would be lessened with more tiles in the pipeline, as data would remain in the accelerator for longer. Here the actual time spent on the simulations is smaller than the time spent on host-side application code.

Finally, for the tests with the RVU architecture, we execute a data crunching kernel composed of a series of arithmetic operations that can represent a liner-algebra equation. This is done because it exposes the architecture more of the architectural complexity of an architecture with internal registers and more diverse operations [4]. In this example, each equation is composed of 3 arithmetic instructions (i.e., ADD, MUL, and SUB) that will operate with 8 kB vectors on of 8 MB of data. The host will offload all the instructions to the PIM, dispatching load instructions that will access the main memory, the arithmetic instructions that will operate in data stored in the internal registers, and the store operations that will save data back to the memory. The results of this operation can be seen in Figure 8. On these tests, this architecture depends on the host for issuing instructions [28], this impact is seen as most of the metrics and simulation time are spent on the host code. Furthermore, the simulation time is significantly higher than the other architectures as we implemented a much more complex and in-depth PIM-simulation. For the purposes of demonstration, both ComputeDRAM and ISAAC PIM-simulations are comprised of fixed function implementations that replicated the behavior of the architecture in code, adding the metrics according to the description provided in the original papers [16, 38]. The RVU PIM-simulation covers many more implementation cases than the ones demonstrated here, but the idea is that the researcher can increment their simulation as much as they deem necessary.

5.3. Multi-Thread Simulation Time Evaluation

We set out to showcase the speedup of simulating multiple threads in Sim²PIMs truly multi-core environment. For this multi-thread experiment, we selected from the previously presented architectures the one which is most suitable for multi-thread operations without any modifications. In this case, the RVU architecture. RVU is capable of operating on different control flows simultaneously without the problem of a fixed vector width (ComputeDRAM) or being part of a larger pipeline (ISAAC). We test the multi-thread PIM simulation on a simple embarrassingly parallel kernel that performs the vector sum (*vecsum*) over 64MB data, varying from 1 to 8 perfectly balanced threads. This way, we avoid complications dealing with complex vector algorithms and inter-thread communication.

In Figure 9 the bars represent the number of simulated cycles for the application with varying numbers of active threads. To evaluate the effectiveness of isolating simulation and application threads in different physical cores, the lines in Figure 9 represent the execution speed (ms) of three different Sim²PIM configurations: a single-core execution, a dual-core execution (simulator + application), and the standard Sim²PIM with dedicated cores. The *Sim²PIM 1-Core* line represents the single-core execution, meaning the entire framework and the application threads share a single core. While the OS can dynamically schedule them, most applications effectively run sequentially with the PIM-simulator. This is similar to many other simulators (e.g., [11, 13] that do not support parallel simulation execution readily. The results clarify that forcing the PIM-simulator and backbone to share a core with the application severely harms the framework performance. This is directly proportional to the number of threads disputing for core time, which results in an increased number of context switches between the application threads and between framework and application threads.

Thus, when we remove the framework from this dispute (by placing it in an exclusive core), as shown in line *Sim²PIM 2-Cores* in Figure 9, there is a significant release in pressure for the application core. For this example, the *vecsum* application is lightweight and straightforward on the host side, there is still contention between application threads disputing the same core, but it is significantly smaller. Finally, we achieve the most efficient execution when we execute Sim²PIM with dedicated cores (line *Sim²PIM N-Cores* in Figure 9) for each application thread and the framework. This way, the application parallelism works in favor of the framework, as long as there are enough free cores to operate in all threads in parallel. This impact might be more meaningful for larger applications on the host side when deciding how to test the application.

Our experience with simulators like Gem5 [11] is that although they can simulate very different architectures and

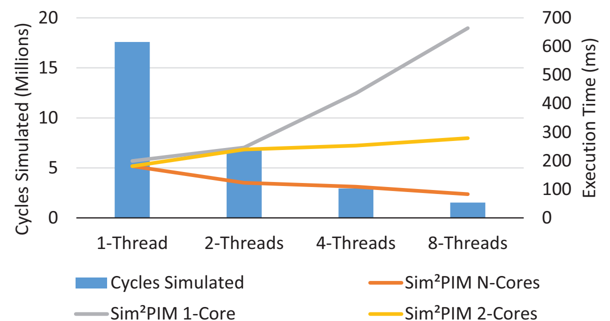


Figure 9: Simulated Cycles and Simulation Time for a 64MB *vecsum* application offloaded by the Xeon CPU to the PIM device using three different simulation configurations.

