UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JOÃO PAULO CARDOSO DE LIMA

PIM-gem5: a system simulator for Processing-in-Memory design space exploration

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Luigi Carro

Porto Alegre
March 2019

CIP — CATALOGING-IN-PUBLICATION

Lima, João Paulo Cardoso de

PIM-gem5: a system simulator for Processing-in-Memory design space exploration / João Paulo Cardoso de Lima. – Porto Alegre: PPGC da UFRGS, 2019.

87 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: Luigi Carro.

1. Processing-in-memory. 2. System simulator. 3. 3D-stacked memory. 4. . I. Carro, Luigi. II. Título.

“A very large part of space-time must be investigated,
if reliable results are to be obtained.”

— Alan Turing

ACKNOWLEDGMENTS

I want to thank and dedicate this thesis to every person who has been involved in my academic life. First, I would want to give special thanks to Professor Luigi Carro for his guidance and words of motivation and inspiration. I want to convey my gratitude to Paulo and Rafael for the discussions and cooperation in many works. Finally, I would like to thank all my beloved relatives and friends, who have been so supportive along the way of my academic journey.

ABSTRACT

Processing-in-Memory (PIM) has been recently revisited to address the issues of memory and power wall, mainly due to the maturity of 3D-stacking manufacturing technology and the increasing demand for bandwidth and parallel access in emerging data-centric applications. Recent studies have shown a wide variety of processing mechanisms to be placed in the logic layer of 3D-stacked memories, not to mention the already available 3D-stacked DRAMs, such as Micron's Hybrid Memory Cube (HMC). Most of the studies in PIM architectures use the HMC as target memory, since its logic layer is suitable for placing processing logic in the memory device. Nevertheless, the lack of tools for rapid prototyping can be a limiting factor to explore new architectures, mainly when computer architectures aim to simulate system integration. In this document, we present a PIM support for the broadly adopted gem5 simulator and a methodology for prototyping PIM accelerators. Using the proposed simulator, computer architects can model a full environment and address open problems in the PIM research field. Also, we present two case studies of a fixed-function and a programmable logic PIM placed alongside each vault controller, and we highlight the generic points of our implementation which can be used to the exploit efficiency of new PIM accelerators.

Keywords: Processing-in-memory. System simulator. 3D-stacked memory. .

PIM-gem5: um simulador de sistemas para exploração de espaço de projeto em arquiteturas de processamento em memória

RESUMO

O conceito de Processamento em Memória (PIM) está sendo revisitado recentemente para tratar de problemas relacionados ao gargalo de memória e energia dos sistemas computacionais atuais. A retomada à pesquisa em PIM deve-se principalmente à maturidade da tecnologia de fabricação de circuitos 3D e à crescente demanda por banda de memória e acesso paralelo em novas aplicações que são centradas em dados. Para conciliar aceleração e eficiência energética em aplicações emergentes, estudos recentes investigaram diferentes projetos de circuitos digitais de processamento para a camada lógica de memórias 3D, sem mencionar as memórias em produção como o Hybrid Memory Cube (HMC) da Micron, que integram camadas de circuitos lógicos e DRAM por vias de alta velocidade. A maioria dos estudos em arquiteturas PIM usa o HMC como memória alvo, já que sua camada lógica é adequada para inserir lógica de processamento no dispositivo de memória. No entanto, a falta de ferramentas para prototipagem rápida pode ser um fator limitante para explorar novas arquiteturas, principalmente quando estas arquiteturas necessitam simular a integração de sistemas para avaliar e testar alguma solução em nível de sistema. Neste documento é apresentado um suporte para o simulador gem5 que permite a simulação de novos projetos e uma metodologia para prototipagem de aceleradores PIM. Usando o simulador proposto é possível modelar um ambiente completo e abordar problemas em aberto no campo de pesquisa de PIM. Além disso, dois estudos de caso de arquiteturas PIM são apresentados: um projeto do tipo função fixa e outro de lógica programável, e destacam-se os pontos genéricos da implementação do simulador que podem ser utilizados para a exploração de eficiência de novos aceleradores PIM.

Palavras-chave: Processamento em memória, Simulador de sistema, memórias 3D.

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic and Logic Unit
AVX	Advanced Vector Extensions
BPO	Bounded-Operand Processing-in-Memory Operation
CAS	Column Access Strobe
CGRA	Coarse-Grain Reconfigurable Array
CISC	Complex Instruction Set Computer
CPO	Compound PIM Operation
CPU	Central Processing Unit
DDR	Double-Data Rate
DLP	Data-level Parallelism
DRAM	Dynamic Random Access Memory
EDP	Energy Delay Product
FCFS	First-Come-First-Served
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FR-FCFS	First-Ready First-Come-First-Served
FSM	Finite-State Machine
FU	Functional Unit
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HMC	Hybrid Memory Cube
ILP	Instruction-level Parallelism
ISA	Instruction Set Architecture
LSU	Load-Store Unit

LPDDR	Low-Power Double-Data Rate
NDP	Near-Data Processing
OoO	Out-of-Order
OS	Operating System
PCE	Pointer-Chasing Engine
PIM	Processing-in-Memory
RAS	Row Access Strobe
RISC	Reduced Instruction Set Computer
RMW	Read-Modify-Write
RTL	Register-tranfer Level
RVU	Reconfigurable Vector Unit
SerDes	Serializer-Deserializer
SIMD	Single Instruction Multiple Data
SSE	Streaming Single Instruction Multiple Data Extensions
TLB	Translation Lookaside Buffer
TSV	Through-Silicon Via
VPU	Vector Processing Unit

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Over the past decades, microprocessor performance has improved, and the energy cost per operation has decreased by many orders of magnitude, which have been resulted from equal efforts made in technology scaling and micro-architectural research (DANOWITZ et al., 2012; HU; STOW; XIE, 2018). However, as technology scaling slows down and we come to effectively notice the end of Moore's Law, the current process of manufacturing technology becomes more difficult and less beneficial to scale and shrink (KISH, 2002; TRACK; FORBES; STRAWN, 2017). To make matters worse, the scaling trends of off-chip memory bandwidth, which has historically been known as the major bottleneck of traditional architectures, are not as promising as the trends of on-chip computing capability.

As the traditional 2D manufacturing process could not diminish the performance gap between processor power and memory bandwidth, emerging die-stacking technologies have caught the attention of the memory industry to mitigate the effect of the bandwidth wall. By stacking several memory dies on top of each other and connecting them through dense vias, the 3D integration enabled high-bandwidth and high-capacity memory systems (HU; STOW; XIE, 2018). Although modern high-speed links and 2.5D designs have been used to deliver improved off-chip bandwidth, they still provide a lower bandwidth between the core die and the DRAM stacks (HASSAN et al., 2016), not to mention that energy consumption remains as a bottleneck to the overall system (HADIDI et al., 2017).

Due to constraints compelled by the end of Dennard scaling (DENNARD et al., 1974; ESMAEILZADEH et al., 2013), computer architects are required to come up with new designs to extract performance in order to provide more speed-up and consumes less energy. A viable approach to achieve such computing capacity consists of avoiding data movements by performing computation where the data resides, which is the main purpose of Processing-in-Memory (PIM) and Near-Data Processing (NDP) concept (SIEGL; BUCHTY; BEREKOVIC, 2016).

The main idea behind developing a PIM approach was to eliminate or at least to lower the memory wall (STANLEY-MARBELL; CABEZAS; LUIJTEN, 2011), the bandwidth wall (KAGI; GOODMAN; BURGER, 1996) and the power wall (POLLACK, 1999) gaps created by bringing data to be processed into the main processor. However, the insertion of 2D-integrated PIM brought some implications and problems that were not

completely solved and, thus, corroborated as the main reasons for a not widely PIM adoption. Only with the advent of 3D-stacking technologies, which enabled new opportunities for integration of distinct technology process, that PIM research was revived.

1.1 Current problem

3D-stacking PIM presents a massive change to the current hardware architecture design and, consequently, reveals new challenges in how applications can extract enormous acceleration. Researches have historically relied on software simulators to enable architectural studies and evaluate their impact on benchmarks and real-life applications. The primary goal of such simulators is to allow design space exploration of PIM architectures and, in some cases, serve as a virtual prototype that enables earlier software design. Some simulators used in previous PIM architectural researches include: gem5 (BINKERT et al., 2011) PimSim (XU et al., 2018), Clapps (OLIVEIRA et al., 2017a), SiNuca (ALVES et al., 2015) and zsim (SANCHEZ; KOZYRAKIS, 2013) and other in-house simulators (AHN et al., 2015).

However, most of these simulators struggle to deliver generic models for implementing PIM logic and a complete simulation platform encompassing both micro-architectural and system-level aspects. One of the essential requisites for PIM acceptance is easy programmability and system support, which may include code offloading, data coherence, virtual address translation, and other issues depending on the type of PIM logic. There is an increasing need for architectural and system-level approaches to solve these issues. Thus, the lack of tools for rapid prototyping can be a limiting factor to explore new solutions for PIM architectures, mainly when one wants to simulate full-system integration.

Many studies have relied on analytical models or separate simulators for evaluating their PIM designs (DRUMOND et al., 2017; ALVES et al., 2016). Nonetheless, analytical models do not provide important design metrics, and trace-based simulation is not as accurate as a real memory model driving the CPU model. Although this can be useful for a proof of concept, it can also be limiting for a proper design space exploration and also for serving as a virtual prototype, which requires more accurate simulations.

On the other hand, some of the more recent works have used highly accurate models in their experiments (OLIVEIRA et al., 2017a; LLOYD; GOKHALE, 2018), which yields useful statistics for a hardware-focused analysis. Although this approach can also

provide rapid prototyping, the System-C models experience high simulation time, and most of the hardware-related metrics may not be useful for system-level evaluation.

1.2 Motivation

Despite all advances related to different types of PIM architectures, some issues are still major concerns to enable the adoption of PIM at the system-level. Problems associated with the programming model, data mapping, runtime scheduling support, granularity of PIM scheduling and applicability to emerging memory technologies are just some of them (GHOSE et al., 2018a). The increasing interest in evaluating how PIM architecture can affect the entire compute-stack is essential not only to enable GPPs and GPUs to use PIM capabilities, but also to allow in-memory accelerators.

The primary goal of such system-level exploration is to allow designers to identify memory and PIM logic characteristics that affect the efficiency of PIM execution. For instance, one can evaluate the trade-off between the number of PIM engines and memory bandwidth, or affinity of PIM engine and data location. However, many optimizations can only be made at execution time and, then, must be made by a system mechanism to avoid increasing the energy of PIM computation.

The complex interactions between the CPUs, Input/Output devices, memory system, and PIM logic are captured by using full-system simulation. As few previous simulators cannot provide a complete environment for the system and architectural PIM research, there are open questions not only for evaluating the benefits of more accurate models, but also for investigating novel interactions between system-level and PIM capabilities. Likewise, virtual prototypes are essential to the industry, since they provide high-speed functional software models of physical hardware, enabling concurrent hardware and software development. Thus, a complete simulation environment can also serve as a virtual prototype.

1.3 Objectives

The main goal of this dissertation is to provide system-level information, which is translated into statistics, for different types of PIM logic. As much work has been done in system simulation, the first challenge consists of providing support for PIM simulation, which includes different programming and execution models, as well as correct models of memory system and system integration.

Secondly, as the nature of PIM can widely vary, generic PIM mechanisms become the second challenge of this work. It is expected to allow the development of system-level solutions which apply to a wide variety of PIM architectures. Finally, the last challenge resides on demonstrating the ability to use the simulator to perform design space exploration in novel PIM proposals.

1.4 Contributions

In this document, PIM-gem5, a PIM support for the broadly adopted gem5 simulator (BINKERT et al., 2011) and a methodology for prototyping PIM accelerators is presented. By using the PIM support described in this work, computer architects can model a full environment and address open issues on PIM research, such as connection to host processors, offloading mechanisms, Instruction Set Architecture (ISA) modifications, data coherence protocols, and address translation methods, just to list some of them.

The validation of the simulator includes two case studies of PIM logic placed alongside each vault controller of HMC, and we highlight the generic points of our implementation which can be used to exploit the efficiency of new PIM accelerators. Using the proposed implementation, we show the potential for energy reduction and performance improvement of different applications when compared to traditional architectures. Also, we demonstrate how any processor available in the gem5 platform can be simulated in a PIM fashion.

1.5 Document organization

This thesis is structured as follows: an overview of 3D-stacked memories and the research field of Processing-in-Memory is presented in Chapter 2. In Chapter 3, we present the recent developments on modeling and simulation tools, as well as system-level challenges that prevent PIM architectures to be broadly adopted.

Then, the implementation of the support for PIM simulation in gem5 is reported in Chapter 4, which is broken down into details of memory and PIM instance modeling, system integration, and host interface. Two case studies of PIM architectures are described in Chapter 5, which captures the potential of PIM-gem5 to simulate different PIM types. In Chapter 5.3 we present the experimental setup used to validate the memory modeling and evaluate the performance of the case studies. Then, in Chapter 6, we present the main results of the HMC and case studies simulation. Finally, some considerations about this

thesis and opportunities for future works are discussed in Chapter 7.

2 BACKGROUND

As the Dynamic Random Access Memory (DRAM) has been a de facto standard for main memory, some basic concepts of recent development on memory systems based on DRAM, which includes high-performance interfaces for 3D-stacked DRAM, are presented. Next, an overview of the research field on PIM architectures and taxonomy is described, as well as basic notions of simulation platforms for enabling the design and exploration of computer-system architecture.

2.1 3D high-density DRAMs

3D high-density memories rely on multiple stacked DRAM dies to provide high bandwidth and capacity to meet the demand of today's system workloads. Most of the today's major memory manufacturers develop 3D-DRAMs, such as Samsung's DDR4, Tezzaron's DiRAM4, AMD and Hynix's High Bandwidth Memory (HBM) and Micron's Hybrid Memory Cube (HMC). Due to the limitation of parallelism and bandwidth of Double Data Rate (DDR) interfaces for high-performance computing, different industry leaders have gathered efforts to propose high-performance RAM interface for through-silicon vias (TSV)-based stacked DRAM memories.

Although significant changes can be seen in these new interfaces over planar DRAM devices, 3D high-density memories use the same basic DRAM circuitry, array organization, and DRAM operations. For details of DRAM technology and devices, we refer the interested readers to (JACOB; NG; WANG, 2010; HANSSON et al., 2014; KIM, 2016; GHOSE et al., 2018b).

The HBM and HMC interfaces are the most prominent specification in the present time. For instance, the HBM is a JEDEC's standard composed of four DRAM dies and one single logic die at the bottom. Each DRAM die consists of 2 channels, where each channel has 1 Gb density with a 128-bit data interface and 8 independent banks (STANDARD, 2013; LEE et al., 2014). The logic die is divided in PHY, which is responsible for interfacing DRAM and memory controller, TSV arrays, and a direct access port for testing. HBM communicates with memory controller through a 2.5D interposer, which has 1024-bit. The available bandwidth with an 8-channel read operation is 128 GBps at 1.2V.

2.1.1 HMC architecture

As described in the last specification (CONSORTIUM et al., 2015), the HMC is a package containing either four or eight DRAM die and one logic die stacked together and connected by TSV, as shown in Figure 2.1. Within each cube, the memory is organized vertically into *vaults*, which consist of a group of corresponding memory portions from different DRAM dies combined with a *vault* controller within the logic die, as shown in the red region of Figure 2.1. Each HMC contains 16 or 32 *vaults* depending on the version and each *vault* controller is functionally independent to operate upon 16 memory banks in the eight DRAM layers configuration. The available bandwidth from all *vaults* is up to 320 GBps and is accessible through multiple serial links, each with a default of 16 input lanes and 16 output lanes for full duplex operation. All in-band communication across a link is packetized and there is no specific timing associated with memory requests, since *vaults* may reorder their internal requests to optimize bandwidths and reduce average latency.

A request packet includes an address field of 34 bits for internal memory addressing (vault, bank and DRAM address) within the HMC. The address mapping is based on the maximum memory block size (32B, 64B, 128B OR 256B) chosen in the address map mode register. This mapping algorithm is reffered to as ẗow interleave, and forces sequential addressing to be spread accross different vaults and then across different banks within a vault, thus avoiding bank conflict. The user can select a specific address mapping scheme to optimize the bandwidth based on the characteristics of the request address stream.

Some of the benefits of the HMC over the traditional DRAM modules can be summarized as:

Capacity: one of the advantages of the HMC architecture resides on the capacity and density problem of current DRAM devices. Due to the difficult of scaling DRAM cells, DRAM density have slowed in recent years. With stacked DRAM dies, a single cube can contain a multiple of 4 or 8 times the storage in the same package footprint as a single DRAM device. Even improvements in traditional devices, such as DDR4 standard, has 3D stacking extension to increase density without increasing pin count.

Parallelism and Aggregated Bandwidth: the high bandwidth of HMC is achieved by combining dense TSV (thousands of TSVs in each cube) and transferring at a high frequency. In addition to the TSVs, each cube has several high speed serialized links to

provide high-bandwidth to off-chip processors. Since each *vault* is operated independently, each one with one or more banks, there is a high level of parallelism inside of a cube. Each *vault* is practically equivalent to DDRx channels. With 16 or 32 vaults per cube, the *vault-parallelism* adds an order of magnitude within a single package. As vertical stacking allows a greater number of banks per vault, bank-level parallelism also increases the bandwidth within each vault.

Energy Efficiency: by reducing the length and capacitance of the connections between the memory controller and the DRAM devices with short TSV bus, the HMC is more efficient than traditional DDRx memories. The first measurements of HMC indicates 3.7 pJ/bit for the DRAM layers and 6.78 pJ/bit for the logic layer, while existing DDR3 modules spend 65 pJ/bit (JEDDELOH; KEETH, 2012). In a design space exploration, Weis et al. (2011) demonstrate that 3D-stacked memories like HMC can be 15x more efficient than an equivalent LPDDR from Micron. Current estimations considering the DRAM process scaling were not available in the literature.

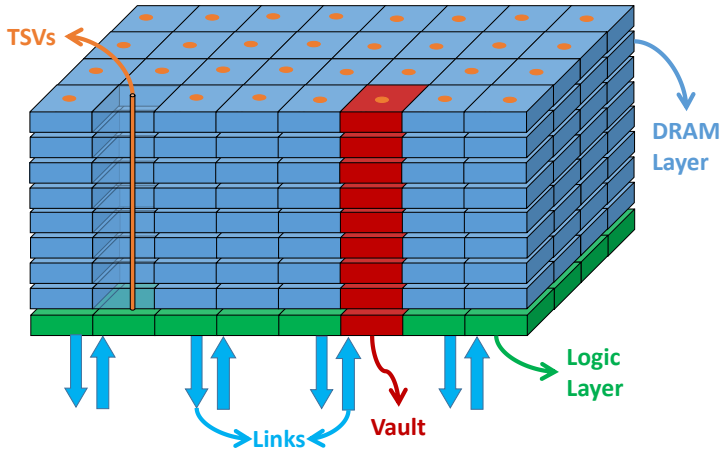
Interface Abstraction: Differently from DDRx systems, a CPU must use a general protocol to communicate with the cube or a topology of cubes that decouple memory controller functions from CPU. The CPU sends read and write commands, instead of traditional RAS and CAS commands, that are converted into device-specific commands within the *vault* controller. This effectively hides the natural silicon variations and bank conflicts within the cube and way from the host CPU. The DRAM can be optimized on a vault-basis without exposing the change to the CPU. (SCHMIDT; FRÖNING; BRÜNING, 2016; JEDDELOH; KEETH, 2012).

Near-Memory computation: as presented in the specification (Hybrid Memory Cube Consortium, 2013b), the logic die of HMC not only plays a role as vault controller, but also supports a set of atomic operation instructions. These instructions operate on 16-bytes memory operands and writes the result back to the DRAM arrays following a read-modify-write sequence.

2.2 Processing-in-Memory

Modern applications, such as data analytics, pattern recognition and bioinformatics, can benefit from PIM since they present poor temporal locality and can use in-memory computing instead of passing the data back and forth through the memory hierarchy (ZHU et al., 2013; SIEGL; BUCHTY; BEREKOVIC, 2016).

Figure 2.1: HMC overview



Source: provided by the author

2.2.1 History

The concept of PIM as presented today appeared in the 90s, where the main idea behind developing a PIM approach was to eliminate or at least to lower both the memory wall (STANLEY-MARBELL; CABEZAS; LUIJTEN, 2011), the bandwidth wall (KAGI; GOODMAN; BURGER, 1996) and the power wall (POLLACK, 1999) gaps. The first studies on 2D-integrated PIM brought some implications and problems that were not completely solved and, thus, corroborated as the main reasons for not having 2D-integrated PIM largely adopted. Only with the advent of 3D-stacking technologies, which enabled new opportunities for integration of distinct technology process, and also with the arising of data-intensive workloads that the PIM research field was revived and leveraged.

In the past decades, several attempts were made to include processing logic in different locations of the memory system. The main contributions are listed below and classified according to the location where processing units were placed:

Processing in the DRAM Module or Memory Controller: some recent works have examined how to process near the memory, but not within the DRAM chip, to leverage conventional DRAM modules or memory controllers (SESHADRI et al., 2015; ASGHARI-MOGHADDAM et al., 2016; HASHEMI et al., 2016). This approach was proposed mainly to reduce the cost of PIM manufacturing, as DRAM chip remains unmodified and it avoids the use of still costly 3D-stacking technology. However, it may suffer from challenges in programmability and consistent PIM interface to tackle the problems of address translation and cache coherence challenges. Moreover, the efficiency of PIM execution in this approach is limited, since it cannot take advantage of internal

memory bandwidth as in-DRAM chip mechanisms and 3D-stacked accelerators do.

Processing in 3D-stacked Memory: meanwhile, several works (KLEINER; KUHN; WEBER, 1995; LIU et al., 2005) studied the possibility of stacking memory dies and interconnecting them through very small vias, granting the emerging of 3D-stacked processing-in-memory approach. Moreover, the evolution of Through-Silicon-Via (TSV) technique (DAVIS et al., 2005; SAKUMA et al., 2008) solved some problems present on previous versions of 3D-stacked memory like thermal dissipation influences making feasible the production and exploitation of stacked memories as done by the HMC (Hybrid Memory Cube Consortium, 2013a) and HBM (STANDARD, 2013) products. Consequently, since 2013 3D-stacked PIMs have regained focus with different projects approaches, varying from multicore systems placed into the logic layer as in (PUGSLEY et al., 2014; AHN et al., 2016; AZARKHISH et al., 2016; DRUMOND et al., 2017; SCRBAK et al., 2017), alternative cores (NAIR et al., 2015; KERSEY; KIM; YALAMANCHILI, 2017), Single Instruction Multiple Data (SIMD) units (SANTOS et al., 2017; OLIVEIRA et al., 2017b), Graphics Processor Units (GPUs) (ZHANG et al., 2014; PATTNAIK et al., 2016) to Coarse-Grain Reconfigurable Arrays (GAO; KOZYRAKIS, 2016; FARMAHINI-FARAHANI et al., 2015a). Despite all advances related to 3D-stacked memory PIM, some issues are still major concerns in the recent 3D-PIM architectures such as how to perform offloading of instructions from the host processor to the PIM unit, data coherence among host processor and PIM units.

