# PiMulator: a Fast and Flexible Processing-in-Memory Emulation Platform

Sergiu Mosanu, Mohammad Nazmus Sakib, Tommy Tracy II, Ersin Cukurtas, Alif Ahmed, Preslav Ivanov,
Samira Khan, Kevin Skadron, Mircea Stan
University of Virginia School of Engineering and Applied Science

***Abstract*—Motivated by the *memory wall* problem, researchers propose many new Processing-in-Memory (PiM) architectures to bring computation closer to data. However, evaluating the performance of these emerging architectures involves using a myriad of tools, including circuit simulators, behavioral RTL or software simulation models, hardware approximations, etc. It is challenging to mimic both software and hardware aspects of a PiM architecture using the currently available tools with high performance and fidelity. Until and unless actual products that include PiM become available, the next best thing is to *emulate* various hardware PiM solutions on FPGA fabric and boards. This paper presents a modular, parameterizable, FPGA synthesizable soft PiM model suitable for prototyping and rapid evaluation of Processing-in-Memory architectures.**

**The PiM model is implemented in System Verilog and allows users to generate any desired memory configuration on the FPGA fabric with complete control over the structure and distribution of the PiM logic units. Moreover, the model is compatible with the LiteX framework, which provides a high degree of usability and compatibility with the FPGA and RISC-V ecosystem. Thus, the framework enables architects to easily prototype, emulate and evaluate a wide range of emerging PiM architectures and designs.**

**We demonstrate strategies to model several pioneering bitwise-PiM architectures and provide detailed benchmark performance results that demonstrate the platform's ability to facilitate design space exploration. We observe an emulation vs. simulation weighted-average speedup of 28× when running a memory benchmark workload. The model can utilize 100% BRAM and only 1% FF and LUT of an Alveo U280 FPGA board. The project is entirely open-source.**

***Index Terms*—FPGA Emulation, Fidelity, Simulation Wall, Processing-in-Memory (PiM), High-Bandwidth Memory (HBM).**

## I. Introduction and Motivation

Processing-in-Memory (PiM) is a topic of great interest to computer architects aiming to design systems less affected by the *memory wall* problem that stresses the growing disparity between microprocessor and memory speeds [1]. In addition to enabling access to the high internal-memory bandwidth, processing-in-memory can reduce data movement, utilization and reliance on memory bus bandwidth. Moreover, it can reduce cache pollution, latency, and energy consumption associated with data movement via the memory bus [2], [3].

Consequently, numerous PiM architectures have been developed, classified into a taxonomy based on memory technology, computation location, computation type, and extracted parallelism [4]. To evaluate these architectures, researchers often have had to develop in-house tools or modify and combine capabilities of existing tools, leading to low-fidelity measurement results and an arduous PiM evaluation experience. PiM simulation frameworks [5], [6] formalize this approach of fusing and PiM-augmenting several well-established simulation tools. However, such heterogeneous computing architectures introduce a high level of complexity that is difficult to simulate, leading to limited, low-fidelity, and low-performance tools. Computer systems complexity is known to advance faster than computer simulation performance, a phenomenon known as the *simulation wall* [7]. As a result, better approaches are necessary. Modeling performance is shown to be significantly higher with FPGA-accelerated simulation [8] or approximated FPGA-emulation [9], [10]. An alternative approach to harness an FPGA is to implement a structurally accurate target model - a *prototype*. This approach enables faithful emulation on the FPGA at only an order of magnitude lower clock rate than the original target. Infrastructure for FPGA-emulated processor systems has long been under development [11], and a handful of RISC-V soft-cores are now available for FPGAs [12]–[14]. However, no framework so far enables fast and accurate FPGA-emulation of the detailed aspects of main memories (e.g., shared bus, granular data organization, bank state and timing) to facilitate prototyping of various PiM architectures on different memory standards and technologies. To the best of our knowledge, we are the first to develop such a framework as we propose PiMulator: a soft-memory and soft-PiM infrastructure for PiM prototyping on FPGA boards.

