PiMulator: A Processing-in-Memory Emulation Platform

Dissertation

presented to the faculty of the

University of Virginia School of Engineering and Applied Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

by

Sergiu Mosanu

May, 2023

sm7ed@virginia.edu

PiMulator: A Processing-in-Memory Emulation Platform

A

Dissertation

Presented to
the faculty of the School of Engineering and Applied Science
University of Virginia

in partial fulfillment
of the requirements for the degree

Doctor of Philosophy

by

Sergiu Mosanu

August 2023

APPROVAL SHEET

This

Dissertation

is submitted in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Author: Sergiu Mosanu

This Dissertation has been read and approved by the examing committee:

Advisor:	Mircea R. Stan
Advisor:	
Committee Member:	Kevin Skadron
Committee Member:	Samira Khan
Committee Member:	Barry W. Johnson
Committee Member:	Steven M. Bowers
Committee Member:	Ted Way
Committee Member:	

Accepted for the School of Engineering and Applied Science:

[illegible]

Jennifer L. West, School of Engineering and Applied Science

August 2023

To my family and friends. ♡

Abstract

Motivated by the memory wall problem, researchers propose numerous Processing-in-Memory (PiM) architectures to bring computation closer to data. Evaluating the performance of these emerging architectures is challenging due to the lack of tools that accurately mimic both software and hardware aspects. This thesis presents PiMulator, an open-source platform for system-level PiM emulation, suitable for rapid prototyping and evaluation of PiM architectures.

At its core, PiMulator incorporates MEMulator, a main-memory emulation model implemented in SystemVerilog, enabling users to generate any desired memory configuration on the FPGA fabric with complete control over the PiM logic units. Furthermore, we develop and implement the FreezeTime mechanism, effectively extending the emulated memory capacity by synchronizing the limited FPGA chip memory resources with the board's DDR4 and HBM2 resources. This approach offers flexibility in modeling memory and logic behavior without compromising emulated time accuracy.

The platform integrates the Memory+PiM model into the LiteX framework, ensuring compatibility with a robust FPGA and RISC-V ecosystem. We adopt a system emulation approach that combines CPUs, memory controllers, memories, interconnect, and peripherals using soft cores synthesizable on FPGA boards. This enables architects to prototype, emulate, and evaluate various PiM architectures and designs at the system level. PiMulator facilitates high-speed and high-fidelity modeling and evaluation of emerging memory and PiM architectures with workloads of interest, utilizing soft cores synthesizable on FPGA boards.

This thesis demonstrates strategies to model several pioneering PiM architectures and provides detailed benchmark performance results, showcasing the platform's ability to facilitate design space exploration.

Acknowledgments

This dissertation signifies the fruitful culmination of several invaluable collaborations and serves as a pivotal stepping stone into the next phase of my life. I am overwhelmed with gratitude for the numerous forms of support I've received from the academic and industry communities, as well as from family and friends. These collective efforts have profoundly and positively impacted both me and my work. I seize this moment to express my profound appreciation and reciprocate with a solemn pledge of future support.

First, I extend my heartfelt gratitude to my advisor, Professor Mircea R. Stan, who warmly welcomed me to UVA and firmly believed in my capabilities, guiding me through intriguing research projects. His adept navigation toward our ultimate goals and insistence on adopting a top-down approach ensured that we worked on essential aspects that matter and will lead to success. Beyond being an extraordinary research advisor, he has provided me with invaluable life and career guidance, helping me evolve into a more competent scientist, engineer, instructor, researcher, and individual. I offer my earnest thanks, dear Mircea, for your unwavering support and patient guidance. And I sincerely apologize for the instances when I didn't quite meet expectations.

Next, I express my deep gratitude to Professor Kevin Skadron, who has been akin to a mentor in my journey. His participation in most of our work meetings was instrumental in shaping our innovative research solutions, while his critical feedback tremendously improved the quality of our published papers. His help fostering networking opportunities and establishing external collaborations was invaluable, for which I remain eternally grateful. In the same spirit, I convey my sincere appreciation to my doctoral dissertation advising committee members - Professors Samira Khan, Barry W. Johnson, Steven M. Bowers, and Dr. Ted Way. Their respective contributions have profoundly enriched my academic experience and enabled the successful completion of my doctoral journey.

I have been extraordinarily fortunate to collaborate closely with many esteemed industry researchers who have generously shared their wealth of knowledge. My heartfelt thanks go to the Micron Automata team and the vibrant summer internships at Microsoft Research and at Microsoft Azure ML, all of which played critical roles in honing my technical skills and expanding my research horizons. The individual contributions of Gavin Huggins, Nicholas Terrell, Paul Dlugosch, Indranil Roy, Matt Grimm, Trishul Chilimbi, Blake Pelton, Daniel Lo, Ted Way, Charles Fu, Valentin Churavy, and Doug Burger not only positively influenced the success of the projects we undertook but also significantly fostered my professional growth.

A vital part of my academic journey has been the support from the UVA ECE and CPE departments, whose contributions are indispensable. My deepest gratitude goes to Beth A. Eastwook-Beatty, Jayne Weber, Natalie Ann Edwards, and Terry B. Tigner for their steadfast support and understanding throughout the academic process. My thanks also extend to Gary Li for his technical expertise and assistance with our IT infrastructure needs. In addition, I am deeply thankful to all the professors who imparted their wisdom and supported me during my stints as TA and co-instructor.

The invaluable contributions of my peers have been an integral part of this journey. Our camaraderie, insightful debates, shared advice, and mutual growth significantly enriched both my graduate student life and the quality of research leading to this dissertation and our numerous published papers. I extend my profound respect and admiration to all individuals from the High-Performance Low-Power (HPLP) Laboratory — past, present, and future members, and the many other collaborators. In particular, I wish to acknowledge: Mohammad Nazmus Sakib, Yimin Gao, Khyati Kiyawat, Tommy Tracy II, Vaibhav Verma, Xinfei Guo, Ted Xie, Arijit Banerjee, Luisa Patricia Gonzalez Guerrero, Mohamed El-Hadedy, Nathan Brunelle, Alif Ahmed, Alec Roelke, Mandi Das, Ben Melton, Pai Wang, Mateja Putic, Robert (Trey) West, Yunfei Gu, Rahul Sreekumar, Rocio Elisa Pantoja Rodriguez, Muhammed Ceylan Morgul, Jun-han Han, Jay Sheth, Vinay Iyer, Jesse Moody, Robert Costanzo, Pouyan Bassirian, Karina Torres Castro, Henry Bishop, and others whose names I may have unintentionally missed. Your collective insights, dedication, and friendship have immeasurably shaped my work and personal growth. Thank you all!

My family has unquestionably been the cornerstone of my strength and success. I am incredibly grateful to my wife, Rocio Elisa, whose steadfast love and support have guided me throughout this journey. To my parents and brother, your unwavering backing and the pride you take in my academic endeavors have fueled my passion. My little son, Marcus, from the moment I felt the flutter of your first heartbeats to the present day, filled with your heartwarming smiles, has been a constant source of motivation and joy. I am deeply grateful to my extended family, uncles, and cousins, who have helped me feel at home and offered unwavering support. Each of you has played a pivotal role in my academic journey, and I am infinitely thankful for this.

Lastly, I want to acknowledge the generous funding support from various organizations, including the Semiconductor Research Corporation (SRC), Multi-functional Integrated System Technology (MIST), DARPA under the PERFECT program, and the CRISP JUMP center. Their financial backing was instrumental in bringing my research aspirations to fruition.

In closing, I am profoundly grateful for all the experiences, opportunities, and support that I have been fortunate to receive during my doctoral journey. This dissertation is a testament to the contributions of all those acknowledged here and many more who have remained unnamed. From the bottom of my heart, thank you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

1.1 Thesis Statement

Comprehensive FPGA-based memory emulation enables fast, high-fidelity design-space exploration and evaluation of processing-in-memory architectures as part of a whole system stressed with heavy workloads.

1.1.1 Elaboration

The thesis statement encapsulates key challenges and discoveries addressed in this work, which are expanded upon below.

- "Comprehensive" implies the detailed construction methodology encompassing every memory architecture aspect for PiM kernel modeling and system-level evaluation. This includes the interface, data bus, data layout, memory bank states and latencies, topology, organization, and pertinent physical effects.
- "FPGA-based memory emulation" indicates the development and use of an FPGA-synthesizable soft memory model using SystemVerilog, facilitating memory system emulation via FPGA

reconfigurable hardware resources. This represents a departure from traditional approaches, where other system components were emulated on the FPGA while the memory was used as-is, simulated, or approximated with delay elements and clock frequency scaling.

- "Enables fast, high-fidelity" signifies that the emulation leverages FPGA capabilities, allowing the memory model to operate at hardware speed, with numerous behavioral aspects running in parallel. Furthermore, this approach ensures high accuracy as functionality, behavior, and performance correspond to RTL structural and signal representations.
- "Design-space exploration and evaluation" suggests that the memory model's flexibility supports various design variations, enabling rapid evaluation of distinct architectural choices during the design process.
- "Of processing-in-memory architectures" indicates the memory model's compatibility to incorporate a broad spectrum of PIM architectures.
- "As part of a whole system" emphasizes the integration of the memory and PIM model within a comprehensive system, including processors, cache hierarchy, memory controllers, buses, and peripherals. These soft modules, implemented on the FPGA, form the logic representation of the whole target architecture, facilitating system-level evaluation of the studied PiM architectures.
- Lastly, "stressed with heavy workloads" underscores the ability to execute real, compute-intensive benchmarks on the emulated system. Exploiting the emulation's speed and level of detail, large-scale applications, which previously required days, can now be completed in mere hours, offering valuable performance insights under realistic conditions.

This thesis presents a comprehensive FPGA-based emulation framework for emerging memory technologies and Processing-in-Memory (PiM) architectures. Chapter 4 introduces MEMulator, a <u>m</u>emory <u>e</u>mulation <u>m</u>odel (MEM) seamlessly integrated into an entire computing system. To address the challenge of emulating large memory capacities on FPGAs, Chapter 5 presents the

FreezeTime technique, a solution for modeling virtual time and enabling broader architectural virtualization. Finally, Chapter 6 unveils PiMulator, detailing the implementation, modeling, and evaluation of PiM kernels and their associated logic behavior. These components - MEMulator, FreezeTime, and PiMulator - form the foundation of our proposed emulation framework.

1.2 Advancing Memory Modeling

Main memory is a critical SoC and system design component that directly affects performance, power, and cost. There are various types of memory, such as DDR, LPDDR, GDDR, HBM, HMC, Wide I/O, and NVMM, each optimized for select core metrics, such as capacity, bandwidth, and efficiency, to meet the requirements of different computing platforms. Computer architects can maximize system performance by selecting the appropriate memory architecture and optimizing the memory access method (ex. compression, clustering, deduplication, tiering [1], disaggregation, processing-in-memory [2]).

1.2.1 Prototyping Memory Architectures

However, as architectures become increasingly specialized and complex, such as Google's TPU [3], AMD's EPYC Rome, or Intel's Cascade Lake SP [4], designing a memory system that best meets the performance requirements is a daunting task. To validate promising analytical results, it is necessary to experiment with various "what-if" scenarios for the internal memory architecture and analyze the effects of different internal memory components' arrangements on overall system performance. These experiments must be performed fast and with high accuracy/fidelity, thus facilitating iterative improvement toward a design that optimizes the desired metrics, such as performance per watt or cost. The first goal of this work was to develop an open emulation model that enables memory architects to create new memory standards or more intelligent memories by prototyping new ideas or porting innovations from one memory type to another. For example, new techniques such as bank grouping and channel splitting were first introduced with GDDR and

HBM and were later adopted to DDR4 and DDR5.

1.2.2 Modeling Memory Systems, Co-design

Optimizing system performance requires comprehensive modeling and evaluation of systems with different memory architectures or hybrid heterogeneous memory hierarchies. Researchers should have the ability to easily switch between different memory types or create hybrid heterogeneous memory hierarchies in order to optimize system performance for various use cases by combining the strengths of different memory types. Recent trends in high-performance computing point to the increasing adoption of hybrid computing systems. For example, the compute express link (CXL) allows the host CPU to access additional persistent memory in conjunction with the DRAM memory, thereby improving the performance of high-capacity workloads of interest [5]. Likewise, the recently introduced AMD Instinct MI300 [6] combines 13 chiplets to create a single chip featuring a CPU, a GPU, and HBM3. This design allows the CPU and GPU to operate simultaneously on the same data in memory using a zero-copy mechanism, which leads to power savings, improved performance, and easier programming. Furthermore, the AMD Versal family of devices combines CPU, RPU, PL (LUTs, LUTRAM, FFs, BRAM, URAM), DSP, AIE, PCIe, DDR and HBM and more interfaces linked with a NoC, a heterogeneous platform for efficient high-performance computing [7].

Inspired by the above-mentioned developments, the second goal of this work was to develop a fast modeling framework that facilitates design and analysis from a system-level perspective. We envision a framework that enables architects to run applications of interest and observe the impact of localized design decisions at a module level on overall system performance. Therefore, researchers can optimize performance in a holistic approach by co-designing multiple system modules, such as the CPU, GPU, and memory in ensemble. By considering the interdependencies between these modules, co-design enables a more comprehensive optimization of the hybrid system. In this thesis, we present a flexible implementation of a whole system utilizing soft IP modules synthesized on an FPGA board, capable of running real workloads at hardware speed. This thesis describes a new, parameterized implementation of a memory emulation model in System Verilog that can be configured to model many desired memory types and integrate it into a whole compute system.

1.2.3 Breaking the Simulation Wall

Simulation frameworks play a crucial role in research for evaluating memory and computer architectures. However, they struggle to achieve both speed and fidelity when modeling complex architectures with heavy workloads and large datasets [8]. The increasing complexity of computer architectures tends to outpace the performance improvements of computer simulations, leading to a phenomenon known as the ”simulation wall” [9]. Consequently, a comprehensive evaluation of a full system can take weeks or months, rendering it impractical for iterative improvements.

In response to these challenges, emerging memory systems and processing-in-memory (PiM) architectures have turned to FPGA-based emulation as an effective alternative. PiM architectures encompass a wide range of kernels, such as CPU-like, FPGA-like, DPU-like, and ASIC-like, and can be placed at various levels, including the subarray, bank, bank group, chip, rank, or DIMM. This diversity in configurations, coupled with concurrent activity and communication between kernels, contributes to the significant complexity of PiM architectures.

FPGA-based emulation offers several advantages over simulation on a CPU or FPGA-accelerated simulation. By constructing memory systems using FPGA resources, emulation provides higher fidelity and evaluation speeds, as well as increased system transparency. This approach is well-suited for handling the intricate designs and high event rates of PiM architectures, allowing researchers to implement and test their designs directly on FPGA hardware as opposed to relying on slower, less accurate simulations [10–14].

In summary, the utilization of FPGA-based emulation offers a promising solution for overcoming the limitations of traditional simulation methods in the face of increasing architectural complexity. By harnessing the power and flexibility of FPGAs, researchers can efficiently explore and validate advanced memory system designs and PiM architectures, paving the way for further innovation and performance optimization.

1.2.4 Enablers of FPGA-based Architecture Emulation

Several factors make FPGAs suitable for architecture modeling and evaluation. Developing IP for FPGAs became more accessible, productive, and agile following the advancement and adoption of high-level synthesis and hardware construction languages [15–17]. In addition, the emergence of system generation and runtime management tools have facilitated reliable, reproducible, flexible, and automated integration of said IP. The levels of abstraction in these tools further enhanced compatibility with various hardware backends [10, 18]. Next, riding Moore's Law, technology scaling and combining several chiplets on an interposer led to large FPGAs packing plentiful quantities of FFs, LUTs, DSPs, BRAM, UltraRAM, and other resources in a single package. Moreover, with the increased demand for reconfigurable high-performance computing, high-end FPGA accelerator cards featuring peripherals such as PCIe interface and DDR4 / HBM2 memory became readily and economically available in the cloud. Hence, architects seek to fit larger architectures on a *single* high-end FPGA board and evaluate them even on a limited budget. Finally, state-of-the-art emulation frameworks [10–12, 19] demonstrate the feasibility of modeling aspects of large, high-performance computing architectures using miniaturized systems implemented on an FPGA board, running only an order of magnitude slower while preserving fidelity, accuracy, and level of insight. Therefore, researchers can conveniently assemble systems using existing or newly developed prototype IPs, host the logic on a high-end FPGA board, and use this infrastructure to efficiently run heavy workloads of interest.

FPGA-based emulation, utilizing soft-core prototypes, serves as a valuable approach for validating simulation results with the high fidelity of a hardware system, while extracting crucial insights related to performance, hardware cost, power, and area. An apt analogy can be drawn between FPGA-based emulation and wind tunnel testing, commonly used to validate the aerodynamic properties of structures, vehicles, and aircraft. Just as wind tunnels employ miniatures or actual objects to emulate environmental factors such as wind, tire rotation, and road behavior (using a large treadmill), FPGA-based emulation leverages LUTs, FFs, and BRAM to construct intricate computer architectures, including CPUs, memory systems, and PiM kernels. By running workloads such as operating systems and compiled applications on this emulation platform, researchers can assess the system's performance at hardware speed and with the fidelity of the final target architecture. This wind tunnel-like approach allows for the comprehensive evaluation of computer architectures, ensuring their efficacy and reliability before implementation in real-world scenarios.

1.3 Architectural Virtualization with *FreezeTime*

Computer architects and technology developers employ simulation and emulation tools to model, evaluate and thus guide the development of future technologies and architectures. These tools allow users to estimate the cost and performance benefits of new ideas and different architectural topologies [20]. Ideally, the simulation tools will accurately and efficiently evaluate complex architectures and provide macro/micro-architectural insights. However, as computer architecture complexity increases, computer simulation becomes prohibitively slow, leading to accuracy trade-offs [9]. Modeling and evaluating multiple iterations of large, heterogeneous architectures with heavy workloads of interest is a high-performance computing (HPC) problem. Previous research demonstrates that FPGA-accelerated simulation, emulation, and prototyping, using realistic models, are capable of delivering high evaluation speed, fidelity, and system transparency [10, 21, 22].

1.3.1 Challenges with Design Mapping on FPGA

The capacity or capabilities of high-end FPGAs still limit the emulated design size and complexity. The emulated systems may consist of larger and faster memory hierarchies than on the FPGA board, such as large caches and high-bandwidth/capacity main memory. Similarly, they may consist of numerous heterogeneous computation instances, such as CPU, GPU, AI/ML processing units, and accelerator cores, which do not fit even when miniaturized and on the largest of FPGAs. Finally, it can prove impossible to map and model all the target system communication infrastructure, such as the buses and the network between the many modules. As we develop system-wide emulation IP models, for example the memory and PiM emulation models described in this thesis, and utilize them to evaluate emerging heterogeneous architectures, we are facing the above-mentioned challenges. *FreezeTime* is a technique we adopted to overcome these challenges.

In the MEMulator and PiMulator frameworks, one of the key aspects being modeled is the memory device's data layout. Considering that Field-Programmable Gate Arrays (FPGAs) possess significantly fewer memory resources compared to Dynamic Random-Access Memory (DRAM) modules (approximately 0.05%), a mechanism is required to emulate the presence of a larger data capacity. The adopted solution utilizes the FPGA chip's Block RAM (BRAM) and UltraRAM resources to represent a subset of memory rows per bank, in conjunction with a data synchronization engine that ensures data coherence between these rows and the board memory resources. Although the synchronization process introduces additional delays that may impact the modeled time, the subsequent section delineates the methodology employed to conceal these induced time intervals.

1.3.2 Remedy - Synchronized Architectural Virtualization

To address these challenges, we propose *FreezeTime*: a coarse architectural virtualization approach facilitating cycle-accurate emulation of large systems on high-end FPGA boards. The model logic has to emulate the behavior of numerous compute instances or other taxing aspects of a system that can not fit on the FPGA (e.g., memory size and speed). We present a virtualized serial execution approach that leverages a subset of the architecture mapped on the FPGA to account for the whole system's behavior using auxiliary mechanisms, blocks of clock cycles, and stalls. For example, to emulate large scratchpad memories on an FPGA with insufficient BRAM, the model can synchronize data between small BRAM pools and board DDR memory, which will incur additional latency. Similarly, the workload on multiple parallel compute instances can be modeled by queuing the operand data and directing it to a subset of compute instances. The FreezeTime technique is to freeze a whole emulated module's activity and state during the additional time intervals necessary for the action on the virtualized modules to elapse. Consequently, the model logic accounts for the activity and cycle-accurate timing of *all* architectural modules with only the subset of instances that fit on the FPGA board, effectively *virtualizing* aspects of the system.

1.4 Modeling Processing-in-Memory Architectures

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 [23]. In addition to enabling access to the high internal-memory bandwidth, bringing computation closer to memory can reduce data movement, thus reducing 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 [24, 25].

1.4.1 Drivers of PiM Architecture Research

Numerous applications can greatly benefit from Processing-in-Memory (PiM) architectures, thanks to their unique advantages. Firstly, applications with low computational intensity (low computation per datum) and memory-bound characteristics can leverage the high in-memory bandwidth and computational parallelism provided by PiM architectures. Secondly, PiM is particularly well-suited for power-efficient computing, as data migration from memory to the processing unit consumes significantly more power compared to the operation itself, while in-memory computation offers improved performance per Watt efficiency [26].

A diverse array of application domains can reap the advantages of PiM architectures, encompassing bioinformatics, image and video analytics, encryption and compression, as well as general data manipulation. Furthermore, high-performance computing applications such as databases, search, graph applications, and machine learning stand to benefit. Given the ever-increasing demand for computational capacity, speed, performance, and data volume, PiM architectures are set to play

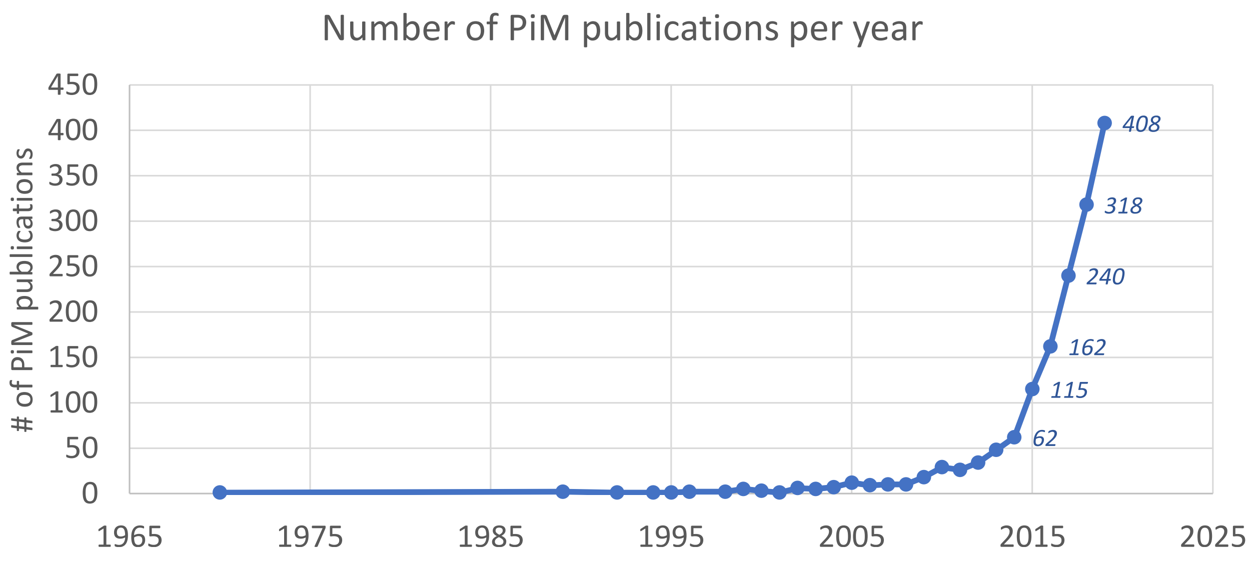


Figure 1.1: PiM-related publications per year suggesting exponential increase in interest.

a pivotal role in augmenting processing units, ultimately facilitating higher throughput, reduced latency, and superior energy efficiency.

1.4.2 Interest in PiM Architecture Research

Consequently, numerous PiM architectures have been developed, classified into a taxonomy based on memory technology, computation location, computation type, and extracted parallelism [27]. The dramatic increase in PiM-related research publications over the past decade, as depicted in Fig. 1.1, demonstrates a significant growth in interest within the scientific community [28]. The concept of PiM was first introduced by Harold Stone in 1970 through his paper titled “A logic-in-memory computer” [29]. This topic remained relatively dormant for approximately four decades, with only a few publications annually. However, around 2009-2010, the research interest in PiM experienced a remarkable surge. The number of annual publications grew exponentially, reaching up to 408 papers in 2019. This rapid acceleration in PiM research can be attributed to various factors, such as advancements in technology, the increasing demand for high-performance and energy-efficient computing, and the emergence of new application areas. As a result, there has been a notable shift in the landscape of PiM research, with several products already available in the market. The exponential growth trend, as shown in Fig. 1.1, signifies the potential impact of PiM technology on future computing systems, warranting further investigation and development in this domain. Moreover, this trend highlights the necessity for a unified, fast, flexible, and high-fidelity modeling and evaluation framework that can serve as a baseline methodology for the evaluation and continued development of PiM architectures. Establishing such a framework will enable researchers and practitioners to effectively explore and optimize PiM technologies, further advancing the field.

1.4.3 Challenges in Evaluating PiM Architectures

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. The lack of a dedicated PiM framework further compounded the difficulties faced during the modeling and evaluation of pioneering PiM architectures. Researchers had to rely on manual calculations, statistical models, and improvised methods, resulting in outcomes that were hard to replicate and scale.