Processing within the memory chip or memory array: several recent works have investigated how to perform memory and arithmetic operations directly within the memory chip and also the memory array. These works take advantage of architectural properties of memory circuits and add bulk operations to them as new functionality to the memory chip. They can significantly improve computational efficiency and do not require 3D integration, although they still face the same challenges of processing in 3D-stacked memory. Most of the mechanisms in the literature following this concept rely on the bulk copy, data initialization (SESHADRI et al., 2013), bulk bitwise operations (SESHADRI et al., 2017; LI et al., 2016; KANG et al., 2017; ANGIZI; HE; FAN, 2018), and simple arithmetic operation (SHAFIEE et al., 2016; CHI et al., 2016) in different memory technologies.

Despite all advances related to 3D-stacked and in-DRAM PIM, some issues are still major concerns, such as how to perform offloading of instructions from the host processor to the PIM unit, cache coherence and interconnection communication network

between host processor and PIM, and also between PIM units.

2.2.2 Taxonomy of PIM logic

Regarding execution model, two main classes of processing units are identified by Loh et al. (2013): Fully programmable PIM and Fixed-function PIM.

- **Fully programmable PIM:** This class comprises full or simplified processors which fetch, decode and execute instructions from code offloaded to the PIM accelerator. Since existing core processors are used as PIM units, existing compilers can be used without any change. On the other hand, it generally requires programmer's efforts to manage the communication between host processors and PIM, and also requires the use of external libraries such as OpenMP and MPI (FANG et al., 2012; DRUMOND et al., 2017; NAIR et al., 2015).
- **Fixed-function PIM:** This class provides pre-defined simple FUs, or fixed operations based on existing memory access instructions. An extension of *LOAD* and *STORE* instruction could encode PIM operations directly in the opcode or as a special prefix in case of General Purpose Processors' ISA. Fixed-function PIM operations can be divided into Bounded-operand PIM Operation (BPO) and Compound PIM Operation (CPO). BPO comprises a single operation or a limited set of operations on single or multiple data (e.g., add and multiply-accumulate), while CPO includes a dynamic number of operations and arbitrary number of memory locations.

The authors do not limit their classification as the only one possible. In fact, they acknowledge that fixed-function can be more complex than the examples shown in their position paper. The exploration of fixed-function PIMs can lead to partially programmable designs, although they cannot be classified as fully programmable PIM since they do not present a mechanism for fetching instructions, translating virtual addresses, and other mechanisms present in fully programmable cores. In addition, there are several challenges regarding system-level decisions and compiler tools in fixed-function PIMs, since a new ISA organization, a different virtual memory translation, and data coherence mechanism, and other requirements may be expected.

In recent 3D-stacked PIM, most of the studies consider a logic layer organized into *vaults* similar to the one available on HMC devices, where simple processing units are placed within each *vault* to minimize communication costs (Hybrid Memory Cube

Consortium, 2013a). Also, in recent PIM architectures based on HMC architecture (AHN et al., 2016; SANTOS et al., 2017), each processing unit is placed next to its home *vault* router to reduce routing complexity and improve performance. Regarding communication costs, *fully programmable* PIM usually requires a complex memory hierarchy. Inside the *vault*, area and power dissipation are critical constraints that can lead to performance limitations (ECKERT; JAYASENA; LOH, 2014; ZHU et al., 2016; LIMA et al., 2018). In this case, communication between different cores across different *vaults* tends to harm performance, and/or require more attention from the PIM programmer (AHN et al., 2016).

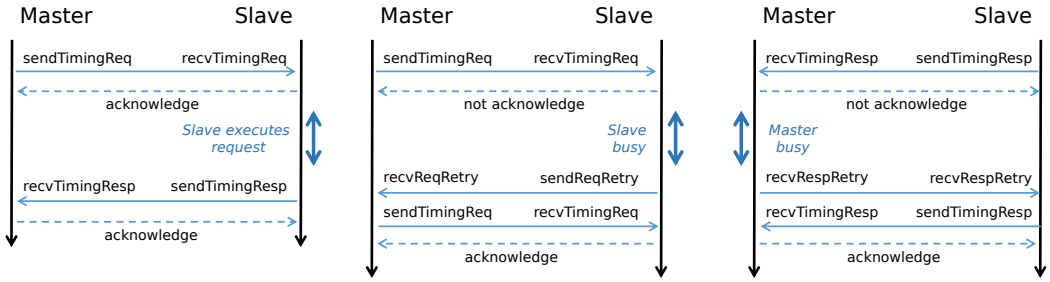
2.3 System simulation basics

Before diving into the implementation of the PIM support in gem5, some details regarding simulation mode, accuracy of simulation and interfaces for connecting models have to explained.

Although cycle-based models can be easily portable to different simulation frameworks, as they do not depend on any specific event semantics, a huge penalty is paid on simulation speed. Event-based models, instead, are only updated when part of the model has to be changed and, then, it skips ahead to the next event. The simulation speed is increased by orders of magnitude. Even though event-based models are tied to a particular simulation platform, many models created on it can also be applied to any discrete event simulator. The gem5 simulator (BINKERT et al., 2011) was chosen as basis of this work due to mature event-based models.

Apart from this level of abstraction, which is suitable for system simulation, gem5 enables three more modes. These simulation modes are closely coupled with the master/slave port interface, which are the most basic, rigid interface protocol in the simulator to implement communication between CPU models and memory system models. As summarized in Figure 2.2, the master/slave protocol allows four sequences of message exchanging in which the request/response is straightforwardly transmitted, or either the slave or master module denies the request/response when busy. These ports implement three different memory system modes: timing, atomic and functional. The timing mode is used to provided correct simulation results dynamic behavior since it makes use of events. Atomic and functional modes, instead, are used in special circumstances that do not require accurate results of the memory system, such as in fast-forwarding simulations and loading binary from the host to the simulated system's memory.

Figure 2.2: Three behaviors of master/slave protocol employed in the gem5 simulator



Source: provided by the author

In addition to hardware simulation modes, gem5 enables architects to use two modes with different system-level accuracy. The full system mode simulates not only the hardware, but also system details similar to emulators like QEMU[1] and hypervisors. This allows gem5 to execute unmodified OS binaries and investigate the impact of operating system, Input/Output devices and other low-level details Likewise, a less accurate system model is supported in the system emulation mode. This mode restricts gem5 to use only basic OS syscalls, which are imitated, rather than executed from the OS binaries. This mode does not require any OS kernel, device drivers, disk image or interaction with the OS.

The models are generally integrated in full-system simulation to form a complete architectural-exploration framework. For more details, we refer the readers to gem5 tutorial and documentation (LOWE-POWER, 2017; LOWE-POWER, 2019; BINKERT et al., 2011).

[1] https://www.qemu.org

3 RELATED WORK

This chapter presents the recent development of tools for exploring memory and processing-in-memory design space, as well as system-level challenges that prevent PIM architectures to be broadly adopted.

3.1 Memory modeling tools and simulators

When it comes to DRAM design space and architectural research, the tools are divided into DRAM bank/chip modeling and DRAM controller simulation tools. Developed by HP Labs, CACTI (CHEN et al., 2012) is one of the most cited tools for DRAM modeling. CACTI produces as output energy and timing parameters, and also an estimation of performance and power for future DRAMs, but it is not suitable for exploring advanced DRAM architectures or devices (WEIS et al., 2017). DRAMSpec (WEIS et al., 2017), in turn, generates datasheet timing and current parameters, as well as it provides support for novel memory devices, such as HMC. The authors claim that their tool enables to explore DRAM design space and key parameters very fast with enough accuracy. Also, the estimation tool is seamlessly integrated into gem5 (BINKERT et al., 2011).

The second type of tools features the DRAM controller simulation driven by previously collected memory-access patterns or by a simulated processor. Some examples of planar DRAM simulators include: DRAMSim2 (ROSENFELD; COOPER-BALIS; JACOB, 2011), Ramulator (KIM; YANG; MUTLU, 2016), DRAMSys (JUNG; WEIS; WEHN, 2015). DRAMsim2 provides a cycle-accurate simulation of DRAM using Verilog descriptions. Although current research efforts try to abstract from low-level accuracy to speed-up simulation time, DRAMsim2 is still a largely adopted tool. Ramulator offers an abstract C++-based simulation which part of the model needs to be clocked at a specific point in time. DRAMSys leverages transactional-level modeling to provide rapid simulation in conjunction with approximated timing to enable characterization of new DRAM subsystems and even allows the evaluation of 3D-stacked memories.

While previous mentioned simulators were focused on planar memories, recent simulators have been focusing on enabling 3D-stacked models, which includes HMC-Sim (LEIDEL; CHEN, 2016), CasHMC (JEON; CHUNG, 2017), Clapps (OLIVEIRA et al., 2017a) and gem5's simple HMC model (AZARKHISH et al., 2015). HMC-Sim provides cycle-accurate memory simulation for any of the supported HMC 1.0 and 2.0 configura-

tions. Though, the approach taken to implement HMC instructions harms scalability for future PIM extension and also the interoperability between different host operating systems. CasHMC is a C++ simulator that implements most of the HMC resources, as packet error detection, link flow control, and HMC instructions. The drawbacks of this tool are due to its offline simulation, which only uses an external memory trace as input and the lack of PIM extension. Clapps (OLIVEIRA et al., 2017a) is a cycle-accurate trace-driven HMC simulator that includes the HMC commands and a generic interface for custom PIM logic. As this tool is built on System-C library, it can be integrated into other simulation platforms such as gem5.

A previous attempt of modeling HMC architecture on gem5 was made by Azarkhish et al. (2015), and their contribution was added to gem5's repository. However, their experiments did not achieve the vault bandwidth reported by Micron due to mistakes in the memory interleaving. In fact, each vault controller received a single contiguous memory range of 256 MB, preventing the cube to exploit both vault-level and bank-level parallelism in sequential request streams. In addition to that mistake, this model of HMC had three modes of interconnection, but all modes presented distortions to the HMC specification. These issues are related to an incorrect behavior on the address range assigned to each serial link, either limiting each link to a specific memory range or adding extra latency to provide complete memory range to all links.

3.2 Simulating a PIM-based architecture

Most of the recent PIM works focus on fully-programmable cores, which are generally simulated by adjusting constraints of 3D integrated circuits in existing simulators and by taking advantage of existing execution models and compilers. For example, zsim (SANCHEZ; KOZYRAKIS, 2013) is a fast and scalable simulator that supports thousand of core simulation, including PIM models (GAO; AYERS; KOZYRAKIS, 2015; GAO et al., 2017; SONG et al., 2018). As the full-system support is compromised to enable scalability, zsim provides a collection of lightweight user-level virtualization to ease execution of unmodified benchmarks. SiNuca (ALVES et al., 2015) is an accurate and validated simulator focused on Non-Uniform Cache Architectures (NUCA) simulation, which was also used in the previous PIM studies (ALVES et al., 2016; SANTOS et al., 2017). Though, this tool requires traces generated on a real machine without the influence of Operating System or other processes.

Some past studies that consider a reconfigurable logic in-memory generally extend existing simulators. For instance, the CGRA model of (FARMAHINI-FARAHANI et al., 2015a) placed near commodity DRAM devices is simulated in the gem5 simulator (BINKERT et al., 2011), and the reconfigurable (GAO; KOZYRAKIS, 2016) is built on zsim (SANCHEZ; KOZYRAKIS, 2013).

On the other hand, fixed-function in-memory processing, which includes the HMC and the works of (AHN et al., 2015; GAO; SHEN; ZHUO, 2018; NAI et al., 2017; OLIVEIRA et al., 2017a), relies on a more varied design methodology which generally includes custom or in-house tools. Although there exist numerous PIM simulators, they still lack dealing with many challenges and difficulties in the PIM simulation. The first issue resides on the necessity of coupling a significant number of different tools to represent a whole computing system and its respective modules. In (XU et al., 2018), the authors presented a PIM simulator that relies on the integration of three memory simulators to support different memory technologies and one architectural simulator to provide interconnection and description of CPU architectures.

Likewise, in (YANG; HOU; HE, 2019) is presented a PIM architecture for Internet-of-Things applications which relies on the integration of one simulator for simulating both PIM and host processing elements and a tool for estimating power consumption. Coupling several simulators to represent the desired computing system incurs drawbacks to the design life-cycle, making this simulation approach prohibitive. When considering different simulation environments, the architectural designers must have complete and in-depth knowledge about the simulators features, which in turn demands time-consuming tasks. Also, since the involved simulators may have different accuracy levels, system modeling patterns, and technological constraint representations, the result of the simulation might not present the desired precision.

Although (YANG; HOU; HE, 2019) utilizes the same architectural simulator for all the hardware components, different simulation accuracy level components are instantiated to compose the whole system. Thus, the simulation approach followed by (YANG; HOU; HE, 2019) not only needs a particular synchronization mechanism but also does not reflect a real scenario where the host processor is represented by an event-detailed processor description and the PIM elements are described only with atomic and no-delayed operations.

Meanwhile, other simulators require the generation of trace files as input to feed them. The main drawbacks inherit from the trace-based simulation approach are the ne-

cessity of previous execution of the target applications in a real machine and the gathering of relevant information such as executed instructions and data access addresses. Although (XU et al., 2018) and (OLIVEIRA et al., 2017a) are built over architectural cycle-accurate simulators, the PIM modeling and measurements are done by analyzing memory traces gathered during the simulation.

To summarize the main aspects desirable for PIM simulation tools, we gathered some features from the related simulators in Table 3.1. The cells marked as "-" indicate that the feature is provided, but the behavior is not correct, which is the case of many HMC models in current gem5-based tools. According to the characteristics listed in this table, PIM-gem5 is the only one to support full-system simulations, accurate HMC models and system-level mechanisms for PIM designs.

Table 3.1: Summary of features

Simulator	Processor sim	HMC model	System-level support for PIM[1]	PIM extension	Full-system support
HMC-Sim (LEIDEL; CHEN, 2016)	x	x	x	✓	x
CasHMC (JEON; CHUNG, 2017)	x	✓	x	✓	x
Clapps (OLIVEIRA et al., 2017a)	x	✓	x	✓	x
Zsim (SANCHEZ; KOZYRAKIS, 2013)	✓	✓	x	x	x
SiNuca (ALVES et al., 2015)	x	✓	x	✓	x
PimSim (XU et al., 2018)	✓	x	x	✓	✓
gem5's simple HMC (AZARKHISH et al., 2015)	✓	-	x	✓	✓
(YANG; HOU; HE, 2019)	✓	-	✓	✓	✓
PIM-gem5	✓	✓	✓	✓	✓

Source: Provided by the author

3.3 System-level challenges for PIM adoption

Data-copy between PIM and host cores is a frequent challenge to be faced since most of PIMs share their embedded memory in a regular storage mode. The investi-

[1]We consider if there is any available mechanisms for PIM offloading, virtual address translation, data coherence and so on.

gation of the cache coherence problem is restricted mainly to three alternatives: non-cacheable data, fine-grain, and coarse-grain coherence. The use of non-cacheable memory region forces the host CPU to directly read data from memory (AHN et al., 2016; NAI; KIM, 2015; FARMAHINI-FARAHANI et al., 2015b), while the fine-grain solution uses traditional coherence protocols and flushes data back to the memory when needed (BOROUMAND et al., 2017). Other works use coarse-grain coherence or coarse-grain locks to prevent the host core to access data being used in PIM (AHN et al., 2015; HSIEH et al., 2016a; SESHADRI et al., 2013; SESHADRI et al., 2015). Another concern on PIM system-level infrastructure is the virtual address translation. For instance, IMPICA (HSIEH et al., 2016b) proposes in-memory support for address translation and pointer chasing operation, and DIPTA (GOKHALE; LLOYD; HAJAS, 2015) avoids traditional page frame translation and stores translation information next to the data to eliminate translation overhead in near-memory architectures.

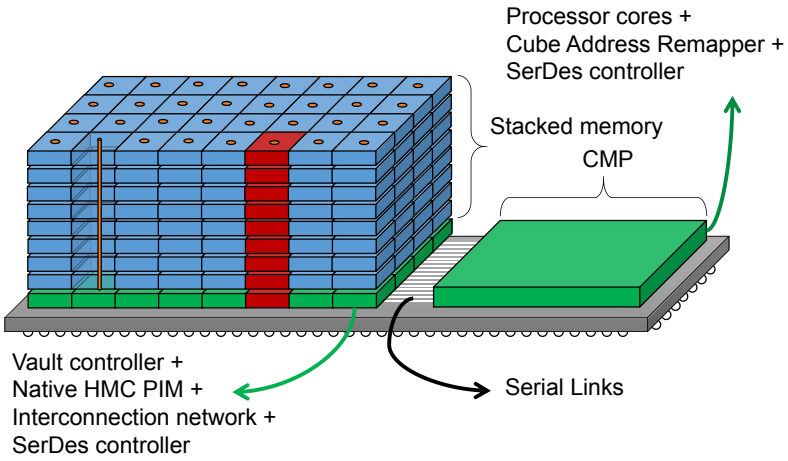
The PIM concept introduces new challenges in how programmers will interact with in-memory processing logic. The discussion about programming model involves how much control the CPU has over the PIM and the type of integration of PIM instruction using compiler-based methods and libraries. Recent works that tackle the problem of deciding between host and PIM instructions (AHN et al., 2015; HSIEH et al., 2016a). The PIM-enabled instruction (PEI) (AHN et al., 2015) consists of an ISA extension for PIM operations and a data locality-aware mechanism to decide whether instructions should be executed on the host CPU or PIM. Unlike PEI, the work of (HSIEH et al., 2016a) proposes a compiler-based mechanism to transparently offload instructions to PIM and also map data based on the code offloaded.

Another open issue in PIM research resides on determining the ideal place to store data and the ideal allocation of the workload to the in-memory compute units. Besides, (HSIEH et al., 2016a) that provides a programmer-transparent mapping of data and PIM instructions, the work of (SURA et al., 2015) hides memory latency by combining programming language features, compiler techniques, and operating system interfaces to map data and optimize data access. Nonetheless, there are still many opportunities for optimizations that take advantage of memory architecture properties, such as vault and banks location, to decide where data should be mapped.

4 SIMULATOR SUPPORT FOR PIM ARCHITECTURES

This section presents the implementation of reusable models for PIM architectural research in the gem5 simulator. The elements described here include system and hardware abstractions applicable to a wide variety of PIM styles. The following subsections report many decisions related to modeling accurately 3D-stacked memories, offloading mechanism, interconnection, PIM logic, and energy model. We divided the presentation into three major blocks: HMC modeling, PIM logic modeling, and, finally, host interface and system integration. An overview of the system integration that we intend to represent using these three blocks is shown in Figure 4.1.

Figure 4.1: An example of system integration: a PIM-enabled stacked memory connected to a Chip Multiprocessor (CMP) via serial links in a 2.5D package



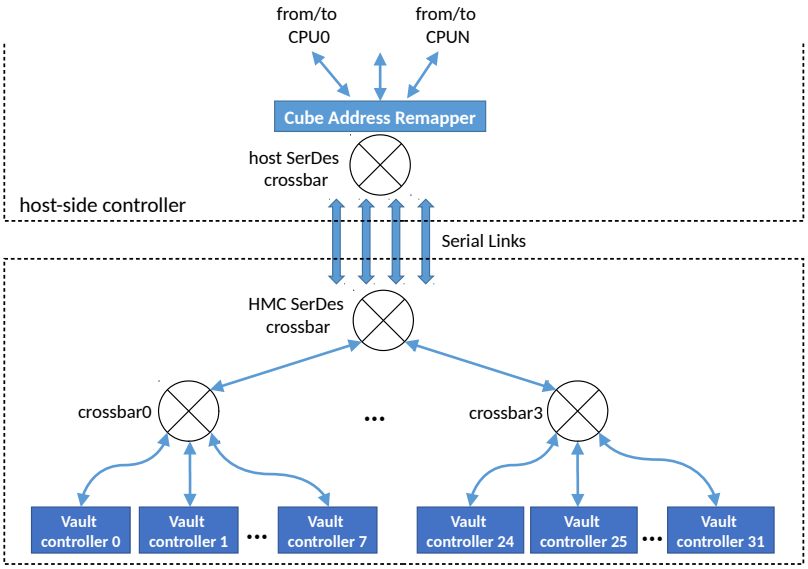
Source: provided by the author

4.1 HMC modeling

To model the HMC and also enable the extension for future PIM studies based on HMC, the specification (Hybrid Memory Cube Consortium, 2013b), related papers (JEDDELOH; KEETH, 2012; PAWLOWSKI, 2011; WEIS et al., 2011; HADIDI et al., 2017; HADIDI et al., 2018; HANSSON et al., 2014) and recent proposals of PIM interfaces were studied (AHN et al., 2015; SANTOS et al., 2017; OLIVEIRA et al., 2017a). The HMC is composed of three major blocks: the high-speed serial links (SerDes), switching interconnection and the vault controller. An overview of the objects that compose the HMC model is shown in Figure 4.2, and they are described in the following subsections. However, before diving into the challenges of implementing and extending the three ma-

jor hardware components of HMC, we focus on the difficulties found to represent the HMC interleaving correctly.

Figure 4.2: HMC model broke down into serial link, crossbar switch and vault controller objects



Source: provided by the author

4.1.1 Address mapping

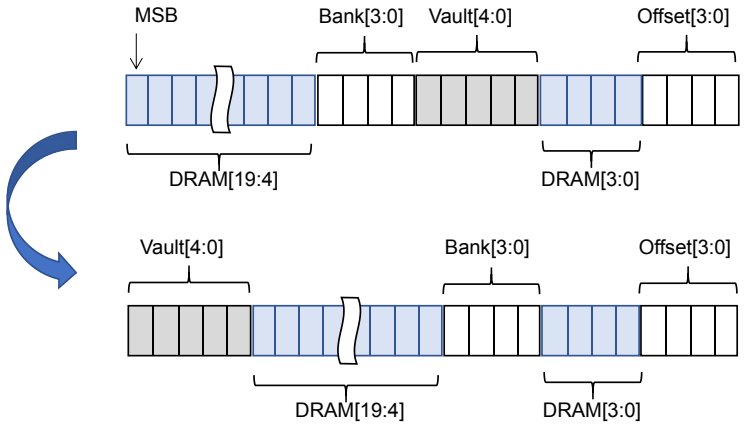
To correctly represent the HMC in the gem5 simulator, we had to make significant changes in the memory mapping mechanism to provide vault-level parallelism and suitable interconnection latencies in the way that it is specified in Hybrid Memory Cube Consortium (2013a). The first challenge is associated with the correct implementation of memory interleaving, which was already misinterpreted in a previous study presented in Subsection 3.1.

As aforementioned in Subsection 2.1.1, the maximum memory block size dictates the pattern of memory ranges addressed by a vault in the low-interleave mapping algorithm. In the gem5 model, these address ranges are given either by a single address range or a range list. Each vault controller receives a range list where the range addresses are spaced at regular intervals, which depends on the number of vaults and the configured block size.

The default approach provided by gem5's source code is heavy to simulate and cannot scale for ordinary memory sizes. This happens because, in a device of 8 GB, it can represent up to 1M entries of address range to form an aggregate range of a single vault

controller. Thus, iterating over a 1M entries to find the address range makes this choice prohibitive in terms of memory usage and simulation time. Then, a different interleaving mode is needed, and we provide the *Cube Address Remapper* to correctly forward requests from the serial link to the vault controllers.