PiMulator enables architects to generate a configurable memory and PiM model. The memory model emulates *components* such as the shared data bus, row buffer, and subarrays, *behaviors* such as latencies and bank states, and thus *operations* such as burst write/read, refresh, bank interleaving. Accounting for both memory structure and behavior facilitates deployment of the desired PiM logic at any desired location. Finally, to demonstrate the capabilities of the PiMulator platform, we present strategies to prototype and emulate pioneering bitwise-PiM architectures such as RowClone [19], LISA [20], and Ambit [21]. These prior works demonstrate high-bandwidth, low-latency, and energy-efficient bulk bitwise PiM operations at the subarray level. Emulating these architectures requires modeling the above-mentioned detailed aspects of the main memory and showcases the strength of our proposed framework. We summarize the contributions of this paper as follows:

- We present a parameterizable, structurally accurate main memory and PiM model implemented in System Verilog, synthesizable on FPGA boards, that allows prototyping of various PiM architectures by injecting PiM logic into reserved placeholders.

978-3-9819263-6-1/DATE22/©2022 EDAA

| Approach | Fidelity | Speed | Underlying Memory Model | Design Space Exploration | Full System Evaluation | Affordability and Adoptability |
|---|---|---|---|---|---|---|
| Verilog Behavioral Simulation [15] | High | Low | Interface, State, Timing, Data flow | Flexible, Behavioral | Compatibility, Corectness | Affordable, Tedious |
| Software Simulation [5], [6] | Low, Medium | Low | Timing | Flexible | OS/Application, Power, Performance | Affordable, Familiar |
| FPGA Accelerated Simulation [7], [8] | Medium | Medium | Timing | Flexible | OS/Application, Power, Performance | Cloud price, Familiar |
| Approximate FPGA Emulation [9], [10] | Low, Limited | High | Interface, Approximate Timing, Data flow | Constrained | OS/Application, Approximate Performance | Platform price, FPGA toolflow |
| FPGA Emulation *PiMulator* | High | High | Interface, State, Timing, Data flow and layout | Full flexibility | OS/App, Hardware Model, Power, Performance, Area | Cloud price, FPGA toolflow, LiteX |
| DRAM use violation [16] | Maximum | Realtime | Physical memory | Constrained | OS/App, Real hardware | Platform price, FPGA toolflow |
| Hardware tape-out [17], [18] | Maximum | Realtime | Physical memory | Constrained | OS/App, Real hardware | Prohibitively Expensive, ASIC & PCB toolflow |

TABLE I: Comparison among PiM system prototyping and evaluation approaches over selected attributes.

- For increased usability, we integrate the memory and PiM model into the LiteX [14] framework, facilitating easy interfacing, system generation, and deployment of the target system on a preferred FPGA board.
- We demonstrate strategies for prototyping and design-space exploration of bitwise-PiM architecture evaluations by exploring various subarray configurations for Ambit [21], including inter-subarray links as described in LISA [20].
- We present memory emulation vs. simulation performance results running a memory micro-benchmark [22]. We explore the main differences between emulation and other means such as simulation or hybrid frameworks in order to bolster the main advantages of emulation, such as high performance and fidelity.
- We demonstrate that our emulation framework is $\sim 28\times$ faster (weighted average) than simulation [23] for memory operations and even higher for PiM architectures.

## II. PiM Prototyping Framework Attributes

In order to support the increase in heterogeneity, complexity, and parallelism that PiM architectures bring to the field, a PiM modeling and evaluation framework must offer additional attributes, as summarized in Table I. In this section, we perform a comparison between different modeling and evaluation approaches over several attributes of interest and make a case for FPGA-based PiM emulation.

### A. Fidelity

High fidelity is critical during PiM architecture modeling for providing convincing results and insight. We define high fidelity microarchitectural modeling as cycle-accurate, cycle-exact, including RTL structural correspondence and hardware signal granularity. These features lead to accurate workload execution modeling on a realistic hardware abstraction and enable hardware-software co-design. Achieving fidelity with software simulation comes at a high cost in performance, motivating framework developers to opt for statistical, analytical, or simplified models. Conversely, FPGA emulation intrinsically delivers high fidelity and speed by harnessing the structural correspondence between the model and the target architecture.