Moreover, the absence of standardized benchmarks and metrics for evaluating PiM architectures has made it challenging to compare and contrast different designs in a consistent and meaningful manner. This lack of standardization can impede the development and adoption of new PiM architectures, as researchers struggle to demonstrate the true advantages of their designs without a common ground for comparison.

PiM simulation frameworks [30,31] 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 simulation tools. Computer systems complexity is known to advance faster than computer simulation performance, a phenomenon known as the *simulation wall* [9]. As a result, better alternative approaches are necessary. Modeling performance is shown to be significantly higher with FPGA-accelerated simulation [21] or approximated FPGA-emulation [11, 12].

1.4.4 FPGA-based PiM Architecture Emulation

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 high-speed cycle-accurate FPGA-emulated processor systems has long been under development [22], and a handful of RISC-V soft-cores are now available for FPGAs [16, 18, 32]. 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 layout, bank state and timing) to facilitate prototyping of various PiM architectures on different memory standards and technologies. This thesis describes the effort to develop such a framework called 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 as part of a whole compute system. The memory model emulates *operations* such as burst write/read, refresh, bank interleaving, *behaviors* such as latencies and bank states, and *components* such as the shared data bus, row buffer, and subarrays. 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 demonstrate strategies to prototype and emulate both *pioneering bitwise-PiM* architectures such as RowClone [33], LISA [34], and Ambit [35] as well as generic PiM architectures such as Fulcrum [36] and BLIMP [37]. These prior works demonstrate high-bandwidth, low-latency, and energy-efficient PiM operations at the subarray, bank, and chip level. Emulating these architectures requires modeling the above-mentioned detailed aspects of the main memory and showcases the strength of our proposed framework. We also develop features to support modeling of memory physical effects and conduct a case study on RowHammer [38] modeling. We incorporate the memory and PiM model as part of a compute system consisting of a processor system, a memory controller and the emulated memory. We use open-source soft IP blocks that can be updated and modified as desired. The result is an emulated system that is composed only of soft IP blocks that can run at a high speed on a high end FPGA board. Finally, we release the platform together with helpful documentation and tutorials, and encourage users to get started with modeling, emulating, and evaluating their designs.

1.5 Contributions

In this thesis, we present a comprehensive framework that combines MEMulator, FreezeTime, and PiMulator for computer system emulation, design space exploration, and evaluation. MEMulator features a parameterizable, synthesizable memory emulation model (MEM) that generates an FPGA synthesizable logic representation of the modeled architectures, facilitating system-level evaluation with heavy workloads. FreezeTime and Architectural Virtualization mechanisms further enhance the capabilities of this framework. Building upon the MEMulator memory model, PiMulator allows users to inject logic for PiM kernels, enabling evaluation and development of PiM architectures. We summarize the combined contributions of MEMulator, FreezeTime, and PiMulator as follows:

- We present the first end-to-end full system emulation framework consisting of soft, open-source, configurable, and FPGA synthesizable modules for the CPU, bus interconnect, memory controller, main memory modules, and peripherals. We leverage high-end FPGA boards to host the emulated system logic, thus enabling researchers to rapidly and accurately evaluate a wide variety of their designs with heavy workloads of interest.
- We implement the framework using separate abstraction layers for the target system, logic model, and host FPGA. We provide a script that maps the target system description to a suitable logic model and use LiteX to integrate the soft modules and automatically generate the RTL and bitstream.

- We present an open-source, FPGA synthesizable memory emulation model in System Verilog that implements the main memory components, models their behavior, and accurately emulates memory operations. The memory model is flexible, parameterizable, and feature-rich to efficiently emulate different memory types. We integrate the memory emulation model into the LiteX framework to achieve full system emulation.

- We run several benchmarks of interest on systems with different CPU and memory configurations and measure significant evaluation speedup compared to computer simulation and matching accuracy with RTL behavioral models. We use both mid and high-end FPGA boards and observe matching target performance results, ensuring that modeling accuracy is preserved independent of the FPGA host.

- We present a coarse system-level to module-level sync-by-stall mechanism that ensures cycle-accurate emulation of a target system while facilitating architectural virtualization through time multiplexing. The novelty comes from the coarse virtualization approach: spatially at the block level and temporally at the operation level.

- We present architectural virtualization for all parts of a system: compute, memory, and interconnect. To this end, we describe how this virtualization can be used to emulate complete system-on-chip (SoC) architectures without the need for custom timing models.

- We compare the accuracy and evaluation speed between FPGA-based emulation, FPGA-accelerated simulation, and simulation with gem5 and Verilator. We also describe how the proposed techniques can enhance the capabilities of state-of-the-art emulation platforms.

- We implement and test a parameterizable, structurally accurate main memory and PiM model in System Verilog, synthesizable on FPGA boards. The memory model supports comprehensive emulation of many memory flavours and allows prototyping of various PiM architectures by injecting PiM logic into reserved placeholders.

- For increased usability, we integrate the memory and PiM model into the LiteX [18] framework, facilitating easy interfacing, system generation, and deployment of the target system on a preferred FPGA board. We demonstrate correct operation of the emulated system consisting of a number of VexRISC-V [16] soft cores, LiteDRAM memory controller, and different memory flavors and PiM architectures modeled with MEMulator and PiMulator.

- We demonstrate strategies for prototyping and design-space exploration of bitwise-PiM and more general/complex PiM architectures. We conduct PiM architecture evaluations by exploring various subarray configurations for RowClone [33] and Ambit [35], including inter-subarray links as described in LISA [34]. Next, we showcase emulation and explore various configurations of PiM architectures that together cover a larger PiM space, such as Flucrum [36], DRISA [39], Sieve [40] and BLIMP [37]. Finally, we assist pioneering users of the platform to model, run and evaluate their desired emulated PiM architectures. The focus is on new, viable PiM architectures that are of interest and use to academic and industry partners.

- We present memory emulation vs. simulation performance and accuracy results running a memory micro-benchmark [41] as well as common user workloads and benchmarks such as Linux boot. We evaluate the memory model alone, as well as the whole system, and compare emulation, FPGA-accelerated simulation and computer simulation results. 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 optimize the memory, PiM and system emulation framework for performance, efficient resource utilization, and ease of use. At this time, the emulation framework is $\sim 28\times$ faster (weighted average) than simulation [42] for memory operations alone and even higher for PiM architectures. For the memory and PiM model alone, we efficiently utilize the available BRAM, and only 1% of the available LUTs, leaving sufficient resources for the remaining system components. We demonstrate PiMulator system operation hosted on different high-end FPGA boards, especially on boards available in the cloud.

Chapter 2

Insights into Memory and PiM

2.1 Background on Memory and PiM

This section explains the memory and processing in memory concepts relevant to this research work in detail.

2.1.1 Memory Hierarchy

Computer architectures balance the strengths and weaknesses of several memory types to achieve high performance, reliability, endurance, low power, and low cost. Several factors such as the technology process characteristics (speed, power, density), physical aspects of the data retention mechanism, the volatile nature, or the addressing circuit delays motivate the use of a memory hierarchy: near-instant level 1 instruction and data cache, swift level 2 cache, agile level 3 and last level cache, fast DRAM and non-volatile main memory, moderately-slow flash, slower disk, and finally tape or other cheap and reliable archival methods. Cache memory is low density, high cost, and power-hungry, therefore only present in small amounts, up to several MB. Caches hold the hottest operand data and make the main memory seem faster to the processor. DRAM memory is high density and low cost, but less fast, volatile, and suffers from leakage, requiring frequent

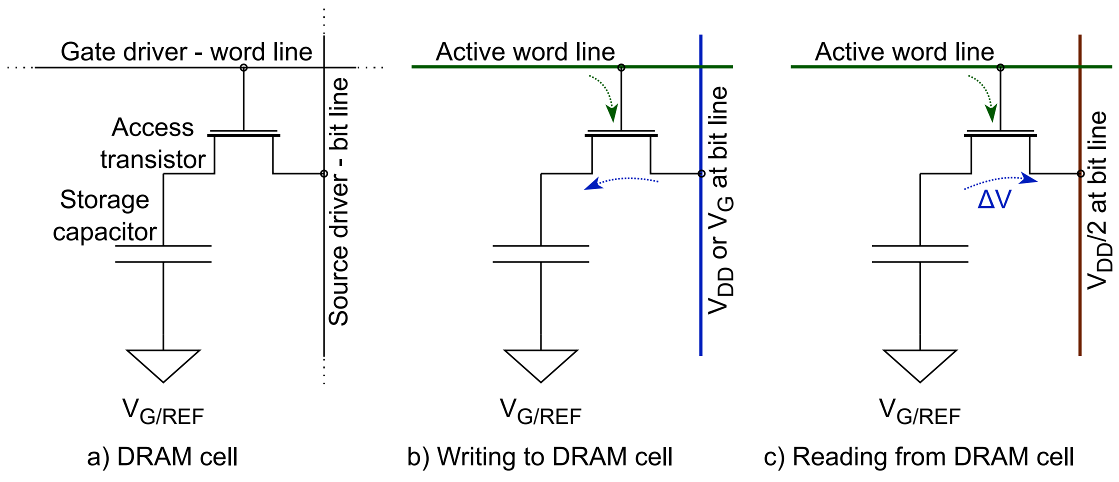


Figure 2.1: DRAM cell circuit components, access signals, and operation

refreshing, which results in increased power usage and makes the memory unavailable during the refresh process duration. Non-volatile main memories do away with the refreshes, however, the technology is young(er) and under-performs DRAM in speed and endurance. Main memory is suitable for the bulk of application data and is a perfect candidate for PiM architectures for applications with low computation intensity.

2.1.2 DRAM Memory Cell and Subarray Operation

All memories, at their core, utilize an elementary unit of memory mechanism that allows writing, storing and reading a bit of information. A DRAM memory cell consists of an access transistor with the drain connected to a charge storage capacitor, as shown in Figure 2.1a. Applying a voltage to the transistor gate allows current to flow through the transistor, charging or discharging the capacitor. The charge level on the capacitor represents a logic value of 0 or 1. Driving the word line activates the access transistor, thus allowing to write the bit line logic value to the storage capacitor, shown in Figure 2.1b. Similarly, placing the bit line in a pre-charged floating state and then driving the word line allows sensing the storage capacitor charge, shown in Figure 2.1c. Reading the logic value of a cell changes the charge level of the capacitor and is therefore a destructive process.

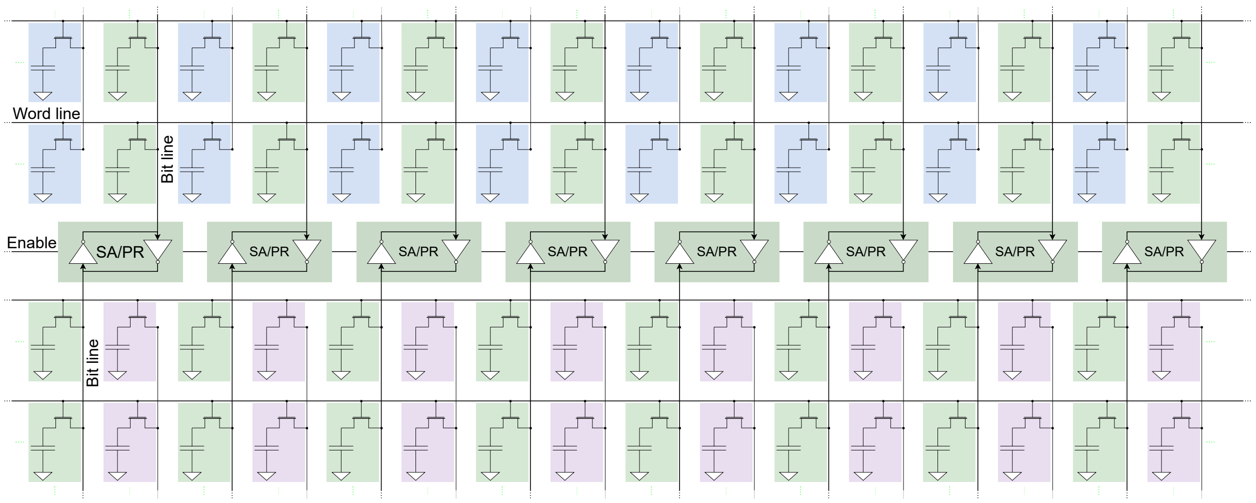


Figure 2.2: DRAM open bit line architecture subarray circuit region

DRAM cells are arranged in a two-dimensional array, forming rows and columns, as shown in Figure 2.2. The gates of a row's transistors connect to a word line, while the sources of a column's transistors connect to a bit line. This allows sharing of the word and bit line wires between multiple cells, which reduces cost. Additional circuitry at the end of each bit line, known as a sense amplifier, facilitates bit line precharging, sensing, amplifying, then value holding and writing to the cell [43]. A sense amplifier can be abstracted as a chain loop of two inverters, similar to an SRAM cell. Figure 2.2 depicts the DRAM cells in the 4 rows and 8 columns near 8 sense amplifiers of a subarray circuit region. Placing the row of sense amplifiers (row buffer) in the middle of a subarray has the advantage of shorter bit line wires, which leads to improved efficiency, performance, and manufacturing yield. The sense amplifier and precharge circuit is larger (wider) than a dram cell therefore the columns of neighboring subarrays (shown with light blue and purple) are fit in between the subarray columns (light green). Given that the word line is shared, an entire row has to be read to determine the value of one or a few cells in the row. For reading, first, all bit lines are precharged and left to float, then the row word line is driven high, which opens the pass transistors. The bit lines of a subarray receive a precharge from the sense amplifier, setting them to $V_{DD}/2$ which places the inverter chain in a metastable state. Upon activation of the word line, the access transistor triggers a slight deviation ΔV in the bit line voltage caused by the charge on the capacitor. This deviation prompts the sense amplifier to deviate from its metastable state, which is quickly amplified to an absolute state, either 0 or 1. Concurrently, the value is latched, driving the bit line and causing restoration of the capacitor charge. Keeping the high voltage set on the word line will allow the sense amplifier to write back the previously read values, thus restoring the original charge on the capacitor. Due to the leakage current through the pass transistors, each cell charge leaks with time and requires periodic refreshes.

We accomplish an accurate emulation of DRAM cells on an FPGA by utilizing BRAM or LUTRAM resources for data storage. Additionally, state machines monitor the circuit state, controlling data access during specific time frames. These periods include when the bit lines undergo precharging, when charge sensing and amplification occurs, and during the RC delays associated with writing to the capacitor. The state machines also manage data access during the cell, row, or array refresh duration and transitions from several read/write processes, as well as other latency-related activities, ensuring a comprehensive emulation of DRAM cell behavior.

2.1.3 DRAM Bank Structure

A cell in the memory array is identified by its row and column addresses. A row address decoder translates the row address to activate a specific row's word line. As a result, the row data is latched into the sense amplifiers, which act as a row buffer for the row of interest. Next, a column multiplexer/demultiplexer translates the column address to interact with a specific value in the row buffer. During a read (write) operation, data is multiplexed (demultiplexed) from (to) the row buffer to (from) the data bus. The number of peripheral components per column (sense amplifier, Mux, DeMux) is higher than per row (row address decoder). Therefore, memory arrays are designed to have more rows than columns (e.g., 2^{17} rows vs. 2^{10} columns) to maximize density. Handling so many rows reliably and at high speed is achieved by splitting the array into subarrays, each subarray equipped with its row of sense amplifiers and local bit lines. Similarly, the many columns in a subarray are split into sections, with local word line drivers, forming MAT/Tile regions [44]. For example, a common design configuration is to allocate $2^8 = 256$ rows per subarray, further

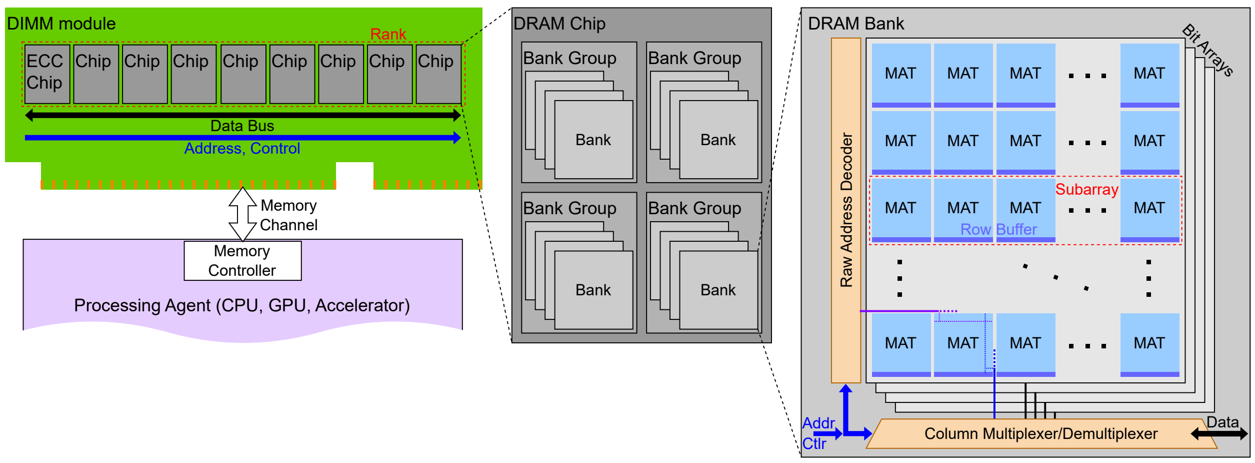


Figure 2.3: DIMM module structure, chip data layout

dividing it into MATs of $2^8 = 256$ columns, with global word/bit lines driving much shorter local word/bit lines. For a device width higher than 1 bit, the row-column address pair and control signals are sent to a group of arrays, each contributing 1 bit to the data bus. The memory arrays, together with the associated peripheral components, form a memory bank. The bank is the smallest structure with which the memory controller can directly interact. A bank receives address, controls, commands, and data signals and is characterized by its state.

2.1.4 DIMM Module, Chip, Bank Groups Structure and Operation

The data layout hierarchy for a DIMM module is shown in Figure 2.3. A memory chip consists of several memory banks, each identifiable by a bank address. Banks can be grouped into bank groups, each identifiable by a bank group address. All banks and bank groups share the chip's address, control, and data bus signals. Bank plurality facilitates high bus utilization, achieved by scheduling operations on banks that are in active state. Similarly, bank grouping facilitates prefetching [45]. A rank consists of several memory chips on a circuit board sharing the same address and control signals but contributing different data bus bits. This aggregation approach makes it possible to use multiple chips for more memory capacity and have a narrow data bus interface at each chip. One or more ranks form a dual (2x32/64-bit) inline memory module (DIMM). A memory controller interfaces with one or more DIMMs via a memory channel. The memory channel consists of control signals, an address bus, and a 64(+8ECC)-bit wide data bus, i.e., 8-byte words. Some memory types, such as GDDR and HBM, have wider data buses. During read/write operations, the memory controller manages data flow between the cache and the memory channel in bursts that fill a 64-byte cache line, a process known as prefetching. Prefetching also alleviates the mismatch between the interface and memory core speed [45]. With the introduction of dual data rate (DDR) SDRAM, the gap between the interface and the memory core speed continued to grow, requiring larger prefetching. The use of bank groups is an alternative to increasing the prefetch size by allowing simultaneous independent prefetching at each bank group. Moreover, the memory controller engages with each bank independently and consecutively to increase DIMM bus utilization and fill multiple cache lines in close succession, a process known as bank interleaving.

The design of the Dynamic Random-Access Memory (DRAM) cell remains largely standardized; however, all subsequent elements within the memory architecture are meticulously optimized to align with critical metrics such as power efficiency, throughput, and cost. Various parameters, including the number of ranks, chips, chip data width, bank groups and banks within each group, rows and columns, the size of subarrays, the frequency of the memory core and interface, latencies, on/off-chip error correction mechanisms, addressing schemes, among others, collectively influence the cost, bandwidth, latency, throughput, reliability, and endurance of the final memory module. In order to accommodate a broad configuration space, our methodology embraces a parameterized implementation for the memory model within System Verilog. Consequently, the memory model assumes its desired configuration at the synthesis phase.

2.1.5 DRAM Bank State and Timing

The read, write, and refresh DRAM operations incur several kind of latencies that have to be accurately respected to ensure data and hardware integrity. These latencies form a memory timing specification. Other than delays associated to the operation of the memory cell and array, the memory timings specify memory core and interface clock frequency, interface protocol timings, data bus switching delays, and power budget limits. A list of memory timings and their description is presented in Table A.1. In addition to memory timings, during operation, the memory arrays are in active states, describing physical behavior, with limited permitted action sequences. This behavior is best represented and modeled using a finite state machine.

2.1.6 RowHammer and Physical Effects

Repeatedly accessing a particular row multiple times in a modern DRAM chip causes bit flips in cells in the rows neighboring the accessed row in a predictable and consistent manner. It is caused by a hardware failure mechanism called DRAM disturbance errors, which is a manifestation of circuit-level cell-to-cell interference in a scaled memory technology [38]. RowHammer is a vulnerability of DRAM memory caused by the physical properties of the technology, with previously demonstrated exploits. Due to the lack of cost effective prevention mechanisms, and with the continuing technology scaling, it is expected that RowHammer effects will only worsen. Other physical effects of interest are aging, manufacturing defects, non-uniform charge leakage, and others. A number of prevention mechanisms have been proposed and have to be thoroughly investigated to be adopted by the industry.

2.1.7 Emerging Memory Technologies

Emerging Non-Volatile Main Memory (NVMM) technologies such as Phase Change Memory (PCM), Spin-Transfer Torque RAM (STT-RAM), Resistive Random-Access Memory (RRAM), and 3D XPoint are promising technologies for the future of memory systems. They offer desirable properties such as high density, byte-addressability, non-volatility, low cost, absence of refresh disruptions, and energy efficiency. However, they are not (yet) ideal, with disadvantages such as high write latency, high write power consumption, and limited write endurance. As competitive alternatives to Dynamic Random Access Memory (DRAM), they bring many research challenges to system architectural designs [46]. Other than the nature of the memory cell physics mechanism and its effects on the array design, emerging memories share the same structure and similar state and timing behavior, therefore can be modelled using the same tools without significant changes to the memory model.

2.2 Real-World Processing-in-Memory Architectures

Numerous PiM architectures have been proposed over the last decades and studied in a number of survey papers [27, 47–49]. Additionally, we refer to a list of PiM architectures of interest with a brief description and taxonomy classification in [28]. PiM architectures can be characterized by several distinct attributes. The computation location indicates where in the memory the computations are performed (e.g., module interface, chip, bank or subarray) while the computation type can be the memory circuit itself, fixed or configurable logic resources, programmable cores or a combination of them. The memory technology indicates the host memory technology that the PiM architecture is implemented on. Finally, the computation parallelism specifies the level of parallelism that can be exploited as a result of the auxiliary computation enabled by the PiM kernels. PiM architectures introduce additional costs in terms of design, manufacturing steps, reduced memory density, power consumption, and overall system modifications, therefore have to be thoroughly evaluated and proven to bring benefits that outweigh the costs.

In the last few years PiM architectures started to transition from research to startup and industry products. In this section we describe several such products, existing at the time of writing this thesis [50, 51], and speculate on how PiMulator can be used to emulate, model, evaluate and develop new aspects of these PiM architectures.

2.2.1 UPMEM PiM Architecture

UPMEM, founded in 2015, announced the first real-world PiM architecture in 2016 and began commercializing it in 2019. UPMEM's architecture augments standard DIMM modules with a large number of DRAM Processing Units (DPUs) located near the DRAM banks. UPMEM fabricates PiM-enabled DIMM chips in a DRAM technology process. The first generation of such chips was 8-bit wide data width, with 8 chips forming a single rank, and the DPUs ran at 267 MHz. Later generations expanded to 16 chips forming two ranks with DPUs running at 350-425 MHz. With the DPU located close to the memory bank, the energy spent for data access is reduced by 20x. Due to the DRAM manufacturing process applied for the DPU logic, the energy spent for the operation itself is doubled, resulting in an overall energy reduction of 17x. The DRAM process adds several constraints, such as 3x slower transistors, 10x lower density, 3x reduced routing metal layers, and others. These mandate for strong design choices, and a high-fidelity emulation platform can come in handy.

The UPMEM architecture consists of the memory array at the memory banks, a first processor as the DPUs, a processor control interface interfacing with the central processor and making the memory banks available to the central processor, resembling the accelerator model of computation. UPMEM PiM-enabled memory DIMMS can coexist with standard memory DIMMs in a system.

Each UPMEM DIMM chip contains 8 64MB banks, each bank paired with a DPU. Therefore, there are 8 DPUs per chip and 64 DPUs per rank, and 4GB of memory capacity. The DPU resembles a highly pipelined processor, connected to a register file, 24KB instruction and 64KB scratchpad caches, and DMA engine with a 64-bit wide access to the DRAM bank. The DPU had to be pipelined into 14 stages in order to meet the desired high frequency using the slow DRAM process. This design approach is quite innovative and highlights a real-world constraint that perhaps would be otherwise ignored by a computer architect using simulation tools alone.

The DPU consists of 6 main stages that are further pipelined into 14 stages to meet an Fmax of 425MHz in a DRAM manufacturing process. It can support up to 24 hardware threads to enable *tasklets*. The *Dispatch* stage deals with thread selection, *Fetch123* fetches the instruction from I-cache, *ReadOP123* reads the operands from the register file, *Format* formats the operands, *ALU1234* implements the operation and writes to scratch and *Merge12* formats the result.

Example systems consist of dual socket central processors with 20 UPMEM DIMMs of 2 ranks

(16 chips) and 2 conventional DDR4 DIMMs for a total of 160GB PiM enabled memory with 2560 DPUs; or a single central processor with 10 UPMEM DIMMS of 1 rank (8 chips) and 2 conventional DDR4 DIMMs for a total of 40GB of PiM enabled memory and 640 DPUs.