Figure 4.3: Address map fields 256 Byte maximum block size and 8GB



Source: provided by the author

The *Cube Address Remapper* is intended to be implemented in the host-side HMC controller, and it enables modules within the HMC to forward packets using contiguous ranges seamlessly. This module moves the bits responsible for the vault addressing (*Vault[0:4]*) to the most significant part of the address as shown in Figure 4.3. By rearranging the address of memory packets before they are forwarded in the cube, we provide the so-called low-interleaved mode without extra overhead in simulation. Thus, for the purpose of ease the simulation, each vault controller receives a single address range and the components within a cube (serial links and switching interconnection) can correctly forward the packets using a reduced list of address ranges. The bitfields presented in Figure 4.3, with the exception of *Vault[0:4]*, are used to address the memory within a vault. *DRAM[4:19]* bits are used to address the row, *Bank[4:19]* bits address the bank, *DRAM[0:3]* bits, if applicable, are used to address the block of a given bank *b* and row *r*, and *Offset[0:3]* bits to address unaligned requests.

4.1.2 High-speed serial links

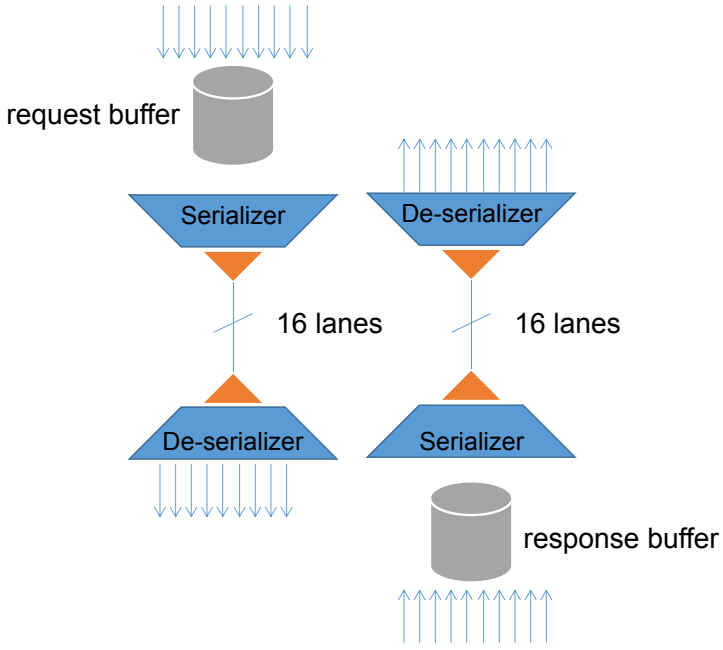
All HMC I/O is implemented as multiple serialized, full duplex links. Each serial link is composed of SerDes lanes, e.g., 16 input lanes and 16 output lanes by default, that carry system commands and data. Other SerDes configurations are also possible,

such as half-width and quarter-width setups. All in-band communication across a link is packetized, and a packet specifies single, complete operations. There is no specific time associated with each memory requests, and responses are generally returned in a different order than requests are made since there are multiple independent vaults answering requests.

Each serial link is composed of buffered slave and master ports. In timing mode accesses, an event on the master port side is responsible for handling a transmission list, i.e. packets in the request buffer. Likewise, the slave port side has a buffer for the responses not yet sent. In addition to that, the master port presents a simple traffic control by marking outstanding packets in the response buffer. By tracking outstanding responses, it is possible to maintain a maximum number of packets inside the cube and prevent data contention.

Table 4.1 shows the parameterized items of this model. These parameters are mostly related to static latencies, bandwidth and number of buffer entries. As the serial link object inherits the address range of memory and routing objects, the four links receive the address range of an HMC crossbar switch. To provide a full address range across all serial links, we added a crossbar switch to forward the requests coming from four serial links to one of the four internal crossbar switches with limited address ranges.

Figure 4.4: Crossbar switch model broke down into ports, buffers and switching layers



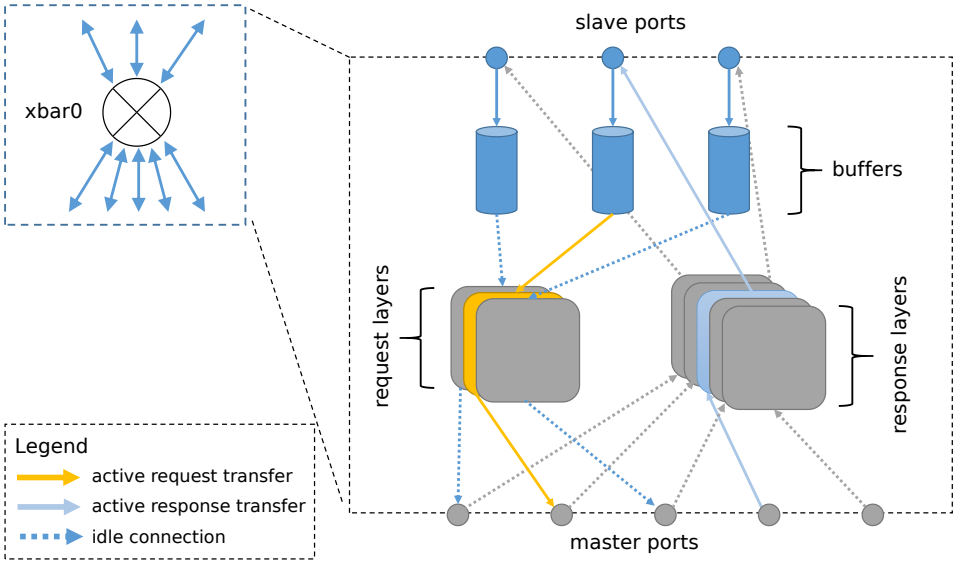
Source: provided by the author

4.1.3 Switch interconnection

The implementation of the interconnection network between serial links and vault controller is an open discussion. Although some past works have analyzed the impact of a Network on Chip on real hardware (HADIDI et al., 2017; HADIDI et al., 2018), this issue generally does not receive much attention, since different interconnection networks can provide the maximum bandwidth. On the other hand, the first announcement of HMC in the Hot Chips Symposium made by Jeddeloh and Keeth (2012) considers a crossbar switch as an alternative, which we have followed in this work.

A crossbar switch is an example of implementation of how the aggregate bandwidth from all vaults is made accessible to the Input/Output links. This portion of HMC is responsible for data routing and data buffering between Input/Output links and vaults, which is implemented by non-coherent buffered crossbar model. A detailed representation of this crossbar model is shown in Figure 4.5.

Figure 4.5: Crossbar switch model broke down into ports, buffers and switching layers



Source: provided by the author

Each link is associated with eight local vaults that form a quadrant. Accesses from a link to a local quadrant may have lower latency than accesses coming from a link of a distinct quadrant. Bit fields within the vault address designate both the specific quadrant and the vaults within that quadrant. Each internal crossbar is associated with eight local vaults, and the transfer of requests from its corresponding serial link has reduced latency. This behavior models the four quadrants present in HMC. We use four crossbar switches to simulate the quadrants specified in (Hybrid Memory Cube Consortium, 2013b). In

Table 4.1 we present the parameterized items of our crossbar switching model. These parameters are mostly related to latencies, bandwidth and number of buffer entries.

Table 4.1: List of parameterized items in the serial link and crossbar switch model

Model	Parameter	Description
Serial Link	Lane width	Number of parallel lanes inside the serial (Bytes)
	Link speed	Speed of each parallel lane inside the (GBps)
	Request buffer size	Number of requests to buffer
	Response buffer size	Number of responses to buffer
	Total controller latency	Static latency experienced by every packet (ns)
Crossbar switch	Width	Data width (Bytes)
	Clock frequency	Fixed clock frequency of all operations
	Request buffer size	Number of requests to buffer
	Response buffer size	Number of responses to buffer
	Frontend latency	Static latency experienced by every packet regardless if the bus is busy or can transmit
	Forward latency	Static latency to forward a packet
	Response latency	Static latency to transmit a packet in the response direction

Source: provided by the author

4.1.4 Vault controller

In this section, the mechanism and behavior of a memory controller targeted for HMC and PIM architectures are described. Our model of *vault controller* is based on the generic DRAM controller proposed by Hansson et al. (2014). Their fast, event-based model captures the behavior of modern DRAM devices, which can model either DIMM modules or can easily be adapted to model other DRAM interfaces such as Wide-IO (KIM et al., 2012), HBM (STANDARD, 2013) and HMC (CONSORTIUM et al., 2015).

As shown in Figure 4.6, the controller splits write and read queues for incoming requests and a shared queue for responses. The DRAM organization is captured by determining parameters such as bus width, page size, burst length, number of banks, ranks, and devices. Likewise, DRAM operations are modeled after event-based state machines that take DRAM timings as parameters. The controller model captures a subset of the DRAM bank state changes, along with data bus occupancy, refresh events, and data response events. For more details of the DRAM finite state machine, we refer the readers to Hansson et al. (2014).

Scheduling: The memory controller schedules read and write requests observing the DRAM timings and bank state machines with the goal of maximizing the efficiency

and exploiting the maximum bandwidth. Two levels of scheduling are provided. The first is related to read/write switching policy. The controller handles requests from read queue as the default state and implements a write drain mode as described in Narancic (2012). A parameterized threshold is used for forcefully changing the DRAM state from read to write queue, and a minimum number of writes to switch back to read queue. The second level of scheduling selects the request from either read or write queue as demanded by the first scheduling level, and it is tightly coupled to the row buffer policy. Two simple scheduling algorithms are available: First-Come-First-Served (FCFS) and First-Ready-FCFS (FR-FCFS) (RIXNER et al., 2000). Also, two basic row-buffer policies, namely closed and opened page, and two variations, namely adaptive closed and adaptative opened page, are provided.

The controller model uses the transaction-level gem5 port interface presented in Section 2.3. The flow control is made by monitoring queue size when receiving a packet or after a response event. The model has two timing parameters to capture the static frontend and backend latency of a memory controller. The frontend latency represents the pipeline stages and complexity of the controller design. The backend parameter captures the PHY and IO latency, thus allowing us to study the impact of different memory interconnection, e.g., Package-on-Package, Dual Inline Memory Module and Through Silicon Vias;

PIM logic interface: A PIM unit can be connected to arbitrary locations in the memory system, e.g., behind a crossbar switch, attached to a dedicated port of the memory controller, behind the row buffer, etc. To model such different locations, we provide an interface to connect PIM logic using dedicated slave and master ports to the memory controller. By using them, one can model the placement of PIM units inside a vault to take advantage of reduced communication costs. The specific location can be imitated by adjusting the timing parameters. In turn, to model PIM units externally to a vault, the implementation is straightforward since crossbar switch models are readily available.

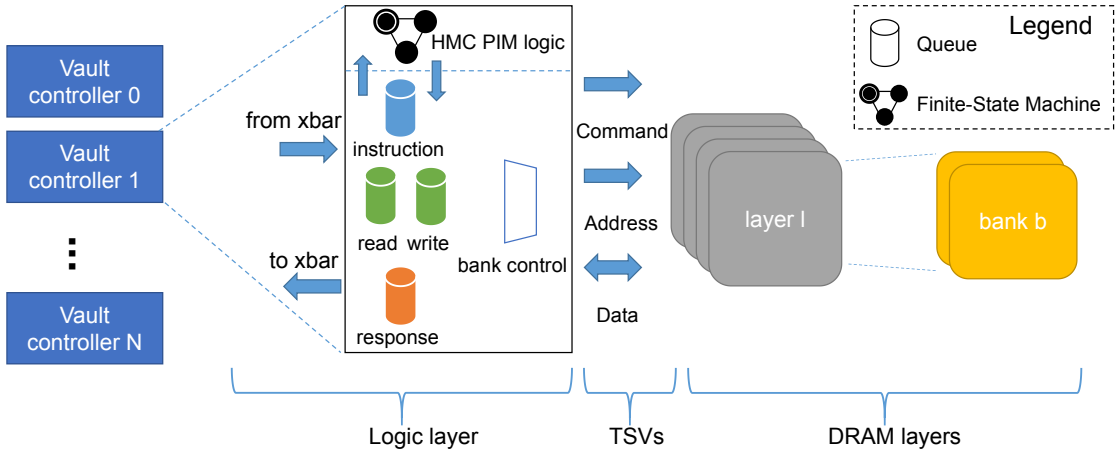
Regarding modeling specifics characteristics of HMC, we modeled the layers as ranks that share the same TSV connection in burst operations. The baseline address mapping of HMC is *RoBaVaOf*[1]. As the *Cube Address Remapper* rearranges the address as depicted in Figure 4.3, the address mapping had to be changed to *VaRoBaOf* to keep the bank-level parallelism.

Transactional operations, such as the read-modify-write (RMW) commands in HMC and the instructions presented in Section 5.2, require changes to the scheduling al-

[1]With Ro, Ba, Va and Of denoting row, bank, vault and offset, respectively, and going from MSB to LSB.

gorithm to keep data consistency. Instead of handling the requests strictly following one of the scheduling algorithms mentioned above, the memory controller must also check flags and timestamps of requests to the same address in the read and write queue. The changes made to both scheduling algorithms, FCFS and FR-FCFS, are responsible for skipping requests to the same address or overlapping address of an in-flight transactional request, then postponing ordinary requests till the transactional one has been finished. More details of transactional requests are presented in Section 4.3.5, which describes two concurrency models for lock-release and RMW requests that apply to different types of PIM commands.

Figure 4.6: Vault controller broke down into queues, bank control, PIM FSM, DRAM commands and DRAM organization (layers and banks)



Source: provided by the author

4.2 PIM logic modeling

As described in Section 2.2.2, the current taxonomy of PIM architecture is divided into two classes which have different requirements when modeling and simulating them.

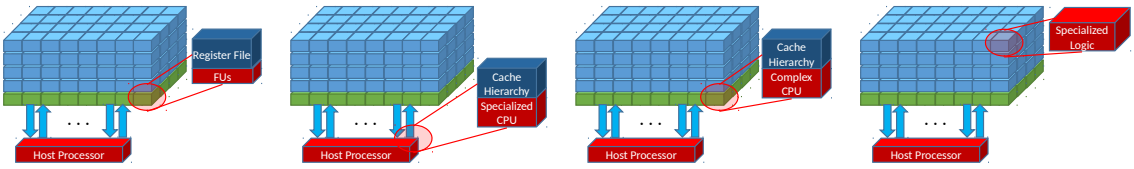
4.2.1 NDP and PIM with fully-programmable cores

First, fully-programmable PIM can be simulated in gem5 by adjusting parameters to represent TSV interconnection and by taking advantage of existing CPU models and compilers. In general, in-order CPUs are used as PIM logic since area and power are primary constraints for the logic layer of 3D-stacked DRAM memories (PUGSLEY et al., 2014; AHN et al., 2016; SCRBAK et al., 2017; DRUMOND et al., 2017). Regarding

software support for code generation and optimization, traditional tools for parallel programming can be applied, such as the m5thread library in syscall emulation mode, and unmodified pthread, OpenMP or other libraries, which can only be used in full-system mode.

In the present form, this work provides a correct and validated model of HMC to be used in fully-programmable PIM research. Since many CPU and ISA models are readily available in gem5, one can concentrate efforts in proposing novel techniques and tools for improving performance based on the application's characteristics and also for dealing with thermal constraints, to list some open problems. As the HMC model is solved and tested, the modeling of NDP/PIM architectures becomes straightforward. One of the decision resides on where to place ports of existing CPUs into the memory system. Figure 4.7 demonstrates different positions to place the processor cores inside or close to HMC.

Figure 4.7: Different position to place logic near memory



Source: provided by the author

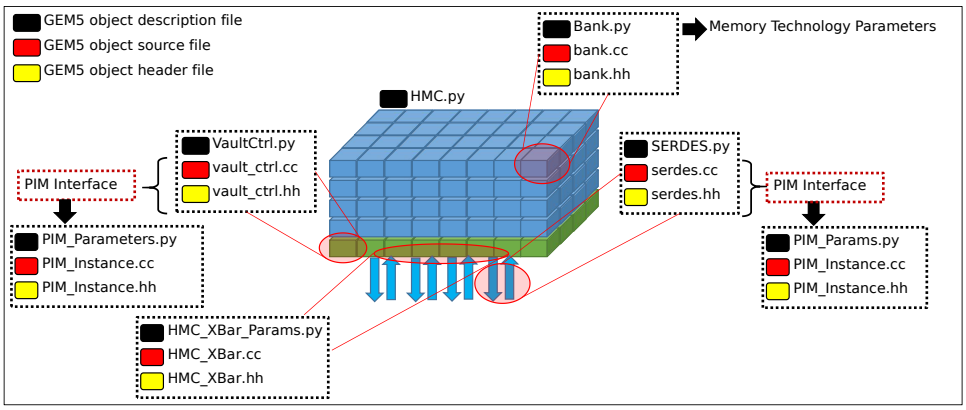
4.2.2 Fixed-function and Functional Unit-centered PIM

On the other hand, fixed-function PIM not always relies on existing models or hardware descriptions. Thus, modeling the behavior of a specialized processing unit is probably the first step. Figure 4.8 presents an overview of the source files used to describe the behavior and parameters of PIM instances and HMC model. Each module is defined by at least three files: the description file, where the parameterized values are exposed to the Gem5 simulation engine in Python, and source file and header file, where the behavior of the module is written in C++. To build the models related to fixed-function and FU-centered PIM, an architect must start by defining the communication protocol, datapath, and control unit. To couple processing units to a vault controller or to a crossbar switch, a PIM instance must implement the PIM interface (master/slave ports) to be able to receive memory packets that encapsulate PIM commands and data.

The PIM instance must implement a command queue and a decoder for the packets containing a PIM instruction. The datapath and control unit must follow one of the approaches to the event-based simulation: either by updating the model on a cycle-by-cycle basis or based only on meaningful events. The behavior of a PIM instance is arbitrary and can be implemented using a myriad of elements, such as pipeline stage and different issues (e.g., Load Store Unit, Arithmetic and Logic Unit (ALU), and Floating Point Unit). Statistics, such as busy cycles and the total of instructions ALU accesses, are optional, though they are valuable for energy consumption estimation.

Two models were created to evaluate the benefits of PIM in different classes of applications, which will be described in more details in Section 5. Before that, we present the basic support in host CPU (offloading, virtual address translation, data coherence).

Figure 4.8: Overview of source files used to implement fixed-function PIM logics in the gem5 simulator



Source: provided by the author

4.2.3 Power and energy

Originally, gem5 integrates McPAT (LI et al., 2009) and CACTI (CHEN et al., 2012) for area and power modeling of chip multiprocessors, which can still be used for power estimation of PIM with existing programmable cores. However, fixed-function designs need more details of the synthesis as input to an analytical model of the power consumption. Thus, we can offer a similar estimation by using results backed on the synthesis of RTL models and the statistics obtained during application execution. For complete results, the analytical model takes statistics, power models and design constraints to estimate the energy consumption of PIM cores. Statistics such as idle and busy cycles, number of floating point, integer and logic operations, just to list some of them, are mul-

tiplied by the static and dynamic power of each corresponding module of the processing core to give an estimate of the energy consumption.

4.3 Host interface and system integration

Several related works propose PIM architectures based on fine-grain offloading of PIM instructions, that is, PIM instructions are emitted one by one from a host CPU to the PIM logic. In fact, the native instructions of HMC already use a similar mechanism. In computer modules such as Micron's FPGA card AC-510, the PIM commands are generated in the FPGA's logic and transmitted to the HMC device following the protocol specified in Consortium et al. (2015). Unlike this FPGA board, new PIM proposals have to find a suitable mechanism to perform the communication between host and accelerator (AZARKHISH et al., 2015) or create new ones.

This section describes an offloading mechanism implemented in an Out-of-Order CPU model. To enable the use of this mechanism by compiler-based tools, a PIM ISA extension was added to the gem5's x86 ISA to support the binary code generated by the PRIMO Compiler (AHMED et al., 2019). The offloading mechanism described here is motivated by the proposal of (SANTOS et al., 2017) to make use of constrained PIM logic and to avoid sophisticated Instruction Level Parallelism (ILP) techniques in PIM units. Also, to prevent duplicating hardware and add more processing logic into the HMC logic layer, virtual address translation and data coherence protocol are described in this section.

4.3.1 ISA extension

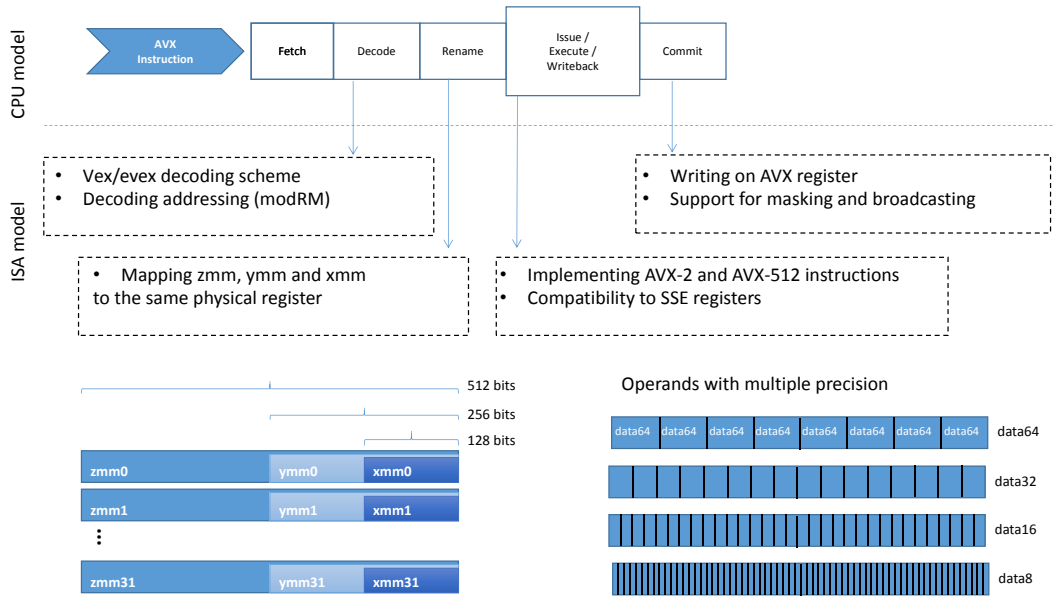
Giving support to a new ISA extension requires at least adding new decoding schemes into the decoder files. Though, this can also include more work to support new architectural registers and addressing modes in the CPU model. In this section we present the changes made to an existing ISA in two steps: a) adding an ISA extension described in a mature reference manual and supported by a commercial compiler, and b) adding an ISA extension for PIM operations, which a new academic compiler (AHMED et al., 2019) generates binary codes. The Intel's X86 family is the chosen ISA that serves as the basis for both extensions. The first step implements the Intel's Advanced Vector Extensions (AVX) ISA, and the second one implements a PIM ISA described in Ahmed et al. (2019).

The current version of gem5's X86 ISA available in the public repository[2] does not support any version of the AVX, only the Streaming SIMD Extensions (SSE). Then, we started by understanding how the ISA is simulated in a CPU model to include AVX-512 instructions. Also, the implementation of AVX to the X86 ISA was desirable for the case studies presented in Section 5.