### B. Target vs Model Speed

High performance is crucial because it enables faster evaluation of increasingly complex architectures with heavy workloads of interest. PiM architectures can be emulated and evaluated at high speed by harnessing the spatial, parallel, and reconfigurable nature of FPGAs and board peripherals.

### C. Underlying Memory Model

Modeling and evaluating PiM architectures with high fidelity requires a detailed underlying memory model that accounts for data flow, data layout, latencies, technology process, area and power budget, etc. Compared to software simulation, an FPGA emulation approach can preserve the spatial nature of memories and PiM architectures, thus providing further hardware insights.

### D. Design Space Exploration

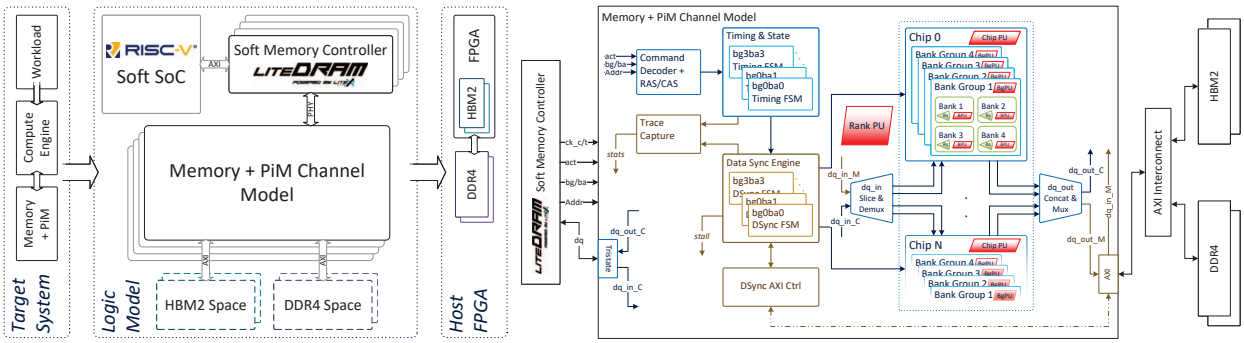
PiM architectures introduce many design variables [4] resulting in the need to perform a complex and vast design space exploration (DSE). In addition to speed and fidelity, a framework must have a high degree of flexibility to accommodate a vast design space. FPGA emulation offers software simulation-like flexibility at hardware-like speed and fidelity.

### E. Full System Evaluation

Ultimately, a good PiM prototyping framework must facilitate PiM architecture evaluation as part of a full system and software stack. The metrics of interest include correctness of operation, application performance, reduction in data migration and cache pollution, energy savings, hardware cost, etc. FPGA emulation outperforms other approaches in measurements and capabilities [10], [24].

### F. Affordability and Adoptability

Finally, to be user-friendly, a framework must be affordable, available, and easy to use. Several cloud providers now offer




(a) (b)

Fig. 1: (a) PiMulator top-level diagram. The *target* system is modeled with open-source IP primitives and synthesized on a *host* FPGA board. Workloads run directly on the hardware configuration at FPGA speed. (b) Memory and PiM emulation model block diagram consisting of memory components (blue), PiM units (red), auxiliary structures (brown), and peripherals.

high-end FPGA boards at competitive prices. With the advancement of high-level synthesis, hardware construction languages, and frameworks such as LiteX [14], FPGA tools are becoming more user-friendly.

In this work, we propose a framework for emulating PiM architecture target systems using highly configurable and FPGA synthesizable soft IPs for processor, memory controller, memory, and PiM, structured as shown in Figure 1a.

## III. Proposed FPGA-based Emulation Model

In this section, we describe the proposed memory and PiM emulation model shown in Figure 1b. The model implements memory and PiM logic structures such as interface, command decoder, controls, data bus and organization, and PiM processing units, emulating associated behaviors such as memory state and latencies on the FPGA fabric. Moreover, we harness FPGA board memory resources (i.e., DRAM, HBM) to expand the emulated memory capacity using a data synchronization engine (DSync). We implement the PiMulator model in System Verilog, allowing for future low-level modifications over the RTL and enabling higher performance than high-level languages.