Emulating the UPMEM architecture in PiMulator can be achieved by connecting the UPMEM PiM processor logic to the PiMulator Bank array interface dedicated for PiM data access. The UPMEM PiM processor logic will account for the computational functionality. For timing, a PLL can generate a clock of frequency matching the UPMEM PiM scaled down according to the system scale factor. The Bank state and timing model can model additional latency aspects.

2.2.2 Samsung HBM-PiM

Samsung unveiled the High Bandwidth Memory (HBM) with integrated Artificial Intelligence (AI) processing capabilities, known as HBM-PIM, in early 2021 [52–54]. It is also known by its aliases, Aquabolt-XL and FIMDRAM [55]. Despite the HBM's wide, high-bandwidth, multi-channel interface, it still falls short for many memory-bound Machine Learning (ML) workloads. To address this limitation, Samsung's design dedicates several layers of the HBM dies to either FIMDRAM or HBM-PIM dies.

Each HBM-PIM die features four pseudo channels, each enhanced by one Programmable Computing Unit (PCU) per two bank cell arrays, facilitating direct access to two bank arrays by the PCU. The PCU microarchitecture comprises 16 16-bit SIMD Floating-Point Units (FPUs) capable of executing multiple multiply-add instructions. The PCU includes a 32-bit RISC-like processor with a 5-stage pipeline. The Instruction Set Architecture (ISA) consists of 9 instructions: 4 floating point arithmetic, 2 data, and 3 control operations. The circuit consists of control, file registers, FP16 Multiplication, and FP16 Addition execution logic. Instructions are delivered as DRAM commands, adhering to JEDEC controller standards. The operand data can originate from bank array cells, row buffers, registers, or the result bus.

The HBM-PIM architecture supports simultaneous bank activation, enabling concurrent access to multiple rows across all banks. Samsung envisions integrating the HBM-PIM architecture with high-performance GPUs or other high-end System-on-Chips (SoCs).

The emulation of this architecture necessitates instantiating multiple PiMulator instances to model the HBM pseudo-channels. Additionally, users need to configure the interface, command decoder, data bus, and data layout to reflect the HBM and PIM data flow. Auxiliary multiplexers must be set up to link a PCU to two banks, modeled using true dual-port BRAM. The user must also implement the PCU processor within the PiM template kernels. Lastly, configuring further timing and state behaviors is necessary, which must be incorporated into the Timing FSM models.

2.2.3 SK Hynix Accelerator-in-Memory (AiM)

In early 2022, SK Hynix announced the development of a PiM AI accelerator based on the GDDR6 memory module, referred to as the GDDR6-AiM [56–58]. The architectural blueprint of the GDDR6-AiM features 16 AI processing units (PUs) located next to the 16 banks of the GDDR6 memory module, establishing a one-to-one correspondence between the PUs and the memory banks. Every PU encompasses multiply-accumulate (MAC), element-wise multiplication (EW-MUL), and activation function (AF) logic units. Additionally, two global buffers, each with a 1KB SRAM capacity, are implemented for storing computation vectors. The MAC unit's structure includes multipliers and several layers of adders, which perform the accumulation operation and subsequently output the result or pass it through AFs. The GDDR6-AiM capitalizes on the extensive device width of the GDDR6 chip, targeting a burgeoning range of AI applications.

Further, the architecture includes specialized commands and features that facilitate the simultaneous activation of four or all sixteen banks. These commands also control the PiM logic related to computation and data management operations. Multiple banks are activated simultaneously by sending identical word line addresses to the banks. A reserve capacitor is also incorporated to offer additional power, thereby alleviating the Four Activate Window constraint (t_{FAW}).

PiM operations employ multiple banks for reading operand data and writing the resultant output, necessitating a data bus design that accommodates this specific data flow pattern. The MAC unit incorporates 16 multipliers operating on BF16 data, followed by an adder tree that accumulates the computed products. The kernel functions in a Single Instruction, Multiple Data (SIMD) fashion. Finally, the AF logic unit supports many activation functions such as (Leaky) ReLU, Sigmoid, GELU, Tanh, among others, of which some are implemented using a look-up table (LUT). Beyond these microarchitectural components, the GDDR6-AiM also introduces a software and firmware stack to enhance user-friendliness.

The GDDR6-AiM architecture can be emulated on an FPGA using PiMulator tools. This involves configuring the memory model to accommodate GDDR6 specifications, defining the PU logic within the given PiM templates, optimally utilizing the dual port BRAM memory and data bus, extending the command decoder table implementation, and incorporating the computation states and delays in the timing state machine model.

2.2.4 Samsung AxDIMM

In August 2021, Samsung launched another Processing-in-Memory (PIM) architecture, AxDIMM [59, 60]. This architecture aims to augment acceleration to DIMM modules. The design strategy capitalizes on rank-level parallelism by introducing an FPGA device between the DDR4 DIMM channel interface and the two concurrently interfaced DDR5 ranks, deviating from the traditional sequential rank interfacing. The host communicates with the module through the DIMM memory channel and interfaces with the FPGA via a DDR4 Slave PHY. Subsequently, the FPGA processes host requests utilizing the programmed compute IP while concurrently interfacing with the two DDR5 rank chips via two Memory Interface Generator (MIG) IP PHYs. The FPGA logic kernels can be customized to boost various applications of interest.

One studied application is accelerating Deep Neural Network (DNN) computation and table lookups as part of a personalized recommendation algorithm for social network users. Another example includes in-memory acceleration of database operations, such as scan queries, where a memory crossbar facilitates multiple kernel memory accesses to both ranks.

In many ways, the AxDIMM module mirrors numerous FPGA accelerator cards, with the primary distinction being the DIMM interface of the module rather than the conventional PCIe interface. To model the AxDIMM in PiMulator, users must instantiate a hierarchy of DIMM interfaces. This includes an initial interface from the host, followed by an additional interface for each rank, with the FPGA logic kernels acting as a compute node in the wire.

Chapter 3

Modeling Methods

3.1 Related Simulation Frameworks

Employing a computer architecture simulation tool running on a general-purpose processor system is the most straightforward, cost-effective, and flexible initial approach for evaluating a target architecture with relevant workloads.

3.1.1 The gem5 Simulator

The gem5 simulator [61], among numerous other tools, stands out as a prevalent, versatile, and widely adopted computer architecture simulation framework, enabling high-accuracy simulations of processor, cache, and memory system configurations. The initial release of gem5 integrated the notable features of both M5 [62] and GEMS [63] simulators, encompassing support for multiple Instruction Set Architectures (ISAs) and a broad spectrum of CPU models, such as in-order and out-of-order processors. Over time, community-driven updates have solidified gem5's position as a vital tool for researchers and educators in the computer architecture field, with many enhancements and new features outlined in [64]. The framework boasts modularity, extensibility, usability, and improved support for various ISAs, memory models, and other architectural components. For example, users can model main memory using distinct memory simulation tools within gem5, such as DRAMsim3 [42], Ramulator [65], or alternative memory simulation tools. However, achieving high accuracy and fidelity using simulation models running on a CPU entails a considerable performance trade-off.

3.1.2 PIMSim

PIMSim [30] is a flexible PiM system simulator that facilitates circuit, architecture, and system-level research on heterogeneous Processing-in-Memory architectures. The framework offers a range of speed versus accuracy tradeoffs through three distinct simulation modes: fast, instrumentation-driven, and full-system. PIMSim provides detailed performance and energy models to simulate PiM-enabled instructions and in-memory processing logic across various memory devices, employing multiple distinct memory simulation tools.

The frontend of PIMSim incorporates an application partitioner that identifies and allocates PiM instructions to PiM kernels executed within memory. The PiM kernel logic can be configured as processors or specified accelerators. Similar to other simulation tools, PIMSim accurately simulates diverse aspects of the system and PiM architecture by combining several simulation tools and managing their input and output traces, yielding reliable performance estimates. However, simulation times range from 4×10^4 slower than target execution in the fast simulation mode to 3×10^5 slower in the full-system mode.

3.1.3 MultiPIM

MultiPIM [31] is a comprehensive, general-purpose Processing-in-Memory simulation framework designed to support multiple memory stacks, such as multiple Hybrid Memory Cube (HMC) configurations. One primary goal of MultiPIM is to accurately simulate architectural details and programming interfaces, which are essential for practical PiM systems. The framework consists of a frontend and a backend, where the frontend primarily manages non-memory instructions and cache accesses, while the backend simulates the latency of memory requests issued from the frontend. MultiPIM leverages existing simulation frameworks for CPU, PiM kernels, and memory systems to simulate system-level PiM architectures. The framework generates memory interconnections using user-defined connections between memories and CPUs and employs a packet-routing scheme among memory nodes, complemented by a crossbar switch implementation. Additionally, MultiPIM supports virtual memory, further enhancing its versatility. For PiM-specific functionality, the framework introduces a coherence directory for PiM cores and provides two offloading interfaces. With these features, MultiPIM offers a powerful, flexible, and accessible platform for simulating and exploring PiM architectures in various configurations.

3.1.4 PIMulator-NN

The PIMulator-NN [66] simulation framework, distinct from this thesis work, serves as an event-driven, cross-level platform for evaluating processing-in-memory based neural network accelerators. By integrating various circuit-level simulation frameworks, PIMulator-NN accurately models PiM architecture details and assesses analog computation units' area, latency, and energy consumption. This renders it particularly apt for simulating PiM architectures that employ memory fabric for logic operations.

The authors showcase PIMulator-NN's utility by implementing several PiM designs and conducting comprehensive simulations and evaluations for a memristor crossbar-like PiM architecture. This process includes examining power, performance, and area outcomes while capturing the impact of different design choices. As an event-driven simulator for neural network workloads execution, PIMulator-NN fuses multiple tools that model atomic aspects of the architecture. Users can make assumptions and approximations about the hardware, memory, and data mapping to optimize for simplicity and performance. The results provide valuable estimations of the performance, latency, energy, and area of these architectures, positioning PIMulator-NN as a significant contribution to the field of PiM-based neural network accelerators.

3.2 Challenges in Computer Architecture Simulation

Computer architecture simulation, employing tools like gem5 [61] or DRAMsim3 [42], can be slow due to several factors that contribute to performance limitations when running on even high-end CPUs. In the context of computer architecture research, these simulations are designed to model intricate systems and execute detailed analyses of processor and memory behaviors. Consequently, they often trade off speed for accuracy and precision [67].

One primary bottleneck is the complexity of accurately modeling the architectural components and their interactions. Simulations often rely on detailed cycle-accurate models, which require a vast number of calculations per cycle to represent the behavior of processors, caches, memory controllers, and other architectural elements. As a result, the computational overhead for these simulations increases significantly, leading to extended execution times [68].

Another factor that slows down computer architecture simulation is the need to model the full system, including the operating system and applications. Booting the OS and running applications within the simulation can take a considerable amount of time, as the simulation has to perform millions of instructions for each task [64]. Additionally, the simulation of large datasets and memory-intensive workloads further increases the simulation time.

Furthermore, traditional CPU-based simulators may be unable to leverage the full potential of modern multi-core processors, as their execution can be inherently sequential due to dependencies between architectural components. This lack of parallelism in simulations can further contribute to slow performance [69].

In summary, computer architecture simulation tools like gem5 and DRAMsim3 often exhibit slow performance due to the trade-off between accuracy and speed, the complexity of modeling intricate architectural components and interactions, and the limitations in leveraging the full potential of modern multi-core processors.

3.3 Enhancing Simulation Performance with Emulation

PiM architectures introduce an additional degree of complexity, aggravating the simulation performance issue. For example, PIMSim [30] attempts to alleviate the simulation performance issue by offering three simulation modes: *full-system mode* models a complete system employing detailed coherence mechanisms, *instrumentation-driven mode* models the PiM component driven by real-time traces extracted by instrumenting a full system, and *fast mode* which simulates only the primary components and quickly processes the input traces to generate "believable" performance and energy results. Even with the simplifications and accuracy tradeoffs, PIMSim is $10^4\times$ to $10^5\times$ slower than the target system. Likewise, MultiPIM [31], in terms of simulation performance is on average $10^5\times$ slower than the real target architecture being simulated.

FPGA-accelerated simulation and FPGA-based emulation is shown to run only at a factor of $10\times$ to $100\times$ slower than the target architecture modeled while preserving functionality, accuracy, and level of insight.

3.3.1 FireSim

FireSim [10] is a versatile and widely-used framework for simulating RISC-V processors, multicore systems, and networks while running Linux and other workloads. Adopting an FPGA-accelerated simulation approach, the framework emulates system processors and peripherals using soft IP modules synthesized onto FPGA and supports a variety of peripherals, including UART and Ethernet. FireSim incorporates FASED [21], an FPGA-accelerated memory simulation tool that models memory architectures, accounts for state and timing, and facilitates system-level memory evaluation. Despite its significant speedup and minimal loss of accuracy when compared to standalone cycle-accurate memory simulators, extending FASED to model various PIM architectures remains challenging due to the absence of structural component models and limited bandwidth between the simulation model and host memory. As an alternative, integrating the memory and PIM model introduced in this study can effectively enhance the FireSim framework, providing support for a range of PIM architectures. Designed to run on AWS F1 or the Alveo U250 board, FireSim interfaces with the IP shell and combines various tools and scripts, including AWS, Chisel, compiler, and FPGA vendor tools. The framework is not an emulation but rather a hybrid mix of FPGA-implemented architecture simulation models and atomic soft-core modules.

3.3.2 LiME

The LiME framework, presented in [11], utilizes a hard processor system interfacing with the DRAM memory on a Zynq UltraScale+ board to emulate high-performance computing (HPC) systems. The framework directs the memory access of the processor to the programmable logic, where it employs a loopback technique to approximately emulate different memories by inserting variable delay and throttle units on the loopback path to host memory. LiME makes use of techniques such as frequency scaling and runtime configurable delays to model the target system frequency and approximate the timing of different memory types.

The framework demonstrates fast full system modeling, system-wide frequency scaling, observability, and a near-memory configurable accelerator. It showcases how a slowed down and miniaturized system can serve as a model platform to accurately, transparently, and quickly emulate larger HPC systems and emerging memory architectures. While LiME provides overall insight into memory performance, it does not model the memory aspects past the memory interface.

3.3.3 MEG-HMC and MEG-HBM

MEG-HMC [19] and MEG-HBM [12] are frameworks that employ soft RISC-V cores, an adaptable memory controller, and HMC/HBM memory to emulate HPC systems featuring HMC/HBM, enabling users to efficiently integrate emerging memories into compute systems. These frameworks facilitate system-level integration, software-hardware co-optimization, and offer support for virtual memory for the study of near-memory accelerators. Users can leverage the RISC-V processor system, configurable memory controller, adaptation module layer, and performance monitor to model their desired architecture and extract useful insights on memory utilization. The MEG framework is particularly suitable for HMC/HBM integration and near-memory processing studies, while other memory flavors can be approximated with configurable delay elements.

3.3.4 PiDRAM

PiDRAM [14] demonstrates RowClone FPM [33] data copy, Ambit-like [35] AND-OR-NOT operations and true random number generation using DDR3 memory by violating the memory timings. PiDRAM makes use of an in-house developed memory controller and a RISC-V processor system to issue the precisely timed memory commands to a DDR3 memory module on an FPGA evaluation board. The framework also offers a PiM ISA extension, drivers, a software library, compiler and OS support to facilitate end-to-end processing-using-memory. PiDRAM uses the memory as is, while PiMulator emulates the memory as a soft IP module on the FPGA fabric.

3.3.5 RAMP-Gold

RAMP Gold [69] is an innovative and cost-effective FPGA-based architecture simulator, developed at the UC Berkeley Parallel Computing Lab. It has been designed to facilitate the early exploration of design space in manycore systems. With its high throughput and cycle-accurate full-system simulation capabilities, the prototype effectively uses a single Xilinx Virtex-5 FPGA board to simulate a 64-core shared-memory target machine with the capability to boot real operating systems. The system uniquely segregates the modeling of target-system timing and functionality, enabling accurate and efficient modeling of complex functions over multiple clock times. The functional model, composed of a multithreaded pipeline, executes hardware functionalities of the target cores, while the timing model meticulously tracks the performance and event timing of each simulated core and other system components, such as the L1/2 caches and the DRAM controller. Debugging and control are realized through an Ethernet link between the FPGA and the host machine, allowing seamless interaction.

The prototype's performance was evaluated using a modern parallel benchmark, with the results indicating a speedup of two orders of magnitude in comparison to a widely-used software-based architecture simulator. In a 2010 mid-sized FPGA setup, RAMP Gold was limited by the BRAM for total simulated cache size. However, even with a single, pipelined and multithreaded instance of the architecture, it demonstrated a staggering 250 times faster simulation speed over a widely-used simulation tool, underscoring the potential of FPGA-based simulation. RAMP Gold represents a foundational effort, significantly influencing the development of the RISC-V architecture and FireSim framework over the years.

3.3.6 Industrial Tools for Emulation and Prototyping

Industrial emulation tools distinguish themselves from previously discussed solutions by facilitating direct emulation and verification of intricate hardware architectures, with the additional benefits of swift compilation times and reliable results. Notable among these are Cadence Palladium, Siemens Veloce, and Synopsys ZeBu [70]. These systems, tailored for pre-fabrication hardware verification, harness extensive platforms integrating multi-node CPU and FPGA servers, underpinning a robust framework for hardware simulation, emulation, and verification processes. They offer optimal verification capabilities during the early design cycle, precisely when RTL is most susceptible to changes. Further, they incorporate comprehensive systems merging exemplary virtual platforms, hardware emulation, and FPGA prototyping technologies, which are the foundation of advanced verification methodologies. Capitalizing on unique architectures, commercial FPGAs, and avant-garde emulation techniques, these tools deliver enhanced performance, effectively catering to the escalating verification needs across diverse sectors like automotive, 5G, AI, and data center SoCs.

3.4 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 3.1. 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.

3.4.1 Fidelity

High fidelity is critical during PiM architecture modeling and evaluation for providing convincing results and insight. Although not an exact quantifiable metric, high fidelity typically suggests cycle-accurate, cycle-exact microarchitectural modeling, including RTL structural correspondence and hardware signal granularity. These features lead to accurate workload execution modeling on 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, which result in reduced fidelity. Conversely, FPGA emulation can intrinsically deliver higher fidelity at a reasonable performance by using a structural correspondence of the model with the targeted architecture.

3.4.2 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.

Approach	**Fidelity**	**Speed**	**Underlying Memory Model**	**Design Space Exploration**	**Full System Evaluation**	**Affordability Adoptability**
Verilog Behavioral Simulation [71]	High	Low	Interface, State, Timing, Data flow	Flexible, Behavioral	Compatibility, Correctness	Affordable, Tedious
Software Simulation [30, 31]	Low, Medium	Low	Timing	Flexible	OS/Application, Power, Performance	Affordable, Familiar
FPGA Accelerated Simulation [9, 21]	Medium	Medium	Timing	Flexible	OS/Application, Power, Performance	Cloud price, Familiar
Approximate FPGA Emulation [11, 12]	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, Power, Performance, Area, Hardware Model	Cloud price, FPGA toolflow, LiteX
DRAM use violation [14]	Maximum	Realtime	Physical memory	Constrained	OS/Application, Real hardware	Platform price, FPGA toolflow
Hardware tape-out [72, 73]	Maximum	Realtime	Physical memory	Constrained	OS/Application, Real hardware	Prohibitively Expensive, ASIC & PCB

Table 3.1: Comparison among PiM system prototyping and other evaluation approaches over selected attributes.

3.4.3 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.

3.4.4 Design Space Exploration

PiM architectures introduce many design variables [27] resulting in the need to perform a complex and vast design space exploration (DSE). In addition to speed, fidelity, and generated insight, a framework must have a high degree of flexibility and capability to accommodate a vast design space. Both software simulation and FPGA emulation offer similar high flexibility, but only emulation allows high performance.

3.4.5 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, 12].

3.4.6 Affordability and Adoptability

Finally, to be user-friendly, a framework must be affordable, available, and easy to use. Several cloud providers now offer high-end FPGA boards at competitive prices. With the advancement of high-level synthesis, hardware construction languages, and frameworks such as LiteX [18], FPGA

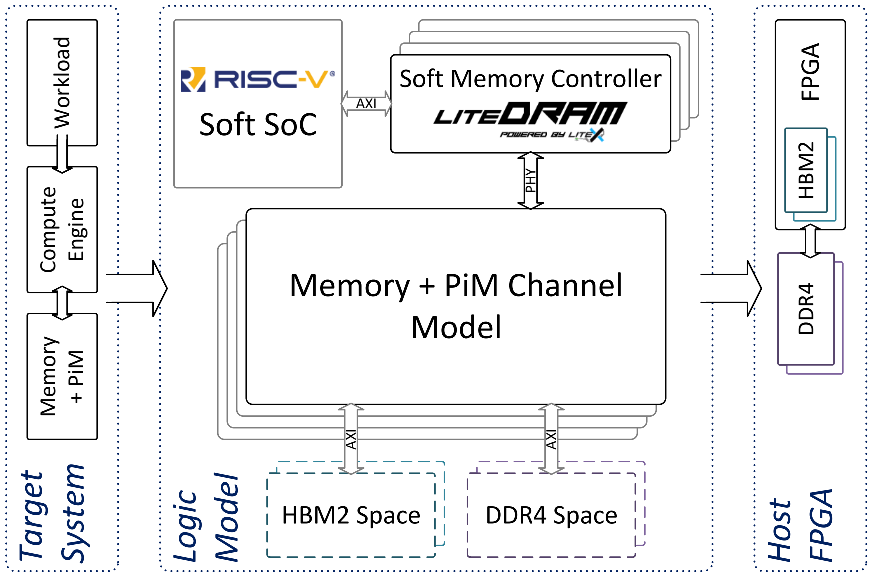


Figure 3.1: PiMulator top-level diagram. The *target* system is modeled with open-source IP primitives and synthesized on a *host* FPGA board. Workloads run directly on the hardware configuration at FPGA speed.

tools are becoming more user-friendly.

3.5 Resulting Platform Organization

When combining all the attributes mentioned above, we believe that an FPGA platform is the best fit as a host for PiM modeling and evaluation. 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 3.1.

3.5.1 Target System

The PiMulator user will begin by describing the *target system* in detail using configuration files. The processor system description will include aspects such as number of cores, cache sizes, and operating frequency. The memory system description will include aspects such as number of memory channels, memory type, interface width and frequency, memory timing parameters and data layout. The PiM configuration description will consist of number of PiM units, their location, type and operating frequency, together with instructions and controls. Finally, the application description will consist of the executable, or source code together with compilation scripts. We categorize these configuration files and scripts as part of the target system layer. At this layer the user is only concerned with correct target system description and runtime monitoring. For increased usability, the configuration scripts, as well as the collected metrics, will be similar in format to the standard adopted by gem5 and DRAMsim3.

3.5.2 Logic Model

We develop scripts that take as inputs the target system configurations and generate the *logic model*. At this layer the user is concerned with correct representation of the target system components with soft logic modules. The processor system is modelled by a system of VexRISC-V [16] soft cores equivalent to the target processor. The memory channels are modeled with instances of LiteDRAM memory controller. Finally, each memory and PiM system is modeled in detail using the memory and PiM model that is the basis of PiMulator. We expect PiMulator users to implement the PiM logic of their architectures of interest into reserved placeholders, and provide testbenches to easily validate their designs. Similarly, unsupported features of emerging memories can be modeled by expanding the state machine with new states and/or timing monitors.

3.5.3 Host FPGA

Finally, the logic model is synthesized onto a *host FPGA*. Here, the user is concerned with efficient utilization of the host FPGA board resources, such as BRAM/URAM, LUTs, FFs, board memory resources, and runtime at a sufficiently high frequency.

We adopted such a layered approach to ensure correctness via separation of concerns. For example, the host FPGA can influence the evaluation runtime duration, by accommodating larger memory models or achieving a higher operating frequency, but can not influence the final results of the target system.

Chapter 4

MEMulator: System-level Memory Emulation

In the field of computer architecture modeling and evaluation, utilizing computer simulation tools such as gem5 [61], SST [74], ZSim [75], and Intel® CoFluent™ Studio [76] has proven to be a highly effective, expedient, and adaptable method. These tools employ accurate cycle-based models for main memory simulations, such as DRAMsim [42, 77, 78] and RAMulator [65]. Researchers employ these tools to examine and assess innovative computer architecture proposals, outline the intended architectures, run relevant applications on simulated representations of these architectures, collect measurable data, and detect performance constraints.

However, as the complexity of the architecture soars, these tools manifest prohibitively slow simulation speeds. Furthermore, introducing novel features necessitates modifications to the existing models, raising concerns regarding the accuracy, correctness, and preservation of simulation fidelity. Drawing parallels to the fields of architecture, civil, and mechanical engineering, where concepts that appear sound in simulation must be validated in a wind tunnel, computer architectures can similarly be emulated on FPGAs. The objective of this chapter is to present a memory emulation model that serves to both complement and validate existing memory simulators.

4.1 System Emulation with Soft Modules, including Memory

Modeling a computing system utilizing a logic layer composed entirely of soft cores offers significant advantages. It enables customization, configuration, and evaluation of various aspects of interest by emulating high-performance systems running real-world workloads. An all-encompassing system, built out of open-source soft cores that are easily modifiable, parameterizable, synthesizable on FPGA, and representative of high-performance computing architectures, can function as a holistic, system-level rapid prototyping framework.