The primary challenge for enabling AVX extensions resides on creating new register modes to represent vector operands. AVX extension specifies scalar and SIMD operations with registers and memory operands of different sizes, such as 128 bits (*xmm* registers), 256 bits (*ymm* registers) and 512 bits (*zmm* registers). A binary code targeting this ISA extension may contain the three operand sizes mixed. Then, another feature of the expansion must also be implemented for representing a smaller vector register contained within a larger vector register.

To correctly model a *zmm* register as a composition of a *ymm* register, which in turn is composed of an *xmm* register, it requires understanding and changing register reordering, mapping architectural registers to organization registers, and the basics of gem5's pipeline stages. Figure 4.9 presents an overview of the vector register arrangement to enable AVX-512 SIMD operations with up to 512 bits operands. Also, Figure 4.9 depicts the modules in X86 ISA and Out-of-Order (OoO) CPU model which were changed to support AVX-512 ISA.

Figure 4.9: Main modifications to include AVX-512



Source: provided by the author

[2]https://github.com/gem5/gem5

To not burden the scope of this document with a detailed description of AVX extension, some of the contributions related to Intel's AVX are summarized in the list below:

- Vector register operands in the OoO CPU model.
- New memory addressing modes, such as VSIB, vector addressing, compressed displacement.
- Support for Evex and Vex prefixes and corresponding decoding schemes.
- Support for mask register and predicate instructions.

4.3.2 ISA extension - PIM

Regarding the insertion of PIM ISA extension to gem5, three main challenges can be pointed out: a) how to distinguish instruction packets from ordinary read/write request packets, b) which type of PIM instruction can be supported and which modules of the host processor they must require, and c) how to send the instruction fields for decoding in the PIM logic through a packet in the memory system.

The first challenge demands one to encapsulate the PIM instruction as a memory request based on the behavior and data-path of a store operation. A store request is more suitable for performing instruction offloading, since data field is already expected to be sent and the operation is non-blocking for other memory requests. To distinguish the packet with PIM instruction in a CPU and throughout the memory system from the store request (WriteReq), a special flag (PIMInst) is set to the packet.

As fixed-function and FU-centered PIM have a limited ISA, and not all instructions available in the x86 family have a corresponding instruction in PIM, we classify these PIM instructions according to their dependence on the host CPU. To clarify that, let us consider the behavior of three arbitrary instructions: a) *register/register* instructions that only take PIM registers as source operands, modifies and store in PIM registers, b) *register/memory* instructions that take one memory operand as a source and store in PIM registers, or also take PIM register as source and store in the memory, and c) *CPU-register/PIM-register* that take one host register as a source operand and store in a PIM register or vice-versa. Thus, the presence or absence of memory operand or register operand captures the three different types of PIM instructions, which solves the second challenge.

Register/register instructions only require the host CPU to decode and carry the

instruction fields to the execution stage so that the Load-Store Queue (LSQ) unit will be able to assemble a request with the instruction fields in the data field. This brief bypass solves the third challenge, since the PIM registers are statically indexed and do not require register renaming. *Register/memory* instructions require address translation before assembling the request in the LSQ unit with the physical address and size of a memory operand. Also, *register/memory* instructions may require invalidating conflicting cache lines and checking any violation of the memory reordering mechanism present in OoO organizations. Unlike *register-register* instructions, *CPU-register/PIM-register* instruction requires dependence control and register renaming for reading and writing in host registers. Also, having a host register as destination requires handling the response coming from a PIM unit in the host write-back stage, which is the only operation which is blocking and synchronous.

Some examples of PIM instructions are shown in Figure 4.10. A register on PIM logic is addressed by the vault index (*Vault_0*) and register index (*Reg_0*). This PIM ISA extension provides fine-grain code offloading of PIM instructions to HMC architecture and other similar architectures (SANTOS et al., 2017). More details of the offloading mechanism are described in Section 4.3.3.

Figure 4.10: Example of PIM instructions

```
// register-register instructions
PIM_VPERM Vault_0_Reg_1, Vault_0_Reg_1, Vault_0_Reg_0
PIM_VADD  Vault_0_Reg_0, Vault_0_Reg_0, Vault_0_Reg_1
PIM_VMOVV Vault_0_Reg_1, Vault_0_Reg_1

// register-memory instructions
PIM_LOAD  Vault_0_Reg_1, ptr [rsp + 1024]
PIM_STORE ptr [rsp + 1536], Vault_0_Reg_1

// CPU-PIM register instruction
PIM_VMOV_PIMtoM zmm0, Vault_0_Reg_1
PIM_BROADCASTRD  Vault_0_Reg_1, zmm1
```

Although a ISA model may have a different way to represent operands and operations, all ISA models use the gem5's ISA description language to automate the generation of source code to emulate an ISA. For more details of inserting instructions in gem5, we refer to (ZHANG, 2015).

4.3.3 Offloading PIM instructions

As described in Section 3, prior studies have investigated different offloading methods for host-PIM logic interactions. The approach taken in this work is based on past studies (NAI et al., 2017; AHMED et al., 2019; SANTOS et al., 2017), where a PIM ISA extension allows instructions targeted to the host CPU and PIM units to be mixed in the same code, as illustrated in Figure 4.11. As described in Section 4.3.1, this hybrid code partially fulfills the code offloading to HMC and similar PIM architectures. Though, to finally carry out the code offloading, the host processor has to fetch, decode and issue the PIM instructions transparently to the PIM logic without or with minimal timing overhead, which is the focus of this section.

Figure 4.11: Example of hybrid code mixing X86 and PIM ISA

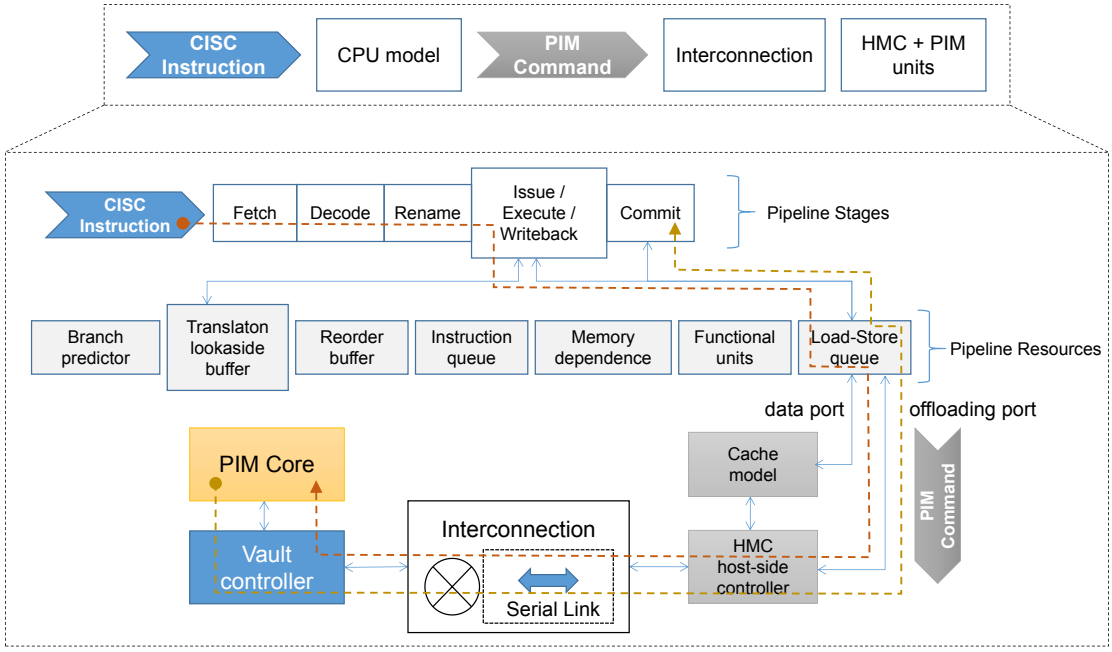
```
mov     r10, rdx
xor     ecx, ecx
PIM_256B_LOAD_DWORD   RVU_3_R256B_1, pimword ptr [rsp + 1024]
PIM_256B_VPERM_DWORD RVU_3_R256B_1, RVU_3_R256B_1, RVU_3_R256B_0
PIM_256B_VADD_DWORD   RVU_3_R256B_0, RVU_3_R256B_0, RVU_3_R256B_1
PIM_256B_STORE_DWORD pimword ptr [rsp + 1536], RVU_3_R256B_0
mov     eax, dword ptr [rsi + 4*rcx + 16640]
imul    eax, r9d
add     eax, dword ptr [rsp + 1536]
mov     dword ptr [r10], eax
inc     rcx
```

In our modeling, we built a generic PIM ISA upon the x86 ISA, and we use a two-step decoding mechanism to reuse modules present in any host CPU, such as TLB, page walker and even host registers. By reusing such components, we prevent hardware duplication in PIM logic and maintain software compatibility and memory protection. The workflow of the two-step decoding is presented in Figure 4.12.

The first step consists of decoding a Complex Instruction Set Computer (CISC) format instruction in the host CPU decoder. The decoded fields are used to calculate a virtual address if the instruction implies a memory request, or to access a host register, and also to encapsulate a PIM instruction into a Reduced Instruction Set Computer (RISC) machine format in the execution stage. From the host CPU view, all PIM instructions are seen as a store operation, which is issued to the LSQ unit, and then transformed into a memory request. In the LSQ unit, the *RISC* format instruction is copied to the data field of a new memory packet. From the LSQ unit to the memory system, the instruction is seen as a regular memory packet. The second step takes place on the memory side when the

packet arrives in the vault controller. The instruction fields are extracted from the packet and decoded by the PIM unit, where data will be finally be transferred or modified.

Figure 4.12: Flow of a PIM instruction in host CPU and memory system



Source: provided by the author

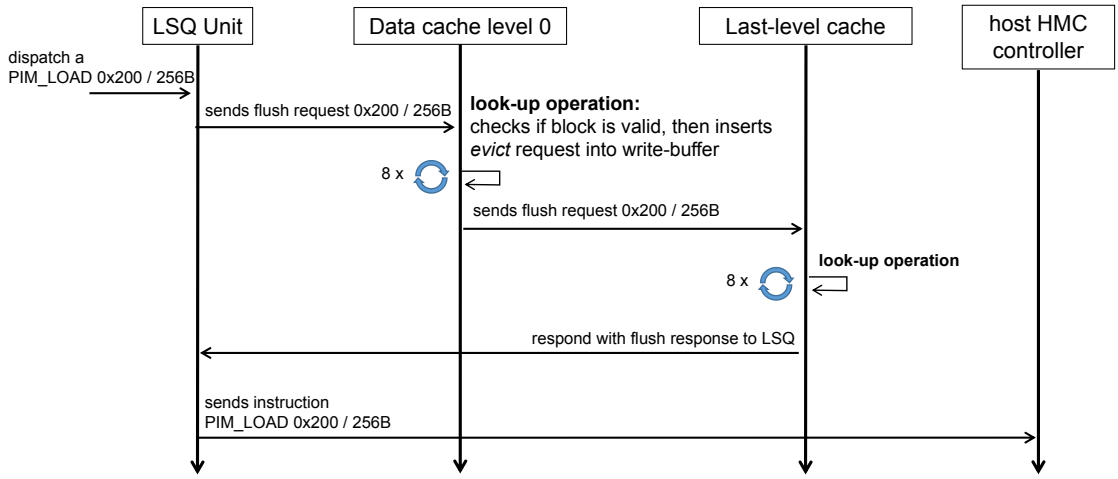
Despite the modifications on the host decoder and copy of instruction fields, the majority of changes were made in the LSQ unit as illustrated in Figure 4.12. This unit is responsible for violation checking between native load/store and RVU load/store requests, and for emitting *flush* requests to the cache memories and PIM instructions to the 3D-stacked memory. An exclusive offloading port connects the LSQ directly to the HMC controller on the host-side. The PIM instructions are dispatched in a pipeline fashion at each CPU cycle, except for PIM memory access instructions that need its data updated in the main memory and invalidated in the cache memories.

Before sending a memory access instruction, the LSQ unit unit emits a *flush* request to the data cache memory port to be handled in the cache memories, which is discussed in Subsection 4.3.4.

4.3.4 Cache coherence

Cache coherence in PIM architectures must not only keep shared data between cache memories and processors, but also between cache memories and main memory used in PIM mode. Since both PIM and host instructions have access to shared data,

Figure 4.13: Sequence diagram depicting the interaction between LSQ unit and cache model to provide data coherence for PIM instructions



Source: Provided by the author

a coherence mechanism is needed. However, traditional coherence mechanisms such as MOESI may not be enough to keep data coherent in PIM because such protocols will require intense traffic of snooping messages in a narrower communication channel, which may be a source of bandwidth overhead. Furthermore, time and bandwidth overheads are important issues that should be taken into consideration during the PIM design.

An algorithm to maintain coherence between host's cache memories and PIM-enable memories is shown in Figure 4.13. We included this protocol in the LSQ unit and cache model, which is triggered by PIM instructions with memory operands. Before sending a memory access instruction, the LSQ unit emits a *flush* request of the corresponding size (ranging from 4 Bytes to 8 KBytes) to the data cache memory port. The flush request is sent to the first level data cache and then is forwarded to the next level until it arrives in the last level cache, where it is transmitted back to the LSQ unit. At each cache level, a specific HW module interprets the *flush* request and triggers lookups for cache blocks within the address range of a PIM instruction that originated the *flush* request. If there are cache blocks affected, they will either cause a *write-back* or an *invalidate* command that is enqueued in the write-buffer and finally, the flush request is enqueued. For multi-core systems, an existing cache coherence, such as MOESI (SUH; BLOUGH; LEE, 2004), will be in charge of keeping data shared coherent.

4.3.5 Data coherence inside HMC

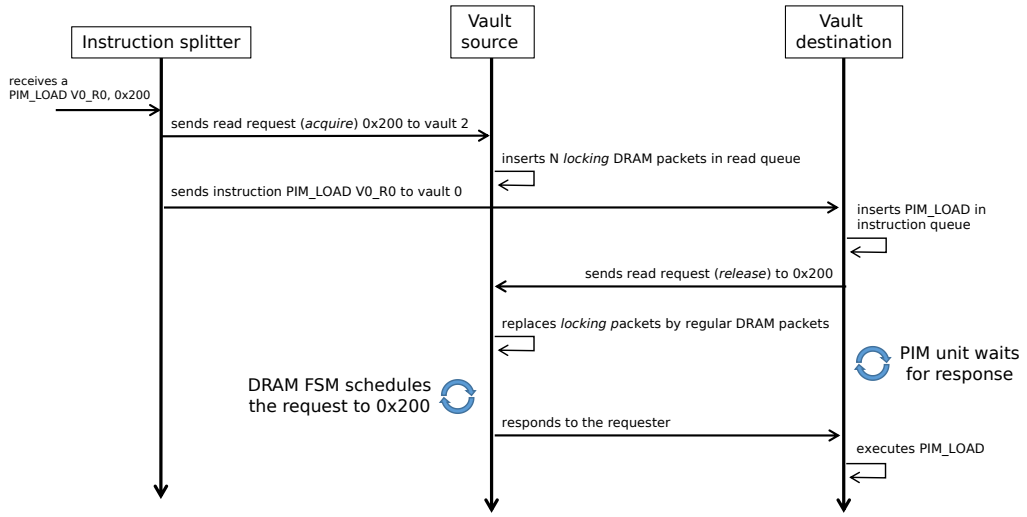
Since the instructions in the PIM logic share the same address space with *read / write* requests originated by the host CPU, it is likely to occur a data race within the HMC device. Likewise, even excluding the interference of requests from the host processor, a code region that triggers multiple PIM instances is prone to have a data race between requests from distinct instances. Leaving it unhandled can potentially cause data hazard, incorrect results and unpredictable application behavior. To solve this problem, we present two data racing protocols to keep requests ordered and synchronized: RMW and lock-release protocol.

The RMW protocol is designed to keep coherence of atomic HMC commands, and it is only applicable for the requests which are made by the PIM unit of the same vault. When the vault controller receives an HMC instruction, a read request is inserted in the read queue and marked with a specific flag, *RMW flag*. While there is a request with RMW flag in the read queue, no other read or write request to the same address can be scheduled. Then, as soon as the response is ready and the data is modified by simple functional units placed in the logic layer, the entry with the RMW flag can be removed from the read queue.

The lock-release protocol is used in the case study architecture described in Section 5.2. Although this protocol has the same purpose as RMW, it applies to non-atomic operations and does not restrict to perform the three operations (read, modify and write) in the same vault. The PIM instances of the case study are not limited to *read/write* requests within the same vault, which makes the first protocol inadequate. The second protocol keeps coherence and racing of host-PIM and PIM-PIM communication using an *acquire-release* transaction protocol.

To do so, we define three commands to use within the *inter-vault* communication subsystem: *memory-write* and *memory-read* and *register-read* requests. These requests can be used with either *acquire* or *release* flag and carry a sequence number related to the original PIM instruction. Given the request specification, we added a module between the HMC serial link and the crossbar switching tree. This module is responsible for emitting *acquire* requests, and also for splitting PIM instructions greater than 256 Bytes into smaller instructions, since each PIM unit of the case study can operate on operands of up to 256 Bytes. The *acquire* request is a dummy packet used to block a memory range in the scheduling algorithm of a vault controller, or also to insert a blocking entry of register

Figure 4.14: Sequence diagram depicting the interaction between vault controllers and the instruction splitter in a PIM_load instruction



Source: Provided by the author

read instruction in the PIM Instruction Queue.

Summarizing, when a PIM instruction is dispatched from the LSQ unit, it crosses the HMC serial link and arrives at the Instruction Splitter module, *acquire memory-read* or *acquire memory-write* requests are generated for memory access instruction or *acquire register-read* requests for modifying instructions involving different vaults. The interaction between the Instruction Splitter and the vault controller involved in this transaction is shown in Figure 4.14. As soon as these requests are enqueued, the module takes PIM instructions greater than 256 Bytes and split them into smaller instructions to fit within the maximum supported vector width, which is generally sized by the row buffer size (e.g., 256 Bytes). Thus, a *PIM_4096B_load* instruction must be split into 16 *PIM_256B_load* instructions with updated physical address and, then, they are forwarded to 16 distinct instances (from Vault 0 to Vault 15). Finally, when the PIM instruction is decoded in the Finite-State Machine (FSM), its LSQ unit generates a *memory-write*, *memory-read* or *register-read* request with a *release* flag. In the target vault controller, the *release* request will either unlock the *register-read* instruction in the Instruction Queue or remove a blocking request in the memory read or write buffer.

5 CASE STUDIES, METHODS AND MATERIALS

In this chapter, we will describe two PIM logic designs, namely Pointer-Chasing Accelerator (SANTOS et al., 2018b) and Reconfigurable Vector Unit (SANTOS et al., 2017; LIMA et al., 2018), proposed in recent studies. These PIM designs have two aspects in common: they implement operations considering it should be placed in the logic layer of HMC devices and also they rely on an offloading mechanism not clearly defined. Thus, they are candidate to make use of the HMC modules, as well as the modules for offloading mechanism proposed in the PIM support for gem5 simulator.

5.1 Pointer-Chasing Engine

In the first case study, an approach to process complex data structure by tackling the bottleneck of pointer chasing operations is presented. The mechanism of Pointer-Chasing Engine (PCE) uses in-memory speculation, which is based on the observation that a wider cache line can benefit pointer-chasing lookups since data nodes are generally allocated in memory contiguously (SANTOS et al., 2018b).

The PCE's ISA has a single *FIND* instruction responsible for triggering the traversing along a targeted data structure. The information that enables different node structures to be interpreted in PCE is encoded in the instruction fields, and it can be summarized as:

- ***structure type*** - This field indicates the type of data structure to traverse, whether it is a linked list, hash, or b+tree.
- ***base address*** - The address of a node where the search begins. It is the starting point and the first node to be loaded by PCE.
- ***key offset*** - The distance between the key or array of keys and the beginning of its node.
- ***key size*** - The size of key in bytes or the number of pages.
- ***next address offset*** - The distance between the pointer to the next node (or an array of pointers to its children) and the beginning of its node.
- ***gold*** - The value used as a key in the search.
- ***structure size*** - The size of a single element in the data structure, generally referred as a node.
- ***operand size*** - This parameter is used to configure the PCE size and also to group

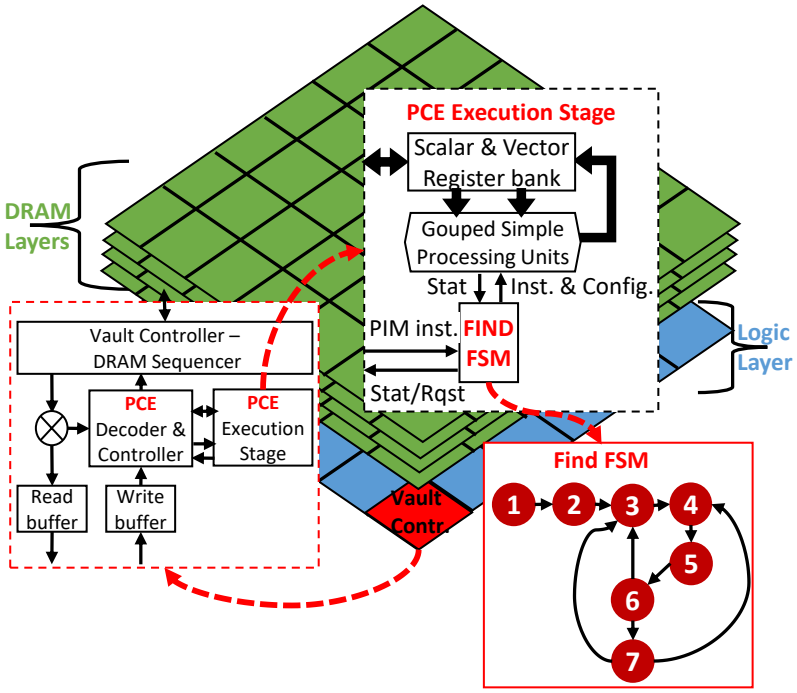
PCEs to form larger speculative operands. A single running PCE varies from 64 bytes to 256 bytes of speculative load size, a group of PCEs can vary from 256 bytes to 8192 bytes.

The entire PIM setup contains one PCE attached at each vault controller of HMC. Each PCE unit is composed of a finite-state machine that decode the *FIND* instruction into a set of fixed operations. Each PCE instance is composed of an 8x256-byte register file, a specialized Finite-State Machine (FSM), and a Functional Unit (FU) capable of executing scalar and vector operations such as *addition*, *comparison* and *gather/scatter* micro-instructions. The FSM is responsible for managing requests and triggering operations according to the information decoded from the *FIND* instruction. Four main operations are managed by the FSM:

- ***Load Generation*** - The FSM generates a *LOAD* operation with the address provided by a previous load or by a *FIND* instruction.
- ***Check Data*** - FSM checks if the data being searched has been reached. It selects the correct operand from the current vector register.
- ***Address Translation*** - Virtual addresses are translated directly by the PCE using SIMD operations. The algorithm for virtual address translation is the direct segment (BASU et al., 2013), which uses only the *base*, *limit* and *offset* registers.
- ***Internal Find*** - A PCE instance forwards the *FIND* instruction to another PCE when the required data lies on another *vault*.