### A. Interface, Command Decoding and Controls

The memory and PiM model implements a native, configurable DIMM interface compatible with most memory standards. We drive the model with a clock with double the frequency of the interface clock to model the dual data rate (DDR). According to memory standards truth tables, the interface signals are translated into memory commands by the command decoder module, which also implements the Row Address Strobe (RAS) and Column Address Strobe (CAS) control logic. During an activate (ACT) command, the address is recorded as *row* for the indicated bank, keeping track of the active rows on each bank. Similarly, during read or write commands, the address is recorded as *column*, while a burst counter automatically increments, keeping track of the active columns on each bank. Finally, the write or read state of each bank is determined, allowing control of both the local data arrays as well as the *inout* data port and data strobe signals.

### B. Bank State and Timing

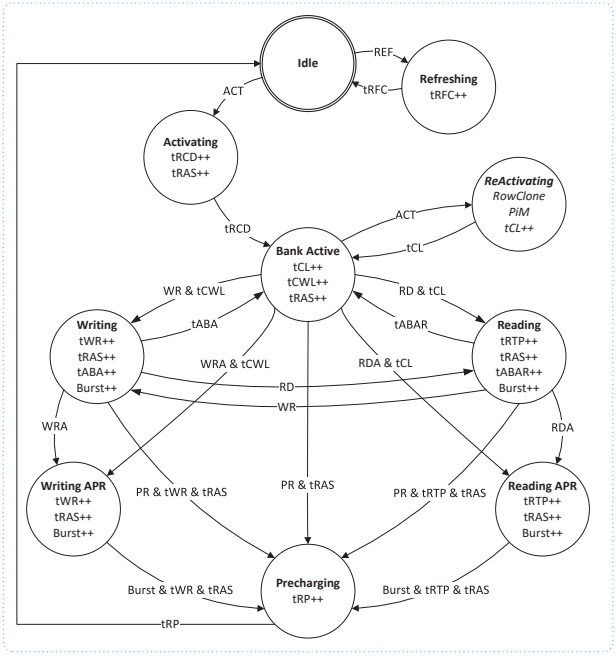
We model the state of each bank and the latencies associated with state transitions using the state machine depicted in Figure 2a. We base the state machine design on common memory standards [25] and model latencies using counters that, together with the decoded commands, condition the associated state transitions. For example, the state machine will switch from *Bank Active* to *Reading* upon receipt of *RD* command and expiry of $tCL$ counter. With this approach, we model the state and latencies of different memory standards and technologies and accommodate additional PiM latencies with minor modifications. We implement the state machine by using FizZim [26], which is an easy-to-use graphical tool for designing state machine RTL modules, thus facilitating ease of development and adaptability of the framework.

### C. Memory hierarchy: Chip, Bank Group, Bank and Subarray

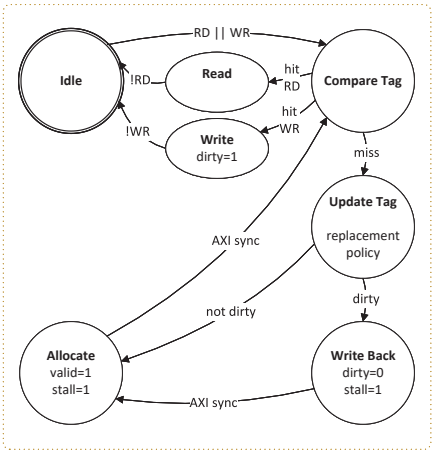
We model memory hierarchy, data flow, and data layout by harnessing the FPGA BRAM resources. The memory interface data bus bits are sliced over the few Chip modules and further de-multiplexed into bank group and bank modules and stored in arrays. Due to the limited BRAM (similarly, LUTRAM) resources on the FPGA, only a tiny part of the total emulated memory capacity fits on the FPGA fabric. Although small, the memory emulated on the FPGA fabric allows modeling several whole rows at each bank module, equivalent to $\approx 0.05\%$ of a DDR4 array. For these rows to act as any rows in the memory array, they are handled by an auxiliary data synchronization engine (DSync), which also models row subarray membership.

