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

BRUNO ENDRES FORLIN

Improving the Efficiency of Multi-Threaded Processing In-Memory

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

Advisor: Prof. Dr. Luigi Carro
Coadvisor: Prof. Dr. Paulo Cesar Santos

Porto Alegre
March 2022

CIP — CATALOGING-IN-PUBLICATION

Forlin, Bruno Endres

Improving the Efficiency of Multi-Threaded Processing In-Memory / Bruno Endres Forlin. – Porto Alegre: PPGC da UFRGS, 2022.

100 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Advisor: Luigi Carro; Coadvisor: Paulo Cesar Santos.

1. Processing in-memory. 2. Memory energy. 3. Multi-thread. 4. Simulation. 5. Thread-communication. I. Carro, Luigi. II. Santos, Paulo Cesar. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões
Vice-Reitora: Profª. Patricia Pranke
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. Claudio Rosito Jung
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“Aos meus pais e à Luíza.

Que voemos juntos.”

AGRADECIMENTOS

Realizar um mestrado inteiro à distância, em tempos de pandemia exigiu muito mais da minha psique do que eu poderia imaginar. Felizmente eu não tive que carregar esse peso sozinho. Várias pessoas me ajudaram nessa jornada de dois anos.

Ao meu orientador Prof. Luigi Carro, que além de conseguir melhorar uma linha de pesquisa inteira com um simples e-mail, segue sendo um exemplo de bom senso, carinho com o aluno e com as pessoas que estão por trás dos números. Agradeço profundamente toda a ajuda que recebi, tenho o maior respeito pelo professor que és.

Ao meu co-orientador Dr. Paulo Cesar Santos, à parte do conhecimento que dividiu comigo nos últimos anos, agradeço pela atenção diária nesse período turbulento. Agradeço pela amizade, conversas e discussões, sei que sou um pesquisador melhor graças ao nosso convívio. Sem o suporte e apoio constante dos meus orientadores, eu (muito facilmente) poderia ter me perdido no trajeto. Serei eternamente grato.

Também gostaria de agradecer aos colegas do grupo, pelas dicas e suporte durante esses dois anos, que possamos nos rever pessoalmente algum dia. Para completar, não posso deixar de fora minha família, sempre pronta pra me acolher e discutir o futuro. E em especial gostaria de agradecer à Luíza, companheira de quarentena e de vida, vamos chegar mais longe juntos.

ABSTRACT

Processing-in-Memory (PIM), with the help of modern memory integration technologies, has emerged as a practical approach to mitigate the memory wall and improve performance and energy efficiency in contemporary applications. Novel memory technologies and the advent of 3D-stacked integration have provided means to compute data in memory, either by exploring the inherent analog capabilities or by tight-coupling logic and memory. PIM devices aim to explore the entire memory bandwidth, leveraging the application's data parallelism in different ways. With general-purpose programming models and hardware devices that can be accessed independently, it is only natural that programmers try to exploit thread-level parallelism in the applications.

Shared data structures inevitably appear with general-purpose threads and must be handled correctly to maintain memory consistency. Whether this maintenance is done by software or hardware, data must still travel between different memory regions. Current commercial PIM devices ignore data transfer in their designs and leave this task to the host processor, sending data through the memory bus to the host caches, where it will be rearranged and sent back to memory. We argue that this process goes against the principles of PIM design by increasing data movement between PIM and host. We demonstrate this inefficiency analytically and, with experiments, develop a power model that can extract an upper and lower bound for communication energy in the host. Depending on the host processor used, relaying data through the caches can cost 3 $\times$ more energy than the DRAM access, highlighting the heavy energy costs involved in using the host for communication.

To correctly execute these experiments, we need to run benchmarks tightly integrated with the host processor while extracting its metrics. There is a lack of tools capable of quickly simulating different PIM designs and their suitable integration with multiple multi-core host processors. Thus, this dissertation presents *Sim*^{2}*PIM*, a Simulation Framework for PIM devices that seamlessly integrates any PIM architecture with a multi-core host processor and the memory hierarchy. By analyzing data-sharing corner cases, this work shows that this communication, if executed through the host, can hinder the benefits of PIM devices. We use the simulator to demonstrate that if the PIM device relies on the host for data-sharing, communication between PIM

units scales faster with the data size compared to computation. In some cases, it can cost 86% of the total execution time.