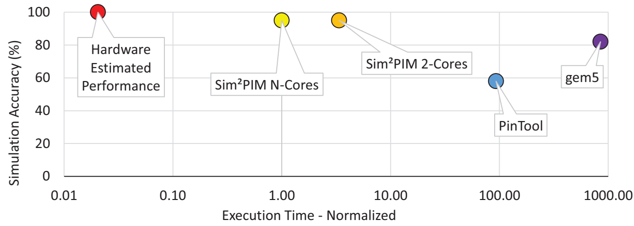


Figure 10: Execution time and accuracy for a *vecsum* application with eight threads in different simulators.

enable the design of new architectures, the designer is forced to simulate the host architecture. If the designer desires to couple the PIM with a different host, there is a need to implement the new host and its features. Quickly changing a software or hardware parameter and rerunning the simulation is not an option as the simulation can take hours, even in the most straightforward modes. Tools based on Intel's Pin [32] offer very fine control over the code currently executing, allowing the user to follow branches and see accessed virtual memory addresses without recompiling code. However, the JIT model of instrumentation and execution makes the simulation speed slow and makes quick testing a nuisance.

Besides, trace-based simulators, which require application traces, suffer from another challenge: changes in application code require the traces to be reacquired. Figure 10 presents a comparison between our previous experiments with these tools showing the accuracy of the metrics concerning simulation estimates of the actual hardware performance [4, 28] (y-axis). The simulation speeds (x-axis) are normalized to the run-time of the execution speed of Sim^{2}PIM. In case of a lack of available cores, executing applications serially in Sim^{2}PIM as shown in line *Sim^{2}PIM 2-Cores* in Figure 9, slows down the simulation and prevents us to evaluate the interaction of these multiple requests on shared resources, such as memory access bandwidth. Even then, Sim^{2}PIM is orders of magnitudes faster than the other base simulation options, either trace-based or full-system simulators.

6. Conclusion

It is more apparent than ever that the architectural community must propose new paradigms to keep up with computing trends. PIM is a clear contender to take the top spot of energy efficiency and acceleration. However, there must be leaps in architectural designs and the support environment to accelerate this development, including simulators. The Sim^{2}PIM framework brings a fast and accurate simulation tool for single and multi-threaded applications to the hands of researchers and designers. Sim^{2}PIM makes few compromises, guaranteeing fast simulation speeds and high accuracy, as long as the host hardware is available for testing. We have released Sim^{2}PIM as an open-source tool, and it is available to download at https://pim.computer/

7. Future Development

Sim^{2}PIM as a framework already has a high complexity in its inner workings. However, there is still a lot of functionalities that can be added to the simulator backend or as part of the functional simulation. As future work, we are testing ways to implement Sim^{2}PIM as an instrumentation tool for current commercial PIMs. Also, as the framework controls the creation of threads, there is space for testing how a PIM-aware scheduler could affect performance during the lifetime of host threads. The framework makes it easy to test on the current host system as is. However, we mean to test its integration with memory simulators to try and change the memory behavior for host-side applications as well.

References

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

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, K. Choi, A scalable processing-in-memory accelerator for parallel graph processing, in: Int. Symp. on Computer Architecture (ISCA), IEEE, 2015.

[3] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, R. Das, Compute caches, in: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017.

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

[5] S. Hamdioui, L. Xie, H. A. Du Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, J. van Lunteren, Memristor based computation-in-memory architecture for data-intensive applications, in: 2015 Design, Automation Test in Europe Conference Exhibition (DATE), 2015.

[6] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization, in: 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2013, pp. 185–197.

[7] H. A. D. Nguyen, J. Yu, M. A. Lebdeh, M. Taouil, S. Hamdioui, F. Catthoor, A classification of memory-centric computing, J. Emerg. Technol. Comput. Syst. 16 (2).