### D. Data Synchronization Engine

PiMulator makes use of a data synchronization engine (DSync) module per bank, which handles the bank module rows. The DSync implementation is based on a standard cache design [27] as depicted in Figure 2b, with block size equal to a whole bank module row. We select the block size to be equal to a whole row to facilitate emulating PiM architectures taking




(a) FSM for modeling bank state and timing based on existing standards and counters.




(b) FSM for bank data synchronization engine (DSync) with FPGA board memory resources.

Fig. 2: Emulation model Finite State Machines (FSM).

advantage of data locality by utilizing the entire row [19]–[21], [28].

A tag table keeps track of the status (valid, dirty) of the rows in the bank module, as well as of the subarray number and address of the row data in the board memory resources. Upon a read or write, the row address is compared with values in the tag table. If present and valid (a hit), the bank module row index is returned for use, and no stall signal is issued. In case of a miss, a bank module row is selected for update by the replacement policy. Users can choose between FIFO, random, or implement their desired replacement policy. If the selected row is dirty, its data is written to the board memory prior to fetching the data of the new row. A controller (DSync AXI Ctrl) links the data synchronization engines with board memory resources such as HBM and DDR using a fixed addressing scheme that maximizes bandwidth utilization. A stall signal has to be issued during the *Write Back* and *Allocate* states in order to pause all modeled activity (processor, controller, memory, and PiM) and accurately preserve the emulated time ticks. Thus, a full-sized memory array is modeled with FPGA BRAM and board memory resources, with the maximum emulated capacity being determined by the latter.

### *E. Templates for PiM logic*

PiM modules that can easily be customized with any logic are included inside the bank, bank group, chip modules, and at the rank level. They are connected to the local data bus and can access the bank module(s) they encompass. After executing the logic, the results are written back. The control signals are passed from the top PiMulator module. We opted for such a distributed PiM model to facilitate emulation of PiM architectures of different topologies such as [28]–[30].

### *F. Top PiMulator Module*

The top module implements the DIMM interface, including the data and strobe tri-state logic, and the AXI interfaces to the board memory resources. Moreover, it instantiates and wires all the modules described above. The configuration parameters are passed top-down, resulting in a modular, parameterizable model for memory and processing in memory emulation on FPGAs. The model supports hardware signal observability and real-time statistics collection such as the number of memory and PiM operations, bank state dynamics, DSync hitrate, etc. The model can be interfaced with a processor system or PCIe DMA via a memory controller. We verified the correct operation of multiple model configurations under possible usage scenarios such as sequential and random burst read/write and bank interleaving.

### *G. Integration with LiteX*

We wrap the top PiMulator System Verilog module in LiteX [14] and interface it with the LiteDRAM memory controller and VexRiscV processor system by defining a *target* script. Moreover, we implement the *platform* definition for the Xilinx Alveo U280 FPGA board and test operation using the provided utilities and standalone applications. The LiteX framework provides numerous open-source IPs, supports several soft cores and 60+ FPGA boards from different vendors. Moreover, it provides utilities for development, build, SoC generation, communication, application compilation, and deployment. We believe these features benefit the PiMulator framework in terms of future full-system and full-stack development and facilitate user adoption.

## IV. Strategies for Emulating Bitwise-PiM

In this section we describe how we can leverage the model entities described above to emulate bitwise-PiM architectures.
**Emulating RowClone:** RowClone [19] describes two methods