When an instruction is offloaded to the HMC, it is forwarded to the corresponding *vault* of the *base address* field. Then, the algorithm presented in Santos et al. (2018b) starts to traverse the target data structure, which is illustrated by the seven states shown in Figure 5.1. The algorithm iteratively loads nodes from a contiguous memory region, stores them in registers and compares several keys speculatively to a gold. While the search is not finished, the PCEs calculate the physical address of the pointer to the next node, which may hit in the speculative region or may require another memory load. If no key matches up with the gold value, the flow comes back to the iterative part where the PCEs emit read requests to a contiguous memory region.

Figure 5.1: Overview of the PCE mechanism



Source: Santos et al. (2018b)

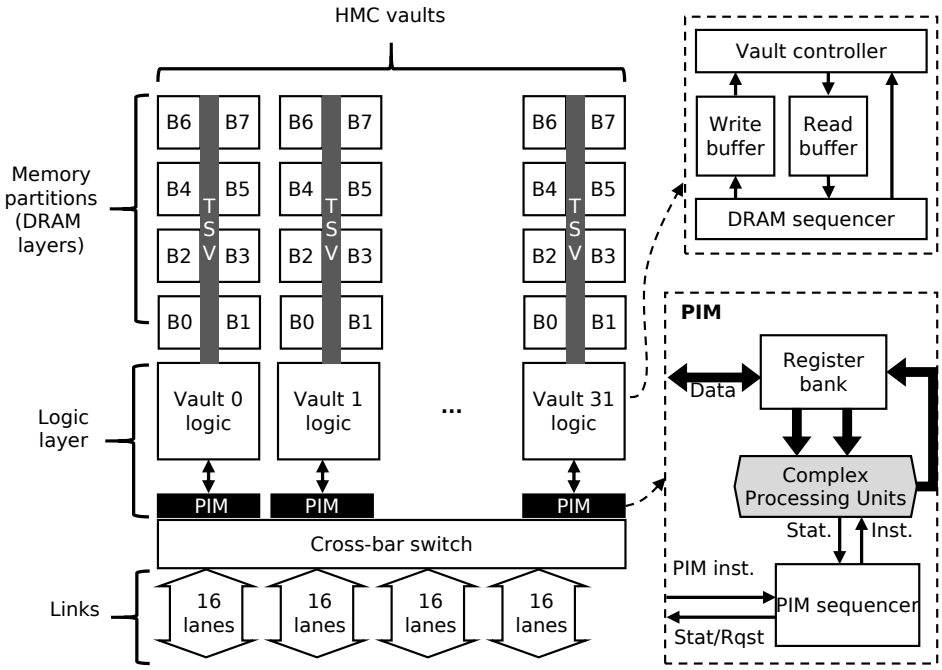
5.2 Reconfigurable Vector Unit

The Reconfigurable Vector Unit (RVU) micro-architecture (SANTOS et al., 2017) was chosen as a case study architecture to be placed in the logic layer of HMC. An RVU core comprises a set of 32x8-byte multi-precision Functional Units, an 8x256-byte register file, and a Finite State Machine (FSM) to control the flow of PIM instructions (SANTOS et al., 2017; LIMA et al., 2018; SANTOS et al., 2018a).

The PIM architecture proposed by Santos et al. (2017) considered several RVU cores, one in each vault and placed alongside the vault controller to reduce communication costs. Each RVU operates on variable operand sizes, varying from 4 Bytes to 256 Bytes, using scalar or SIMD instructions based on the native HMC ISA and the AVX ISA. Nonetheless, by using larger PIM instructions and an instruction splitting mechanism, it is possible to trigger several RVU instructions at once. Thus, when a PIM instruction aggregates RVU cores, they enable massive and adaptive in-memory processing capability and can achieve a peak compute power of 2.5 TFLOPS (LIMA et al., 2018).

Regarding programmability, each RVU only expects to receive PIM instructions in the RISC format as described in Section 4.3.3. Thus, the responsibility for finding DLP and delivering the instruction is entirely up to a compiler and host processor. Hence, the performance that this PIM architecture can achieve depends on the efficient exploitation

Figure 5.2: Overview of the RVU architecture



Source: (SANTOS et al., 2017)

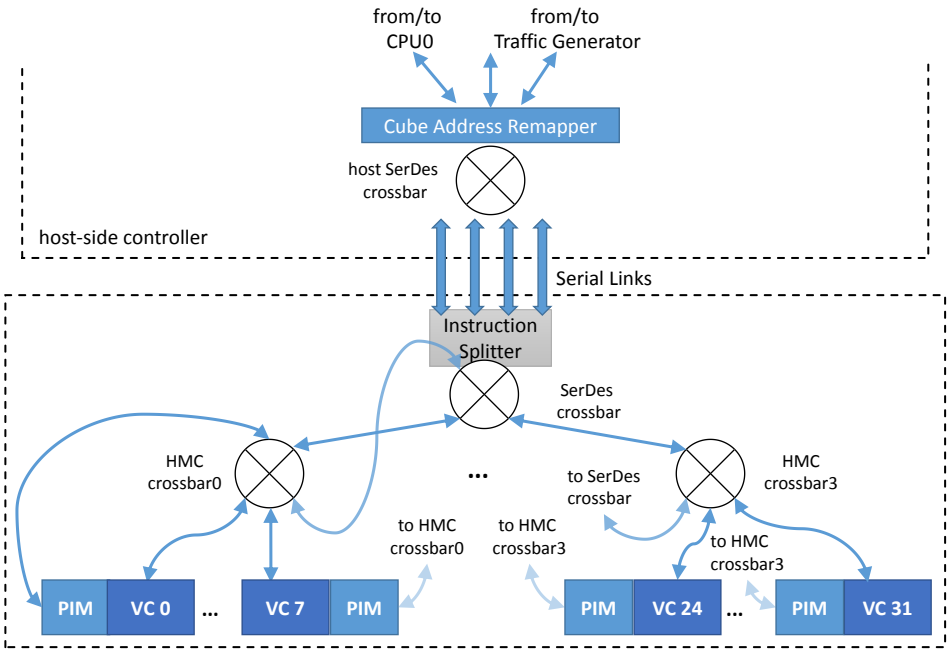
of DLP and data location by PRIMO compiler (AHMED et al., 2019) and other system-level techniques.

The RVU ISA has the following class of instructions: *memory access*, *register transfer*, and *data modification*. *Memory access* instructions, such as RVU *LOAD*, RVU *STORE*, and RVU *BROADCAST* with memory operand, rely on the existing host address translation mechanism, but the request is made by RVU's LSU. The *register transfer* instructions can move host register to PIM register and vice-versa, and also between PIM registers. Data modification instructions include arithmetic and logic operations, which represent the majority of the instructions described in the ISA specification.

All instructions are addressed by the destination RVU and an architecture-specific flag is set in the packet to differ it from typical *read* and *write* requests. The payload of an instruction packet comprises mandatory fields, such as opcode, register indexes for source and destination operands, data size, vector width, and optional fields, such as immediate, data from host register, and up to two physical addresses.

RVU instances are not limited to *read* and *write* requests within the same vault. Also, a PIM instance may use registers from other instances since the hybrid code considers all instances as part of the same execution flow sharing register files. So, we had to enhance the crossbar switching tree presented in Section 4.1.3 with bridges and ports for requesting in the opposite direction, thus enabling *inter-vault* requests inside the HMC.

Figure 5.3: Setup mechanism used for all experiments with fixed-function PIM



Source: provided by the author

Coupled with the data coherence mechanism presented in Section 2.1, this allows the RVU architecture to: a) synchronize and keep the order of memory requests as soon as the PIM instruction arrives in the PIM logic, and b) emit a read or write request from a RVU located in an arbitrary vault to any other *vault* of the same HMC.

5.3 Experimental setup

In this section, we present the experimental setup used to validate the memory modeling and evaluate the performance of the case studies of PIM architectures.

5.3.1 HMC architecture setup

The chosen DRAM organization is based on Micron's specification (Hybrid Memory Cube Consortium, 2013b) targetting high-end HMC devices, and the DRAM timings are provided by DRAMSpec (WEIS et al., 2017). The organization and timing parameters are summarized in Table 5.1. The latency of serial links were estimated based on the works of (KIM et al., 2013; AHN; YOO; CHOI, 2016) and the interconnection latencies were based on the (HADIDI et al., 2018). The organization shown in Figure 5.3 is used in the validation of HMC model and the case studies of PIM architectures.

Table 5.1: HMC configuration

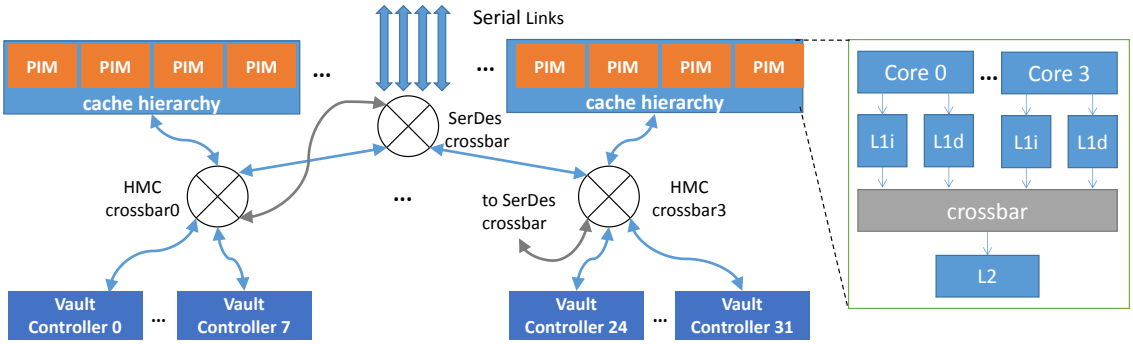
Vault controller	
Number of Vaults	32
Number of DRAM layers	8
Banks/vault	16
Memory size	8GB
Row buffer size	256B
Burst width	8B
CL,RP,RAS,RCD	9.9ns, 7.7ns, 21.6ns, 10.2ns
Row-buffer policy	close adaptive
Crossbar Switching Tree	
SerDes Crossbar	
Bus width	128B
Clock frequency	2.5 GHz
Latency frontend, forward, response	2, 2, 2
Buffer size req/resp	32, 32
HMC Crossbar	
Bus width	32B
Clock frequency	2.5 GHz
Latency frontend, forward, response	2, 2, 2
Buffer size req/resp	32, 32
Serial Link	
Number of links	4
Number of lanes	16
Lane Speed	40 Gbps
Delay	2 ns
Buffer size req/resp	64, 64

Source: provided by the author

5.3.2 PIM multi-core setup

Figure 5.4 presents an overview of the multi-core setup using the PIM concept. The reduced latency provided by TSVs is captured by connecting the CPU ports to the internal crossbar switch of HMC. Each 4-core group is composed of private and cache memories, which are attached to a quadrant crossbar. Following the HMC architecture presented in Figure 5.3, each quadrant crossbar (four in total) has access to the full memory range. Hence, the request latency may also depend on the distance between the PIM unit and vault controller, as well as queuing constraints. For this reason, data mapping and CPU affinity optimization would benefit multi-core setups, since the PIM units has reduced cost to memory access to the same quadrant.

Figure 5.4: Setup mechanism for PIM multi-core simulation



Source: provided by the author

5.3.3 Fixed-function PIM for pointer-chasing operations

Table 5.2 describes the configuration of the baseline systems employed to evaluate the PCE mechanism, and also the internal configuration of the PCE. The baseline is an ARM-A57 + 1MB of last-level cache. In our analysis, we explored different configurations for the PCE, ranging its operand size capabilities from 64 bytes to the limit of 8192 bytes. We extrapolate the last-level cache size for the baseline to 2M Bytes, aiming to evaluate the benefits of a larger cache memory for the pointer-chasing operation.

In this section, we also present energy results of the proposed architecture. We estimated energy by synthesizing a Hardware Description Language of PCE using Cadence RTL Compiler Tool with a technology node of 32nm. We also consider the power consumption of the host processor connected to the pointer-chasing accelerator. For the baseline ARM processor and cache memories, we are based on McPat (LI et al., 2009) coupled with Cacti (CHEN et al., 2012) tools extrapolated to a technology of 22nm. The HMC organization follows the parameters presented in Section 5.3.1.

The design is evaluated using the same three data-intensive micro-benchmarks employed in related works (HSIEH et al., 2016b; HONG et al., 2016), following the parameters presented in (HSIEH et al., 2016b). The micro-benchmarks are composed of linked lists (varying the memory access pattern from continuous insertions to 25%, 50% and 100% random insertions and deletions), a hash table from (FITZPATRICK, 2004), and the b+tree implementation of DBx1000 (YU et al., 2014).

Table 5.2: Baseline and Design system configuration of PCE

ARM A57
2.5 GHz; 4 cores; NEON Instruction Set Capable;
I/D 64kB L1 Cache 2 Cycles + 16-way L2 Cache 1MB 20 Cycles;
Power - 8W;
PCEs
1.25 GHz; 32 Independent Vector Units;
Vectorial Operations up to 256Bytes per Units;
Vector Register Bank of 8x256Bytes each;
Scalar Register Bank of 8x32 bits each;
Latency (cycles): 1-alu, 3-mul. and 20-div. integer units;
Interconnection between vaults: 5 cycles latency;
Host Processor - 1.2GHz ARM Cortex A8; IL1 64kB + DL1 64kB;
Power - PCE Logic = 3.1W estimated;
Power - Host Processor = 0.6W;

5.3.4 Reconfigurable Vector Unit

To experiment and evaluate the RVU and reconfiguration techniques, the RVU architecture was implemented for simulation and tests as presented in Section 5.2. For compiling the source code application tests and generating the binaries, PRIMO (AHMED et al., 2019) was used as a support compiler tool. Table 5.3 summarizes setup simulated, it comprises an Intel *Skylake* micro-architecture as the host processor and an HMC RVU capable module as main memory. As the RVU core is based on AVX-512 ISA, the x86 processor available on the chosen simulator was modified to support both the targeted PIM ISA and the AVX-512 ISA.

Further, we use a subset of the PolyBench benchmark suite to evaluate the impact of the proposed architecture in most of scientific kernel applications (POUCHET, 2012). Table 5.3 summarizes the setup simulated.

The energy and power models were obtained by synthesizing the VPU design provided by (LIMA et al., 2018). Supported by Cadence RTL Compiler tool, we extracted area, dynamic and static power for this implementation using 32nm process technology.

Table 5.3: Baseline and Design system configuration of RVU

Intel Skylake Microarchitecture
4GHz; AVX-512 Instruction Set Capable; L3 Cache 16MB;
8GB HMC; 4 Memory Channels;
RVU
1GHz; 32 Independent Functional Units; Integer and Floating-Point Capable;
Vectorial Operations up to 256Bytes per Functional Units;
32 Independent Register Bank of 8x256Bytes each;
Latency (cycles): 1-alu, 3-mul. and 20-div. integer units;
Latency (cycles): 5-alu, 5-mul. and 20-div. floating-point units;
Interconnection between vaults: 5 cycles latency;

6 RESULTS AND DISCUSSION

In this chapter, we present the potential for simulation of new PIM architecture based on 3D-stacked memories. To simplify the analysis, we have divided our results into three sections. In the first section, we present the validation of our HMC model. Then, we show potential gains of the fixed-function PIM described in Section 5.1. Finally, we show the results of a programmable FU-centered PIM and their design space exploration.

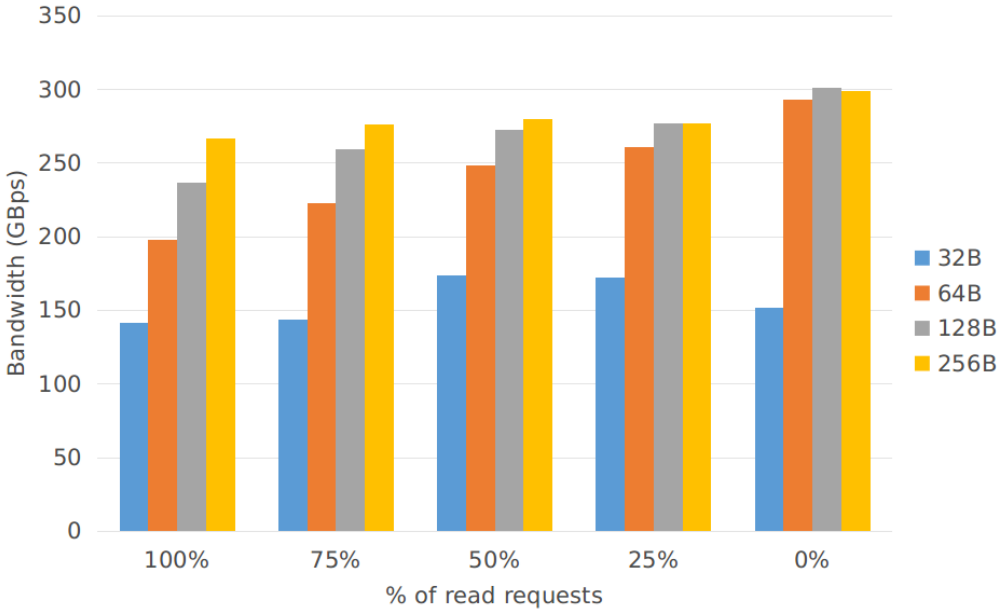
6.1 HMC validation

Since the HMC's memory vendor provides little information about the internal configuration, the validation is not limited to HMC's datasheet, but we also validate our results based on previous studies using the Micron's FPGA card AC-510 (HADIDI et al., 2017; HADIDI et al., 2018) and a thesis that explores the design space of this memory device (ROSENFELD, 2014). Thus, we have to evaluate if the parameters provided in Section 5.3.1 represent the memory by comparing basic statistics, such as memory bandwidth and latency.

A stress test is performed to find the maximum memory bandwidth and latencies in two memory access pattern, linear and random access, through the whole HMC during 10ms. We set up a traffic generator to generate packets at a higher rate to pressure the memory system in the two access patterns. In the linear mode, the traffic generator creates read/write requests in a way that their address pattern allows us to explore the maximum bank-level parallelism. In the random mode, the requests are generated randomly without avoiding bank conflicts. To observe the effects of requests' characteristics, we varied the request size from 32B to 256B, which aims at simulating different cache lines size, and we also varied the read/write ratio from 0% of read requests to 100% with a step of 25%. The traffic generator is attached to a host HMC controller, as presented in Figure 5.3.

Figure 6.1 and Figure 6.2 show that greater request sizes (128B and 256B) enable the vault controllers to achieve a higher bandwidth. The reason for a single request of 256B be more beneficial than four aligned requests of 64B relies on the minimum timing between the requests that a vault controller has to respect. Also, the maximum aggregate bandwidth is provided when 100% of the requests are write ones for any access pattern. Although Rosenfeld (2014) has already observed this behavior that relates higher ratios of read/write to less efficient TSV usage, the bandwidth achieved by their simulations is far

Figure 6.1: Aggregate vault bandwidth for linear access pattern



Source: provided by the author

lower than the results presented in this work. The maximum bandwidth this work could achieve is 299 GB/s for write requests only, and 267 GB/s for read requests only, both using with request size of 256B.

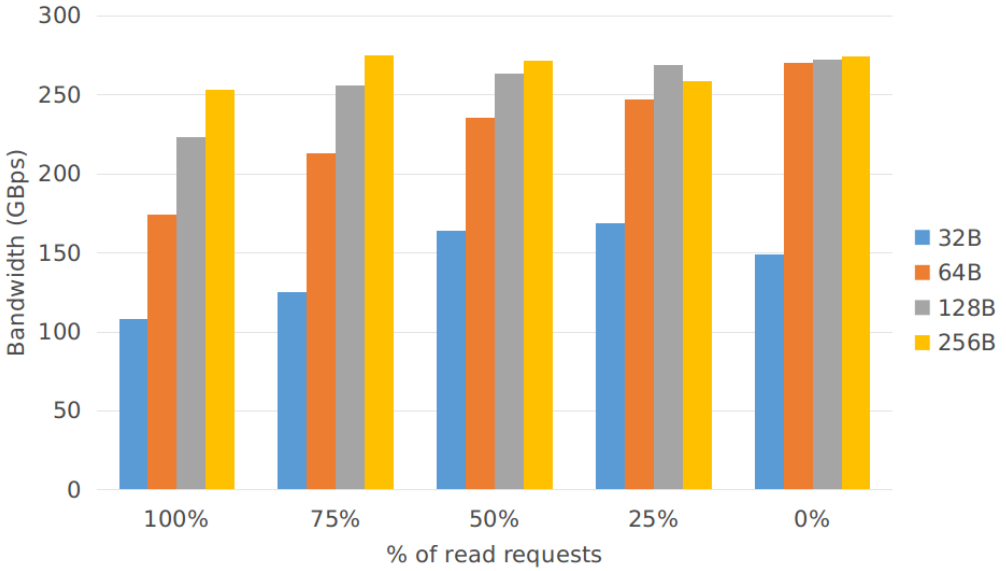
In the random mode, it is expected a significant performance degradation due to bank conflicts. However, the random pattern can also cause write requests to be merged in write queue and read quests be responded from the write queue, thus increasing the vault bandwidth. After all, the random mode reduce the bandwidth by up to 9% when compared to the linear mode.

Figure 6.3 and Figure 6.4 presents the average latency of read requests for the two above-mentioned access patterns. The results presented in Figure 6.3 suggest that the ratio of read/write requests significantly impacts the latency observed for a read request. Also, by increasing the percentage of write requests, the latency perceived by the requester is also increased, since the memory controller has to constantly switch between read to write queue.

In comparison with the random mode when 100% of the requests are read ones, the linear pattern can fully take advantage of the pipeline of the DRAM scheduler and provide the smallest latencies for any requested size. In random patterns, though, the increased latency is due to bank conflicts, that inserts bubbles into the pipeline. However, smaller request sizes provide reduced latencies in random access patterns.

In both access pattern shown in Figure 6.3 and Figure 6.3, the average latency

Figure 6.2: Aggregate vault bandwidth for random access pattern



Source: provided by the author

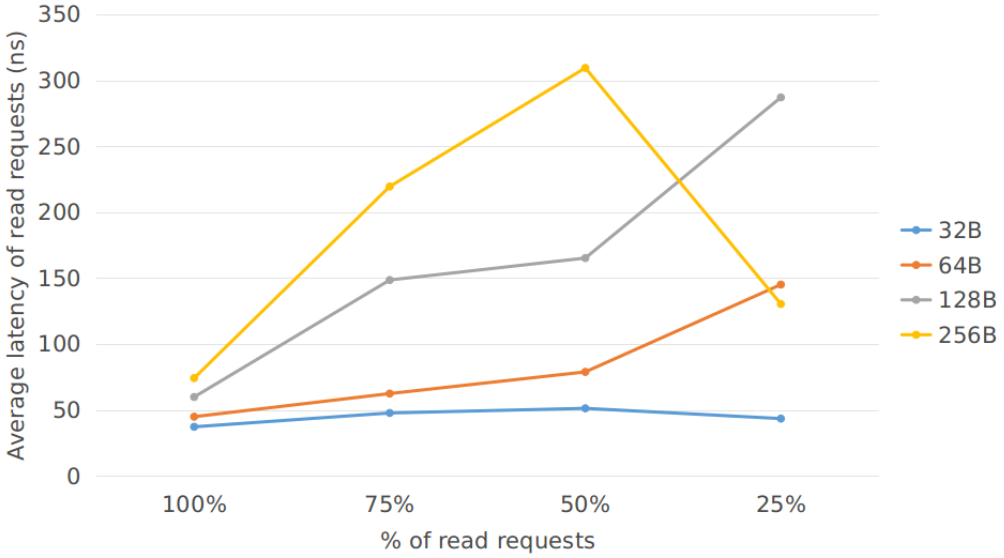
drops when the read/write ratio is 0.25 for 256B request size. This drop is due to the combination of two features of the controller scheduler. Firstly, a 256-byte requests can take advantage of the maximum row-buffer size and the scheduler can optimize the access while reducing the bank conflict. Secondly, as more than 50% of the requests are write ones, the scheduler's FSM spend more time in the write queue, while the read ones are already scheduled and in the fly. Also, this drop is impacted by the disparity of latency between write and read operations, seeing that write operation are faster than read ones.