We propose a PIM-side communication solution that can reduce the performance and energy costs of data-sharing by maintaining data in the memory module. Inter-PIM can access PIM units and their memory spaces independently, decoupling them from the standard DDR memory access pattern while operating without host oversight. We can achieve performance and energy gains on data transfers between PIM units with a low area and power overheads. The Inter-PIM solution reduces the performance costs of inter-thread data movement by around 20% when data is aligned in memory and more than 4× when data is not aligned. Inter-PIM completely avoids using the host hardware to communicate, significantly improving data-sharing energy efficiency by more than 9×.

Keywords: Processing in-memory. memory energy. multi-thread. simulation. thread-communication.

Melhorando a Eficiência de Processamento em Memória em Múltiplas Threads

RESUMO

Processamento em memória (PIM), com a ajuda de modernas tecnologias de integração, emergiu como uma solução prática para o *memory wall* enquanto melhora a performance e efciciência energética de aplicações contemporâneas. Novas tecnologias de memória juntamente com o surgimento de técnicas de integração 3D proveram os meios para computar dados na memória. Seja explorando as capacidades analógicas ou integrando lógica e memória. Dispositivos PIM tem o objetivo de explorar toda a banda da memória, usando o paralelismo de dados das aplicações de diferentes formas. Com modelos de programação genéricos, e dispositivos de *hardware* que podem ser acessados independentes, é natural que programadores tentem explorar paralelismo a nível de *thread*.

Com *threads* de propósito geral, estruturas de dados compartilhados inevitavelmente surgem, as quais devem ser lidadas corretamente para garantir consistência na memória. Independentemente da maneira como essa consistência é mantida, dados devem ser transmitidos entre diferentes regiões de memória. Os atuais dispositivos comerciais PIM ignoram esse aspectos em seus designs e deixam a transferencia de dados à cargo do processador.

Enviando dados através do *bus* de memória para as *caches*, onde eles serão rearranjados e enviados de volta para a memória. Nós argumentamos que esse processo vai contra os princípios de *design* PIM, aumentando os movimentos de dados entre o PIM e o processador. Nós demonstramos essa ineficiência analiticamente e experimentalmente, desenvolvendo um modelo de consumo de potência que consegue extrair limites superiores e inferiores para a comunicação via o processador. Dependendo do processador usado, retransmitir dados através das *caches* pode custar 3 $\times$ mais energia, salientando os altos custos energéticos em usar o processador para esta tarefa.

Para rodar corretamente esses experimentos, nós precisamos executar *benchmarks* muito integrados com o processador, enquanto extraímos suas métricas. Existe uma falta de ferramentas capazes de rapidamente simular diferentes designs PIM e suas integrações com múltiplos processadores *multi-core*. Logo, essa dissertação apresenta

Sim^{2}PIM um simples simulador para dispositivos PIM que integra qualquer arquitetura PIM com um processador *multi-core* e a hierarquia de memória. Analisando casos de compartilhamento de dados, esse trabalho mostra que essa comunicação, se execudada pelo processador, pode minar os benefícios de dispositivos PIM. Nós usamos esse simulador para demonstrar que se o dispositivo PIM depende do processador para compartilhamento de dados, o custo de comunicação entre *threads* escala mais rápido com o tamanho dos dados do que o custo da computação, em alguns casos podendo custar 86% do tempo total de execução.
Nós propomos uma solução interna para o PIM que reduz os custos de performance e energia de compartilhamento de dados, mantendo a comunicação dentro do módulo de memória. Esse mecanismo pode acessar unidades PIM e seus espaços de memória independentemente, se desacoplando do padrão de acesso à memória *DDR*, enquanto opera sem supervisão do processador. Com baixos custos de área e potência, podemos atingir ganhos de performance e energia em transferências de dados entre unidades PIM. A solução *Inter-PIM* reduz o custo de performance de movimento de dados entre *threads* em 20% quando os dados estão alinhados na memória e em mais de 4× quando não estão. *Inter-PIM* evita usar o processador para comunicação, significativamente melhorando a eficiência energética do compartilhamento de dados em mais de 9×.