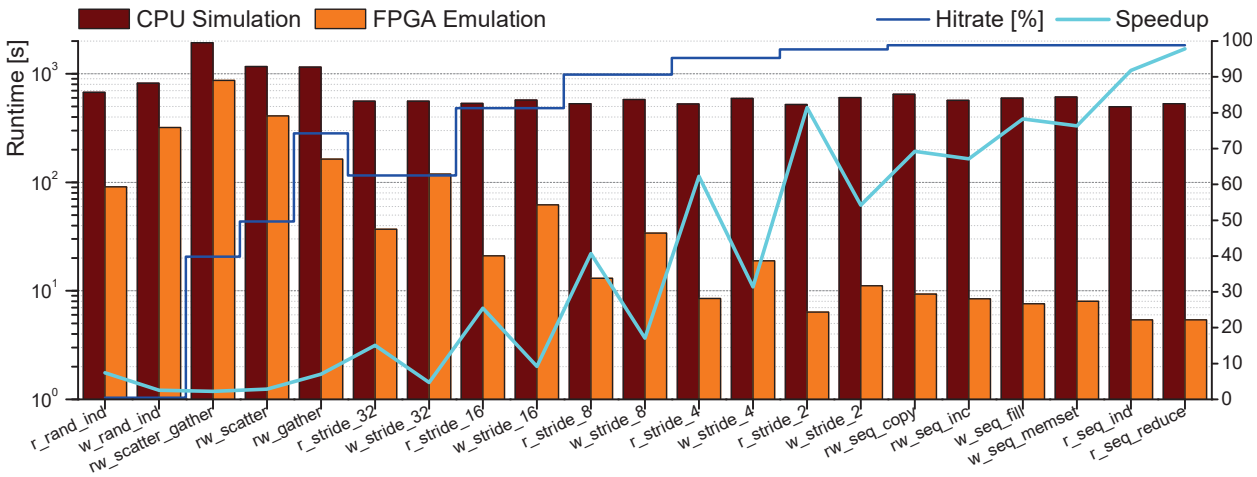



Fig. 3: Runtimes for CPU Simulation and FPGA Emulation of a DDR4 memory system stressed for 1s of real *target* time with different access patterns; ratio between the two runtimes and the corresponding DSync hitrate.

of cloning data locally in memory. In *Fast Parallel Mode (FPM)*, data is read from a source row into the row buffer, then written back to a destination row within the same subarray with the help of a second *Activate* command. In *Pipelined Serial Mode (PSM)*, data is copied from a source row in a bank to a destination row in a different bank via the shared global bus in a pipelined manner. To emulate RowClone-FMP, we augment the DSync tag table to support linking multiple memory rows to one local row, while also accounting for subarray membership. Additionally, we model the bank state and timing associated with the second activate command by dedicating an extra *ReActivating* state, also shown in Figure 2a. Likewise, for RowClone-PSM, we configure the DIMM interface and memory bus to allow data flow between banks.

**Emulating LISA:** LISA [20] further improves RowClone-FPM by enabling fast inter-subarray data cloning with the help of isolation transistors that link the bit lines of neighboring subarrays. Emulating LISA involves modeling neighbor subarray membership on top of the approach to emulate RowClone-FPM, effectively emulating a line network between the subarrays. The same approach can be used to emulate other inter-subarray network topologies, furthering design space exploration.

**Emulating Ambit:** Ambit [21] describes a DRAM subarray design that facilitates bulk AND-OR-NOT operations. First, it utilizes RowClone-FPM to copy the operand data to dedicated subarray rows. Next, triggering a *triple-row activation* computes bitwise AND-OR via a majority function. Similarly, a dedicated word line connected to the negated logical value in the sense amplifier is activated for NOT operation. Finally, it returns the result with a last RowClone-FPM operation. We model the dedicated Ambit subarray rows and the logic operations inside the Bank Processing Unit module using LUTRAM and LUTs while timing is modeled with three repeated activations. We explore several Ambit subarray designs and test these strategies for emulating state, timing, sub-array, data flow, and bitwise-PiM with long test vectors.

## V. Evaluation

We evaluate PiMulator runtime under different memory access patterns using the Hopscotch [22] benchmark suite and compare it with simulation runtime. Hopscotch consists of data-intensive kernels that issue read-only, write-only, and mixed traffic with different access patterns (e.g., sequential, random, strided, tiled, and a combination of these), making it easier to pinpoint the performance bottlenecks of a system. For simulation, we first run the benchmark workloads for an *allowed runtime* of 1s in gem5 [31] under a target system consisting of a 3GHz X86 CPU, 64kB L1 cache, and 8GB DDR4_2400_16x4 memory. We extract the memory access traces and convert them into DRAMsim3 [23] trace format. Finally, we measure execution time for each workload traces in DRAMsim3 on a machine with two 8 core Intel Xeon Silver 4208 2.1GHz processors and 256GB DDR4 memory running Ubuntu 18.04.5 LTS. This runtime accurately measures CPU memory simulation time. For emulation, we feed the memory access traces into an 8GB DDR4_2400_16x4 PiMulator memory model and capture the total number of clock cycles. The total FPGA runtime is evaluated with the clock frequency of 250MHz, extendable up to 333MHz. The measured results are presented in Figure 3. As expected, all kernels completed significantly faster in emulation, registering a speedup of 28× (weighted average over the number of memory accesses). The results also indicate that emulation speed is very high for kernels with high data locality and can be increased by using better fitting DSync replacement policy algorithms that achieve higher hitrates.