Furthermore, this approach preserves the hardware's spatial aspect, granting valuable insights to digital designers and facilitating hardware development. Additionally, the fast runtime achievable at FPGA hardware speeds enables swift evaluation with heavy workloads. While numerous soft-core models exist for CPU, bus, cache, and other peripherals, main memory models typically consist of the hardware as is or a throttled/delayed version. For example, in [11,79] the authors emulate the memory access throughput and latency using frequency scaling and runtime configurable delay units. Our solution addresses this gap by integrating a soft memory model with existing IP for the CPU, controller, interconnect, and peripherals. Other soft models can also be incorporated, such as accelerators, GPU, TPU, reconfigurable fabric, and other compute kernel types.

Previous research, notably RAMP-Gold [69], has made substantial progress toward emulating highly configurable architectures using soft IP modules. Our approach builds upon these lessons, principles, and an array of robust open-source development tools and IP cores. We incorporate our in-house developed emulation model for main memory into a framework that employs numerous other tools and software / hardware components. In the following sections, we detail the technology stack, flow, and system time modeling principles that render our framework capable and versatile.

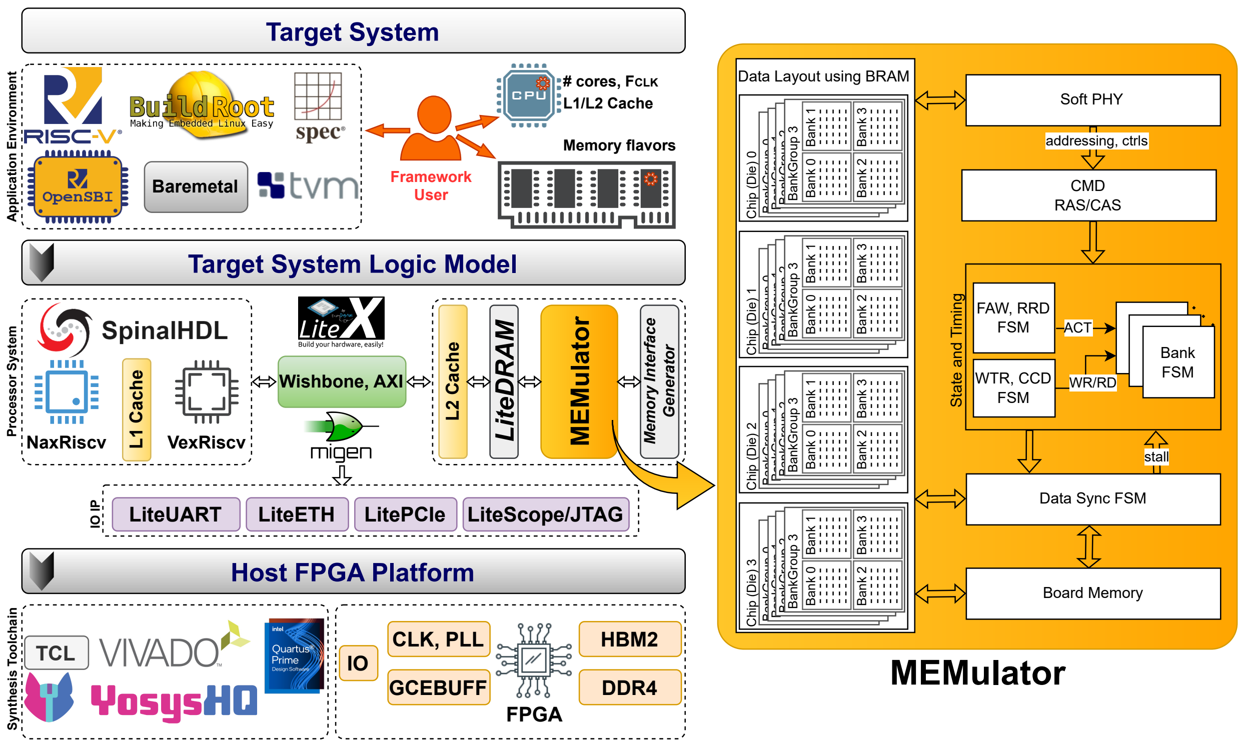


Figure 4.1: Comprehensive Overview of the Framework Technology Stack: Tools and IP Integration, Organizational Structure, and Memory Emulation Model Implementation

4.2 Technology Stack

Building on the principled structure of the framework depicted in Figure 3.1, we develop each framework layer by incorporating various valuable open-source and proprietary tools. The framework's current organization of IP and tools, depicted in Figure 4.1, facilitates the transition from an architecture target description to a suitable logic representation and generates deployable binaries and executables with minimal engineering effort. This technology stack encompasses a broad range of computing systems and offers flexibility for future expansion. Furthermore, it can be cost-effectively executed on entry-level, mid-range, and high-end FPGA hosts from various vendors, with most of the components being open-source. We also aim to showcase the tool space as a technology stack employed by the framework, highlighting the dependencies between the blocks. Furthermore, we emphasize how our IP contributions enhance the original LiteX framework.

4.2.1 Target System

The platform user provides a comprehensive description of the target system to be emulated, encompassing software (compiler, BIOS, OS, application workload) and detailed hardware specifications. The target system's processor configuration may comprise multiple CPU cores with varying complexity and cache hierarchy, running at a particular frequency. Additionally, the memory system may consist of one or more main memory modules, with their microarchitecture meticulously described in DRAMsim-like [42] configuration files. The target system layer also entails user experience aspects such as FPGA programming, user I/O, BIOS and OS boot, workload deployment, and observability. We chose open-source RISC-V ISA implementations and compilation tools [80] and utilized Buildroot OpenSBI Linux [81] for the OS. We leverage RISC-V GCC/G++ cross-compiler tools for constructing workloads and LiteX [18] BIOS and UART/JTAG peripherals for I/O.

Frequently, the target system will be a complex, extensive, resource-demanding architecture operating at a high clock rate. However, on the FPGA, the system must be modeled using soft, scaled-down IP modules composed of FPGA blocks (LUTs, FFs) and operating at a reduced clock frequency. Analogous to how an object in a wind tunnel is often a miniature, simplified yet representative version of the target, exposed to scaled stresses, MEMulator serves as a framework that assembles an entire system from FPGA blocks, paying particular attention to the memory system and subjecting it to corresponding stresses. The framework includes a script that analyzes the performance metrics of both the target and host systems, determining the appropriate Target System Logic Model values. These values are then supplied as parameters to the Target System Logic Model generator.

By default, we experiment with and provide RISC-V and other soft or hard CPU cores featured in the LiteX framework as logic models for the target CPUs. Users have the flexibility to expand this system with other CPU or software stack components (compiler, BIOS, OS) according to their preferences.

4.2.2 Target System Logic Model

The target system logic model uses soft IP blocks to accurately represent the target system, including RISC-V softcore processors such as NaxRiscV [82] and VexRiscV [16]. These processors, implemented in SpinalHDL [83], are optimized for FPGA deployment and offer reliability, performance, scalability, community support, and compatibility with various application modes, such as bare metal, BIOS, RISC-V compiler tools, Linux, and Linux-based operations.

The MEMulator logic layer employs the LiteX framework [18], which supports a wide range of processor cores, languages, and implementations. LiteX uses Migen [17] for Python-based HDL development and offers several IP blocks, including interconnect bus, Wishbone, and AXI. Our memory emulation model connects with the LiteDRAM [84] memory controller, which features a Level 2 cache and supports various memory configurations. This controller has been tested on multiple FPGA boards with DDR3, DDR4, and LPDDR physical memories. The MEMulator memory model connects to the Memory Interface Generator (MIG) to synchronize model data with board memory resources. We also include various Input/Output (I/O) components, such as LiteUART, LiteEthernet, LitePCIe, LiteScope, and LiteJTAG, to enable deployment, interfacing, booting, and high-performance observability and debugging features.

In summary, we use the LiteX framework to integrate, generate, synthesize, and deploy the logic system. LiteX incorporates different IP modules, including ROM, RAM, buses like Wishbone and AXI, interfaces like UART and I2C/SPI, L1/2 Cache, LiteDRAM memory controller, LitePCIe DMA, and other important IP. Additionally, LiteX employs Migen (Milkymist generator) to wrap custom or third-party IPs. We use vendor IP such as the physical memory interface generator (MIG), PLL, gated clock enable buffers, and more. For processors, we use open-source RISC-V cores, including the FPGA-optimized in-order VexRISC-V and the high-performance out-of-order NaxRISC-V implementations.

Memory Emulation Model Overview

We have developed a memory emulation model that accommodates various memory types, the main components shown in Figure 4.1. The model is implemented in SystemVerilog and features the main components of a memory device, such as a DIMM interface acting as a synthesizable soft model of a physical (PHY) layer, a decoder, a timing and state model, a data bus, and RAM as blocks modeling data layout. The memory model is synthesizable to FPGA boards, efficiently utilizes FPGA fabric, and can run at various frequencies. The design is parameterizable and wrapped using a Python Migen script parsing configuration files and passing the values to parameters. The data layout supports Rank, Chips or HBM Dies, Bank Groups, Banks, Subarray rows and columns, together with the data bus and necessary addressing logic. A Data Synchronization Engine combines the memory model with the FPGA board memory resources, facilitating high capacity.

The implemented interface is a soft-PHY (also called NoPHY), which is a logical representation of all interface signals and their behavior. A Command Decoder module decodes these signals into commands, generating controls for row address strobe (RAS) and column address strobe (CAS) processes. A JEDEC-compliant state machine models each memory bank's state, incorporating all memory states, transitions, and protections against prohibited actions. Furthermore, memory timings for state transitions are modeled with counters, ensuring appropriate delays. Additional state machines impose inter-Bank and inter-BankGroup global timings, such as tFAW.

A small fraction of the memory core is mapped to FPGA block RAM (BRAM) resources. With a limited amount of BRAM, only a few full rows per memory Bank are modeled at the same time. The accessed row serves as data storage and an active row buffer. The Data Synchronization Engine manages these rows as floating rows, mapping them to actual row IDs in the emulated memory. This engine also functions as a cache control unit, synchronizing the memory data with the onboard memory data.

Users can easily tailor a memory microarchitecture by modifying an existing configuration in terms of its structure, behavior, and operation. For instance, users can prototype new interfaces, commands, or data hierarchy organizations, such as additional banks or intra-bank features like concurrent row activations at subarrays (subarray interleaving). The memory controller similarly includes multiple configurations to adapt to the desired memory type accordingly. This capability facilitates rapid prototyping and system-level evaluation of new features.

4.2.3 Host FPGA Platform

The soft modules are synthesized and programmed on the FPGA's reconfigurable fabric. The configuration leverages the board peripherals, such as the memory resources, to accurately model the target system. Emulating a high-performance computing system implies a scale-down factor, plus additional stalling for architectural virtualization, as detailed in Chapter 5. The FPGA platform specifics are isolated within the Host FPGA Platform layer. In this layer, the user must consider factors such as run frequency, clock, phase-locked loop (PLL), efficient resource utilization for fitting the design, and optimizations for speed. Separating these layers enables targeting different architectures on various hosts while replicating consistent results.

For instance, users can target a high-end FPGA board, in-situ or in the cloud, or a mid-range FPGA evaluation platform. By fitting the corresponding logic layer on each of these platforms, they can execute experimental runs and ultimately observe the same insightful results. These results remain consistent even though the logic model on the mid-range FPGA board will have fewer allocated resources and longer runtime.

4.2.4 Host CPU Options

For the host CPU, we leverage the popular open-source RISC-V soft-core implementations supported as part of the LiteX framework. These FPGA-optimized implementations efficiently utilize the available resources and meet a target F_{max} of 330MHz. The availability of multicore in-order and out-of-order versions and configurable variations allows modeling a whole system at a reasonably high clock speed for target systems with matching RISC-V processors. Alternatively, the user

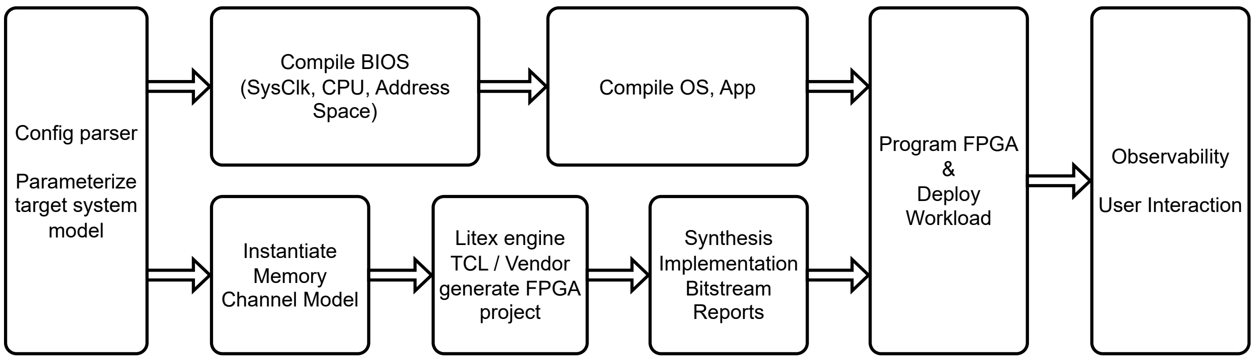

Config parser
Parameterize target system model
Compile BIOS (SysClk, CPU, Address Space)
Compile OS, App
Instantiate Memory Channel Model
Litex engine TCL / Vendor generate FPGA project
Synthesis Implementation Bitstream Reports
Program FPGA & Deploy Workload
Observability
User Interaction

Figure 4.2: Design flow steps from configuration files to logic layer generation, deployment and runtime

can utilize the hardened ARM-based multicore processor system as part of the Zynq UltraScale+ SoC, which LiteX also supports, to emulate a target system with a matching ARM-based processor. At this time, we do not support X86/64 or other ISA implementations.

4.2.5 Design Flow

Our framework implementation enables system generation from configuration files and system integration through an automated script. This script instantiates soft modules for the CPU, memory controller, and memory model, ultimately constructing the final binaries. Concurrently, the framework compiles the necessary tools, BIOS, OS, and application. The user then deploys the gateware, firmware, and software to gather runtime data, as depicted in Figure 4.2.

The design flow illustrates how the framework processes configuration files and host information, leveraging LiteX IP and our mmory emulation model to create a project that accurately represents the target system and maps it to the specified host. The user-defined configuration files, in conjunction with host board details, determine the clock scaling factor between the target system and the logic model. For instance, a 2.5GHz target system can be modeled using a 250MHz miniaturized soft-core on the FPGA, resulting in a 10× scale-down factor. Subsequently, the parameters for each CPU core and memory aspect are supplied to a LiteX system definition script as arguments, which then instantiates soft modules for the processor, memory controller, memory channel model, interconnect, and peripherals. LiteX generates a project structure, and a build TCL script executes the project generation, synthesis, implementation, bitstream generation, and reports logs and resource utilization reports. The final bitstream is deployed on the FPGA, initiating BIOS boot and memory calibration.

LiteX builds the LiteX BIOS, a minimal firmware system serving as a shell for the generated system configuration, providing memory testing and calibration tools, control over memory space and peripherals, and facilitating application or OS boot initiation. The BIOS is also suitable for bare-metal experiments. The UART and Ethernet interfaces serve as valuable system communication channels. The universally present UART enables basic access, file and data transfer, while Ethernet offers faster and more reliable file transfer and network boot capabilities.

Applications can be built separately using RISC-V cross-compile tools and loaded via UART, Ethernet, or SD card. We run Buildroot OpenSBI Linux [81] and other applications, cross-compiling them either using the RISC-V GCC compiler for bare-metal execution or the Buildroot Linux RISC-V GCC compiler for Buildroot Linux execution. Binary files are transferred via UART or Ethernet, and workload deployment resembles interfacing with a remote computer. Users can observe the application through the terminal or trace collection.

4.2.6 LiteX Integration

We encapsulate the top-level MEMulator SystemVerilog module within the LiteX framework [18], enabling its interfacing with the LiteDRAM memory controller and VexRiscV processor system through an appropriate target script. We also define the platform for the Xilinx Alveo U280 FPGA board and validate its functionality using provided utilities and standalone applications. The LiteX framework boasts a vast collection of open-source IPs, compatibility with over 60 FPGA boards from various vendors, support for multiple soft cores, and a comprehensive suite of development, building, system-on-a-chip (SoC) generation, communication, application compilation, and de-

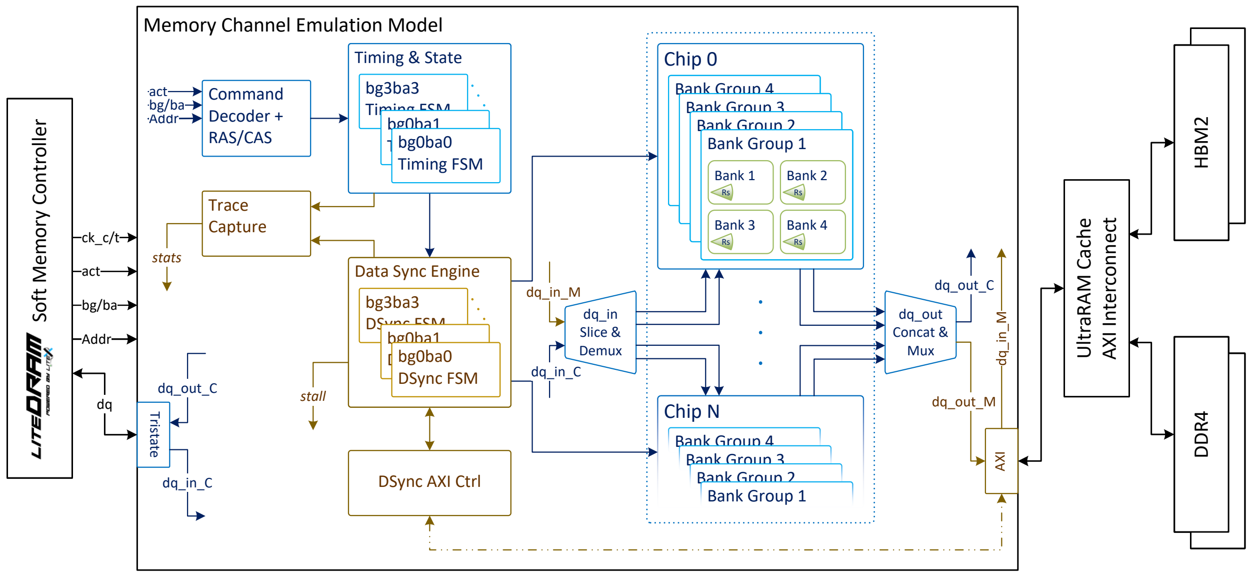


Figure 4.3: Memory emulation model (MEM) block diagram consisting of memory components (blue), auxiliary structures (brown), and peripherals.

ployment tools.

To integrate the memory emulation model into LiteX, we devise a wrapper for the SystemVerilog top module using Migen *Instances*. This wrapper parses configuration files, extracts parameter values, and instantiates a parameterized memory model module. Similarly to the LiteDRAM PHY, we create a ”NoPHY” alternative that connects to the memory model wrapper, bypassing the need for physical FPGA pins wired to board memory. The integration utilizes a 1:1 clock bridge between the memory controller and the memory emulation model, diverging from typical FPGA interfacing with physical memory but aligning with expectations for system modeling and emulation.

4.3 Memory Emulation Model Implementation Details

In this section, we describe the memory emulation model in detail, with the main modules and signals depicted in Figure 4.3. The model comprises of various logic structures, including an interface, command decoder, controls, data bus, and organization. These structures emulate the associated behaviors of memory state and latencies on the FPGA fabric. Additionally, to increase the emulated memory capacity, we leverage the memory resources of the FPGA board, such as dynamic random-access memory (DRAM) and high-bandwidth memory (HBM), through the use of a data synchronization engine (DSync). The MEMulator model is implemented using SystemVerilog Hardware Description Language (HDL) to achieve high performance, which also allows for future low-level modifications at the register-transfer level (RTL).

4.3.1 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). The interface signals are passed to a command decoder module and are translated into memory commands according to memory standards truth tables. The memory commands are used to implement 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.

4.3.2 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 4.4.

The state machine implementation is based on common memory standards [85] while latencies are modeled using counters that, together with the decoded commands, condition the associated state transitions. For example, the state machine switches from *Bank Active* to *Reading* upon

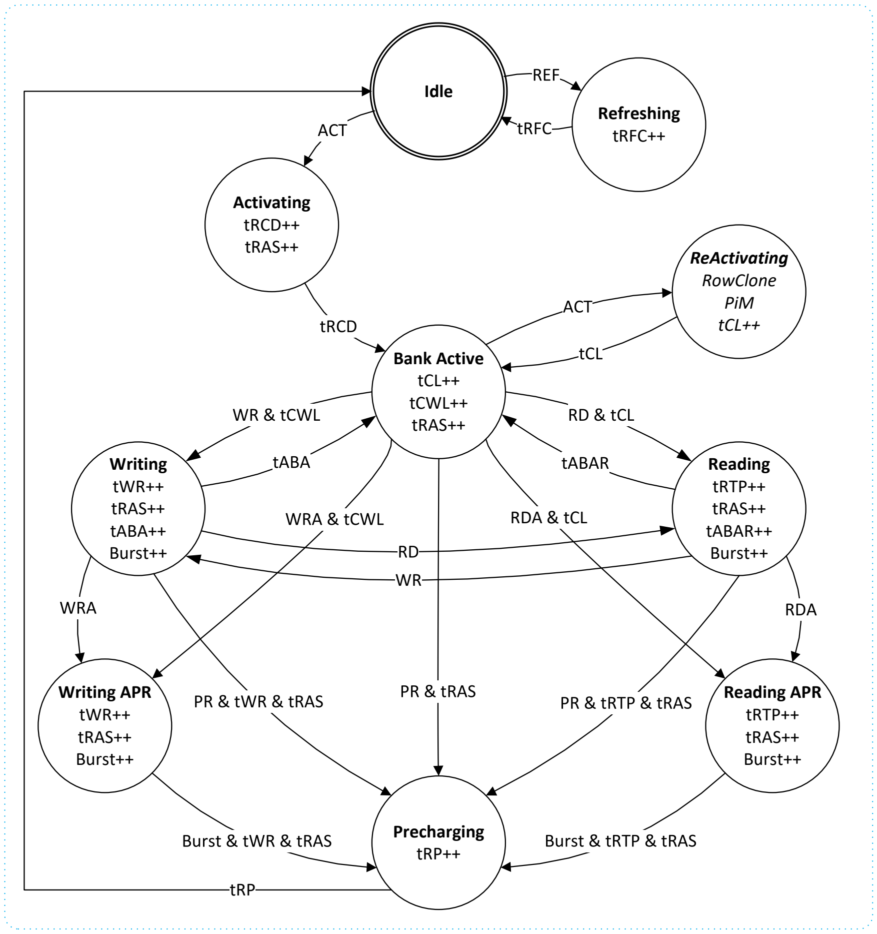


Figure 4.4: Emulation model Finite State Machine (FSM) for modeling bank state and timing based on existing standards and counters.

receipt of *RD* command and expiry of tCL counter. With this approach, the state and latencies of different memory standards and technologies can be modeled. Moreover, it can be easily modified to model additional PiM latencies. We implement the state machine by using FizZim [86], which is an easy-to-use graphical tool for designing state machine RTL modules, thus facilitating ease of development and adaptability of the framework.

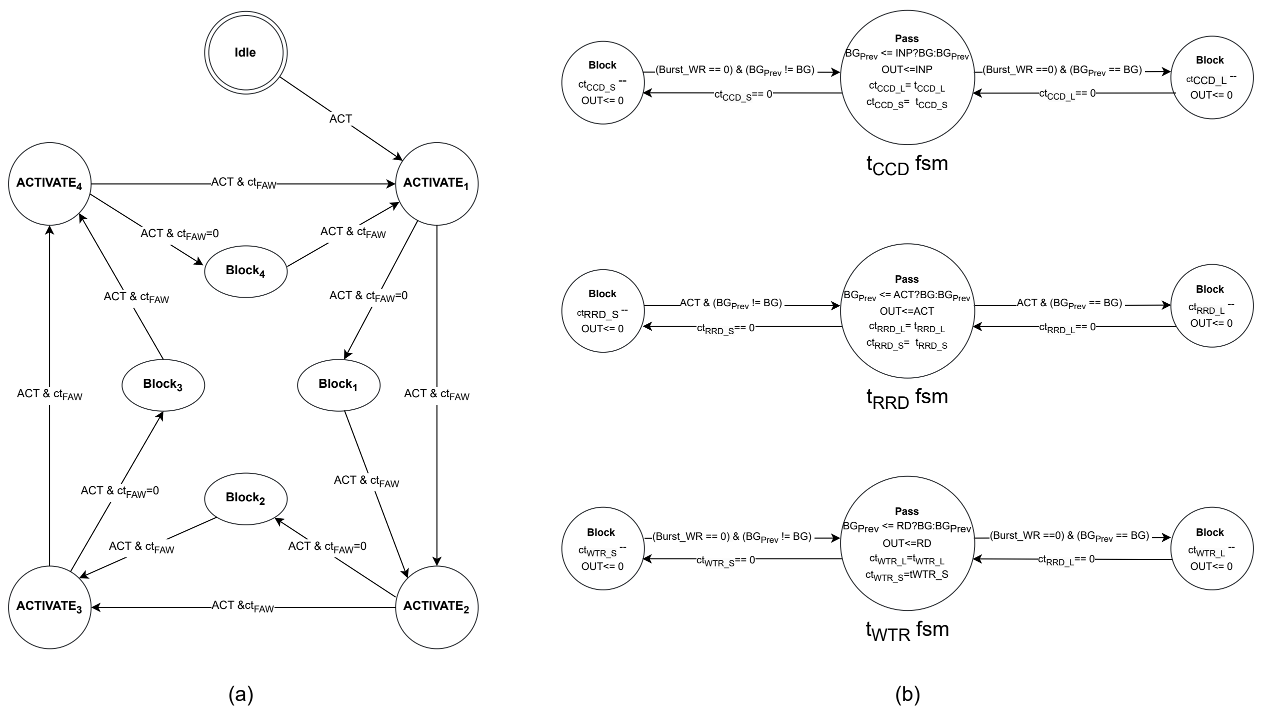


Figure 4.5: FSM model for a) t_{FAW} and b) other ACT global timing rules

4.3.3 Modeling ACTIVATE Timing

The ACTIVATE command initiates the opening of a row within a bank. In relation to the ACTIVATE command, it is important to note the existence of three timing parameters: t_{RRDL}, t_{RRDLS}, and t_{FAW}. In order to issue consecutive ACTIVATE commands to banks in different bank groups or the same bank group, it is necessary to observe the respective row-to-row-delay–short and row-to-row-delay–long. However, the Four Activate Window (t_{FAW}) is a timing restriction, which limits the ability to issue consecutive fifth ACTIVATE commands. The state machines for t_{FAW}, and t_{RRD} are shown in Figure 4.5 (a) and (b), respectively. We model the t_{FAW} state machine in a modular fashion that will allow different activation window modeling for future memory technologies. There is a block state between each ACTIVATE to ACTIVATE state transition that blocks the fifth ACTIVATE command if thet_{FAW} timing is not respected. We use four counters, one for each ACTIVATE state. Once any of the four counter values expire, the value of ct_{FAW} will be high and an ACTIVATE command issued when the ct_{FAW} is high will ensure a successful state transition to an ACTIVATE state.