Palavras-chave: processamento em memória, energia da memória, multi-thread, simulação, comunicação de threads.

LIST OF FIGURES

Figure 1.1 Increasingly larger gap between processor and memory performance along the years. 14
Figure 1.2 Common Types of Processing-in-Memory Devices 15

Figure 2.1 Reconfigurable Vector Unit (RVU) architecture. 21

Figure 3.1 Different modes for threads to share PIM computation 26
Figure 3.2 Different PIM Vector Engines in an 8kB row DRAM device. 27
Figure 3.3 Vector operations required to operate over 1MB of data with different spatial data localities for a monolithic SIMD unit and multiple smaller units in parallel. 27
Figure 3.4 PIM units on DDR DIMM-like module connecting with an unmodified host 31

Figure 4.1 Host loading, rearranging, and then storing data back to memory. 39
Figure 4.2 Amount of bytes read and written in a broadcast operation between the active threads in two 64-byte transfers. 40
Figure 4.3 Amount of bytes read and written in a multicast operation between the active threads in two 64-byte transfers. 40
Figure 4.4 Placement of the Inter-PIM mechanism in between the data-path of the DIMM. 41
Figure 4.5 Inter-PIM communication mechanism and PIM Unit status. 42
Figure 4.6 High level depiction of Inter-PIM hardware, with the IM as part of the package. 44
Figure 4.7 Inter-PIM interconnect description. 45
Figure 4.8 Breakdown of the total memory power consumption by its components. 49
Figure 4.9 Size of each type of cache access for different data transfers. 52

Figure 5.1 Simulators scope when considering system integration. Many more examples exist in all categories. 56
Figure 5.2 Overview of Sim^{2}PIM modular components and execution phases. 59
Figure 5.3 Creation of threads before instrumentation (left) and after (right). 62
Figure 5.4 Interfaces and overheads of offloading data from the application to PIM-simulation. The functionality encapsulated by *Sim^{2}PIM space* is the executable, while the *Application Space* is the original application code. 64
Figure 5.5 Overhead diagram for a multi-thread application on the Sim^{2}PIM with the Hardware Performance Counters (HPC). 67
Figure 5.6 Simulated Cycles and Simulation Time for a 64MB *vecsum* application offloaded by the Xeon CPU to the PIM device using three different simulation configurations. 72
Figure 5.7 Execution time and accuracy for a *vecsum* application with eight threads in different simulators. 73

Figure 6.1 Energy and Cycle results for 8 threads of vecsum kernel with a broadcast access pattern. 77
Figure 6.2 Energy and Cycle results for 8 threads of vecsum kernel with a multicast access pattern. 78
Figure 6.3 Energy and Cycles for 1MB of non-aligned shared data movement between threads. 79

Figure 6.4 Shared data movement in the *GEMM* application. 79
Figure 6.5 Execution cycles and Energy for *GEMM* kernels 80

Figure B.1 Sim^{2}PIM PIM_interface() call graph. 95
Figure B.2 Sim^{2}PIM join_interface() call graph. 96
Figure B.3 Sim^{2}PIM create_interface() call graph. 96
Figure B.4 Sim^{2}PIM main() call graph. 97

LIST OF TABLES

Table 4.1 Area and power overheads for the 3x3 NoC interconnect @45 nm. 46
Table 4.2 Power consumption for each component, calculated with data from Micron's 8GB DDR4-2666 Data Sheet (MICRON, 2017). 48
Table 4.3 Host System used in the data movement test. 51
Table 4.4 Collected metrics from executing data transfer between DRAM chips with 10^6 repetitions 51
Table 4.5 Energy consumption for data access and movement. Extracted from (KESTOR et al., 2013). 53

Table 5.1 Baselines and Case Study PIM Parameters 69
Table 5.2 Average overheads for two different Hardware Performance Counterss (HPCs), unhalted cycles and retired instructions. Measured with 10,000 repetitions in the warm-up phase of two different processors. 70
Table 5.3 Simulated Cycles vs. Simulation time for Sim^{2}PIM and *perf* on the AMD processor. Values represent a single thread and the average of 4 threads. 71