We evaluate a theoretical Ambit+LISA [20], [21] array configuration consisting of variable-sized inter-connected subarrays with several configurations for the dedicated PiM rows, in which all subarrays can compute at once. We stress the model with large input vectors on which repeatedly compute AND, OR, NOT, XOR and NAND at the subarrays, and measure

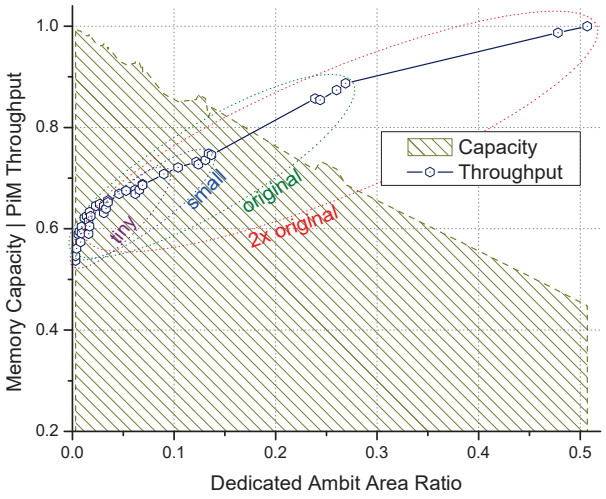



Fig. 4: Pareto curve for Ambit design space exploration.

throughput for each configuration. The results obtained are shown in Figure 4. As expected, throughput increases with the number of dedicated Ambit rows. Moreover, as the number of dedicated Ambit rows starts to dominate the memory space, the throughput increase slows due to the need to clone the operands from neighboring subarrays. This showcases how PiMulator can be used to emulate and evaluate multiple variations of a PiM architecture.

We synthesize the memory + PiM model with only a minimal PiM configuration on an Alveo U280 FPGA board and achieve 100% BRAM and under 1% FF and LUT utilization. The unused resources are to be utilized for PiM emulation and the remaining full system components.

## VI. Conclusion & Future Work

We present PiMulator - an open-source emulation framework for Processing-in-Memory. PiMulator enables fast, flexible, and high fidelity prototyping and evaluation of PiM architectures. With an average speedup of 28×, users can emulate architectures of increased complexity and evaluate them with heavy workloads of interest. Moreover, the modular, parameterizable design and trace collection facilitate design space exploration. Finally, the efficient resource utilization and integration with the LiteX framework result in high usability and availability.

PiMulator is actively developed, with future features such as improved DSync, support for more popular PiM architectures, and full system/stack emulation coming soon.
PiMulator is available at https://github.com/hplp/PiMulator.

## VII. Acknowledgement

We would like to thank Ameen Akel and Sean Eilert from Micron Technology for their valuable and helpful input. This work was supported in part by Semiconductor Research Corporation (SRC), and in part by the NSF I/UCRC on Multi-functional Integrated System Technology (MIST) Center; IIP-1439644, IIP-1439680, IIP-1738752, IIP-1939009, IIP-1939050, and IIP-1939012.