We also use block state to model the row-to-row-delay, t_{RRD}. A register in the pass state keeps track of the bank group information. When an ACT command is issued and the previous bank group and the current bank group match (mismatch), a state transition from pass to block state occurs and it stays there until the counter value of the row-to-row-delay–long (row-to-row-delay–short) expires, ensuring t_{RRD} is respected while issuing consecutive ACTIVATE command. We use a similar approach to model the t_{FAW} and the t_{WTR} which are shown in Figure 4.5

Finally, the ACTIVATE signal passed to the bank FSM will be an AND reduce between incoming ACT and passing (non-blocking) states.

4.3.4 Memory hierarchy: Rank, Chip, Bank Group, Bank and Subarray

Memory hierarchy, data flow, and data layout are modeled by harnessing most FPGA BRAM resources together with addressing logic. The memory interface data bus bits are sliced over the few Chip modules and further de-multiplexed into bank group and bank modules. Due to the limited BRAM (similarly, LUTRAM) resources on the FPGA, only a tiny part of the total emulated memory capacity can be modeled on the FPGA fabric. The memory emulated on the FPGA fabric, despite its small size, 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 controlled by an auxiliary data synchronization engine (DSync), which also models row subarray membership.

4.3.5 Data Synchronization Engine

The Memory and PiM Emulation Model makes use of a data synchronization engine (DSync) per bank module, which handles the use of bank module rows. The DSync implementation is based on a standard cache design [87] adapted for memory modeling, as depicted in Figure 4.6, with

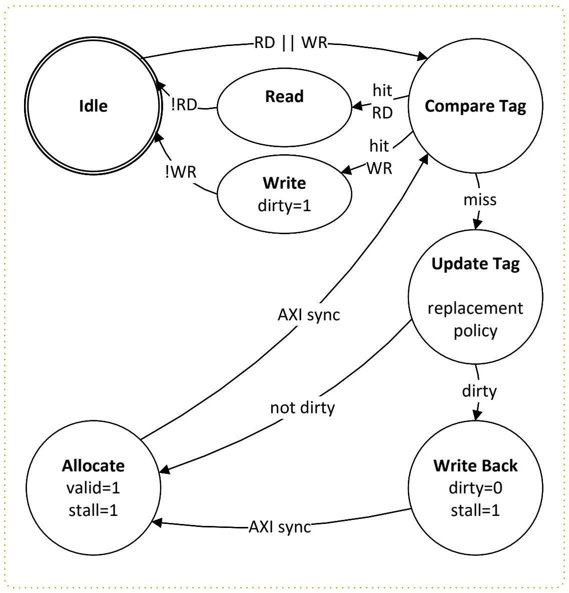


Figure 4.6: Emulation model Finite State Machine (FSM) for bank data synchronization engine (DSync) with FPGA board memory resources.

block size equal to a whole bank module row. The Idle, Compare Tag, Read and Write states, as well as the state transitions, align with states of the bank state machine. This design choice has the advantage of reducing the number of cycles the framework has to stall to synchronize model rows data with board memory resources since the DSync engine will begin processing the miss state while the row is still activating. The DSync begins when a row is activated and maintains it in a hit case until precharged. The miss case time penalty is decreased by overlapping with RCD, CL and RAS delays with the DSync write back and/or allocate time. We select the block size to be equal to a whole row to facilitate emulating PiM architectures taking advantage of data locality by utilizing the entire row [33–35, 39]. The modular design of the framework allows users to modify the block size as desired.

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) in order to prevent the emulated time ticks from running ahead. 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.

4.4 Time Modeling at System Level

The primary challenge is maintaining accurate system-level time synchronization across all components, including processors, caches, buses, memory controllers, memory models, peripherals, accelerators, and upward to the firmware and user interaction level. The primary focus is often on modeling high-performance computing systems, which may exhibit varying operating frequencies. In this section we describe HPC (GHz) target system emulation with FPGA soft cores running at $10\times$ to $100\times$ lower clock frequency.

4.4.1 Time Scaling Frequency

The feasible frequency range for FPGAs is typically between 100 and 330 MHz, with the memory model and LiteX modules successfully tested at the upper limit. Modeling a target system with components at different frequencies can be achieved by identifying the low and high frequencies among the target system components, such as processors and memories, thus determining the range. Subsequently, the appropriate speed for the logic model is determined based on the min and max frequencies, which dictates the system's slowdown factor. An example system is presented in Figure 4.7, illustrating a synthesized logic system featuring multiple CPU cores, caches, intercon-

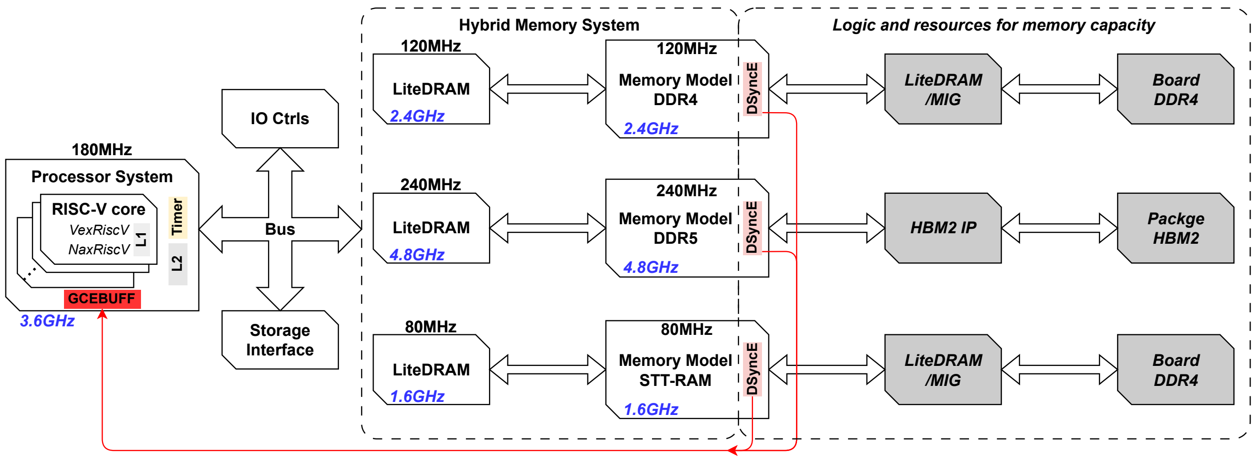


Figure 4.7: System Time Emulation: Depiction of Target System Frequencies (blue) and Corresponding FPGA-based Soft Core Model Frequencies (black) and FreezeTime stalls for Accurate Time Scaling.

nects, and a hybrid memory system consisting of three distinct memory types. The significance of this design lies in the framework's ability to model hybrid memory systems. A hybrid memory system may combine various memory types, such as high-speed but expensive DDR5, more affordable DDR4 for increased capacity, and STT-RAM for persistence. The model utilizes separate instances of LiteDRAM and MEM modules to emulate each memory channel, with each MEM leveraging an on-board memory resource for enhanced capacity. When multiple processors and memories are involved, it is crucial to maintain the slowdown scaling uniformly throughout the system. This factor represents the degree to which the emulated system operates at a slower pace than the actual system.

4.4.2 Virtual Time Modeling

An additional challenge arises from stalling due to memory data synchronization. When one synchronization engine stalls, it necessitates stalling all other components in the system, including the memory model, other memory models, memory controllers, and the processor system. This will further slow emulation time, though improved DSync operation could alleviate this drawback.

4.4.3 Target System Time Emulation

Finally, interpreting the results requires scaling both the bandwidth and runtime. For instance, if a task takes one hour to run on the FPGA with an overall scale factor of 10, the actual runtime in real-life conditions would be only six minutes. User collected traces should include both host time and simulation time, with the latter automatically determined by the framework based on system configuration.

In summary, system-level time modeling faces several challenges, including maintaining accurate time synchronization across all components, operating within FPGA frequency limitations, preserving slowdown scaling, handling stalling, and interpreting the results correctly by applying scaling factors. Addressing these challenges requires a comprehensive understanding of the target system's intricacies and the ability to adapt and adjust models accordingly.

4.5 Modeling Different Memory Types

With the provided full system support (LiteX, VexRiscV CPU, LiteDRAM), users can utilize this model to run heavy workloads that can complete in hours instead of days. MEMulator is challenging the existing simulation approach by developing an emulation alternative that enables faster and detailed design space exploration of different memory types (DDRx, HBM, HMC, STT-MRAM) and even modeling of physical effects. DRAM memories are all based on the same technology principles, with differences only at what is around them. Moreover, emerging memory technologies share a lot of similarities too. We leverage this high similarity between the different memory models to enable emulation of different memory types using a common general parameterizable memory channel model. The base memory channel model implements a DDR4 memory channel, with the ability to customize data layout, latencies, and so on. To modify the DDR4 model for LPDDR users have to instantiate two MEM instances, configure the width to x16, and use a low power FSM. For GDDR, the DDR4 has to be assigned a wide device width and lower number of chips. For HBM the number of MEM instances has to be increased to 8 or 16 to account for

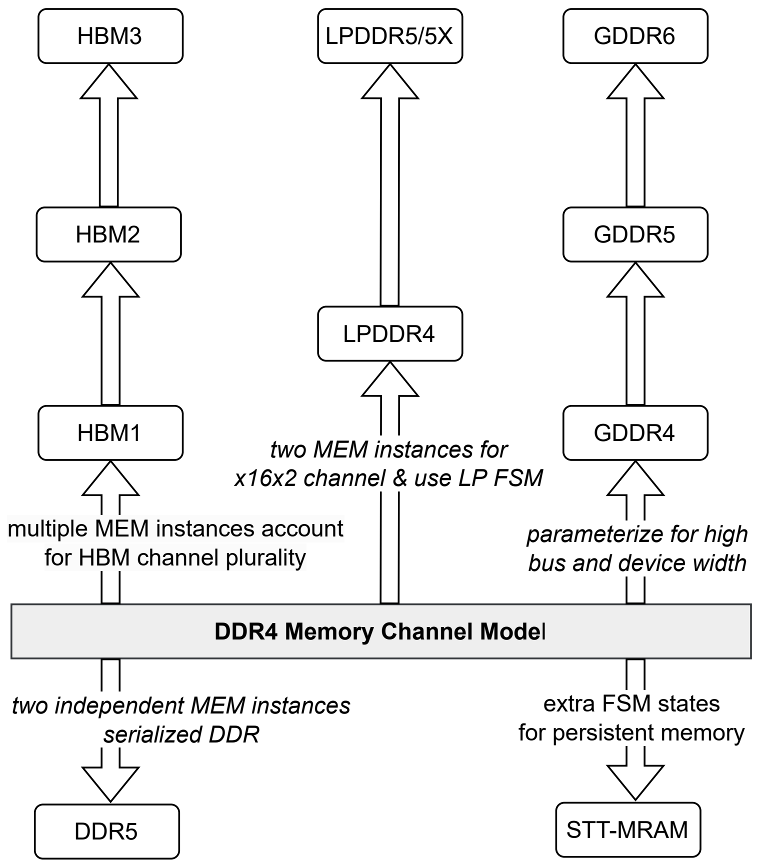

HBM3
LPDDR5/5X
GDDR6
HBM2
GDDR5
LPDDR4
HBM1
GDDR4
two MEM instances for x16x2 channel & use LP FSM
multiple MEM instances account for HBM channel plurality
parameterize for high bus and device width
DDR4 Memory Channel Model
two independent MEM instances serialized DDR
extra FSM states for persistent memory
DDR5
STT-MRAM

Figure 4.8: Using the memory emulation model to generate and emulate different memory types.

independent HBM channels. If necessary, a crossbar can be implemented to enable cross channel data access. The DDR4 model is augmented to support DDR5 channel clock serialization, and two MEM instances will account for the two channels. Finally, for emerging memory technologies, such as STT-MRAM, the FSM has to be modified to use the persistent memory states.

4.5.1 Scope of MEMulator

The scope of MEMulator encompasses prototyping and evaluating what-if scenarios, crafting architectures aimed at reducing memory accesses, fostering the development and experimentation of intelligent memory controllers, modeling physical effects such as aging and RowHammer, and estimating repercussions of DRAM process limitations. Additionally, the memory model is embedded into an entire computing system as part of a flexible, comprehensive memory hierarchy and facilitates the swift and accurate evaluation of high-performance computing systems with real workloads.

A crucial objective of MEMulator is to help guide memory technology development. Within the multitude of potential enhancements to memory systems, only a select few innovations are ultimately assimilated into new standards and consequently adopted into various memory types. Noteworthy instances of such adoption encompass the transition of the bank group concept from GDDR5 to DDR4, and the incorporation of the multichannel interface, as channel splitting, from HBM to DDR5.

Reducing memory accesses can significantly improve both computation performance and energy efficiency. Thus, another key aim of MEMulator is to monitor data transfers, focusing on parameters such as latency and average throughput across varying requests, including those originating from the CPU or GPU. This feature is especially advantageous in the assessment of emerging architectures, such as Processing-in-Memory (PiM), Processing-using-memory (PuM), and Processing-near-Memory (PnM), along with the advanced functionalities of memory controllers.

A vital part of MEMulator's scope is the modeling of memory controller designs. This process is essential for understanding how to use a memory module effectively by maximizing bank activation and refresh interleaving, leveraging active row locality, and ensuring high throughput and low latency. The memory controller deals with transactions from diverse sources, including the host CPU and SIMD-like compute units such as GPUs, accelerators, or vector instruction kernels. These different sources present varied requirements - CPU transactions demand low latency, whereas other sources prioritize bandwidth, highlighting the necessity for the memory controller to maintain a precise balance.

In addition, the memory controller can refine refresh activities and identify RowHammer attacks, along with other tasks related to managing physical effects on memory. Users, for example, can create an error-injection model to imitate aging, RowHammer, or other forms of errors, subsequently experimenting with recovery or mitigation methods. MEMulator provides users with comprehensive component and behavior models, facilitating these experimental procedures.

Emulating the effects of the DRAM fabrication process on the speed and behavior of logic circuits - including registers, addressing, ECC, or other advanced logic - poses a significant challenge. Take, for example, the UPMEM architecture: it shows how the DRAM process's inherent constraints can limit the PiM core's complexity. Crucial factors to consider are switching time, which is generally higher than CMOS, and the reduced number of metal layers that constrain routing options, thereby affecting the design's complexity. These constraints necessitate robust design decisions concerning complexity, pipelining depth, and other elements. In this context, MEMulator is adept at evaluating design prototypes for system-level fit, which involves assessing how well a design performs as part of an entire system.

The final aspect of MEMulator's scope is optimizing a complete computing system's memory hierarchy. This hierarchy comprises L1/2/3 caches, data buses, coherency mechanisms, buffers, memory controllers, a heterogeneous main memory system, and more. By running targeted applications and continuously observing all facets of system behavior, architects are empowered to fine-tune each component. This iterative process allows detecting and eliminating bottlenecks, leading to an optimally balanced system. The result is a memory architecture perfectly tailored from the ground up to meet specific performance needs.

Chapter 5

FreezeTime: System Emulation through Architectural Virtualization

High-end FPGAs enable architecture modeling through emulation with high speed and fidelity. However, the available reconfigurable logic and memory resources limit the size, complexity, and speed of the emulated target designs. In addition to the spatial dimension, this work uses the temporal dimension, implemented with architectural multiplexing coupled with block-level synchronization, to model a complete system-on-chip architecture. Our approach presents mechanisms to abstract instance plurality while preserving timing in sync. We demonstrate this technique by emulating a hypothetical system consisting of a processor and a large SRAM memory. For Linux boot, we measure significant emulation vs. simulation speedup while matching RTL simulation accuracy.

5.1 Using FPGAs to Emulate Large Systems

Full system FPGA-based emulation poses several challenges. The primary challenge is to scale FPGA mapping to extensive and complex target systems. The two main methods for handling this type of scaling are spatially scaling the emulation across multiple FPGAs and temporally scaling instances through time domain multiplexing.

5.1.1 Spatial Scaling across Multiple FPGAs

One method to emulate large systems is to partition the design across multiple FPGAs. This approach introduces numerous challenges and extra steps, such as partitioning the design into small sections that do not exceed the utilization of the individual FPGAs, matching the links between the sections with the physical connections between FPGAs, synthesis, implementation, generation, and handling of multiple bitstreams, etc. [88, 89]. FreezeTime can be combined with this approach to simplify the partitioning process.

5.1.2 Time Domain Multiplexing

An alternative method to emulate large systems is to take advantage of their granular structure and virtualize the individual building blocks (instances) by mapping them to a subset of physical blocks. The virtual instances' emulation will occur according to a scheduling scheme (often round-robin). A trade-off to this method is the increased time required to emulate multiple instances and the need for a dedicated mechanism to store, hold and load the state of the multiple instances. In particular, for the case of memory, it is challenging to achieve faster emulation speed and emulate memories larger than the memory resources available on the FPGA. For example, emerging architectures feature a hybrid mix of memory flavors (DDR6, DDR5, DDR4, HBM, persistent memory, etc.) or accelerators with large scratchpad memory. Modeling the capacity and speed of such systems using Block RAM and FPGA board DDR3/4 cannot be performed accurately with simplistic mechanisms such as delay elements and frequency scaling. For computation, the challenge is to scale the number of instances (e.g., cores or PEs) from few to many and evaluate parallel execution with workloads of interest. Both memory and compute emulation can be augmented using the temporal dimension, namely system stall while preserving state and time synchronization and while virtualizing the memory address space or compute instance plurality.

Other approaches, such as hybrid software simulation combined with FPGA-based emulation, or FPGA-accelerated simulation, manifest lower fidelity and simulation speed than the hardware emulation used in this work.

5.2 FreezeTime Mechanism and Implementation

In this work, we leverage the property that, during FPGA-accelerated architectural emulation, the simulated clock cycles need not coincide with the host FPGA clock cycles. Henceforth in this chapter, we refer to the time seen by the emulated target system as *emulation time* and the time seen by the host fabric as *host time*. We develop a technique called *FreezeTime* to facilitate coarse-grain architectural virtualization. The FreezeTime technique preserves emulation time, synchronized throughout the system, which we achieve by stalling the remaining architecture modules during the runtime of a virtualized module. The stall mechanism has to store the modules state, halt the execution over many clock cycles, then fetch back the state and resume execution. The mechanism requires a shared memory space and additional control logic, such as halting the clock driving the model logic (for example, using a gated clock enable buffer) or inserting auxiliary control logic that prevents updating register values.

The FreezeTime mechanism adds to the toolbox of time modeling. Most commonly, time is modelled using (a) frequency scaling, that allows modeling the baseline bandwidth or throughput, in conjunction with (b) delay elements that model latency, which are often approximated to fixed parameters, runtime variable values, or more elaborate timing models [11, 90]. While helpful, with these two techniques the logic model still adheres to the system clock and time has to pass uninterrupted. With (c) FreezeTime the user has an additional technique that brings the flexibility to model more behavior, using auxiliary clock cycles, without affecting emulated time correctness. In the following, we describe how the FreezeTime mechanism can be implemented in an emulation framework.

5.2.1 Target System Mapping for Virtualized Emulation

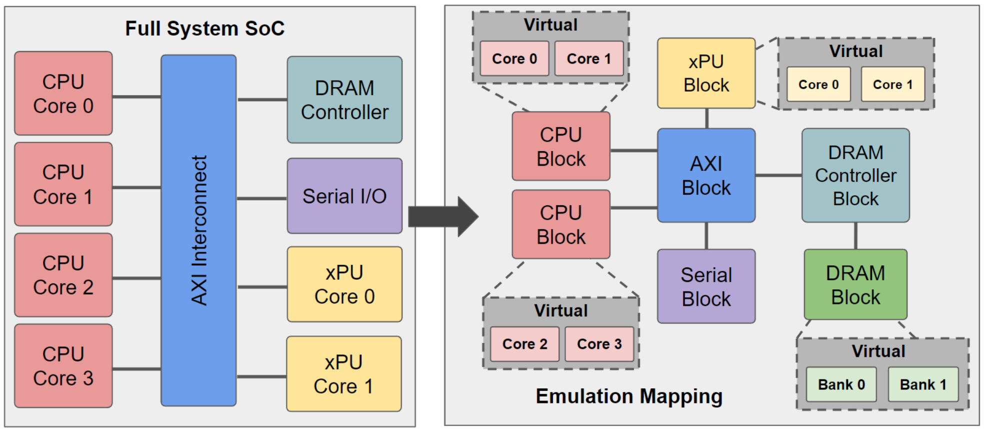


Figure 5.1: FreezeTime top-level diagram with physical to virtual mapping.

To facilitate system-level emulation, we first spatially partition the emulated system by grouping the identical instances into blocks and then connecting the blocks through dedicated interconnects. Each block can then implement its respective instances through virtualization (in time). This concept is demonstrated by an example SoC system in Fig. 5.1. The target system SoC consists of several CPU and accelerator (marked as *xPU*) instances, memory controller, and peripherals. To emulate it, we use a logical system in which the CPU cores and accelerators are virtualized within their respective blocks. We illustrate the virtualized CPU cores across two physical CPU blocks to demonstrate flexibility and balancing between physical and virtual resources. The DRAM is represented by a soft FPGA-based DRAM model in which the memory capacity is virtualized due to FPGA resource constraints. Following the spatial partitioning, we virtualize the contents of the emulation blocks independently, employing a system-wide method for communicating data accesses and synchronization. This communication and synchronization method requires a generic template for virtual instances.

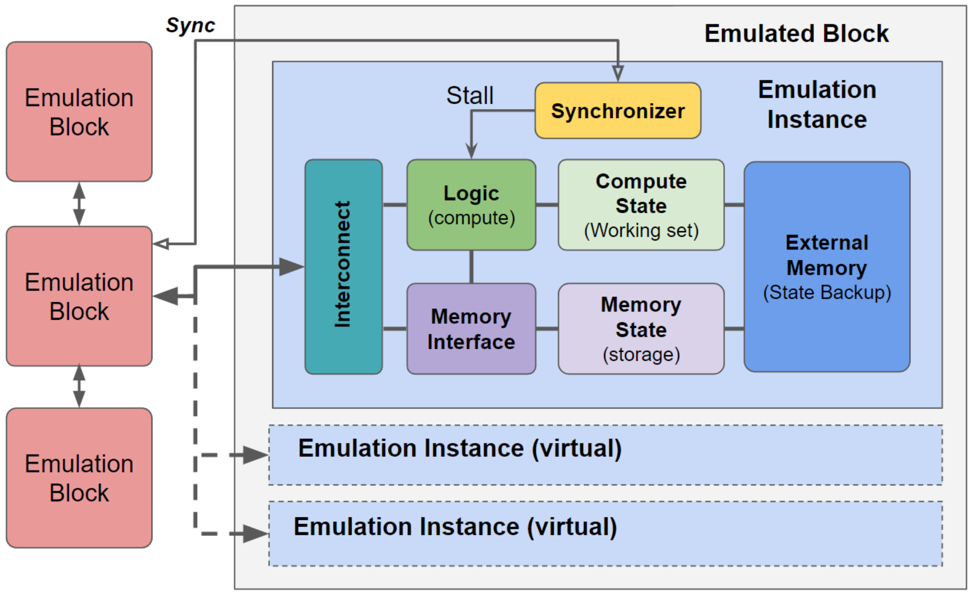


Figure 5.2: Architecture partitioning in *FreezeTime* where emulation blocks group duplicate instances and virtualize them using a template.

5.2.2 Generic Template for Virtual Instances

The internal composition of an emulation block is shown in Fig. 5.2, where several virtual instances are mapped to a common physical emulation instance using time-domain multiplexing. In order to account for modeling both logic, memory, and interconnect, each physical instance is broken down into five main components: 1) the logical emulation and its associated state, 2) the memory emulation and its associated state, 3) the interconnect emulation, 4) a state backup in external memory, and 5) a synchronizer for maintaining synchronization across all emulation blocks. Most emulation blocks will only contain the components related to the virtualized functionality. For example, an AXI interconnect emulation block may only contain an interconnect emulation component and a synchronizer. On the other hand, a CPU block with a large L2 cache may contain all five components. Fig. 5.3 depicts how the above-mentioned components are realized in the virtual and physical form.