Table 6.1 Baselines and PIM Parameters 75
Table 6.2 DRAM, Inter-PIM, and Host energy and power results scaled according to the number of threads. 76

CONTENTS

1 INTRODUCTION **14**
1.1 Motivation **16**
1.2 Research Goals and Contributions **17**
1.3 Dissertation Overview **18**
2 RELATED WORK **19**
3 UNDERSTANDING MULTI-THREADED PIM **25**
3.1 Why Bother with Multi-Thread? **25**
3.2 PIM Architecture **28**
3.3 Integrating With the Host System **29**
3.3.1 in-Memory Mapped PIM 30
3.3.2 Code Offloading 31
3.3.3 Cache Coherence 33
3.3.4 Virtual Memory Support 34
3.4 Communicating Between Threads **35**
3.4.1 Parallel Programming Model 35
3.4.2 PIM Hardware Support 37
4 IMPROVING PIM COMMUNICATION **38**
4.1 Communication Efficiency **38**
4.2 Inter-PIM Hardware **41**
4.2.1 Functional Requirements 42
4.2.2 Interconnection Device 43
4.2.3 Hardware Topology 44
4.2.4 Overheads 45
4.3 Memory Access Power **47**
4.4 Processor and Cache Energy **50**
5 BUILDING A SIMULATOR **55**
5.1 Sim^{2}PIM Framework **57**
5.2 Instrumentation **60**
5.3 Interfaces **61**
5.4 Backbone **63**
5.4.1 Precise measurements 63
5.4.2 Environment Setup 64
5.4.3 Communication Buffer 65
5.4.4 PIM-Control Interface 66
5.5 Application Thread Management **67**
5.6 Thread Synchronization **68**
5.7 PIM-Simulator **68**
5.8 Validating the Simulator **69**
5.8.1 Overhead Evaluation 70
5.8.2 Simulation Time Evaluation 71
6 EVALUATING COMMUNICATION STRATEGIES **74**
6.1 Experiment Setup **74**
6.2 Energy Efficiency **75**
6.3 Communication Patterns **76**
6.4 Experimenting with a Complex Application **79**
7 CONCLUSIONS **81**
7.1 Future Work **81**
REFERENCES **83**

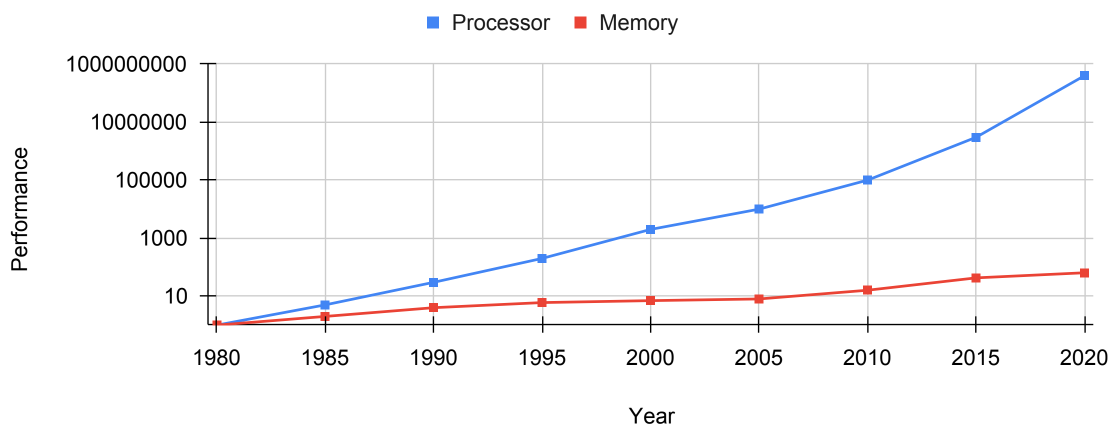
APPENDIX A — HOST TRANSMISSION CODE **91**
APPENDIX B — SIM^{2}PIM **95**
APPENDIX C — RESUMO EXPANDIDO **98**
C.1 Contribuições e Objetivos Alcançados **99**
C.2 Trabalho Futuro **99**