6.2 PIM multi-core

The gem5 platform has been widely used to perform architectural exploration on chip multiprocessors, either creating homogeneous or heterogeneous multi-core systems. Although the modeling of such scenarios is simple, the design of PIM multi-core has open issues regarding thermal and power dissipation, which have become the central problem of many past studies. Figure 6.5 presents the execution time of Polybench kernels when running on 16 in-order cores in PIM style normalized to 1 OoO core in traditional style.

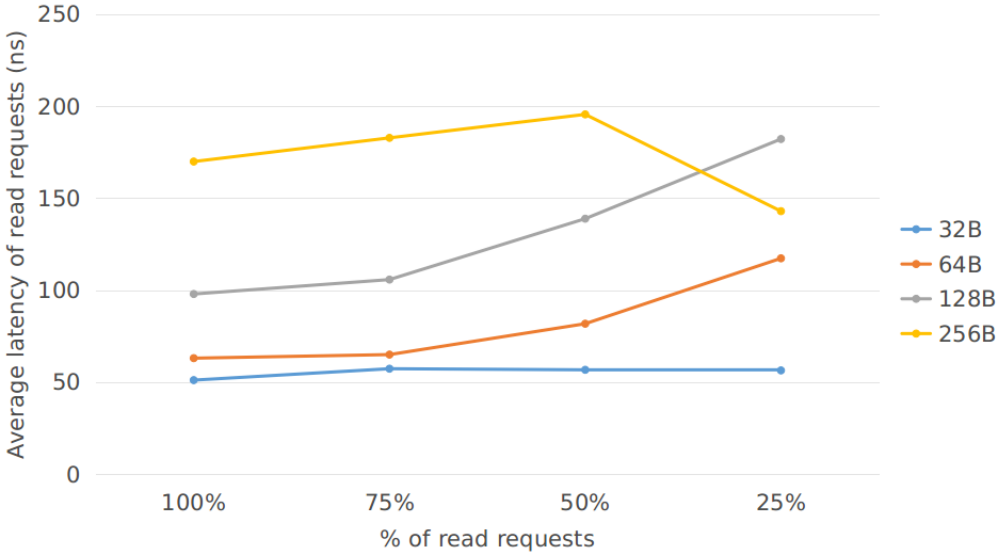
The main challenges on PIM multi-core design resides on the system setup and software infrastructure, such as code partition and thread optimization targeting near-memory architectures rather than traditional memory hierarchy. In Figure 6.5 we could not achieve a significant reduction of execution time since little effort was put in the task partition. For most of the applications, the 16-core setup have lower execution time, which

Figure 6.3: Average latency observed for read requests in linear access pattern



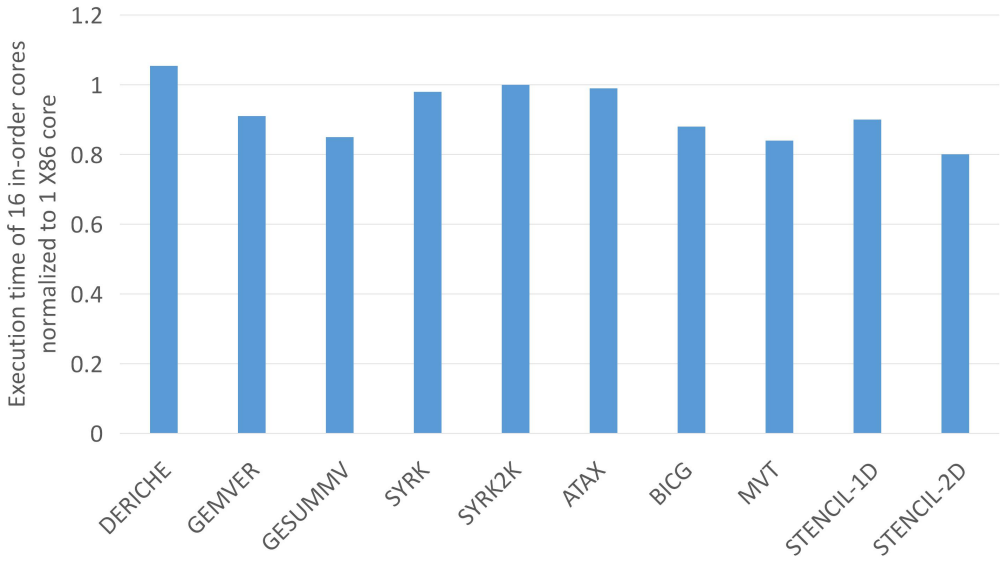
Source: provided by the author

Figure 6.4: Average latency observed for read requests in random access pattern



Source: provided by the author

Figure 6.5: Performance comparison between 16 in-order cores and single OoO core systems



Source: provided by the author

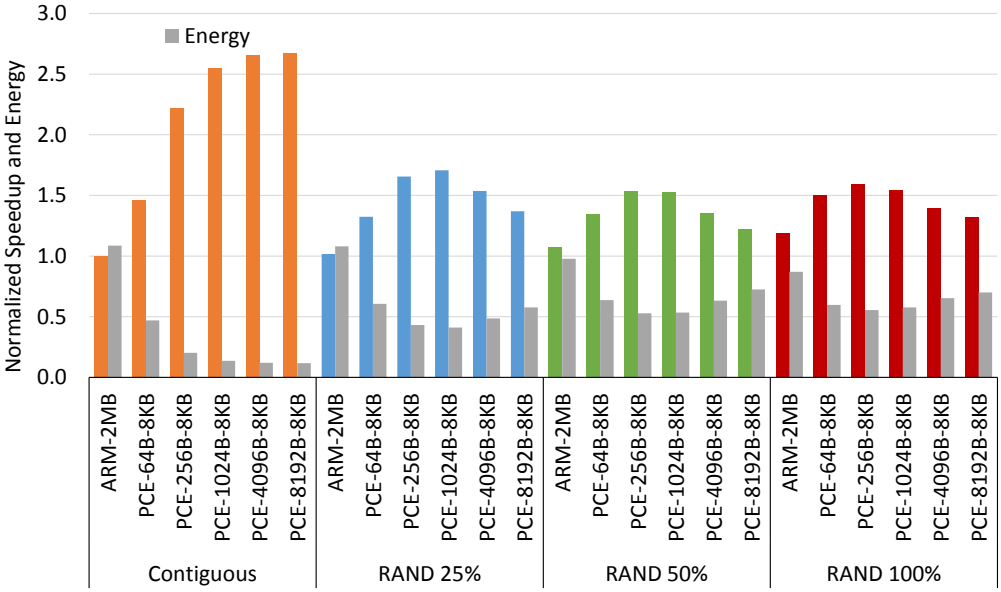
can represent up to 80% of the execution time of a single OoO core setup.

Though this test shows that gem5 can simulate multiple cores, mostly of the efforts have to be made on using software libraries and optimization to improve time performance. On the other hand, when compared to traditional architecture the application's energy consumption can be minimized since off-chip transfer are avoided and cache memories are generally reduced (also due to power and area constraints).

6.3 Fixed-function PIM for pointer-chasing operations

Figure 6.6 presents performance and energy results for traversing a linked list with 100 k nodes using both the ARM processor with an extra last-level cache memory and the PCEs limited to 8k bytes of memory. In the first set of bars (Contiguous), the nodes of a linked list lay contiguously on the memory, which means that processors can take advantage of prefetching techniques. Even so, the PCEs can accelerate the application using the same data chunk size of processors' cache line (64 bytes), since the micro-benchmark presents no data-reuse and the PCEs have no cache latencies. Also, by increasing the cache size of the baseline, no performance improvement is observed, as the spatial locality stands out as the main pattern of this workload. Moreover, as the PCEs are aggregated to enlarge the accessed data chunk, they are capable of increasing the performance by speculatively reading up to 8192 bytes of data from DRAM in parallel, taking advantage of spatial locality. Despite the sequential characteristics of the *FIND* instruction, the spec-

Figure 6.6: Speedup and energy for traversing a linked list with 100k nodes



Source: provided by the author

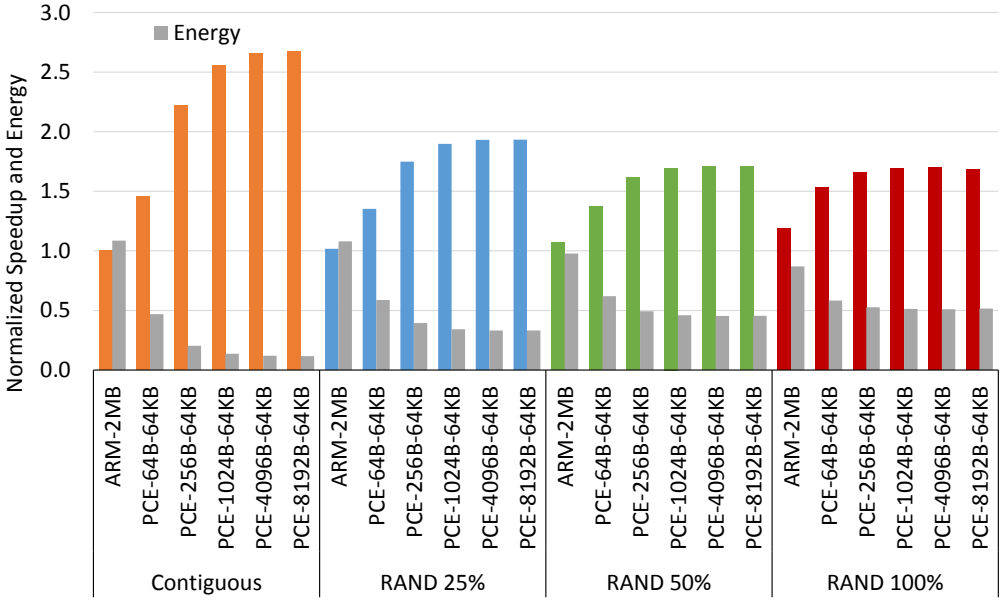
ulative loads mitigate the DRAM access latencies, thus increasing the overall performance by 2.7× when speculating over 8192 bytes of data.

On the other hand, when the entire data is placed randomly in memory (RAND 100% in Figure 6.6), the amount of data reuse increases, leveraging the importance of larger cache memories. However, the performance increment achieved by the ARM processor when doubling its cache size is limited to 18%. In contrast, the PCEs can speedup 1.6× when speculating over 256 bytes of data. Speculating over 8192 bytes of data, though, provides a speedup of 1.3×, showing the benefits that configurable speculation can bring to our evaluation.

Figure 6.7 depicts the results when the PCEs are allowed to use 64k bytes of vector registers. For continuous data (orange bars), increasing the number of registers available for PCE does not lead to performance improvement. However, when workload has random data access, the PCEs can take advantage of extra registers due to the now available temporal locality. Also, the energy savings for each access pattern and each PCE configuration (gray bars) are presented in Figure 6.6 and Figure 6.7. The PCEs can reduce energy consumption to 88% and 50% for contiguous and entirely random data access pattern, respectively.

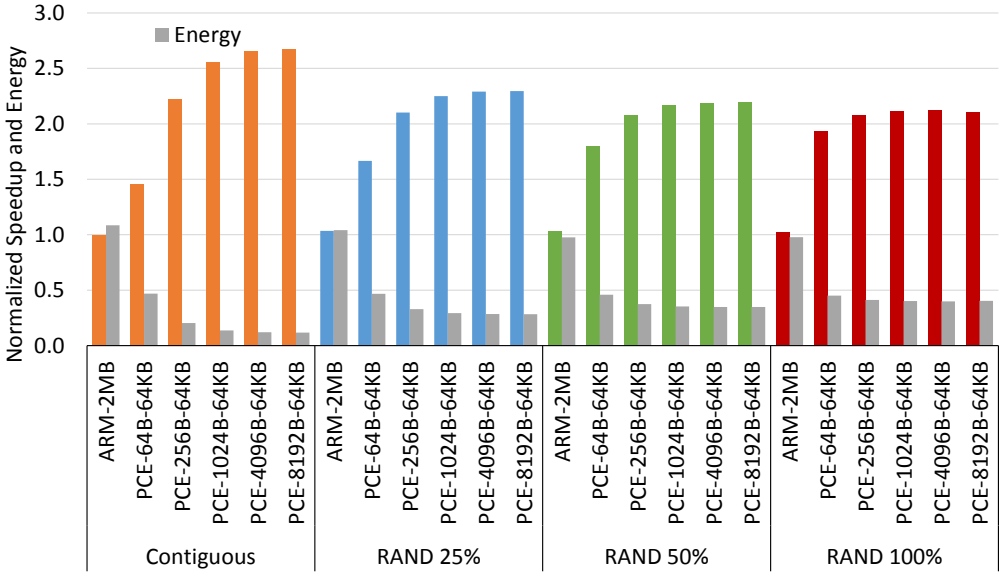
In Figure 6.8, we increased the size of the linked lists to 1M nodes, aiming to analyze the effect of more pressure on the memory hierarchy. As depicted by the red bars of Figure 6.8, the PCEs can accelerate the traversing of fully random data in up to 2.2× when configured with a speculative window of 4k bytes and 64k bytes of registers.

Figure 6.7: Performance and energy for traversing a linked list with 100k nodes - PCE using 64kB of registers



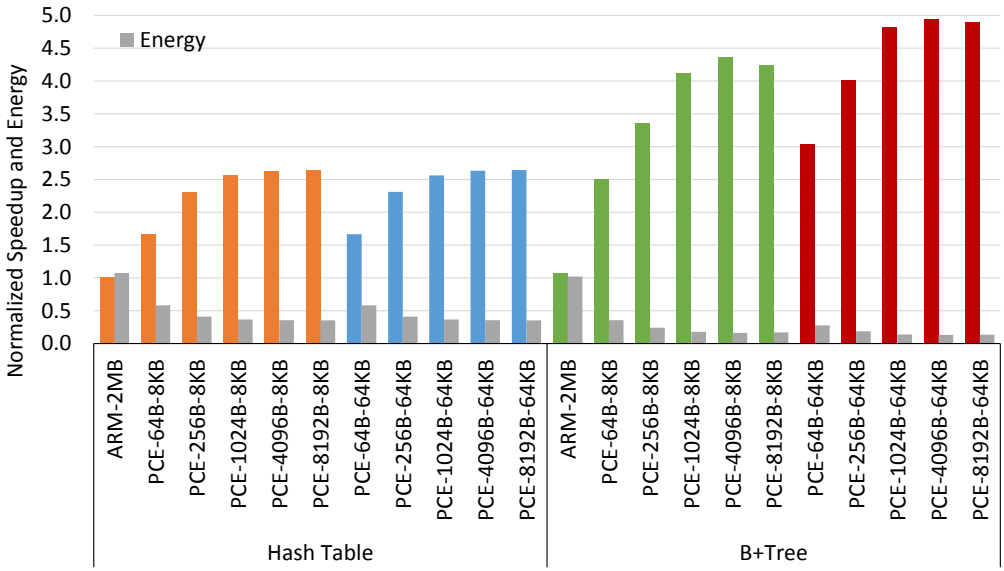
Source: provided by the author

Figure 6.8: Performance and energy consumption for traversing a linked list with 1M nodes



Source: Santos et al. (2018b)

Figure 6.9: Performance and energy consumption for traversing a hash table with 1.5M nodes and a b+tree with 3M nodes



Source: Santos et al. (2018b)

Thus, this case study reduces energy consumption by 60% of the energy consumed by the baseline.

Figure 6.9 presents the results for traversing a hash table and a b+tree with a workload of 1.5M nodes and 3M nodes, respectively. By using 64k bytes of registers and 8192 bytes of speculative request size (blue bars), the PCEs can speed up the operations on hash tables in 2.7× when compared to the baseline. Due to the reduced temporal locality of traversing operations on a hash table, the PCEs cannot take advantage of 8k bytes of total registers (orange bars), or even 64k bytes of registers, seeing that a slight difference is observed when varying the number of registers. Due to the acceleration and reduced hardware, the PCEs consumes 35% of the baseline's energy consumption to accelerate 2.7× the *Find* operation on a hash table structure.

The b+tree traversing presents a more intensive temporal locality, which can be observed on green and orange bars of Figure 6.9. The additional cache size available for the baseline can provide only 7% of performance improvement, showing that the reduced spatial locality dictates an important rule on the overall performance. On the other hand, PCE can take advantage of its configurable operands and SIMD units. Setting the operand size to 4096 bytes represents the best compromise between temporal and spatial locality for this kernel application. Also, as the temporal locality is considerable, the 64k bytes of registers are better exploited. Furthermore, the vector operations and simple TLB design facilitate the calculus of the next addresses (16 per node on b+tree),

which increases overall performance. Thus, this case study can achieve a speedup of 4.94× when compared to the baseline, while the energy reduction is near to 87% for traversing b+trees.

Therefore, by taking advantage of the parallelism present in HMC, the PCEs make use of speculation to perform pointer-chasing operations with reduced energy consumption and improved performance. In the majority of cases, a larger request size can benefit applications because the nodes of a data structure are contiguously allocated in the memory. This pointer-chasing accelerator is capable of accelerating linked list traversal and hash tables by a factor of 2.5, and b+tree by 4.9, while consuming 13% of the baseline's energy on average.

6.4 Reconfigurable Vector Unit

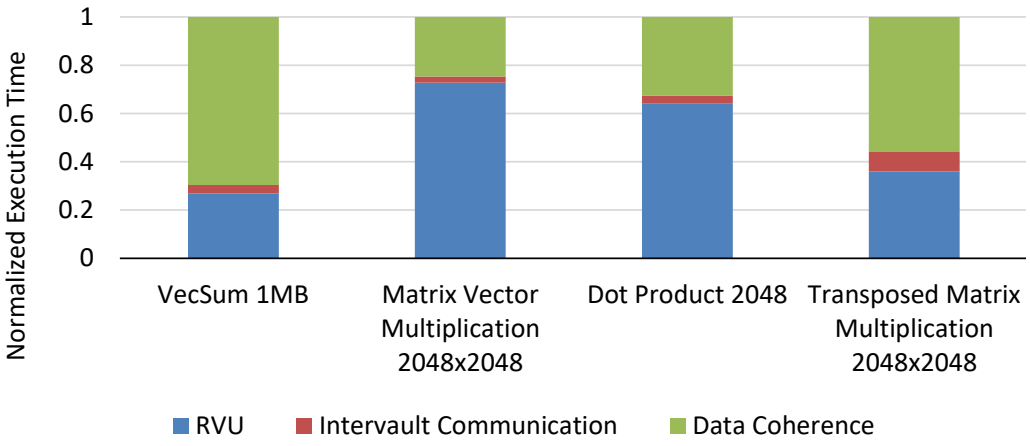
In this section we present the results used to evaluate the RVU architecture in two steps. First, we break down the execution time of some benchmarks into the overhead of cache coherence, intercommunication delay, and the time for processing in RVUs. These results are related to the first degree of reconfigurability, which is entirely given by the vector width of a PIM instruction. Then, we evaluate another level of reconfiguration that optimizes the number of active functional units in each RVU core based on the compute intensity of different kernel applications.

6.4.1 Evaluation of mechanisms to support RVU

Figure 6.10 presents the results for small kernels decomposed into three regions. The bottom blue region represents the time spent only computing the kernel within the in-memory logic. The red region highlighted in the middle depicts the cost of *inter-vault* communication, while the top green region represents the cost to keep cache coherence.

It is possible to notice in the *vecsum* kernel that more than 70% of the time is spent in cache coherence and internal communication, while only 30 % of the time is actually used for processing data. Although most of the *vecsum* kernel is executed in PIM, hence the data remains in the memory device during all execution time and no hits (writeback or clean eviction) should be seen in cache memories, there is a fixed cost for lookup operations to prevent data inconsistency. Regarding the matrix-multiplication and dot-product kernels in Figure 6.10, the impact of *flush* operations is diminished by the lower ratio of PIM memory access per PIM modification instructions.

Figure 6.10: Execution time of common kernels to illustrate the costs of cache coherence and *inter-vault* communication



Source: provided by the author

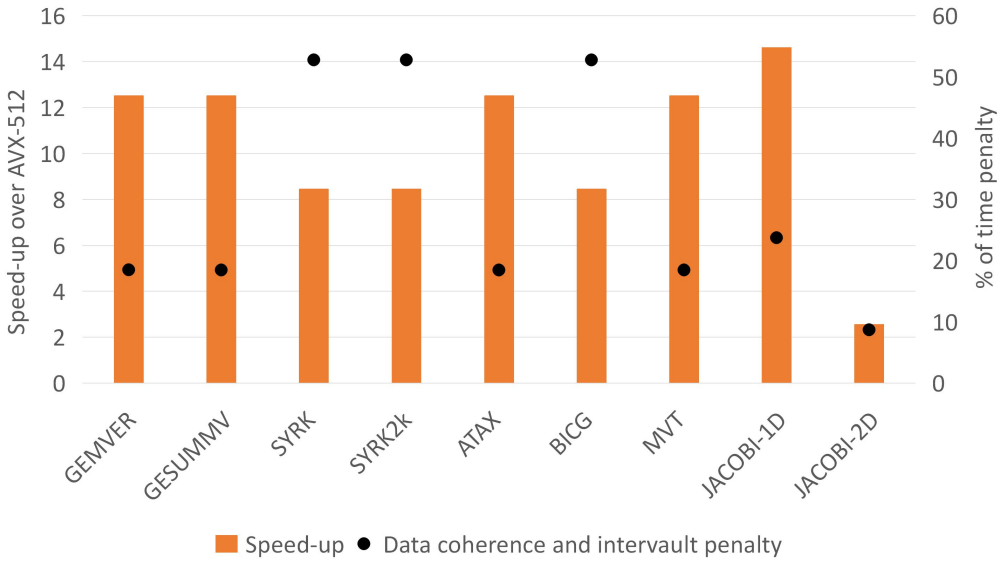
Since the *flush* operation generally triggers lookups to more than one cache block addressed by a PIM instruction, the overall latency will depend on each cache-level lookup latency. Also, for each *flush* request dispatched from the LSQ, all cache levels will receive the operation, but it is executed sequentially from the first-level to last-level cache. Only improvements in lookup time or reduced cache hierarchy would impact in the performance of *flush* operations. On the other hand, *inter-vault* communication penalty generally has little impact on the overall performance. For the transposed matrix-multiplication kernel, it is possible to see the effect of a great number of *register-read* and *mem-read* to different *vaults* inherent to the application loop.