Logic Emulation

The role of the logic component is to emulate the computation behavior of virtualized instances, such as CPU cores or accelerators. A subset of instances is implemented on the FPGA and used to execute the operations of all virtual instances. For virtualization, upon a call for execution, operations are queued for emulation on the first available physical resource, as shown in Fig. 5.3 (a). Execution begins with the complete fetching of the operands, resolving any pending data dependency. During execution, if an operation is emulated following the first batch, the module issues and maintains a stall signal until the result is returned and stored. The stall signal freezes the time and activity of the remaining system, preserving system-level time and activity in sync, as shown in Fig. 5.3 (c).

At any given time, the register values define the state of the computation instances in the system. Preventing the update of the register values saves and preserves the instance state during a stall and halts execution. This can be achieved by delaying the clock edge event or by inserting MUX gates to control the register value update. The loaded state allows resuming runtime upon stall termination. A possible hazard is when data is being burst from outside, such as during an interrupt, in which case it is falsely ignored. Since the FreezeTime approach is to stall all the systems but the virtualized instance to be emulated, the data transmission will be stalled at the source. Resuming execution after a stall relies on the state correctly saved throughout the system.

Memory Emulation

The role of the memory emulation component is to virtualize aspects of a memory hierarchy that are unfeasible to map on an FPGA, such as latency and capacity. We achieve this by controlling the memory operation timing as observed from the system. For example, to emulate a large SRAM or DRAM flavor using the board memory resources (BRAM, DRAM, Flash), the board memory latency can be masked by freezing system time. A general mechanism to emulate memory aspects is to dedicate a small amount of memory local to the FPGA with custom delay and bandwidth and

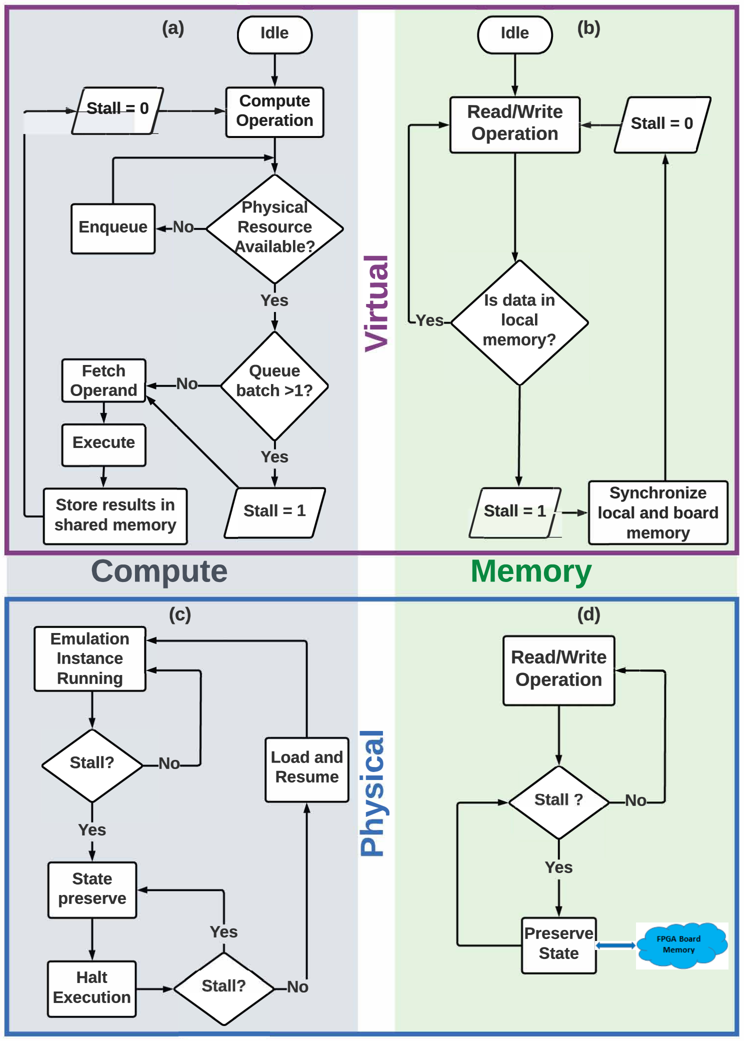


Figure 5.3: Virtual (a and b) and physical (c and d) flowchart of *FreezeTime* components. Compute (a and c), and memory (b and d) constructs are shown with gray and green backgrounds, while the virtual and physical mapping are separated with purple and blue borders, respectively.

use a data synchronization mechanism to expand the emulated capacity to board memory resources. The implementation is similar to a standard cache. If the memory operation data/location is not locally cached, the data synchronization mechanism will sync it with the board memory while issuing a stall signal, as shown in Fig. 5.3 (b). A good example of the above-explained data synchronization mechanism is implemented in PiMulator [90].

Stalling the physical memory model preserves the model state, ensuring that the custom delays observed by the system are consistent. The stall signal does not affect the physical board memory controller.

Interconnect Emulation

The interconnect emulates interfaces between the compute and memory components. This component is assumed stateless and is emulated as part of the compute and memory block interfaces. More advanced communication elements, such as an NoC router, are emulated with computing blocks.

State Backup

The state backup is the primary mechanism for preserving virtual instance states and complete memory contents. This is implemented using an FPGA board memory resource and is shared across multiple emulation blocks. The emulation framework can use the backup memory to store the internal register values of virtual compute instances and load them each time the instance is emulated. For memory emulation, the framework can use this resource to hold the virtual memory data, as shown in Fig. 5.3 (d). Since the backup memory is shared across the many emulation blocks, instances can be virtualized on any available supporting physical modules.

Synchronizer

The role of the synchronizers is to maintain the emulation time and state in sync across virtual instances and other physical emulation blocks. The synchronizer components collect the stall signals from virtualized instances and pass the logical value to the remaining system blocks via a dedicated channel (not shown in Fig. 5.3). Similarly, it receives the synchronization stall signal at the emulation instances and passes the signal to be used by the FreezeTime mechanism. The synchronizer accumulates the stalls issued during virtual instance computation and virtual memory data sync and issues the stall signals at the physical resources modeling the memory and compute, respectively.

We further augment the synchronization technique to facilitate scaling architecture virtualization from several to many virtualized instances. For this, we equip each emulation instance synchronizer with a timer that keeps track of a given instance's execution point. By comparing the timer values of virtualized instances and the rest of the system, we confirm synchronization and can repeat the virtualization over vast virtual architectures.

5.2.3 Sync-by-stall

A key component of the FreezeTime mechanism is the ability to preserve time synchronization by stalling the emulated system. When emulating an architecture with a mix of physical, virtual active, and virtual queued-for-execution instances, the state and time stamp of the latter will deviate and require synchronization. By stalling the system, we allow the virtual queued-for-execution instances to ”catch up” with the system time and state. Stalling can be implemented by generating dynamically stalled clock signals and using them to drive system components. On FPGAs, the recommended practice is to use a Gated Clock Enable Buffer (BUFGCE) since the generated clock signals map to clock tree resources. Alternatively, the stall mechanism can be implemented as part of the design logic. Many compute modules, such as processors, already include stall mechanisms that facilitate debug, interrupt, or hibernate capabilities and can be utilized for FreezeTime. Stalling system time and state augment an emulation framework with the time-multiplexing capability. Architectural time-multiplexing facilitates the emulation of numerous virtual instances on a subset of physically synthesized modules, thus enabling the emulation of larger systems.

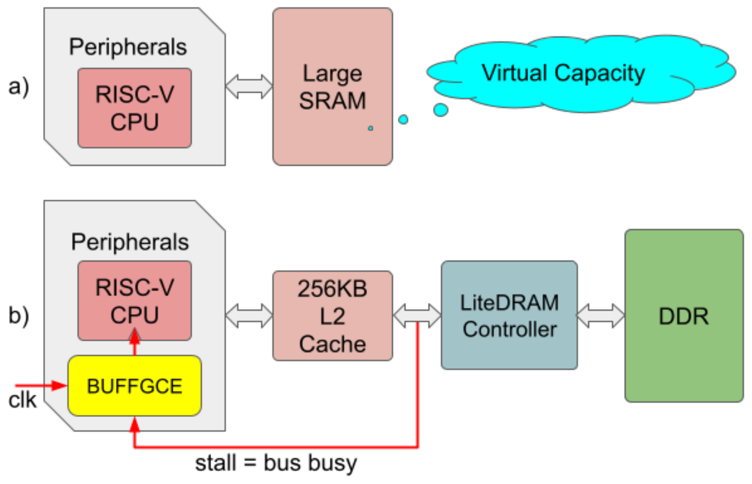


Figure 5.4: Target system to be emulated and b) Logic model that freezes the processor during L2 cache miss state, hiding the auxiliary data sync time, effectively virtualizing large SRAM capacity.

5.3 Experimental Results and Discussion

5.3.1 Experimental Setup

To evaluate the efficacy of FreezeTime, we consider the hypothetical case of a single processor coupled to a large SRAM memory (too large to fit on an FPGA), with a read and write latency of 1 clock cycle, shown in Fig. 5.4(a). While this may be an overly simple example, it serves as a representative case in which more complex system configurations can be inferred. We accomplish this task by modifying an SoC generated with the LiteX framework [18] running on a Xilinx VC707 FPGA (Virtex-7). The framework modifications consist of state machines that monitor and respond to L2 cache to LiteDRAM memory controller bus activity by issuing processor stall signals.

Stalling the processor system allows for auxiliary clock cycles for synchronizing data between L2 and board DRAM memory. Meanwhile, the software on the processor perceives the memory system as a large SRAM. We use the default Linux-capable VexRiscV processor [16] combined with the onboard DDR3 memory via the standard LiteDRAM controller (with accompanying L2 cache), as shown in Fig. 5.4(b). By “hiding” the LiteDRAM controller bus latency, we emulate the behavior of a hypothetical 1 GB SRAM memory. To achieve this, we generate a stall signal based on the LiteDRAM controller bus activity and feed the signal into the processor stall mechanism.

5.3.2 Freezing Time Demonstration

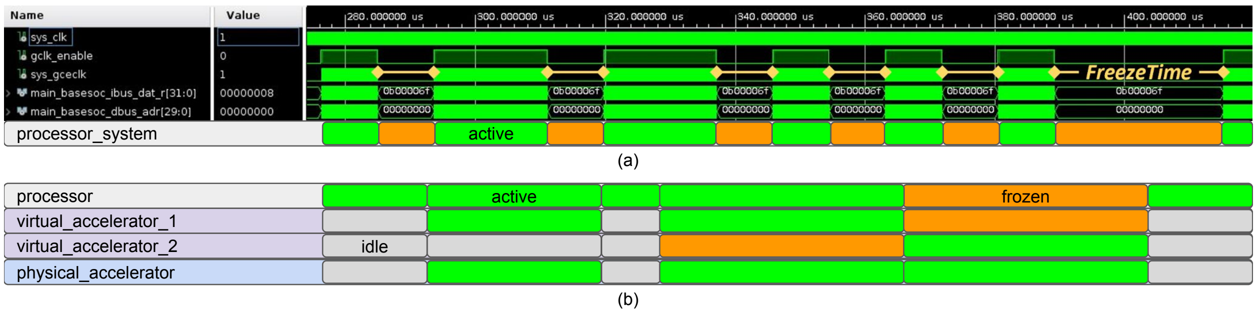


Figure 5.5: Architectural virtualization with time multiplexing: (a) Waveform of stalled VexRiscV CPU using a gated clock enable buffer. (b) Runtime mapping of virtual compute instances to a physical module preserving time in sync.

The challenge is to freeze, then unfreeze, on command, a large portion of an emulated architecture without loss of handshaking or bus data. We ensure that the expected behavior occurs, first in simulation and then by observing signals of interest on hardware using a Xilinx Integrated Logic Analyzer (ILA). A waveform containing several signals of the simulated testbench is shown in Fig. 5.5 (a). All system peripherals are driven by the global system clock signal *sys_clk*, except for the processor system. The *sys_clk* is passed via a clock gate (*BUFGCE* instance), enabled by the *gclk_enable* signal, to generate the gated clock signal *sys_gceclk*, which drives the processor system. As expected, we observe that the processor iBus and dBus are active during *sys_gceclk==1* and stalled during *sys_gceclk==0*. We further validate correct operation by observing that the processor stalls do not miss incoming bus data or signals and can resume execution once the processor is no longer stalled. We then further verify the correct behavior with complete program execution, such as BIOS and Linux boot, in simulation and on hardware with random and bus-generated stalls.

Similarly, in Fig. 5.5 (b), a time-multiplexed system is depicted, comprising a processor and two identical accelerator instances, both of which are supported by a single physical accelerator module. In the event that both virtual accelerator units are busy, the FreezeTime mechanism must be employed. The operand data of *virtual_accelerator_2* is stored in memory and queued for execution while *virtual_accelerator_1* and the processor execute as expected. Once the task on *virtual_accelerator_1* is complete, the *physical_accelerator* becomes available, and the task on *virtual_accelerator_2* resumes, while activity and time on all remaining system modules are frozen.

5.3.3 Benchmark and Run Environments

For comparing FreezeTime against traditional simulation methods, we consider booting an operating system as a representative benchmark. Booting an operating system can stress various parts of the complete system, including the memory subsystem. These reasons make booting an operating system an ideal test case to verify the correct memory modeling in our evaluation system.

To perform this benchmark, we boot the ubiquitous Linux kernel on an FPGA, including our evaluation system, with and without FreezeTime implementation. We verify the correct behavior with complete program execution, such as BIOS and Linux boot, in simulation and on hardware with random and bus-generated stalls. Given the nature of the FPGA, we measure the boot time by counting fabric cycles between two triggers generated by the kernel debug printouts to the attached serial port. Linux Boot time is typically measured with respect to the login prompt; however, we instead define the measurement between the initial boot-loader hand-off message and the execution of the init process. While this is a non-standard approach, the init process message is associated with the final system time timestamp during the boot process, thus providing a consistent measure of simulation time across the various platforms benchmarked in this chapter.

In addition to running in FreezeTime, we also boot Linux on a FireSim [10] single-node simulation instance. We configure a system consisting of a Rocket Chip [91], a large L2 cache, and a timing model for the DDR4 memory. This system does not use FreezeTime; however, it is useful for evaluation as it resembles a hybrid emulation system.

We also benchmark against two traditional software simulators: gem5 [61] and Verilator [92]. The gem5 simulator is a state-of-the-art architectural simulator that uses a functional timing (FT) model approach to simulate both the processor and memory system. The gem5 target system consists of a standard single 64-bit RISC-V core interfacing with a large L2 cache and a simplified DDR3 memory model. The stock Linux image for gem5 included additional features necessary for a software simulation that increased the total boot time. We adjusted the measured host and simulation duration to exclude the extra latency. Verilator, on the other hand, is a logic simulator that accurately and faithfully simulates the synthesizable Verilog hardware description. For the Verilator simulation, we use the LiteX framework to generate an exact Verilog implementation of the target system consisting of a VexRISCV processor coupled with a large SRAM. Benchmarking against gem5 and Verilator gives us an indication of evaluation performance (host time duration) and accuracy, indicated by consistency in simulation time duration.

In all cases, the system clock frequency was set to 150 MHz. Both gem5 and Verilator runs were performed on a machine with two 8-core Intel Xeon Silver 4208 2.1GHz processors and 256GB DDR4 memory running Ubuntu 18.04.5 LTS. The additional resource utilization of FreezeTime for this example setup was under 2% LUTs/FFs since we only use a gated clock enable buffer, control state machines, and observability counters.

In addition, we also run different memory access patterns using Hopscotch [41] benchmark suite. Hopscotch is a benchmark suite that comprises of data-intensive kernels which are designed to issue various types of traffic such as read-only, write-only, and mixed traffic, each with different access patterns. These access patterns may include sequential, random, strided, tiled, or even a combination of these. This diverse range of access patterns makes Hopscotch an ideal tool for identifying the performance bottlenecks of a system. With Hopscotch, users can easily pinpoint the areas where the system is struggling and improve its overall performance. We run selected bandwidth Hopscotch kernels inside Linux on the same logic system on the Xilinx VC707 FPGA, with and without FreezeTime, and observe the impact on the measured bandwidth utilization.

5.3.4 Results and Interpretation

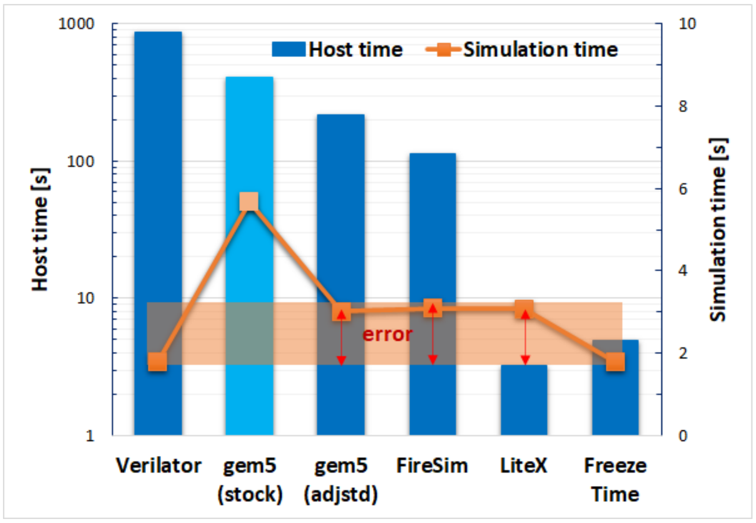


Figure 5.6: Experimental results showing host and simulation time for Linux boot for different evaluation approaches. With FreezeTime, we achieve fast runtime and observe a similar simulation time as Verilator RTL simulation.

We summarize the measured Linux boot results in Fig. 5.6, showing both the host time duration for the different platforms and their respective simulation time duration, and the measured Hopscotch bandwidth utilization in Fig 5.7. We expect the total simulation time to be consistent, given that the target systems and workload are the same. Therefore, the measured simulation times serve as an indirect accuracy assessment. The Verilator RTL is the most accurate since it faithfully simulates the exact target system.

The first observation is that using time multiplexing for architectural virtualization re-introduces additional latency. The results show a $\approx 53\%$ increase in host time between the LiteX system and FreezeTime. However, the added latency is still in the realm of FPGA emulation, with the host time duration much below Verilator RTL simulation ($170\times$), software simulation ($40\times$), and FPGA-accelerated simulation ($20\times$). The latency penalty will increase linearly with the amount of architectural virtualization. For memory capacity, the penalty is fixed per each cache-miss RD/WR operation. For compute instance virtualization, the penalty will depend on the ratio of physical to

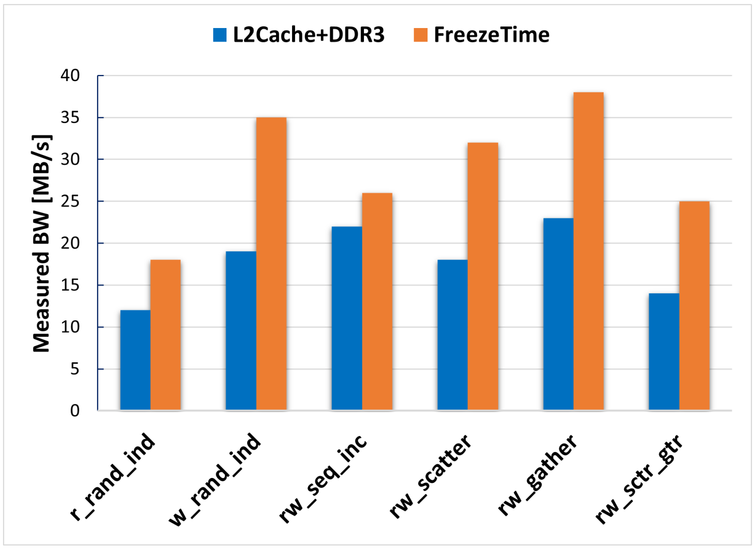


Figure 5.7: Experimental results showing higher measured memory bandwidth utilization when running selected hopscotch bandwidth kernels on the FreezeTime versus the LiteX and system.

virtual and the ratio of computing time to total duration.

Second, we observe that the FreezeTime and Verilator RTL simulation's total simulation times are consistent, demonstrating that the FreezeTime technique is cycle-accurate. In the experiments, we compared the waveforms and ensured that the system stalls entirely hide the auxiliary clock cycles.

Last, for bandwidth utilization, as can be seen in Fig. 5.7, we observe a consistently higher measured bandwidth utilization for the FreezeTime system. In the conventional system, the VexRiscV processor issuing memory commands interfaces with a DDR3 memory via a tiny L2 cache and LiteDRAM memory controller. During cache misses, the memory access latency causes a penalty on the overall bandwidth utilization. With FreezeTime, the processor system, including the timer, is frozen during these DDR3 memory access intervals. Therefore the measured bandwidth utilization is equivalent to the hypothetical case of a processor coupled with a large SRAM memory. Not only does the modeled SRAM size become effectively equal to the DDR3 capacity, but the measured results represent the processor core performance unaffected by host platform or logic model limitations. In this example the user can thus emulate and quickly evaluate processor performance for low operation intensity.

5.4 Related Work and Adoption Opportunities

Here we describe several state-of-the-art emulation frameworks for modeling next-generation high-performance computing systems and explain how FreezeTime technique can enhance their capabilities. LiME [11] employs a hard processor system and reconfigurable accelerator interfacing with the DRAM memory to emulate HPC systems. The FPGA board resources limit the target system processor topology, accelerator count, memory capacity, and other modeled aspects. In addition to using frequency scaling and runtime configurable delays to model the target system frequency and different memory flavors, the FreezeTime technique can expand emulation capability by virtualizing accelerator instances and expanding the memory hierarchy. Similarly, MEG-HMC [19] and MEG-HBM [12] employ soft RISC-V cores, an adaptable memory controller, near memory accelerator logic, and the HMC/HBM memory to emulate HPC systems featuring HMC/HBM. Here, in addition to spatial limitations, only the memory flavor of the host FPGA board can be emulated in the target system. By adopting the FreezeTime mechanism, we can reshape the host physical memory to render aspects of other, different memory flavors using auxiliary state machines and the sync-by-stall mechanism. FireSim [10] employs FPGA-accelerated simulation models for Rocket/Boom cores, memory, and network peripherals to emulate the activity of nodes in a network. Scaling to vast sizes is achieved by partitioning the network over multiple FPGA boards in the cloud. Conversely, with the FreezeTime mechanism, a single FPGA board can host numerous virtualized nodes and network components.

Several works similarly make use of time multiplexing for architectural virtualization. HAsim [93] presents a fine-grained time-multiplexing scheme to model a multicore processor and an on-chip network. In contrast, FreezeTime adopts a coarse, module-level, and operation-level time-multiplexing approach interfacing with a global synchronization mechanism that spans a heterogeneous computing system and memory hierarchy. ProtoFlex [94] employs "transplanting," a hybrid simulation-emulation technique where the FPGA emulates only the common-case logic while the rare, complex behaviors are performed in software simulation. RAMP Gold [22] simulates multicore systems using a functional-timing model efficiently mapped on FPGA. The FreezeTime mechanism facilitates synchronization between the software-simulated and FPGA-emulated regions of a hybrid system and provides additional flexibility to functional-timing units.

Unlike the frameworks mentioned above that employ a miniaturized, abstracted version of the target design, industrial emulation tools such as Cadence Palladium, Siemens Veloce, and Synopsys ZeBu [70] are developed with the purpose of hardware verification prior to fabrication. FreezeTime can help virtualize instances to fit extensive and complex designs on FPGA resources.

Chapter 6

PiMulator: Fast and Flexible Processing-in-Memory Emulation

PiMulator leverages the memory emulation model presented in chapter 4 together with the system-level emulation features and FreezeTime mechanism from chapter 5 and augments the infrastructure with support for PiM kernel logic throughout the memory data hierarchy.