## References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH*, vol. 23, no. 1, pp. 20–24, 1995.
[2] O. Mutlu *et al.*, "A modern primer on processing in memory," *arXiv preprint arXiv:2012.03112*, 2020.
[3] T. Finkbeiner *et al.*, "In-memory Intelligence," *IEEE Micro*, vol. 37, no. 4, pp. 30–38, 2017.
[4] H. A. D. Nguyen *et al.*, "A classification of memory-centric computing," *ACM JETC*, vol. 16, no. 2, pp. 1–26, 2020.
[5] S. Xu *et al.*, "PIMSim: A flexible and detailed processing-in-memory simulator," *IEEE CAL*, vol. 18, no. 1, pp. 6–9, 2018.
[6] C. Yu *et al.*, "MultiPIM: A Detailed and Configurable Multi-Stack Processing-In-Memory Simulator," *IEEE CAL*, vol. 20, no. 1, pp. 54–57, 2021.
[7] H. Angepat *et al.*, "FPGA-accelerated simulation of computer systems," *Synthesis Lectures on Comp. Arch.*, vol. 9, no. 2, pp. 1–80, 2014.
[8] D. Biancolin *et al.*, "FASED: FPGA-accelerated simulation and evaluation of DRAM," in *2019 ACM/SIGDA ISFPGA*, 2019, pp. 330–339.
[9] A. K. Jain *et al.*, "Microscope on Memory: MPSoC-Enabled Computer Memory System Assessments," in *FCCM*. IEEE, 2018, pp. 173–180.
[10] J. Zhang *et al.*, "MEG: A RISCV-based System Emulation Infrastructure for Near-data Processing Using FPGAs and High-bandwidth Memory," *ACM TRETS*, vol. 13, no. 4, pp. 1–24, 2020.
[11] J. Wawrzynek *et al.*, "RAMP: Research accelerator for multiple processors," *IEEE micro*, vol. 27, no. 2, pp. 46–57, 2007.
[12] K. Asanovic *et al.*, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," University of California at Berkeley United States, Tech. Rep., 2015.
[13] C. Papon and SpinalHDL, "VexRISC-V: An FPGA-friendly 32-bit RISC-V CPU implementation." [Online]. Available: https://github.com/SpinalHDL/VexRiscv
[14] F. Kermarrec *et al.*, "LiteX: an open-source SoC builder and library based on Migen Python DSL," in *OSDA/DATE*, 2019.
[15] Micron, "DDR SDRAM Verilog Simulation Models." [Online]. Available: https://www.micron.com/products/dram
[16] A. Olgun *et al.*, "PiDRAM: A Holistic End-to-end FPGA-based Framework for Processing-in-DRAM," *arXiv preprint arXiv:2111.00082*, 2021.
[17] M. Gokhale *et al.*, "Processing in memory: The Terasys massively parallel PIM array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.
[18] Y. Kang *et al.*, "FlexRAM: Toward an advanced intelligent memory system," in *ICCD*. IEEE, 1999, pp. 192–201.
[19] V. Seshadri *et al.*, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *IEEE/ACM MICRO*, 2013, pp. 185–197.
[20] K. K. Chang *et al.*, "Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM," in *HPCA*. IEEE, 2016, pp. 568–580.
[21] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *MICRO*. IEEE, 2017, pp. 273–287.
[22] A. Ahmed and K. Skadron, "Hopscotch: A micro-benchmark suite for memory performance evaluation," in *MEMSYS*, 2019, pp. 167–172.
[23] S. Li *et al.*, "DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator," *IEEE CAL*, vol. 19, no. 2, pp. 106–109, 2020.
[24] S. Karandikar *et al.*, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *ISCA*. IEEE, 2018, pp. 29–42.
[25] J. S. S. T. Association *et al.*, "JEDEC Standard: DDR4 SDRAM," *JESD79-4, Sep*, 2012.
[26] P. Zimmer *et al.*, "FizZim–an open-source FSM design environment," *Enterprise Information Systems*, vol. 9, no. 5-6, pp. 528–555, 2014.
[27] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.
[28] S. Li *et al.*, "Drisa: A dram-based reconfigurable in-situ accelerator," in *MICRO*. IEEE, 2017, pp. 288–301.
[29] M. Lenjani *et al.*, "Fulcrum: a simplified control and access mechanism toward flexible and practical in-situ accelerators," in *HPCA*. IEEE, 2020, pp. 556–569.
[30] F. Devaux, "The true processing in memory accelerator," in *HCS*. IEEE Computer Society, 2019, pp. 1–24.
[31] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.