Considering *flush* operations and *inter-vault* communication as costs that could be avoided, in Figure 6.11 we show the overall performance improvement of an ideal PIM and the time penalty using our proposal in some benchmarks of Polybench Suite. In general, the present mechanism can achieve speedup improvements between 2.5× and 14.6×, while the time penalty represents an average percentage of 29% over the ideal PIM. In general, our proposal provides a competitive advantage in terms of speedup in comparison to other HMC-instruction-based PIM setups. For instance, the proposal presented in (NAI et al., 2017) relies on *uncacheable* data region, hence no hardware cost is introduced. However, it comes with a cost in how much performance can be extracted when deciding if a code portion must be executed in the host core or in PIM units.

Also, the speculative approach proposed in (BOROUMAND et al., 2017) has only 5% of performance penalty compared to an ideal PIM, but the performance can profoundly degrade if rollbacks are frequently made, which will depend on the application behavior. Also, another similar work (AHN et al., 2015) advocates locality-aware PIM

execution to avoid *flush* operations and off-chip communication. However, they do not consider that large vectors in PIM can amortize the cost of cache coherence mechanism even if, eventually, the host CPU has to process scalar operands on the same data region.

Figure 6.11: Execution time of PolyBench applications normalized to AVX-512



Source: provided by the author

This section presented the simulation results of system-level mechanisms for PIM architectures described in Section 4.3, which aims at solving the instruction offloading, data coherence and management of the communication inside HMC to a class of PIM designs. The proposed acquire-release protocol offers programmability and data coherence resources to reduce programmers and compilers' effort in designing applications to be executed in PIM architectures. The experiments show that RVU can accelerate applications up to 14.6× compared to an AVX baseline, while the penalty due to cache coherence and communication represents an average percentage of 29% over an ideal PIM.

6.4.2 Design space exploration on RVU

In a second step, we used the simulator and the RVU model to propose a reconfiguration technique and improve energy efficiency according to the application's demands. The motivation and a description of modifications which have to be made in the RVU design presented by Santos et al. (2017), Lima et al. (2018) are presented in Lima et al. (2019). We varied the amount of active Functional Units, which is given by the #FUnn labels in the following charts. Figure 6.12 presents normalized memory bandwidth and processing power achieved by PIM logic to process kernels with different arithmetic in-

tensity. These kernel applications range from mostly memory-bounded, such as Scale, to mostly compute-bounded kernels, such as Polynomial Equation Solver. Figure 6.12(a) depicts a pure streaming behavior where the number of Functional Units does not impact on the total processing power, neither the average memory bandwidth. As this kernel application is not compute-intensive, the memory bandwidth stands out when the application makes use of largest load/store instructions available.

In contrast to the Stream Scale, the Polynomial Solver Equation shows an opposite behavior to streaming applications, as shown in Figure 6.12(c). The largest vector widths achieve both the highest values of memory bandwidth and processing power. In this case, not only memory bandwidth is required by the application, but also the processing power, which is achieved by the two reconfiguration setups (#FU32 and #FU16). It is possible to notice that the combination of memory- and compute-bound characteristics are found in the Bilinear Interpolation kernel. As shown in Figure 6.12(b), the discrepancy of bandwidth and FLOPS is only observed on the setups #FU1. One can notice that increasing the vector width also increase the memory bandwidth, thus allowing the use of few FUs to achieve the maximum FLOPS.

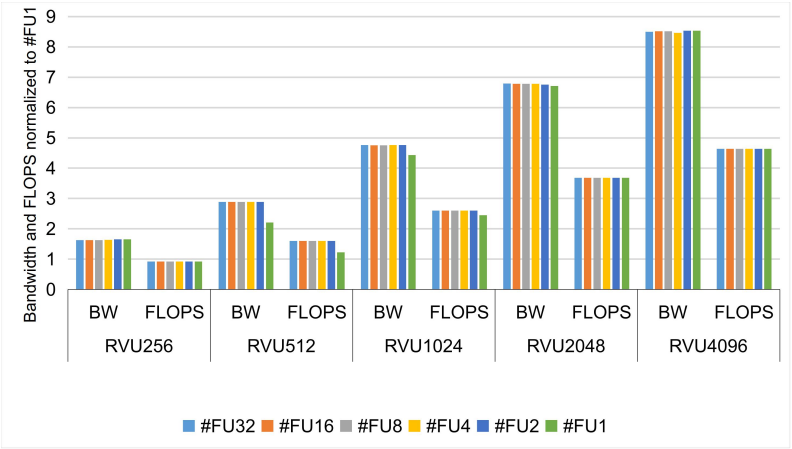
For the kernels presented in Figure 6.12, Figure 6.13 shows speedup and energy results. The streaming-like application in Figure 6.13(a) shows that bandwidth limits the speedup. The reconfiguration setup with fewer FUs is enough to consume data and achieve the same performance of the setups with more FUs. To achieve high performance, more VPUs are required to allow larger load operations. However, this implies that more hardware resources (register file, FSM, and FUs) will be kept in idle mode wasting static power, thus reducing the energy efficiency of those configurations. Similarly, Figure 6.13(c) can achieve the highest performance for different reconfiguration setups, except for the #FU1. In Figure 6.13(d), different points can achieve the low energy consumption of computation. However, as aforementioned, this application combines memory and compute-bound behavior, which means that the most efficient points will occur when a better compromise between memory bandwidth and processing power. Compute-intensive kernels are profoundly impacted by the number of FUs available in SIMD units, as presented in Figure 6.13(e). Although the highest performance is achieved by using the RVU4096 with setups #FU32 or #FU16, the most energy efficient configuration is achieved by using the setups #FU16 and #FU8.

Despite Figure 6.13 have presented different energy consumption and performance points separately, a better metric to show the efficiency of the reconfiguration is the

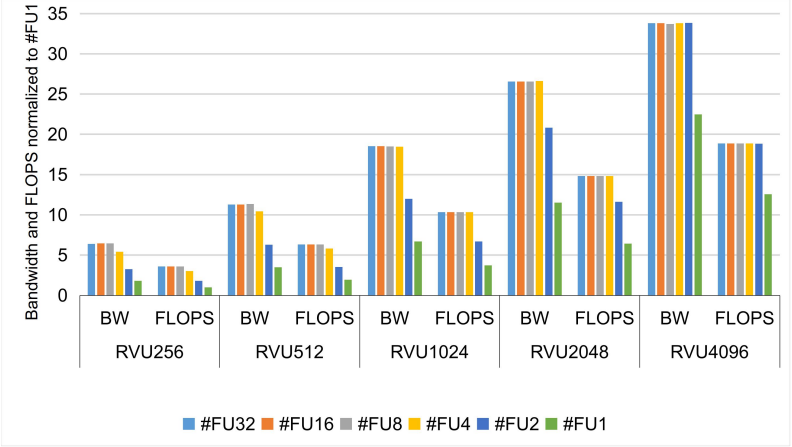
Energy Delay Product (EDP). Figure 6.14 presents the EDP results of several application benchmarks. These applications are classified according to their arithmetic inten-

Figure 6.12: Total memory bandwidth and processing power for applications with different processing requirements. (a) Stream Scale, (b) Bilinear Interpolation and (c) Polynomial Solver

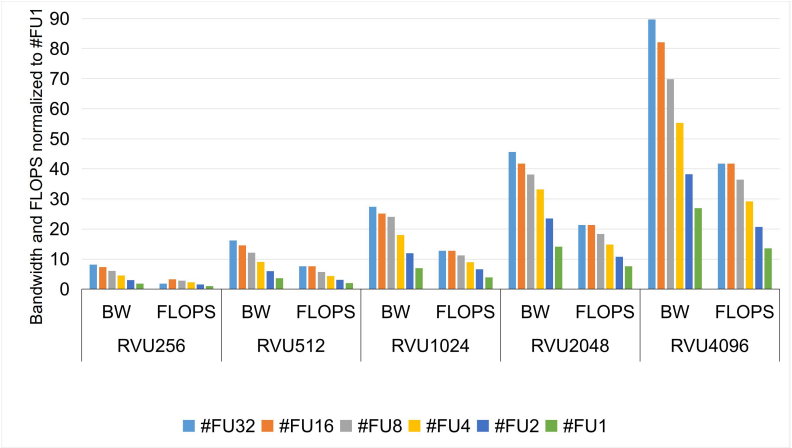
(a)



(b)



(c)



Source: provided by the author

sity into: mostly memory-bound applications, such as Scale, Copy, Fill, Add, Sum and Daxpy, applications with intermediate arithmetic intensity, such as Interpolation and Dot-product, and compute-bounded kernels, such as Matrix-vector Multiplication and Polynomial Equation Solver. All columns were obtained by running the largest vector width (RVU4096) and varying the reconfiguration setups. One can notice from Figure 6.14 that memory-bound applications must use fewer FUs to achieve significant energy efficiency, since the lowest EDP results are obtained with the setup #FU1. On the other hand, compute-bound applications require higher FLOPS, and the setup with lower EDP values are obtained by using a larger amount of FUs.

Therefore, by identifying and taking advantage of the deviations in the compute-intensity, we can reconfigure RVU and lead to more energy savings. Our simulation results show that, for a set of memory-bounded applications, the number of FUs on does not interfere in the system performance so that energy savings can be achieved. On the other hand, compute-bounded applications have their memory bandwidth as FLOPS dictated by the biggest number of active FUs.

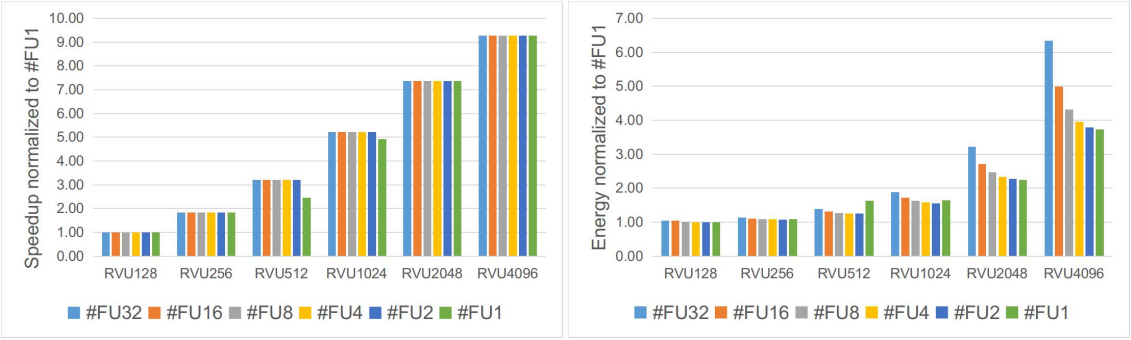
6.5 Applicability to emergent memory technologies

To simulate the performance of a system with main memory different from DRAM, we chose the Resistive RAM (ReRAM) based on the 1-transistor-1-resistor (1T1R) array architecture as it is organized similarly to the conventional 1-transistor-1-capacitor (1T1C) array present in DRAMs. Thus, a ReRAM can be simulated by changing the DRAM timings to ReRAM ones (MAO et al., 2016). Figure 6.15 presents the performance of two scenarios to run an image recognition algorithm, namely Yolo9000, and both setups are based on an HMC using DRAM and ReRAM timings.

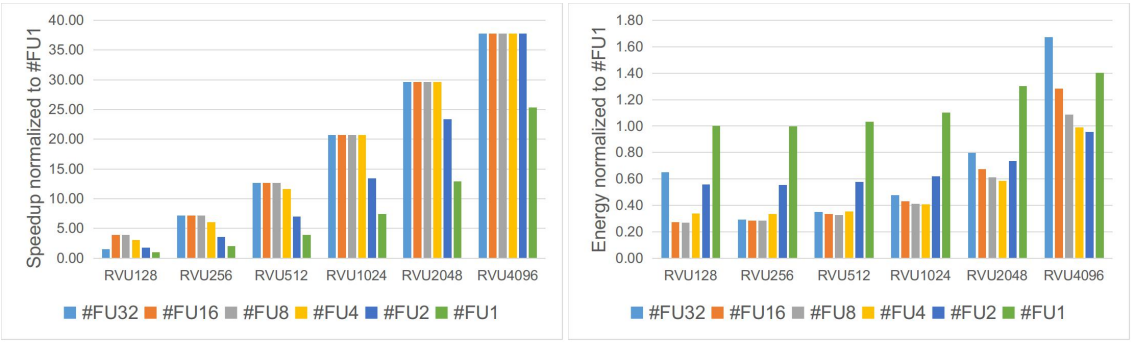
The first architecture presented in Figure 6.15 is composed of 4 SSE-enabled cores connected to HMC's SerDes as presented in Section **??**? **TODO: referencia da secao pim multicore**, which can achieve up to 7 FPS with DRAM. The second scenario presents the results for RVU (setup presented in Section **??**) reaching up to 64 frames per second (FPS). The bars of Figure 6.15 illustrate the main difference between the two designs, where the traditional cores spends more time in generic matrix-multiplication relative to memory access. Due to the large vector units, the PIM approach spends less time processing and more time in memory access, which indicates that convolutional neural networks may not take advantage of full processing power provided by large SIMD instructions of

Figure 6.13: Speedup and energy consumption in three applications. (a) and (b) Stream Scale, (c) and (d) Bilinear Interpolation, and (e) and (f) Polynomial Solver Equation

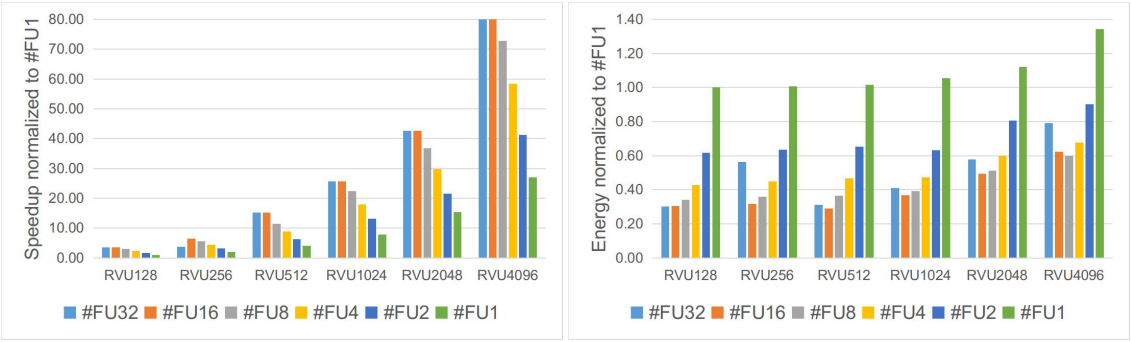
(a) (b)



(c) (d)



(e) (f)

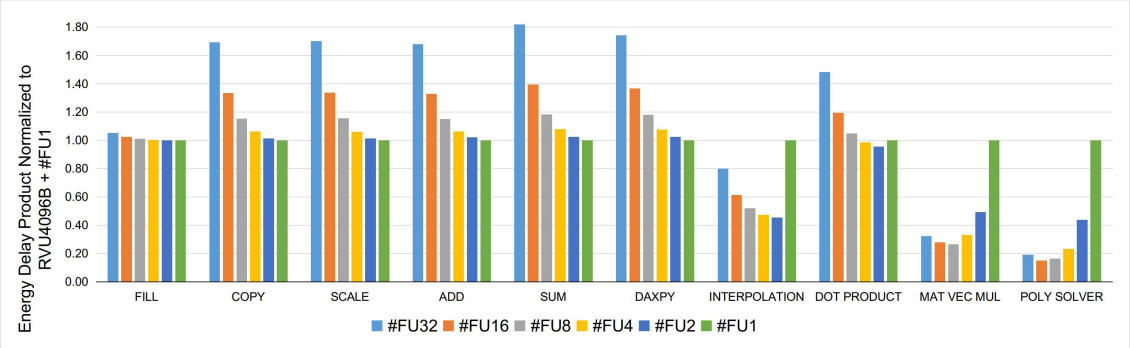


Source: provided by the author

RVU cores.

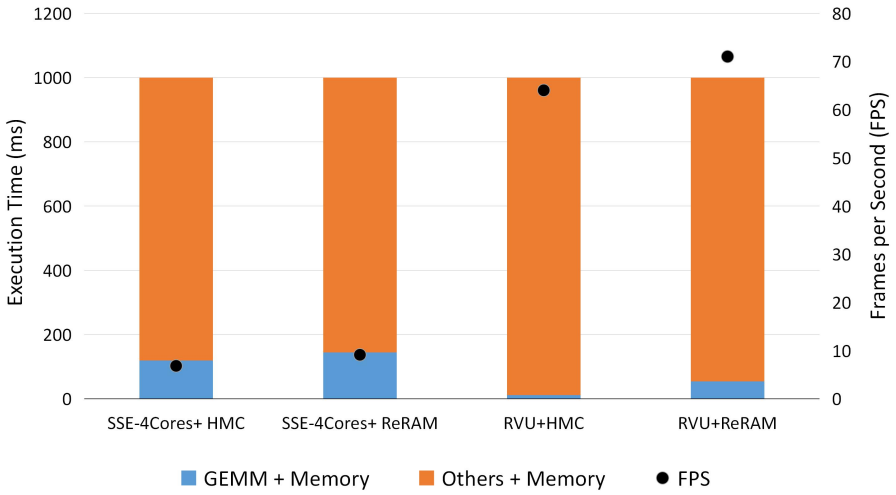
By changing the memory parameters, the improvement on the memory access time provided by ReRAM has improved the overall performance in both cases, as the memory bound portions of Yolo application have significant impact on FPS. Thus, 4 X86 cores and RVU have achieved 10 and 71 FPS, respectively, which represents an improvement of 36% and 10% over DRAM.

Figure 6.14: Energy Delay Product (EDP) results for several application kernels



Source: provided by the author

Figure 6.15: Effect of memory technology on application's performance



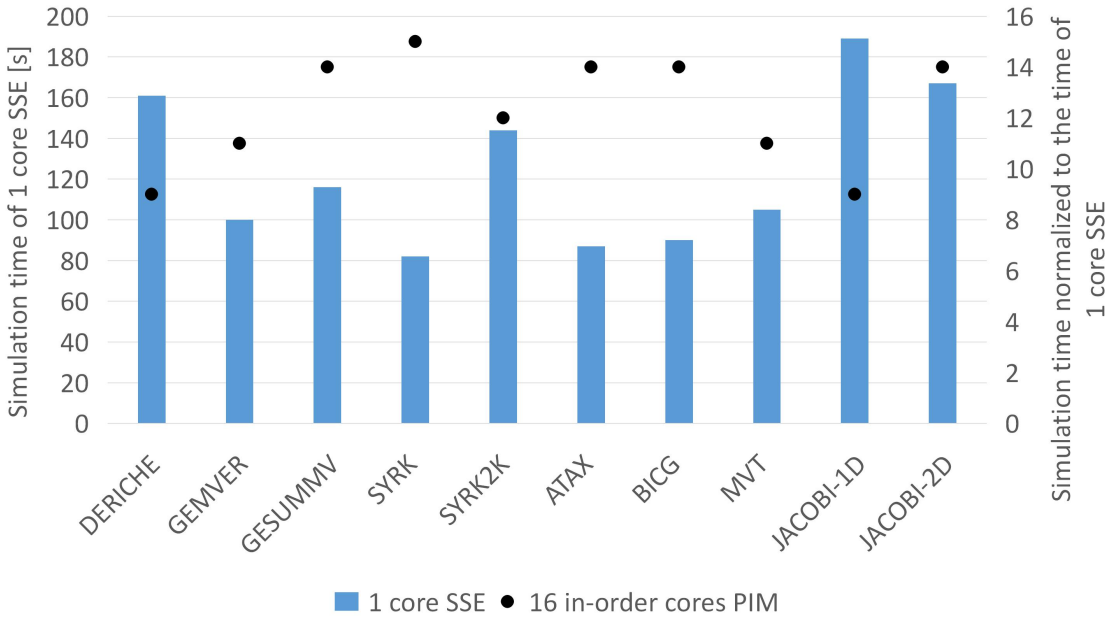
Source: provided by the author

6.6 Simulation time and development efforts

In order to illustrate the performance of the proposed GEM5 support, Figure 6.16 shows the simulation time comparison between two scenario of traditional fully programmable cores. The first architecture is represented by 1 X86 SSE-enabled core connected to the HMC's SerDes and the second one consists of 16 X86 in-order cores connected to the crossbar switches of the HMC, as presented in Section **??? TODO: qual secao**.

Figure 6.16 demonstrates that the simulation time is approximately proportional to the number of simulated cores. The slowdown of simulating 16 cores in comparison to a single core ranges from 9× to 16×. This slowdown is due to the sequential engine of the gem5 simulator, which runs all models in a single core of the host machine. This issue is an open problem in gem5 platform. Although there is a simple approach to parallelize

Figure 6.16: Simulation time of PIM multicore



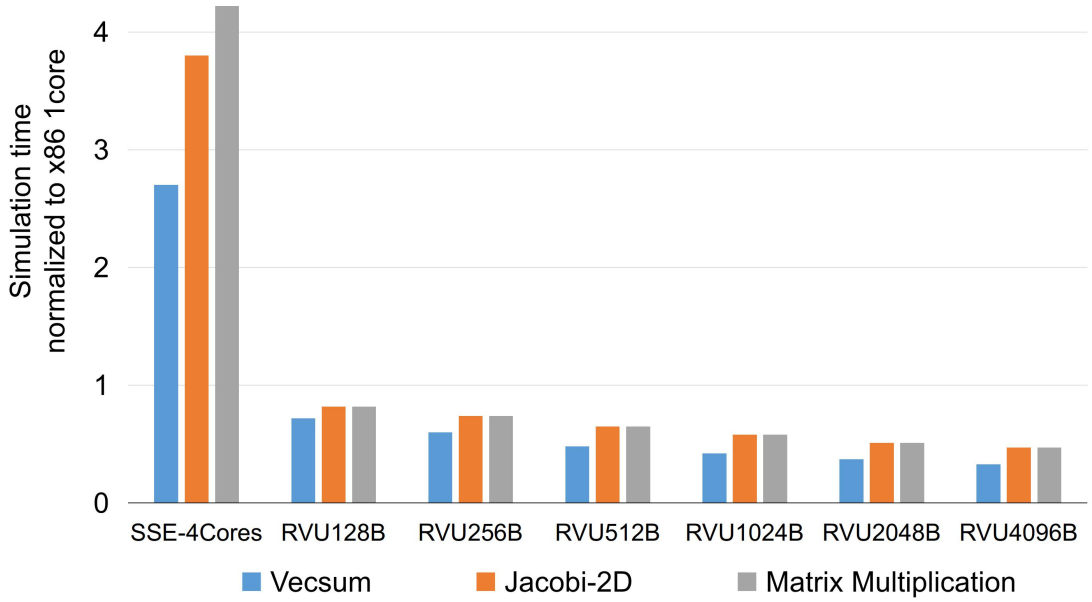
Source: provided by the author

the simulation engine, it is limited to cores without synchronization and requires more development efforts in the overall gem5 engine to be beneficial in terms of simulation time.

Figure 6.17 shows the simulation time comparison between traditional fully programmable cores and fixed-function PIM approach. As fixed-function PIM generally relies on simplified models, their simulation time is drastically reduced. As the simulated PIM reduces the number of instructions executed due to the improvement on vector capabilities, the simulation time is drastically reduced. Also, it is important to notice that the RVU PIM is able to operate through different operand sizes per instructions, thus operating from 128 Bytes to 4096 Bytes of data at once, which further reduces the simulation time.