6.1 Proposed FPGA-based Emulation Model

In this section, we describe the proposed memory and PiM emulation model shown in Figure 6.1. Similar to the emulation model described in chapter 4, 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. The following subsections describe each

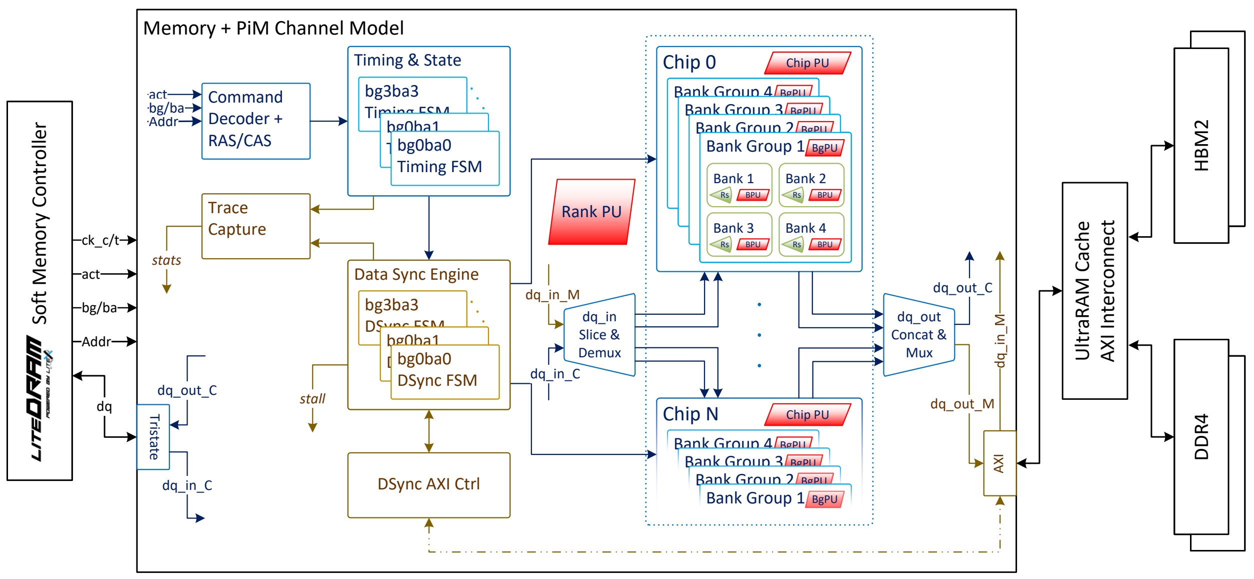


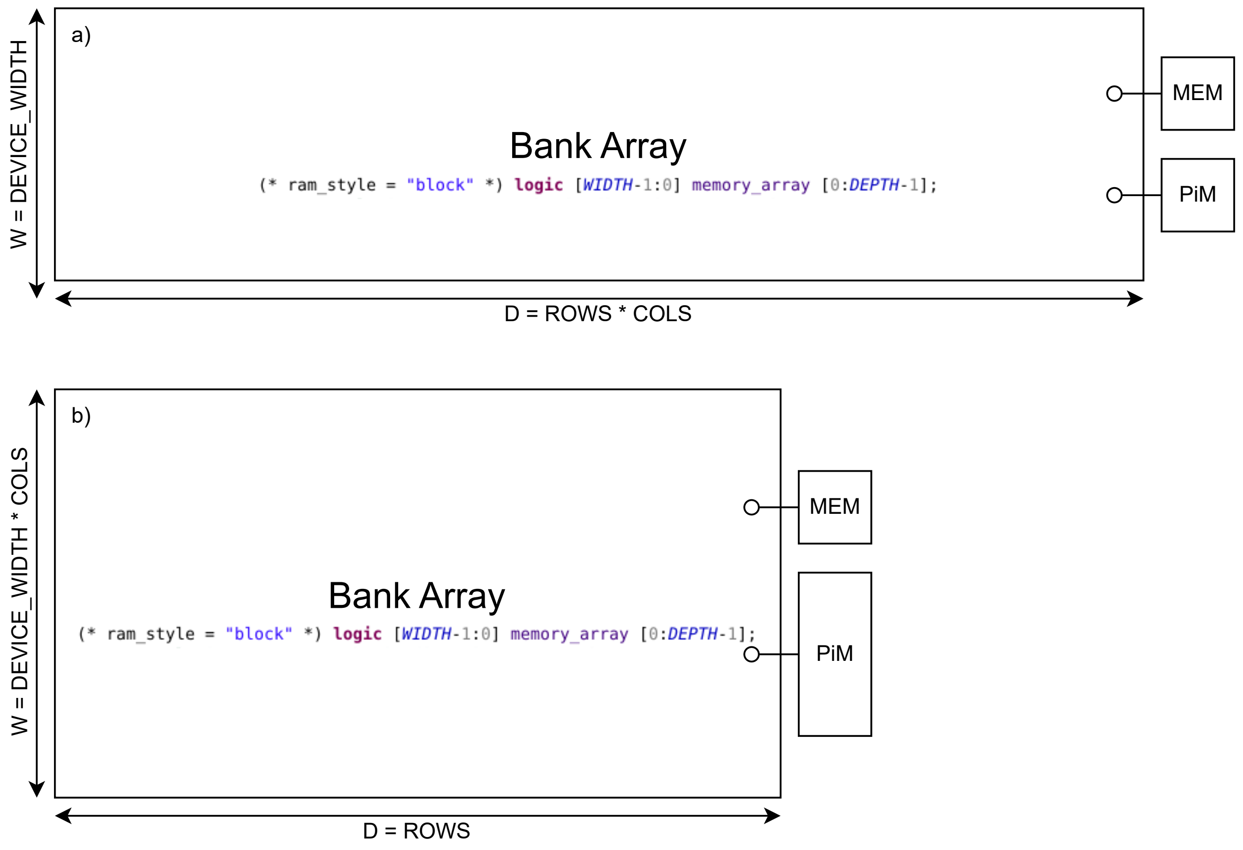
Figure 6.1: Memory and PiM emulation model block diagram consisting of memory components (blue), PiM processing units (red), auxiliary structures (brown), and peripherals.

of the PiM components in detail.

6.1.1 Bank array model for memory and PiM

We present two distinct bank array shapes, as illustrated in Figure 6.2. Each array is implemented utilizing a true dual-port RAM module, where one port is dedicated to the DIMM interface bus and the other exclusively caters to the PiM kernels. The first array shape, characterized by a width of $DEVICE\_WIDTH$, is tailored for conventional memory operations, such as burst read and write of a single row-column word per clock cycle. Addressing is achieved by concatenating the row and column indices. This configuration restricts the PiM kernels to accessing only a single word of data at a time, which is suitable for architectures with PiM kernels external to the bank.

In contrast, the second array shape facilitates access to the entire row buffer, rendering it compatible with a wide range of PiM architectures in which the logic is either integrated within the array, aligned with the row buffer, or situated in proximity to the array. To implement this configuration, we change the array width to $DEVICE\_WIDTH * COLS$. We then configure the MEM port



```
(* ram_style = "block" *) logic [WIDTH-1:0] memory_array [0:DEPTH-1];
```

```
(* ram_style = "block" *) logic [WIDTH-1:0] memory_array [0:DEPTH-1];
```

Figure 6.2: Comparative Bank Array Shapes for Memory and Processing-in-Memory (PiM) Data Storage Architectures. (a) Depicts the array shape tailored for standard memory operations and PiM outside the Bank, while (b) illustrates the array shape that enables access to the entire row buffer, making it suitable for various high-bandwidth PiM architectures.

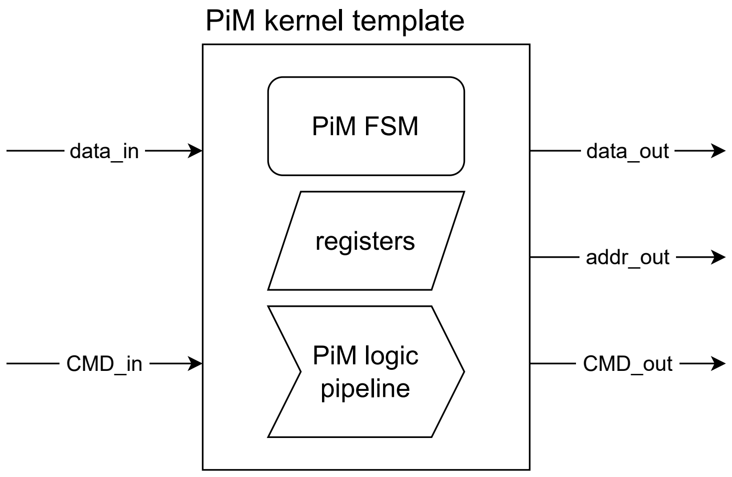


Figure 6.3: PiM kernel minimal logic template.

to solely affect a single-word portion of the interface, while the PiM port can leverage the full interface data.

6.1.2 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 data from the bank module rows that the parent module encompasses. After executing the logic, the results are written back to the bank module rows. 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 [36, 39, 95].

6.1.3 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 hit rate, etc. The model can be interfaced with a processor system or PCIe DMA via a memory controller. We verified the operation of multiple model configurations under possible usage scenarios such as sequential and random burst read/write and bank interleaving and ensured correct behavior.

6.1.4 Integration with LiteX

We wrap the top PiMulator System Verilog module in LiteX [18] and interface it with the LiteDRAM memory controller and VexRISC-V 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.

6.1.5 Strategies adopted to provide increased usability

We opted to implement the memory and PiM logic in SystemVerilog, a low abstraction HDL that preserves complete hardware control for hardware modeling. Nevertheless, for increased usability, the framework provides a rebust and parameterized memory model, PiM kernel templates, FizZim GUI approach for maintaining the FSMs and automated system generation with LiteX.

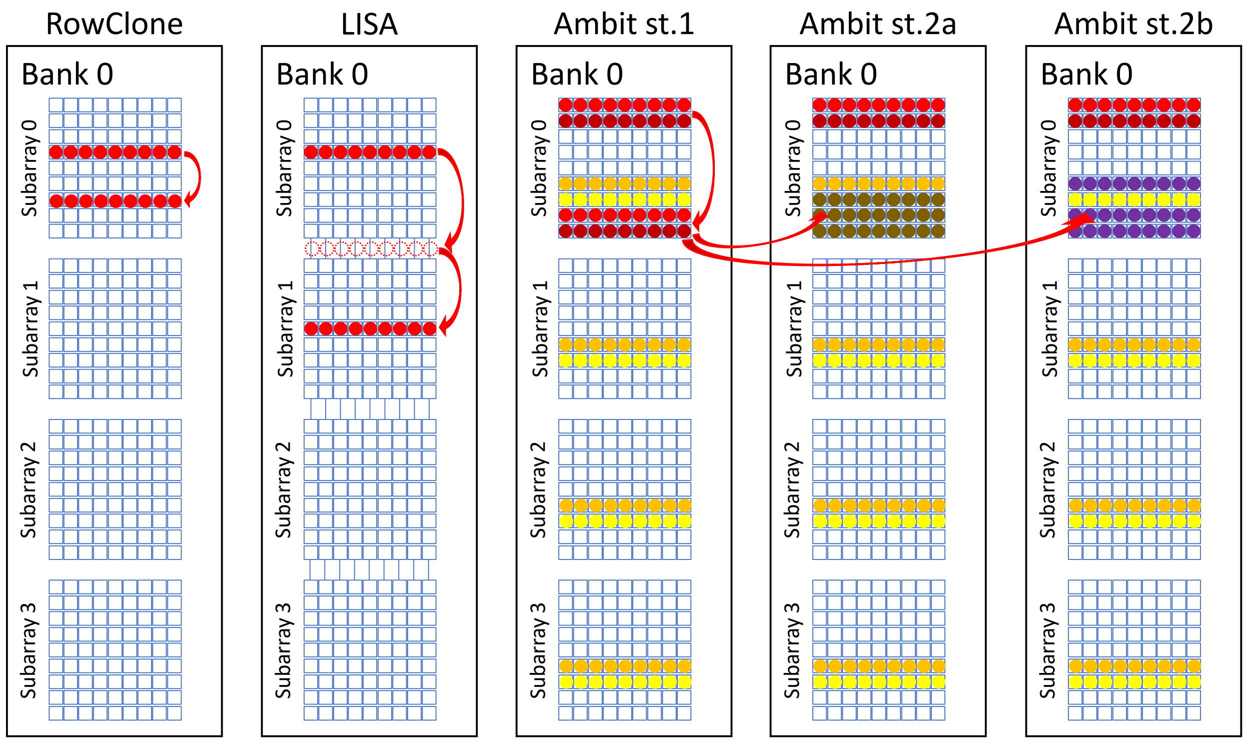


Figure 6.4: Bitwise PiM operations for RowClone, LISA, and Ambit AND/OR.

6.2 Strategies for Emulating Bitwise-PiM and Generic PiM

In this section we describe how we can leverage the model entities described above to emulate bitwise-PiM architectures.

6.2.1 Emulating RowClone

RowClone [33] describes two methods of cloning data locally in memory. In *Fast Parallel Mode (FPM)*, data is read from a source row into the row buffer and 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 4.4. Likewise, for RowClone-PSM, we configure the DIMM interface and memory bus to allow data flow between banks.

6.2.2 Emulating LISA

LISA [34] 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.

6.2.3 Emulating Ambit

Ambit [35] 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, for AND-OR operations, a *triple-row activation* is triggered that computes a bitwise majority function. Similarly, a dedicated word line connected to the negated logical value inside the sense amplifier is activated for NOT operation. Finally, it returns back 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.

6.2.4 Emulating Fulcrum

Fulcrum [36] proposes a more complex and general PiM architecture that places an *AddressLess Processing Unit (ALPU)* at each pair of two subarrays. Additionally, operand data is passed between the subarrays and the ALPU via *Walkers*. The ALPU consists of an ALU, an instruction buffer, a control unit, register and connection busses. To emulate Fulcrum into PiMulator, we have to expand the bank model to enable parallel access to multiple subarrays. Next, we have to instantiate the necessary number of walkers and ALPU instances into the bank PiM template. While this architecture is of high interest to the authors and has motivated the development of several features such as subarray membership and shared PiM kernels, the actual emulation implementation is left as future work.

6.2.5 Emulating Sieve

Sieve [40] constitutes a bit-serial Processing-in-Memory (PiM) architecture designed to execute k-mer matching at the bank subarray level. Its primary components include a matcher circuit per sense amplifier, which ensures parallel execution of query and reference k-mers, along with a pipelined column finder circuit for pinpointing the corresponding column of the reference k-mer.

We implement functional modeling of Sieve within the PiMulator framework using SystemVerilog. The process is part of a broader effort to integrate Sieve with the host system fully and evaluate variations of the Sieve architecture with relevant workloads. PiMulator hosts the Sieve logic within the PiM templates, facilitating direct and seamless access to active rows. Through the true dual-port bank array design implementation in PiMulator, the framework allows the host and the Sieve kernel independent access to the banks, simplifying the model implementation. The framework emulates the additional Sieve logic latencies and commands by modifying the state machine timing model and extending the command decoder table with Sieve logic controls.

The emulation of Sieve reveals a Block RAM (BRAM) memory usage of a mere 0.78% on a subarray featuring 64×8192 bit cells. This low BRAM usage suggests PiMulator's scalability across

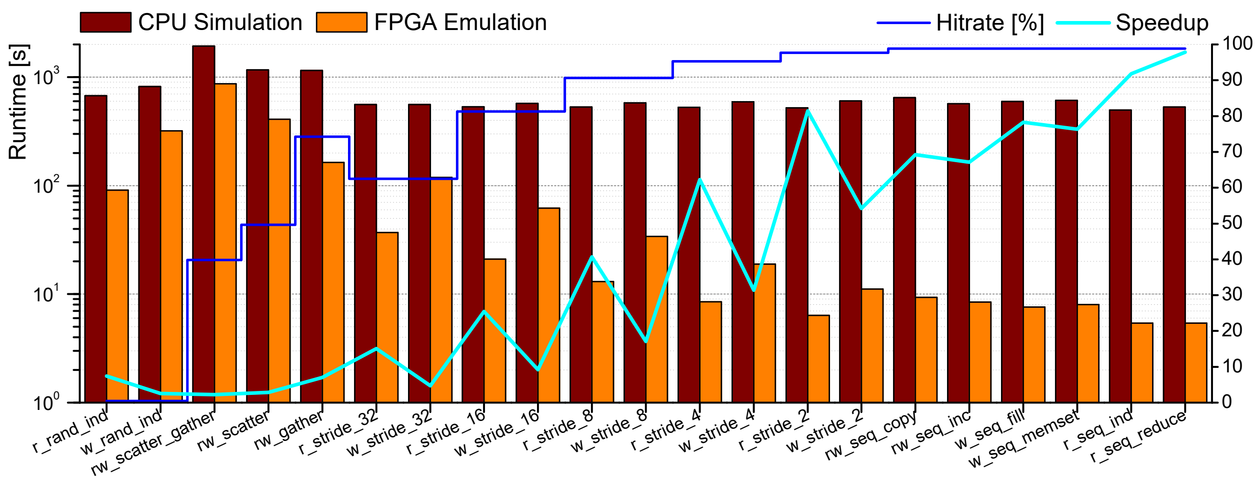


Figure 6.5: Runtimes for CPU Simulation and FPGA Emulation of a DDR4 memory system stressed for 1s of real *target* time with different access patterns; ratio between the two runtimes and the corresponding DSync hitrate.

multiple subarrays, banks, and chips. This modeling exercise has resulted in valuable architectural insights and innovative design concepts that can inform the development of future bit-serial PiM architectures.

6.3 Evaluation

At this time, we evaluate PiMulator runtime under different memory access patterns using the Hopscotch [41] 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 [61] 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 [42] trace format. Finally, we measure execution time for each workload traces in DRAMsim3 on a machine with two 8 core Intel Xeon Silver 4208

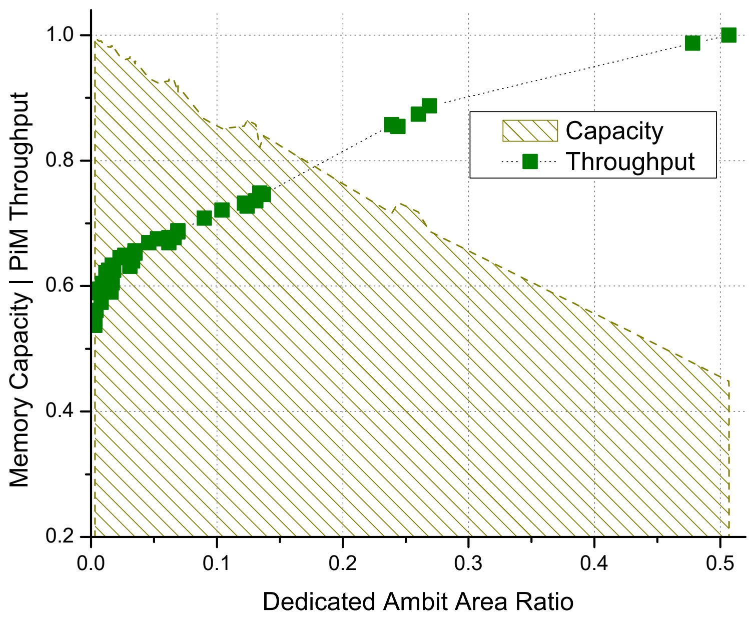


Figure 6.6: Pareto curve for Ambit design space exploration.

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 6.5. As expected, all kernels complete 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 hit rates.

We also evaluate a theoretical Ambit+LISA [34,35] 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 throughput for each configuration. The results obtained are shown in Figure 6.6. 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. It is also possible to expand the memory model local rows to make use of Ultra-RAM resources, thus accommodating an even larger number of rows on the FPGA. Alternatively, the Utra-RAM resources can be harnessed in a cache block, adding locality and diminishing the penalty during a miss by further speeding up part of the HBM2/DDR4 memory accesses. The remaining FPGA resources are necessary to be used to emulate larger PiM units and the remaining full system components such as CPU, caches, bus, controllers, peripherals, accelerators, and observability modules.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis offers a comprehensive exploration of Processing-in-Memory (PiM) architectures and their emulation. The central objective was to propose a robust, high-fidelity framework to model and assess upcoming memory and PiM architectures. To this end, we presented an in-depth theoretical discussion about DRAM memory and PiM architectures, specifically those pertinent to FPGA emulation. We scrutinized various modeling approaches, emphasizing their benefits and shortcomings, ultimately guiding this work's design process.

The MEMulator memory model, a primary component of this thesis, incorporates all significant components and behaviors of a memory device. The framework features a full-system target and logic layer integration and facilitates system-level modeling, evaluation, and co-design, while the layered approach allows for efficient separation of concerns. Furthermore, we demonstrated time modeling strategies such as frequency, throughput, latency, and event synchronization, making MEMulator a comprehensive tool for emulating large, complex, high-performance systems on high-end FPGA boards.

Another significant element of this work is the introduction of FreezeTime, a coarse-grain module-level stall mechanism that freezes simulation time to enable synchronization and cycle-accurate emulation of computer architectures. The power of FreezeTime is its potential to enhance the emulation capabilities of existing emulation platforms, providing near-emulation speed and high accuracy. We demonstrated the viability of this concept by integrating FreezeTime into the open-source LiteX framework, which supports a diverse set of FPGA boards.

This work then introduced PiMulator, a flexible open-source emulation framework for PiM. PiMulator provides a fast, adaptable, and high-fidelity platform for PiM architecture prototyping and evaluation. Notably, it facilitates the emulation of complex architectures under heavy workloads. Our framework exhibits an average speedup of 28x, making it an efficient tool for design space exploration.

This work combines the MEMulator, FreezeTime, and PiMulator into a comprehensive framework for computer system emulation, design space exploration, and evaluation. This integration has resulted in a robust, full-system emulation framework. This open-source, FPGA synthesizable model can generate all system components, from the CPU to memory and peripherals, on the FPGA using only soft cores - a unique feature among existing emulation frameworks.

Finally, we conducted a series of benchmarks on systems with different CPU and memory configurations, validating the speed and accuracy of our emulation framework. Our comparison studies of FPGA-based emulation, FPGA-accelerated simulation, and simulation with gem5 and Verilator further underlined the robustness of our approach. Overall, this work provides a comprehensive, efficient, and robust platform for researchers to evaluate their design in a fast and accurate manner, propelling the PiM architectural research forward. The code base and related dcumentation has been made open source and is available at `https://github.com/hplp/MEMulator` and `https://github.com/hplp/PiMulator`.

7.2 Future Work and Directions

Though we've laid the groundwork with key placeholders and essential components of the framework, there is always room for growth. This involves improving existing attributes, fixing bugs, adding new features, or aligning with emerging standards. The development and upkeep of the framework is truly a life-long mission. The following subsections present our immediate and long-term plans for future work. We hope these enhancements will positively impact both the academic and research communities.

7.2.1 Memory model enhancements

As memory standards continue to advance, the memory emulation model must keep pace and incorporate support for emerging features. Our model is primarily based on the DDR4 memory standard, with a few supplementary functionalities supporting more recent standards. Despite this, there are already opportunities for improvement. The model could gain from more robust parameterization, such as incorporating a diverse range of memory channels, ranks, chip wiring configurations, ECC algorithms, RAS/CAS controls, addressing schemes, and timing behavior.

These proposed enhancements should then be evaluated against RTL behavioral and software simulations with respect to performance and accuracy. Architects can learn insightful lessons by running memory micro-benchmarks, such as Hopscotch, alongside standard user workloads and benchmarks like Linux boot, SPEC CPU2017, and AI/ML training/inference workloads. Moreover, we suggest running multiple system-level simulations versus emulation tests, analyzing the entire memory system, and comparing emulation, FPGA-accelerated, and computer simulation results.

Lastly, an encompassing study on the whole memory hierarchy—spanning from virtual memory, caches, cache coherency, memory controller, addressing schemes, to memory types—should be undertaken. This would allow the extraction of performance data for latency and bandwidth under various execution types (SISD, SIMD) and diverse workloads of interest. Such capability would particularly benefit architects seeking to optimize the entire memory architecture for performance, power, quality of service, and cost-effectiveness.

7.2.2 Models for Comprehensive PiM Taxonomy Support

This thesis outlines various PiM architectures that position PiM kernels anywhere from the sub-arrays to the row buffer, bank, rank, interface, etc. These architectures could leverage diverse memory technologies and involve different types of compute (analog or hardened digital circuit, processor, reconfigurable logic). While we have experimented with emulating innovative and cost-efficient PiM architectures like bitwise and bit-serial, providing comprehensive support for all types of architectures remains a key objective.

To better understand the platform's limitations and motivate the development of new features, we should strive to emulate most of the PiM taxonomy space. An ideal starting point would be implementing the most prevalent real and academic PiM architectures. The selection of architectures should also consider popular applications such as AI/ML, database operations, and bioinformatics. The future task, therefore, involves developing models of popular PiM architectures using PiMulator, accommodating their variations, and running high-demand applications. The ultimate goal is to explore different memory types (HBM, DDR, GDDR, LPDDR), identify and implement any missing features, and ensure full support for the whole PiM taxonomy.

7.2.3 Full-system Integration Enhancements

Embedding the memory+PiM model into a full-scale system presents significant challenges and demands considerable time. However, it remains a cornerstone in our projected work. The existing deployment does not yet consistently deliver reliable functionality, with several key elements still under development or inadequately tested, often underperforming with new configurations.

We have initiated integrating the memory model into the LiteX framework using a soft-PHY, with support for both single and multiple channels. This feature is slated for a stable release in the near future. Concurrently, the FreezeTime solution, responsible for implementing the time-scaling and stall broadcast system, requires further enhancements to improve its automation, system generation, and reliability.

An additional aspect requiring meticulous planning and execution is the implementation of system-level observability and statistical data collection. This feature will help measure performance throughout the system and offer essential insights, facilitate the identification and understanding of issues/bottlenecks, and enrich the framework's benefits.

Finally, we aim to develop a dynamic, self-partitioning addressing scheme that seamlessly bridges the Data Synchronization Engine with the UltraRAM and FPGA board memory resources. This feature will support the emulation of large memory capacity with reduced cache miss penalty.

7.2.4 Firmware and Software Stack

Make the programmer's life easy is one of the core CRISP center missions. Achieving this goal requires more than logical layer infrastructure; it requires comprehensive system, firmware, and software support for PiM architectures. Note that of the many PiM architectures proposed over the years, only the real-world PiM architectures offer firmware and library support.

This research direction could greatly expedite the adoption of PiM architectures. The ultimate objective is the creation of a flexible firmware and software SDK that accommodates a comprehensive PiM taxonomy and facilitates effortless application development and efficient runtime on the PiM / memory-centric compute paradigm. The solution would include memory controller support for PiM commands, cache coherence mechanisms, security and data integrity protections, standalone libraries or OS drivers, compiler support, and libraries for higher level software development.

This endeavor promotes interdisciplinary collaboration among computer architects, embedded and operating systems researchers, as well as compiler and software engineers. The resulting advancements could significantly streamline the adoption of PiM architectures, thereby stimulating further interest in this domain.

7.2.5 Fostering Community Engagement

The evolution and progress of our framework will largely be guided by its users. Their discerning eyes help identify potential issues, while their collaborative efforts contribute to developing solutions and new features. We owe a debt of gratitude to all those who have contributed so far and openly welcome further feedback and insights.

We cordially invite you to join the vibrant PiMulator community, where your work with PiM and research can find a supportive and resourceful base. Engaging with us, you will not just be a user but an integral part of a dynamic community that is collectively shaping the future of memory and PiM emulation. Join, contribute, and let us innovate together in this exciting journey of discovery and growth. Thank you!