[8] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, N. S. Kim, Hardware architecture and software stack for pim based on commercial dram technology : Industrial product, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021, pp. 43–56. doi:10.1109/ISCA52012.2021.00013.

[9] J. Nider, C. Mustard, A. Zoltan, J. Ramsden, L. Liu, J. Grossbard, M. Dashti, R. Jodin, A. Ghiti, J. Chauzi, A. Fedorova, A case study of processing-in-memory in off-the-shelf systems, in: 2021 USENIX Annual Technical Conference (USENIX ATC 21), USENIX Association, 2021, pp. 117–130.
URL https://www.usenix.org/conference/atc21/presentation/nider

[10] P. C. Santos, B. E. Forlin, L. Carro, Sim²pim: A fast method for simulating host independent pim agnostic designs, DATE '21, 2021.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, The gem5 simulator, SIGARCH Comput. Archit. News.

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

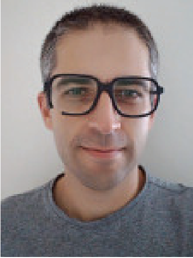
[13] S. Xu, X. Chen, Y. Wang, Y. Han, X. Qian, X. Li, Pimsim: A flexible and detailed processing-in-memory simulator, IEEE Computer

Architecture Letters.
[14] G. F. Oliveira, P. C. Santos, M. A. Z. Alves, L. Carro, A generic processing in memory cycle accurate simulator under hybrid memory cube architecture, in: SAMOS, 2017.
[15] A. BanaGozar, K. Vadivel, S. Stuijk, H. Corporaal, S. Wong, M. A. Lebdeh, J. Yu, S. Hamdioui, Cim-sim: Computation in memory simulator, in: 22nd Int. Workshop on Software and Compilers for Embedded Systems, SCOPES '19, 2019.
[16] F. Gao, G. Tziantzioulis, D. Wentzlaff, Computedram: In-memory compute using off-the-shelf drams, in: Proceedings of the 52nd Annual IEEE/ACM Int Symp on Microarchitecture, MICRO '52, 2019.
[17] L. Xie, H. Cai, 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.
[18] J. Liu, H. Zhao, M. A. Ogleari, D. Li, 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, 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] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, D. Pnevmatikatos, The mondrian data engine, in: Int. Symp. on Computer Architecture, ACM, 2017.
[21] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, H. Kim, Graphpim: Enabling instruction-level pim offloading in graph computing frameworks, in: Int. Symp. on High Performance Computer Architecture (HPCA), IEEE, 2017.
[22] M. A. Z. Alves, P. C. Santos, M. Diener, L. Carro, Opportunities and challenges of performing vector operations inside the dram, in: Int. Symp. on Memory Systems (MemSys), 2015.
[23] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. Costa, J. Doi, C. Evangelinos, et al., Active memory cube: A processing-in-memory architecture for exascale systems, IBM Journal of Research and Development.
[24] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, M. Ignatowski, Top-pim: throughput-oriented programmable processing in memory, in: Int. Symp. on High-performance Parallel and Distributed Computing, ACM, 2014.
[25] A. Drebes, L. Chelini, O. Zinenko, A. Cohen, H. Corporaal, T. Grosser, K. Vadivel, N. Vasilache, Tc-cim: Empowering tensor comprehensions for computing-in-memory, in: IMPACT 2020 workshop, 2020.
[26] P. C. Santos, J. P. Lima, R. F. Moura, M. A. Alves, L. Carro, A. C. S. Beck, Solving datapath issues on near-data accelerators, in: IFIP WG10.2 Working Conference: Int Embedded Systems Symp (IESS).
[27] H. Ahmed, P. C. Santos, J. P. C. de Lima, R. F. de Moura, M. A. Alves, A. Beck, L. Carro, A compiler for automatic selection of suitable processing-in-memory instructions, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019.
[28] P. C. Santos, B. E. Forlin, L. Carro, Providing plug n' play for processing-in-memory accelerators, in: Asia and South Pacific Design Automation Conference (ASPDAC), 2021.
[29] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, O. Mutlu, Google workloads for consumer devices: Mitigating data movement bottlenecks, in: Int. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2018.
[30] J. D. Leidel, 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.
[31] L. Xia, B. Li, T. Tang, P. Gu, P. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, H. Yang, Mnsim: Simulation platform for memristor-based neuromorphic computing system, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.
[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, Association for Computing Machinery, 2005.
[33] C. Yu, S. Liu, S. Khan, Multipim: A detailed and configurable multi-stack processing-in-memory simulator, IEEE Computer Architecture Letters.
[34] D. Sanchez, C. Kozyrakis, Zsim: Fast and accurate microarchitectural simulation of thousand-core systems, in: Int. Symp. on Computer Architecture, 2013.
[35] Y. Kim, W. Yang, O. Mutlu, Ramulator: A fast and extensible dram simulator, IEEE Computer Architecture Letters.
[36] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, S. W. Keckler, Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems, ACM SIGARCH Computer Architecture News.
[37] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, O. Mutlu, LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory, IEEE Computer Architecture Letters.
[38] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, V. Srikumar, Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars, in: Int. Symp on Computer Architecture (ISCA), 2016.
[39] L.-N. Pouchet, Polybench: The polyhedral benchmark suite, URL: http://www.cs.ucla.edu/pouchet/software/polybench.
[40] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, T. C. Mowry, Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology, in: Int. Symp. on Microarchitecture (MICRO), 2017.