Table 6.1 presents an estimate of the development effort put into the gem5 support for PIM presented in this thesis. One can notice that the a number of lines of code is needed to insert a new ISA in the simulator. Though, the efforts to implement and test a PIM logic cannot be underestimated, since synchronization of independent PIM logic is crucial to prevent deadlocks on the memory system.

Figure 6.17: Simulation time of RVU cores



Source: provided by the author

Table 6.1: Estimate of development efforts

Number of lines to write a PIM logic	2,500
Number of lines to extend x86 ISA with AVX	10,000
Number of lines to extend x86 ISA with PIM ISA	2,000
Number of lines added to HMC model	700
Number of lines to provide system-level features	1,500

Source: provided by the author

7 CONCLUSIONS AND FUTURE WORK

In this thesis, we presented PIM-gem5, a set of models, mechanisms, and methods to support PIM design exploration in the gem5 simulator. The proposed simulator provides system-level information, such as statistics regarding execution time, hardware utilization, offloading and data coherence overhead for different types of PIM logic placed in 3D-stacked memories like HMC. We demonstrated how PIM-gem5 can provide support not only for PIM multi-core, but also for fixed-function PIM, which demands different programming and execution models. The simulation time of the case studies had shown that time is not a drawback in any case, which is not much different from the time to simulate a single processor in the original gem5.

We have demonstrated that the proposed HMC model achieves an aggregate bandwidth close to the one specified by HMC consortium. Also, PIM-gem5 provides system-level solutions which apply to a wide variety of PIM architectures, such as data coherence protocols, instruction offloading mechanism, and virtual-to-physical address translation mechanisms inspired in past studies and enhanced in this work. Finally, we demonstrated the potential of the simulator to perform design space exploration in two case studies of PIM architectures, namely Pointer-Chasing Engine, and Reconfigurable Vector Unit, which made use of some generic system-level solutions.

By choosing a well-accepted, flexible simulator like gem5 as the basis of this work, we can take advantage of many features related to mature design patterns for modeling, compatibility with several operating system features and existing models of CPU and ISA, without reinventing the wheel or limiting future expansion of this simulation tool. This description of modifications made to gem5 aims at easing the learning process to include new instructions and processing logic models, thus allowing computer architects to focus on the proposal of new solutions rather than developing a simulator from scratch or based on a tool with a steep learning curve. Though, it requires the architects to get acquainted with gem5's simulation modes and basic objects, such as slave/master ports and event scheduling.

7.1 Future Work

We plan to include new features to PIM-gem5 and continue investigating generic solutions for PIM architectures, mainly for the class of fixed-function and FU-centered

PIM designs. Some topics for future studies and developments are:

- Enable the simulation of new memory technologies and in-situ computing devices, such as memristive devices and in-DRAM circuits.
- Investigate the benefits of multi-core host processors sharing and offloading code to PIM devices.
- Investigate the benefits of static and runtime schedulers for PIM execution to provide seamless data mapping.
- Parallelize the simulation engine to run simulations on multi-core hosts.

7.2 Published Papers

The PIM support for gem5 was used as main tool either for validating an architecture or a compiler in the works listed below.

1. SANTOS, Paulo C. et al. **Processing in 3D memories to speed up operations on complex data structures**. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018. IEEE, 2018. p. 897-900.
2. DE LIMA, João Paulo C. et al. **Design space exploration for PIM architectures in 3D-stacked memories**. In: Proceedings of the 15th ACM International Conference on Computing Frontiers. ACM, 2018. p. 113-120.
3. SANTOS, Paulo Cesar et al. **Exploring IoT platform with technologically agnostic processing-in-memory framework**. In: Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications. ACM, 2018. p. 1-6.
4. AHMED, Hameeza et al. **A compiler for Automatic Selection of Suitable Processing-in-Memory Instructions**. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019. IEEE, 2018. p. 897-900.
5. DE LIMA, João Paulo C. et al. **Exploiting reconfigurable vector processing for energy-efficient computation in 3D-stacked memories**. In: International Symposium on Applied Reconfigurable Computing. Springer, Cham, 2019. p. 28-35.

REFERENCES

AHMED, H. et al. A compiler for automatic selection of suitable processing-in-memory instructions. In: **Design, Automation and Test in Europe Conf. and Exhibition (DATE), 2019**. [S.l.: s.n.], 2019.

AHN, J. et al. A scalable processing-in-memory accelerator for parallel graph processing. **ACM SIGARCH Computer Architecture News**, ACM, v. 43, n. 3, p. 105–117, 2016.

AHN, J.; YOO, S.; CHOI, K. Low-power hybrid memory cubes with link power management and two-level prefetching. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 24, n. 2, p. 453–464, 2016.

AHN, J. et al. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In: **Int. Symp. on Computer Architecture (ISCA)**. [S.l.: s.n.], 2015.

ALVES, M. A. et al. Large vector extensions inside the hmc. In: **Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.: s.n.], 2016.

ALVES, M. A. Z. et al. Sinuca: A validated micro-architecture simulator. In: **HPCC/CSS/ICESS**. [S.l.: s.n.], 2015. p. 605–610.

ANGIZI, S.; HE, Z.; FAN, D. Pima-logic: a novel processing-in-memory architecture for highly flexible and energy-efficient logic computation. In: IEEE. **2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)**. [S.l.], 2018. p. 1–6.

ASGHARI-MOGHADDAM, H. et al. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In: IEEE PRESS. **The 49th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2016. p. 50.

AZARKHISH, E. et al. High performance axi-4.0 based interconnect for extensible smart memory cubes. In: EDA CONSORTIUM. **Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition**. [S.l.], 2015. p. 1317–1322.

AZARKHISH, E. et al. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In: SPRINGER. **International Conference on Architecture of Computing Systems**. [S.l.], 2016. p. 19–31.

BASU, A. et al. Efficient virtual memory for big memory servers. In: ACM. **ACM SIGARCH Computer Architecture News**. [S.l.], 2013. v. 41, n. 3, p. 237–248.

BINKERT, N. et al. The gem5 simulator. **ACM SIGARCH Computer Architecture News**, ACM, v. 39, n. 2, p. 1–7, 2011.

BOROUMAND, A. et al. Lazypim: An efficient cache coherence mechanism for processing-in-memory. **IEEE Computer Architecture Letters**, IEEE, v. 16, n. 1, p. 46–50, 2017.

CHEN, K. et al. Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory. In: EDA CONSORTIUM. **Proceedings of the Conference on Design, Automation and Test in Europe**. [S.l.], 2012. p. 33–38.

CHI, P. et al. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In: IEEE PRESS. **ACM SIGARCH Computer Architecture News**. [S.l.], 2016. v. 44, n. 3, p. 27–39.

CONSORTIUM, H. M. C. et al. **HMC specification 2.0**. 2015.

DANOWITZ, A. et al. Cpu db: recording microprocessor history. **Queue**, ACM, v. 10, n. 4, p. 10, 2012.

DAVIS, W. R. et al. Demystifying 3d ics: The pros and cons of going vertical. **IEEE Design & Test of Computers**, IEEE, v. 22, n. 6, p. 498–510, 2005.

DENNARD, R. H. et al. Design of ion-implanted mosfet's with very small physical dimensions. **IEEE Journal of Solid-State Circuits**, v. 9, n. 5, 1974.

DRUMOND, M. et al. The mondrian data engine. In: IEEE. **Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on**. [S.l.], 2017. p. 639–651.

ECKERT, Y.; JAYASENA, N.; LOH, G. H. Thermal feasibility of die-stacked processing in memory. In: **2nd Workshop on Near-Data Processing (WoNDP)**. [S.l.: s.n.], 2014.

ESMAEILZADEH, H. et al. Power challenges may end the multicore era. **Communications of the ACM**, ACM, v. 56, n. 2, p. 93–102, 2013.

FANG, Z. et al. Active memory controller. **The Journal of Supercomputing**, v. 62, 2012.

FARMAHINI-FARAHANI, A. et al. Drama: An architecture for accelerated processing near memory. **IEEE Computer Architecture Letters**, IEEE, v. 14, n. 1, p. 26–29, 2015.

FARMAHINI-FARAHANI, A. et al. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In: IEEE. **High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on**. [S.l.], 2015. p. 283–295.

FITZPATRICK, B. Distributed caching with memcached. **Linux journal**, Belltown Media, v. 2004, n. 124, p. 5, 2004.

GAO, D.; SHEN, T.; ZHUO, C. A design framework for processing-in-memory accelerator. In: ACM. **Proceedings of the 20th System Level Interconnect Prediction Workshop**. [S.l.], 2018.

GAO, M.; AYERS, G.; KOZYRAKIS, C. Practical near-data processing for in-memory analytics frameworks. In: IEEE. **Parallel Architecture and Compilation (PACT), 2015 International Conference on**. [S.l.], 2015. p. 113–124.

GAO, M.; KOZYRAKIS, C. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In: IEEE. **High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on**. [S.l.], 2016. p. 126–137.

GAO, M. et al. Tetris: Scalable and efficient neural network acceleration with 3d memory. In: **Int. Conf. on Architectural Support for Programming Languages and Operating Systems**. [S.l.: s.n.], 2017.

GHOSE, S. et al. Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions. **arXiv preprint arXiv:1802.00320**, 2018.

GHOSE, S. et al. What your dram power models are not telling you: Lessons from a detailed experimental study. **arXiv preprint arXiv:1807.05102**, 2018.

GOKHALE, M.; LLOYD, S.; HAJAS, C. Near memory data structure rearrangement. In: ACM. **Proceedings of the 2015 International Symposium on Memory Systems**. [S.l.], 2015. p. 283–290.

HADIDI, R. et al. Demystifying the characteristics of 3d-stacked memories: A case study for hybrid memory cube. In: IEEE. **Workload Characterization (IISWC), 2017 IEEE International Symposium on**. [S.l.], 2017. p. 66–75.

HADIDI, R. et al. Performance implications of nocs on 3d-stacked memories: Insights from the hybrid memory cube. In: IEEE. **Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on**. [S.l.], 2018. p. 99–108.

HANSSON, A. et al. Simulating dram controllers for future system architecture exploration. In: IEEE. **Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on**. [S.l.], 2014. p. 201–210.

HASHEMI, M. et al. Accelerating dependent cache misses with an enhanced memory controller. In: IEEE. **Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on**. [S.l.], 2016. p. 444–455.

HASSAN, S. M. et al. Reliability-performance tradeoffs between 2.5 d and 3d-stacked dram processors. In: IEEE. **Reliability Physics Symposium (IRPS), 2016 IEEE International**. [S.l.], 2016. p. MY–2.

HONG, B. et al. Accelerating Linked-list Traversal Through Near-Data Processing. In: **Int. Conf. on Parallel Architectures and Compilation - PACT**. [S.l.: s.n.], 2016.

HSIEH, K. et al. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. **ACM SIGARCH Computer Architecture News**, v. 44, n. 3, p. 204–216, 2016.

HSIEH, K. et al. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In: IEEE. **2016 IEEE 34th International Conference on Computer Design (ICCD)**. [S.l.], 2016. p. 25–32.

HU, X.; STOW, D.; XIE, Y. Die stacking is happening. **IEEE Micro**, IEEE, v. 38, n. 1, p. 22–28, 2018.

Hybrid Memory Cube Consortium. **Hybrid Memory Cube Specification Rev. 2.0**. 2013. Http://www.hybridmemorycube.org/.

Hybrid Memory Cube Consortium. **Hybrid Memory Cube Specification Rev. 2.1**. 2013. Http://www.hybridmemorycube.org/.

JACOB, B.; NG, S.; WANG, D. **Memory systems: cache, DRAM, disk**. [S.l.]: Morgan Kaufmann, 2010.

JEDDELOH, J.; KEETH, B. Hybrid memory cube new dram architecture increases density and performance. In: IEEE. **VLSI Technology (VLSIT), 2012 Symposium on**. [S.l.], 2012. p. 87–88.

JEON, D.-I.; CHUNG, K.-S. Cashmc: A cycle-accurate simulator for hybrid memory cube. **IEEE Computer Architecture Letters**, IEEE, n. 1, p. 10–13, 2017.

JUNG, M.; WEIS, C.; WEHN, N. Dramsys: A flexible dram subsystem design space exploration framework. **IPSJ Transactions on System LSI Design Methodology**, Information Processing Society of Japan, v. 8, p. 63–74, 2015.

KAGI, A.; GOODMAN, J. R.; BURGER, D. Memory bandwidth limitations of future microprocessors. In: IEEE. **Computer Architecture, 1996 23rd Annual International Symposium on**. [S.l.], 1996. p. 78–78.

KANG, W. et al. In-memory processing paradigm for bitwise logic operations in stt–mram. **IEEE Transactions on Magnetics**, IEEE, v. 53, n. 11, p. 1–4, 2017.

KERSEY, C. D.; KIM, H.; YALAMANCHILI, S. Lightweight simt core designs for intelligent 3d stacked dram. In: ACM. **Proceedings of the International Symposium on Memory Systems**. [S.l.], 2017. p. 49–59.

KIM, G. et al. Memory-centric system interconnect design with hybrid memory cubes. In: IEEE PRESS. **Proceedings of the 22nd international conference on Parallel architectures and compilation techniques**. [S.l.], 2013. p. 145–156.

KIM, J. The future of graphic and mobile memory for new applications. In: IEEE. **Hot Chips 28 Symposium (HCS), 2016 IEEE**. [S.l.], 2016. p. 1–25.

KIM, J.-S. et al. A 1.2 v 12.8 gb/s 2 gb mobile wide-i/o dram with 4x128 i/os using tsv based stacking. **IEEE Journal of Solid-State Circuits**, IEEE, v. 47, n. 1, p. 107–116, 2012.

KIM, Y.; YANG, W.; MUTLU, O. Ramulator: A fast and extensible dram simulator. **Computer Architecture Letters**, v. 15, n. 1, p. 45–49, 2016.

KISH, L. B. End of moore's law: thermal (noise) death of integration in micro and nano electronics. **Physics Letters A**, Elsevier, v. 305, n. 3-4, p. 144–149, 2002.

KLEINER, M. B.; KUHN, S. A.; WEBER, W. Performance improvement of the memory hierarchy of risc-systems by application of 3-d-technology. In: IEEE. **Electronic Components and Technology Conference, 1995. Proceedings., 45th**. [S.l.], 1995. p. 645–655.

LEE, D. U. et al. 25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv. In: **Int. Solid-State Circuits Conference Digest of Technical Papers (ISSCC)**. [S.l.: s.n.], 2014.

LEIDEL, J. D.; CHEN, Y. Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations. In: IEEE. **Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International**. [S.l.], 2016. p. 621–630.

LI, S. et al. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In: **MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.: s.n.], 2009. p. 469–480.

LI, S. et al. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In: ACM. **Proceedings of the 53rd Annual Design Automation Conference**. [S.l.], 2016. p. 173.

LIMA, J. P. et al. Exploiting reconfigurable vector processing for energy-efficient computation in 3d-stacked memories. In: SPRINGER. **International Symposium on Applied Reconfigurable Computing**. [S.l.], 2019. p. 28–35.

LIMA, J. P. C. de et al. Design space exploration for pim architectures in 3d-stacked memories. In: ACM. **Proceedings of the 15th ACM International Conference on Computing Frontiers**. [S.l.], 2018. p. 113–120.

LIU, C. C. et al. Bridging the processor-memory performance gapwith 3d ic technology. **IEEE Design & Test of Computers**, IEEE, n. 6, p. 556–564, 2005.

LLOYD, S.; GOKHALE, M. Design space exploration of near memory accelerators. In: ACM. **Proceedings of the International Symposium on Memory Systems**. [S.l.], 2018. p. 218–220.

LOH, G. H. et al. A processing in memory taxonomy and a case for studying fixed-function pim. In: **Workshop on Near-Data Processing (WoNDP)**. [S.l.: s.n.], 2013.

LOWE-POWER, J. **gem5 Tutorial**. 2017. Available from Internet: <http://learning.gem5.org/book/index.html>.

LOWE-POWER, J. **gem5 Documentation**. 2019. Available from Internet: <http://gem5.org/Documentation>.

MAO, M. et al. Optimizing latency, energy, and reliability of 1t1r reram through cross-layer techniques. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, IEEE, v. 6, n. 3, p. 352–363, 2016.

NAI, L. et al. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In: **Int. Symp. on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2017.

NAI, L.; KIM, H. Instruction offloading with hmc 2.0 standard: A case study for graph traversals. In: ACM. **Proceedings of the 2015 International Symposium on Memory Systems**. [S.l.], 2015. p. 258–261.

NAIR, R. et al. Active memory cube: A processing-in-memory architecture for exascale systems. **IBM Journal of Research and Development**, IBM, v. 59, n. 2/3, p. 17–1, 2015.

NARANCIC, G. **A Preliminary Exploration of Memory Controller Policies on Smartphone Workloads**. Thesis (PhD), 2012.

OLIVEIRA, G. F. et al. A generic processing in memory cycle accurate simulator under hybrid memory cube architecture. In: IEEE. **Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2017 International Conference on**. [S.l.], 2017. p. 54–61.

OLIVEIRA, G. F. et al. Nim: An hmc-based machine for neuron computation. In: SPRINGER. **International Symposium on Applied Reconfigurable Computing**. [S.l.], 2017. p. 28–35.

PATTNAIK, A. et al. Scheduling techniques for gpu architectures with processing-in-memory capabilities. In: ACM. **Proceedings of the 2016 International Conference on Parallel Architectures and Compilation**. [S.l.], 2016. p. 31–44.

PAWLOWSKI, J. T. Hybrid memory cube (hmc). In: IEEE. **2011 IEEE Hot Chips 23 Symposium (HCS)**. [S.l.], 2011. p. 1–24.

POLLACK, F. J. New microarchitecture challenges in the coming generations of cmos process technologies (keynote address). In: IEEE COMPUTER SOCIETY. **Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture**. [S.l.], 1999. p. 2.

POUCHET, L.-N. Polybench: The polyhedral benchmark suite. **URL: http://www. cs. ucla. edu/pouchet/software/polybench**, 2012.

PUGSLEY, S. H. et al. Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads. In: IEEE. **2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2014. p. 190–200.

RIXNER, S. et al. Memory access scheduling. In: ACM. **ACM SIGARCH Computer Architecture News**. [S.l.], 2000. v. 28, n. 2, p. 128–138.

ROSENFELD, P. **Performance exploration of the hybrid memory cube**. Thesis (PhD), 2014.

ROSENFELD, P.; COOPER-BALIS, E.; JACOB, B. Dramsim2: A cycle accurate memory system simulator. **IEEE Computer Architecture Letters**, IEEE, v. 10, n. 1, p. 16–19, 2011.

SAKUMA, K. et al. 3d chip-stacking technology with through-silicon vias and low-volume lead-free interconnections. **IBM Journal of Research and Development**, IBM, v. 52, n. 6, p. 611–622, 2008.

SANCHEZ, D.; KOZYRAKIS, C. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In: ACM. **ACM SIGARCH Computer architecture news**. [S.l.], 2013. v. 41, n. 3, p. 475–486.

SANTOS, P. C. et al. Exploring iot platform with technologically agnostic processing-in-memory framework. In: IEEE. **Proceedings of the INTelligent Embedded Systems Architectures and Applications Workshop**. [S.l.], 2018.

SANTOS, P. C. et al. Processing in 3d memories to speed up operations on complex data structures. In: IEEE. **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018**. [S.l.], 2018. p. 897–900.

SANTOS, P. C. et al. Operand size reconfiguration for big data processing in memory. In: EUROPEAN DESIGN AND AUTOMATION ASSOCIATION. **Proceedings of the Conference on Design, Automation & Test in Europe**. [S.l.], 2017. p. 710–715.

SCHMIDT, J.; FRÖNING, H.; BRÜNING, U. Exploring time and energy for complex accesses to a hybrid memory cube. In: ACM. **Proceedings of the Second International Symposium on Memory Systems**. [S.l.], 2016. p. 142–150.

SCRBAK, M. et al. Exploring the processing-in-memory design space. **Journal of Systems Architecture**, Elsevier, v. 75, p. 59–67, 2017.

SESHADRI, V. et al. Rowclone: fast and energy-efficient in-dram bulk data copy and initialization. In: ACM. **Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2013. p. 185–197.

SESHADRI, V. et al. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In: ACM. **Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2017. p. 273–287.

SESHADRI, V. et al. Gather-scatter dram: in-dram address translation to improve the spatial locality of non-unit strided accesses. In: ACM. **Proceedings of the 48th International Symposium on Microarchitecture**. [S.l.], 2015. p. 267–280.

SHAFIEE, A. et al. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. **ACM SIGARCH Computer Architecture News**, ACM, v. 44, n. 3, p. 14–26, 2016.

SIEGL, P.; BUCHTY, R.; BEREKOVIC, M. Data-centric computing frontiers: A survey on processing-in-memory. In: ACM. **Proceedings of the Second International Symposium on Memory Systems**. [S.l.], 2016. p. 295–308.

SONG, L. et al. Graphr: Accelerating graph processing using reram. In: IEEE. **High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on**. [S.l.], 2018. p. 531–543.

STANDARD, J. High bandwidth memory (hbm) dram. **JESD235**, 2013.

STANLEY-MARBELL, P.; CABEZAS, V. C.; LUIJTEN, R. Pinned to the walls: impact of packaging and application properties on the memory and power walls. In: IEEE PRESS. **Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design**. [S.l.], 2011. p. 51–56.

SUH, T.; BLOUGH, D. M.; LEE, H.-H. Supporting cache coherence in heterogeneous multiprocessor systems. In: IEEE. **Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings**. [S.l.], 2004. v. 2, p. 1150–1155.

SURA, Z. et al. Data access optimization in a processing-in-memory system. In: ACM. **Proceedings of the 12th ACM International Conference on Computing Frontiers**. [S.l.], 2015. p. 6.

TRACK, E.; FORBES, N.; STRAWN, G. The end of moore's law. **Computing in Science & Engineering**, IEEE, v. 19, n. 2, p. 4–6, 2017.

WEIS, C. et al. Dramspec: A high-level dram timing, power and area exploration tool. **International Journal of Parallel Programming**, Springer, v. 45, n. 6, p. 1566–1591, 2017.

WEIS, C. et al. Design space exploration for 3d-stacked drams. In: IEEE. **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011**. [S.l.], 2011. p. 1–6.

XU, S. et al. Pimsim: A flexible and detailed processing-in-memory simulator. **IEEE Computer Architecture Letters**, IEEE, v. 1, n. 1, 2018.

YANG, X.; HOU, Y.; HE, H. A processing-in-memory architecture programming paradigm for wireless internet-of-things applications. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 19, n. 1, p. 140, 2019.

YU, X. et al. Staring into the abyss: An evaluation of concurrency control with one thousand cores. **Proceedings of the VLDB Endowment**, 2014.

ZHANG, C. Modifying instruction sets in the gem5 simulator to support fault tolerant designs. 2015.

ZHANG, D. et al. Top-pim: throughput-oriented programmable processing in memory. In: ACM. **Proceedings of the 23rd international symposium on High-performance parallel and distributed computing**. [S.l.], 2014. p. 85–98.

ZHU, Q. et al. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In: IEEE. **3D Systems Integration Conference (3DIC), 2013 IEEE International**. [S.l.], 2013. p. 1–7.

ZHU, Y. et al. Integrated thermal analysis for processing in die-stacking memory. In: ACM. **Int. Symp. on Memory Systems**. [S.l.], 2016. p. 402–414.