Appendix A

DRAM Memory Timings

A.1 DRAM Memory Timings

tCK	clock period
AL	additive latency
CL	CAS latency- Column address strobe latency
CWL	CAS write latency
tRCD	RAS to CAS delay
tRP	Precharge time
tRAS	Active to Precharge delay
tRFC	Delay between the refresh command and the next valid command
tRFC2	Delay of partial refresh
tRFC4	Delay of partial refresh
tREFI	Time interval between consecutive refresh cycles
tRPRE	Minimum pulse width of read preamble
tWPRE	Time between when the data strobe goes from non-valid to valid
tRRD_S	ACTIVATE to ACTIVATE command delay to different bank group
tRRD_L	ACTIVATE to ACTIVATE command delay to same bank group
tWTR_S	Delay from start of internal write transaction to internal read command for different bank group
tWTR_L	Delay from start of internal write transaction to internal read command for same bank group
tFAW	Four activate window is a rolling time frame in which a maximum of four bank activation can be engaged
tWR	Write to precharge delay
tRTP	Read to precharge delay
tCCD_S	CAS_n to CAS_n command delay for different bank group
tCCD_L	CAS_n to CAS_n command delay for same bank group
tCKE	CKE (clock enable) minimum pulse width

Table A.1: Description of Memory timings.

Appendix B

List of Publications

B.1 In progress

- "MEMulator: Comprehensive Memory System Emulation" targeting a journal. In this paper we will demonstrate full system emulation on FPGA, using soft IPs for the CPU, memory controller and the memory model. We will demonstrate support for emulation of many different memory flavors, evaluate runtime using the hopscotch [41], SPEC CPU2017 [96] and ML benchmarks and compare results with computer simulation using gem5 [61] and FPGA-accelerated simulation using FASED [21]. We will also conduct and present a case study for RowHammer [38], highlighting the model's ability to model components and data layout.

- "PiMulator+: System Support for Processing-in-Memory Emulation“ targeting an architecture conference or journal. In this paper we expand PiMulator and MEMulator with support for increasingly advanced PiM architectures such as Fulcrum [36], and system support for PiM at the memory controller, ISA, CPU, compiler and OS.

B.2 Journals/Letters

1. X. Guo, V. Verma, P. Guerrero, **S. Mosanu**, M. Stan, “Back to the Future: Digital Circuit Design in the FinFET Era,” *Journal of Low Power Electronics* (**JOLPE**), Vol. 13, No. 3, pp. 338-355, September 2017. (**Invited Paper**)

B.3 Conferences

1. **Sergiu Mosanu**, Joshua Fixelle, Mohammad Nazmus Sakib, Kevin Skadron, and Mircea Stan. “FreezeTime: Towards System Emulation through Architectural Virtualization.” In 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) - Reconfigurable Architectures Workshop (RAW). IEEE, 2023.

2. **Sergiu Mosanu**, Mohammad Nazmus Sakib, Tommy Tracy II, Ersin Cukurtas, Alif Ahmed, Preslav Ivanov, Samira Khan, Kevin Skadron and Mircea Stan. “PiMulator: a Fast and Flexible Processing-in-Memory Emulation Platform.” In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2022.

3. Yimin Gao, **Sergiu Mosanu**, Mohammad Nazmus Sakib, Vaibhav Verma, Xinfei Guo, and Mircea Stan. “LiteAIR5: A System-Level Framework for the Design and Modeling of AI-Extended RISC-V Cores.” In 2023 36th IEEE International System-on-Chip Conference (SOCC).

4. Elisa Pantoja, Rahul Sreekumar, **Sergiu Mosanu**, Tommy Tracy, and Mircea Stan. “Virtualized Controller for Computational RFID-based IoT Sensors.” In 2023 IEEE International Conference on RFID (RFID), pp. 54-59. IEEE, 2023.

5. Xiangdong Wei, Mohamed El-Hadedy, **Sergiu Mosanu**, Zhengping Zhu, Wen-Mei Hwu, and Xinfei Guo. “RECO-HCON: A High-Throughput Reconfigurable Compact ASCON Processor for Trusted IoT.” In 2022 IEEE 35th International System-on-Chip Conference (SOCC), pp. 1-6. IEEE, 2022. **Best Paper Award**

6. Xinfei Guo, Mohamed El-Hadedy, **Sergiu Mosanu**, Xiangdong Wei, Kevin Skadron, and Mircea R. Stan. “Agile-AES: Implementation of configurable AES primitive with agile design approach.” Integration 85 (2022): 87-96.

7. Mohammad Nazmus Sakib, Hamed Vakili, Samiran Ganguly, **Sergiu Mosanu**, Avik W. Ghosh, and Mircea Stan. “Magnetic skyrmion-based programmable hardware.” In Spintronics XIII, vol. 11470, p. 114703D. International Society for Optics and Photonics, 2020.

8. Mohamed El-Hadedy, Martin Margala, **Sergiu Mosanu**, Danilo Gligoroski, Jinjun Xiong, and Wen-Mei Hwu. “MICRO-GAGE: A Low-power Compact GAGE Hash Function Processor for IoT Applications.” In 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pp. 1-4. IEEE, 2020.

9. Mohamed El-Hadedy, Martin Margala, **Sergiu Mosanu**, Danilo Gligorosk, and Wen-Mei Hwu. “PRO-GAGE: A High Performance Compact GAGE Hash Function Processor for Small Space Technology.” In 2021 IEEE Space Computing Conference (SCC), pp. 9-16. IEEE, 2021.

B.4 Workshops, Posters, Presentations and Demos

1. Khyati Kiyawat, **Sergiu Mosanu**, Mircea Stan and Kevin Skadron. “Open-Source Processing-in-Memory Architecture Design through FPGA Emulation: A Case Study Modeling Sieve.” Presented at the workshop on Open-Source Computer Architecture Research (OSCAR), co-located with the International Symposium on Computer Architecture (ISCA) 2023.

2. **Sergiu Mosanu**, Joshua Fixelle, Kevin Skadron, and Mircea Stan. “FreezeTime: Towards System Emulation through Architectural Virtualization.” In Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA). ACM/SIGDA, 2023.

3. **Sergiu Mosanu**, Preslav Ivanov, Tommy Tracy II, Ersin Cukurtas, Samira Khan, Kevin Skadron and Mircea Stan. “PiMulator: a Processing-in-Memory Emulation Framework.”

DAC Young Fellow 2020.

4. **Sergiu Mosanu**, Xinfei Guo, Mohamed El-Hadedy, Lorena Anghel, and Mircea Stan. "Flexi-AES: A Highly-Parameterizable Cipher for a Wide Range of Design Constraints." Poster, demo, 1-page paper in proceedings of 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 338-338. IEEE, 2019.

5. **Sergiu Mosanu** and Mircea Stan. "AES and SHA Cryptography Library for Chisel." Presentation at Chisel Community Conference 2018, recording available at `https://youtu.be/VNM88i74Ky0`.

6. **Sergiu Mosanu**, Kevin Skadron, and Mircea Stan. "Burrows-Wheeler Short Read Aligner on AWS EC2 F1 Instances." Workshop on Accelerator Architecture in Computational Biology and Bioinformatics HPCA-AACBB 2018 Presentation, see `https://aacbb-workshop.github.io/slides/2018/aacbb-serg-presi.pdf`.

7. **Sergiu Mosanu**, Arijit Banerjee and Mircea Stan. "Modeling and Design of an SLC 3D NAND Flash CLB for 3D CPLDs and FPGAs." Flash Memory Summit 2015.

B.5 SRC TECHCON

- **Sergiu Mosanu**, Tom Tracy, Ersin Cukurtas, Preslav Ivanov, Robert West, Samira Khan, Kevin Skadron and Mircea R. Stan. "PiMulator – a Processing in Memory FPGA Emulation Framework." TECHCON 2020

- **Sergiu Mosanu**, Xinfei Guo, Mohamed Aly, Lorena Anghel, Kevin Skadron and Mircea R. Stan. "Flexi-AES: A Highly-Parameterizable Cipher for a Wide Range of Design Constraints." TECHCON 2019.

- **Sergiu Mosanu**, Samira Khan, Kevin Skadron and Mircea R. Stan. "Near-Storage Computing for Face Detection and Face Recognition." TECHCON 2018.

B.6 Awards

- Co-Instructor for CS 3330 – Computer Architecture, with Prof. Charles Reiss, University of Virginia, Spring 2023
- **SOCC 2022 Best Paper Award** for the paper titled "RECO-HCON: A High-Throughput Reconfigurable Compact ASCON Processor for Trusted IoT", September 2022
- **School of Engineering and Applied Science Teaching Intern Fellowship**, co-teaching ECE4550-0001, Field Programmable Gate Arrays (FPGAs) Design, ECE 4550-003 ASIC/-SoC Design, and ECE6502-004 Advanced Digital Design, with Prof. Mircea Stan, University of Virginia, Spring 2018
- **Outstanding Teaching Assistant Award**, Charles L. Brown Department of Electrical and Computer Engineering, University of Virginia, May 2017

Glossary

Acronyms and Abbreviations	
ACAP	Adaptive Compute Acceleration Platform
AES	Advanced Encryption Standard
AI	Artificial Intelligence
AIE	AI Engine
AMBA	Advanced Microcontroller Bus Architecture
APU	Application Processing Unit
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
AWS	Amazon Web Services
BIOS	Basic Input/Output System
BRAM	Block RAM
BUFGCE	Global Clock Buffer with Clock Enable
CAS	Column Address Strobe
CHISEL	Constructing Hardware in a Scala Embedded Language
CLK	Clock Signal
CMD	Command Decoder Module
CPU	Central Processing Unit
CRISP	Center for Research on Intelligent Storage and Processing-in-Memory
CXL	Compute Express Link

DDR	Double Data Rate
DIMM	Dual In-line Memory Module
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processing
DSync	Data Synchronization Engine
ECE	Electrical and Computer Engineering
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GCC	GNU Compiler Collection
GDDR	Graphics Double Data Rate
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HLS	High-Level Synthesis
HMC	Hybrid Memory Cube
HPC	High-Performance Computing
HPLP	High-Performance Low-Power research lab at University of Virginia
I/O	Input/Output
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
ILA	Integrated Logic Analyzer
IoT	Internet of Things
IP	Intellectual Property
ISA	Instruction Set Architecture
JEDEC	Joint Electron Device Engineering Council
JTAG	Joint Test Action Group

LPDDR	Low-Power Double Data Rate
LUT	LookUp Table
LUTRAM	LookUp Table RAM
MEM	Memory Emulation Model
MIG	Memory Interface Generator
ML	Machine Learning
MUX	Multiplexer
NoC	Network on Chip
NVMM	Non-Volatile Main Memory
OoO	Out-of-Order execution
OS	Operating System
PE	Processing Element
PCIe	Peripheral Component Interconnect Express
PCM	Phase Change Memory
PHY	Physical layer
PiM, PIM	Processing-in-Memory
PL	Programmable logic
PLL	Phase-Locked Loop
PnM	Processing-near-Memory
PS	Processing System
PuM	Processing-using-memory
RAM	Random-Access Memory
RAS	Row Address Strobe
RISC	Reduced Instruction Set Computing
RISC-V	Reduced Instruction Set Computer Fifth Generation
RPU	Real-time Processing Unit
RRAM	Resistive Random-Access Memory

RTL	Register Transfer Level
SDRAM	Synchronous Dynamic Random-Access Memory
SoC	System on Chip
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
STT-RAM	Spin-Transfer Torque Random-Access Memory
STT-MRAM	Spin-Transfer Torque Magnetic Random-Access Memory
TPU	Tensor Processing Unit
TSV	Through-Silicon Via
UART	Universal Asynchronous Receiver/Transmitter
URAM	UltraRAM
UVA	University of Virginia

Bibliography

[1] D. J. Campello, “Optimizing Main Memory Usage in Modern Computing Systems to Improve Overall System Performance,” *FIU Electronic Theses and Dissertations*, vol. 2568, 2016, https://digitalcommons.fiu.edu/etd/2568.

[2] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, “A modern primer on processing in memory,” in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*. Springer, 2022, pp. 171–243.

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[4] M. Velten, R. Schöne, T. Ilsche, and D. Hackenberg, “Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors,” in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, pp. 165–175.

[5] D. D. Sharma, “Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing,” in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2022, pp. 5–12.

[6] Lisa Su, “AMD Keynote,” CES 2023, https://www.youtube.com/watch?v=OMxU4BDIm4M&t=5380s, 2023.

[7] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: VersalTM architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 84–93.

[8] S. Li, "Scalable and Accurate Memory System Simulation," Ph.D. dissertation, University of Maryland, College Park, 2019.

[9] H. Angepat, D. Chiou, E. S. Chung, and J. C. Hoe, "FPGA-accelerated simulation of computer systems," *Synthesis Lectures on Computer Architecture*, vol. 9, no. 2, pp. 1–80, 2014.

[10] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 29–42.

[11] A. K. Jain, S. Lloyd, and M. Gokhale, "Microscope on Memory: MPSoC-Enabled Computer Memory System Assessments," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 173–180.

[12] J. Zhang, Y. Zha, N. Beckwith, B. Liu, and J. Li, "MEG: A RISCV-based System Emulation Infrastructure for Near-data Processing Using FPGAs and High-bandwidth Memory," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 4, pp. 1–24, 2020.

[13] F. Wen, M. Qin, P. Gratz, and N. Reddy, "An FPGA-based Hybrid Memory Emulation System," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021, pp. 190–196.

[14] A. Olgun, J. G. Luna, K. Kanellopoulos, B. Salami, H. Hassan, O. Ergin, and O. Mutlu, "PiDRAM: A Holistic End-to-end FPGA-based Framework for Processing-in-DRAM," *arXiv preprint arXiv:2111.00082*, 2021.

[15] G. Martin and G. Smith, “High-Level Synthesis: Past, Present, and Future,” *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, July 2009.

[16] C. Papon and SpinalHDL, “VexRISC-V: An FPGA-friendly 32-bit RISC-V CPU implementation.” [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[17] S. Bourdeauducq, “Migen (Milkymist generator) - A Python toolbox for building complex digital hardware.” [Online]. Available: https://github.com/m-labs/migen

[18] F. Kermarrec, S. Bourdeauducq, H. Badier, and J.-C. Le Lann, “LiteX: an open-source SoC builder and library based on Migen Python DSL,” in *OSDA 2019, colocated with DATE 2019 Design Automation and Test in Europe*, 2019.

[19] J. Zhang, Y. Liu, G. Jain, Y. Zha, J. Ta, and J. Li, “MEG: A RISCV-based System Simulation Infrastructure for Exploring Memory Optimization Using FPGAs and Hybrid Memory Cube,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 145–153.

[20] A. Akram and L. Sawalha, “A Survey of Computer Architecture Simulation Techniques and Tools,” *IEEE Access*, vol. 7, pp. 78 120–78 145, 2019.

[21] D. Biancolin, S. Karandikar, D. Kim, J. Koenig, A. Waterman, J. Bachrach, and K. Asanovic, “FASED: FPGA-accelerated simulation and evaluation of DRAM,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 330–339.

[22] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic, “RAMP: Research accelerator for multiple processors,” *IEEE micro*, vol. 27, no. 2, pp. 46–57, 2007.

[23] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[24] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," *arXiv preprint arXiv:2012.03112*, 2020.

[25] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning, "In-memory Intelligence," *IEEE Micro*, vol. 37, no. 4, pp. 30–38, 2017.

[26] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 316–331.

[27] H. A. D. Nguyen, J. Yu, M. A. Lebdeh, M. Taouil, S. Hamdioui, and F. Catthoor, "A classification of memory-centric computing," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 16, no. 2, pp. 1–26, 2020.

[28] P. Gu, "List of Process-in-memory (PIM) and Near-data-processing (NDP) papers," 2020. [Online]. Available: https://github.com/miglopst/PIM_NDP_papers

[29] H. S. Stone, "A logic-in-memory computer," *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 73–78, 1970.

[30] S. Xu, X. Chen, Y. Wang, Y. Han, X. Qian, and X. Li, "PIMSim: A flexible and detailed processing-in-memory simulator," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 6–9, 2018.

[31] C. Yu, S. Liu, and S. Khan, "MultiPIM: A Detailed and Configurable Multi-Stack Processing-In-Memory Simulator," *IEEE Computer Architecture Letters*, vol. 20, no. 1, pp. 54–57, 2021.

[32] K. Asanovic, D. A. Patterson, and C. Celio, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," University of California at Berkeley Berkeley United States, Tech. Rep., 2015.

[33] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch *et al.*, “RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 185–197.

[34] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, “Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 568–580.

[35] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, “Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology,” in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 273–287.

[36] M. Lenjani, P. Gonzalez, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron, “Fulcrum: a simplified control and access mechanism toward flexible and practical in-situ accelerators,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 556–569.

[37] S. Rai, A. Devic, and A. Sivasubramaniam, “Scaling (-In-) Memory Computing with BLIMP (Bank Level In-Memory Processing),” 2020.

[38] O. Mutlu and J. S. Kim, “Rowhammer: A retrospective,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

[39] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, “Drisa: A DRAM-based reconfigurable in-situ accelerator,” in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 288–301.

[40] L. Wu, R. Sharifi, M. Lenjani, K. Skadron, and A. Venkat, “Sieve: Scalable In-situ DRAM-based Accelerator Designs for Massively Parallel k-mer Matching,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 251–264.

[41] A. Ahmed and K. Skadron, "Hopscotch: A micro-benchmark suite for memory performance evaluation," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 167–172.

[42] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[43] B. Keeth and R. J. Baker, *DRAM circuit design: a tutorial.* IEEE, 2001.

[44] V. Seshadri and O. Mutlu, "In-DRAM bulk bitwise execution engine," *arXiv preprint arXiv:1905.09822*, 2019.

[45] Graham Allan, "DDR4 Bank Groups in Embedded Applications," Synopsys technical bulletin, https://www.synopsys.com/designware-ip/technical-bulletin/ddr4-bank-groups.html.

[46] H.-K. Liu, D. Chen, H. Jin, X.-F. Liao, B. He, K. Hu, and Y. Zhang, "A Survey of Non-Volatile Main Memory Technologies: State-of-the-Arts, Practices, and Future Directions," *Journal of Computer Science and Technology*, vol. 36, no. 1, pp. 4–32, 2021.

[47] P. Siegl, R. Buchty, and M. Berekovic, "Data-centric computing frontiers: A survey on processing-in-memory," in *Proceedings of the Second International Symposium on Memory Systems*, 2016, pp. 295–308.

[48] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "A review of near-memory computing architectures: Opportunities and challenges," in *2018 21st Euromicro Conference on Digital System Design (DSD).* IEEE, 2018, pp. 608–617.

[49] ——, "Near-memory computing: Past, present, and future," *Microprocessors and Microsystems*, vol. 71, p. 102868, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933119300389

[50] J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking memory-centric computing systems: Analysis of real processing-in-memory hardware," in *2021 12th International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2021, pp. 1–7.

[51] O. Mutlu and J. Gómez-Luna, "Onur Mutlu Lectures, PIM Course: Real-world PIM," 2023. [Online]. Available: https://www.youtube.com/@OnurMutluLectures/videos

[52] S. Newsroom, "Samsung Develops Industry's First High Bandwidth Memory with AI Processing Power," 2021. [Online]. Available: https://news.samsung.com/global/samsung-develops-industrys-first-high-bandwidth-memory-with-ai-processing-power

[53] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim, "25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 350–352.

[54] 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, and 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.

[55] J. H. Kim, S.-h. Kang, S. Lee, H. Kim, W. Song, Y. Ro, S. Lee, D. Wang, H. Shin, B. Phuah, J. Choi, J. So, Y. Cho, J. Song, J. Choi, J. Cho, K. Sohn, Y. Sohn, K. Park, and N. S. Kim, "Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–26.

[56] E. S. Y. L. Kanga Kong, Jaehwan Kevin Kim, "SK hynix Develops PIM, Next-Generation AI Accelerator," 2022. [Online]. Available: https://news.skhynix.com/sk-hynix-develops-pim-next-generation-ai-accelerator/

[57] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, K. Vladimir, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, J. Lee, D. Ko, Y. Jun, K. Cho, I. Kim, C. Song, C. Jeong, D. Kwon, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications," in *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 1–3.

[58] D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G.-M. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, V. Kornijcuk, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Lee, D. Ko, Y. Jun, I. Kim, C. Song, I. Kim, C. Park, S. Kim, C. Jeong, E. Lim, D. Kim, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25V 8Gb 16Gb/s/Pin GDDR6-Based Accelerator-in-Memory Supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep Learning Application," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 1, pp. 291–302, 2023.

[59] S. Newsroom, "Samsung Brings In-Memory Processing Power to Wider Range of Applications," 2021. [Online]. Available: https://news.samsung.com/global/samsung-brings-in-memory-processing-power-to-wider-range-of-applications

[60] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, K. Kim, J. Jung, I. Yun, S. J. Park, H. Park, J. Song, J. Cho, K. Sohn, N. S. Kim, and H.-H. S. Lee, "Near-Memory Processing in Action: Accelerating Personalized Recommendation With AxDIMM," *IEEE Micro*, vol. 42, no. 1, pp. 116–127, 2022.

[61] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[62] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, “The M5 simulator: Modeling networked systems,” *Ieee micro*, vol. 26, no. 4, pp. 52–60, 2006.

[63] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[64] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, “The gem5 Simulator: Version 20.0+,” 2020.

[65] Y. Kim, W. Yang, and O. Mutlu, “Ramulator: A fast and extensible DRAM simulator,” *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[66] Q. Zheng, X. Li, Y. Guan, Z. Wang, Y. Cai, Y. Chen, G. Sun, and R. Huang, “PIMulator-NN: An Event-Driven, Cross-Level Simulation Framework for Processing-In-Memory-Based Neural Network Accelerators,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 12, pp. 5464–5475, 2022.

[67] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, “Accuracy evaluation of gem5 simulator system,” in *7th International workshop on reconfigurable and communication-centric systems-on-chip (ReCoSoC)*. IEEE, 2012, pp. 1–7.

[68] T. Nowatzki, J. Menon, C.-H. Ho, and K. Sankaralingam, “Architectural simulators considered harmful,” *IEEE Micro*, vol. 35, no. 6, pp. 4–12, 2015.

[69] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, “RAMP gold: an FPGA-based architecture simulator for multiprocessors,” in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 463–468.

[70] L. Rizzatti, “Hardware emulation: three decades of evolution. Part III,” *Verification Horizons*, pp. 15–18, 2015.

[71] Micron, “DDR SDRAM Verilog Simulation Models,” https://www.micron.com/products/dram.

[72] M. Gokhale, B. Holmes, and K. Iobst, “Processing in memory: The Terasys massively parallel PIM array,” *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[73] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, “FlexRAM: Toward an advanced intelligent memory system,” in *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*. IEEE, 1999, pp. 192–201.

[74] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis *et al.*, “The structural simulation toolkit,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[75] D. Sanchez and C. Kozyrakis, “ZSim: Fast and accurate microarchitectural simulation of thousand-core systems,” *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[76] A. Barreteau, “System-Level Modeling and Simulation with Intel® CoFluent™ Studio,” in *Complex Systems Design & Management: Proceedings of the Sixth International Conference on Complex Systems Design & Management, CSD&M 2015*. Springer, 2016, pp. 305–306.

[77] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, “DRAMsim: a memory system simulator,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 100–107, 2005.

[78] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “DRAMSim2: A Cycle Accurate Memory System Simulator,” *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[79] A. K. Jain, S. Lloyd, and M. Gokhale, “Performance Assessment of Emerging Memories Through FPGA Emulation,” *IEEE Micro*, vol. 39, no. 1, pp. 8–16, 2018.

[80] K. Asanović and D. A. Patterson, “Instruction sets should be free: The case for RISC-V,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[81] T. Petazzoni and F. Electrons, “Buildroot: a nice, simple and efficient embedded Linux build system,” in *Embedded Linux System Conference*, vol. 2012, 2012.

[82] C. Papon and SpinalHDL, “NaxRISC-V: OoO Superscalar 64-bit RISC-V CPU implementation targeting FPGAs.” [Online]. Available: https://github.com/SpinalHDL/NaxRiscv

[83] C. Papon, “Spinal Hardware Description Language.” [Online]. Available: https://github.com/SpinalHDL/SpinalHDL

[84] F. Kermarrec *et al.*, “A small footprint and configurable DRAM core powered by Migen & LiteX.” [Online]. Available: https://github.com/enjoy-digital/litedram

[85] J. S. S. T. Association *et al.*, “JEDEC Standard: DDR4 SDRAM,” *JESD79-4, Sep*, 2012.

[86] P. Zimmer, M. Zimmer, and B. Zimmer, “FizZim–an open-source FSM design environment,” *Enterprise Information Systems*, vol. 9, no. 5-6, pp. 528–555, 2014.

[87] 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.

[88] W. N. Hung and R. Sun, “Challenges in large FPGA-based logic emulation systems,” in *Proceedings of the 2018 International Symposium on Physical Design*, 2018, pp. 26–33.

[89] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz, “Ramp blue: A message-passing manycore system in fpgas,” in *2007 International Conference on Field Programmable Logic and Applications*. IEEE, 2007, pp. 54–61.

[90] S. Mosanu, M. N. Sakib, T. Tracy, E. Cukurtas, A. Ahmed, P. Ivanov, S. Khan, K. Skadron, and M. Stan, “PiMulator: A fast and flexible processing-in-memory emulation platform,” in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 1473–1478.

[91] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, “The rocket chip generator,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, 2016.

[92] W. Snyder, “Verilator Manual.” [Online]. Available: https://github.com/verilator/verilator

[93] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, “HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing,” in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 406–417.

[94] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, “ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 2, pp. 1–32, 2009.

[95] F. Devaux, “The true processing in memory accelerator,” in *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society, 2019, pp. 1–24.

[96] J. Bucek, K.-D. Lange, and J. v. Kistowski, “SPEC CPU2017: Next-generation compute benchmark,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.