Bruno Endres Forlin received a B.Sc in Electrical Engineering and an M.Sc in Computer Science from the Federal University of Rio Grande do Sul, Brazil, in 2019 and 2022. During his bachelor's degree, Bruno participated in two research projects with international collaboration. The former, from 2016 to 2018, was research on MPSoC security-focused in NoC based attacks in collaboration with TU Munich. The latter was an investigation on the feasibility to create intrinsic PUFs in GPUs, in collaboration with TU Delft. In his master's, Bruno's focus shifted to processing-in-memory architectures, their integration with the host system, and the simulation of such devices. Recently, Bruno was accepted for a Ph.D. position at the University of Twente, in The Netherlands. His research interests include processing-in-memory, parallel systems, and embedded security.

Paulo Cesar Santos completed his Ph.D at Federal University of Rio Grande do Sul (UFRGS) in 2019 and works at the Embedded Systems Laboratory as a postdoctoral researcher in the same university since 2020. He received his Master degree in Microelectronics from UFRGS in 2014, and he received his B.S. in Digital Systems Engineering from the State University of Rio Grande do Sul (UERGS) in 2011. His research interests include Processing-in-Memory systems, Novel Memory Technologies and Architectures, Embedded Systems, High Performance Systems, and Compilers.

Augusto Becker is a Computer Engineering undergraduate student at Universidade Federal do Rio Grande do Sul, in Brazil. He has taken part in projects involving Processing in Memory, Low Energy Software, and Image Processing.



Marco Antonio Zanata Alves is a Professor at the UFPR since 2016. He graduated in Computer Science from UNESP in 2006. He received his master degree in Computer Science from UFRGS in 2007 and his Ph.D. in Computer Science from UFRGS in 2014. He works in the Laboratory of Embedded Systems at UFPR. His research interests include Computer Architecture and Processing in Memory systems.

Luigi Carro received the Electrical Engineering and the M.Sc degrees from Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, in 1985 and 1989, respectively. From 1989 to 1991 he worked at ST-Microelectronics, Agrate, Italy, in the RD group. In 1996 he received the Dr. degree in the area of Computer Science from Universidade Federal do Rio Grande do Sul (UFRGS), Brazil. He is presently a full professor at the Applied Informatics Department at the Informatics Institute of UFRGS, in charge of Computer Architecture and Organization. He has advised more than 20 graduate students, and has published more than 150 technical papers on those topics. He has authored the book Digital systems Design and Prototyping (2001-in Portuguese) and is the coauthor of Fault-Tolerance Techniques for SRAM-based FPGAs (2006-Springer), Dynamic Reconfigurable Architectures and Transparent optimization Techniques (2010-Springer) and Adaptive Systems (Springer 2012). In 2007 he received the prize FAPERGS - Researcher of the year in Computer Science. His most updated resume is located in http://lattes.cnpq.br/8544491643812450.