1 INTRODUCTION

In 2022 we see a stark contrast to the reality of 1959 when the Nobel Prize-winning physicist Richard Feynman addressed the American Physical Society: "There's Plenty of Room at the Bottom." The last 50 years saw unprecedented development due to the advances in computer performance predicted in 1975 by Moore's Law (Moore, 2006; SCHALLER, 1997). However, decades of miniaturization took the transistor to its physical limits, and even though we still see strides in technology, they are presenting diminishing gains (LEISERSON et al., 2020). Performance improvements also began to shift purely from frequency increases and transistor counts due to the breakdown of Dennard Scaling (DENNARD et al., 1974). The industry responded to these phenomena with increasingly complex processing cores and the popularization of multi-core systems.

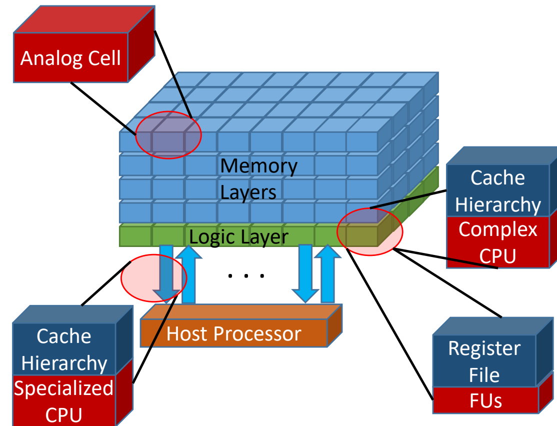
However, the Dark Silicon effect (TAYLOR, 2012) prohibits the entire chip from operating at total capacity at the same time. Furthermore, vector units in these complex processing cores, coupled with the increased number of cores, increased the pressure on the memory system significantly. This pressure aggravated the memory-wall (MCKEE et al., 1994; HENNESSY; PATTERSON, 2011) problem, generated from decades of processor performance gains overtaking memory performance, as shown in Figure 1.1. Cache memories in the processor are inefficient for modern applications that handle vast amounts of data in a streaming fashion (SANTOS et al., 2016; SHAHAB et al., 2018; SANTOS et al., 2017; NAI et al., 2017; GAO;

Figure 1.1 – Increasingly larger gap between processor and memory performance along the years.



Adapted from: (HENNESSY; PATTERSON, 2011).

Figure 1.2 – Common Types of Processing-in-Memory Devices



TZIANTZIOULIS; WENTZLAFF, 2019a). As the widespread adoption of Graphics Processing Units (GPUs) in the last decade has shown, significant gains in performance require new architectures, computing modes, and software that fit critical applications' requirements.

Advances in memory technology, chip stacking, and manufacturing processes have allowed the resurgence of Processing-in-Memory (PIM) as a viable candidate for accelerating memory-intensive applications. There have been several solutions presented in the literature for the memory-wall problem. Despite the similar objective, PIM, Near-Data Accelerator (NDA), and Computing-In-Memory (CIM) devices possess fundamentally different architecture approaches (LOH et al., 2013; NGUYEN et al., 2020). The solutions proposed in (Liu et al., 2018; BOROUMAND; GHOSE, 2018; DRUMOND et al., 2017; AHN et al., 2015a; DEVAUX, 2019; ZHANG et al., 2014a) added complete general-purpose processors to the logic layer. In traditional General-Purpose Processor (GPP), although the performance is commonly measured from the processing logic's point of view, the Last-Level Cache (LLC) is the default data entry point, making it the main bottleneck in terms of on-chip bandwidth. This bottleneck still exists even inside the memory.

Another approach proposes using small functional units (FUs) in the logic layer memory chips. This method is a better fit for power and area-constrained devices such as the ones based on 3D-stacked integration and those that aim to adopt cheaper integration methods, as shown in Figure 1.2. However, it requires innovative programming models, cache coherence, and virtual memory support. Several authors (Lee et al., 2018; SANTOS et al., 2017; NAI et al., 2017; AHN et al., 2015c; CALI et al., 2020; FARMAHINI-FARAHANI et al., 2015; GAO; KOZYRAKIS, 2016; GAO et al., 2017; KIM et al., 2016) proposed the use of custom hardware logic to exploit

the enormous bandwidth available on memory devices fully. However, these works rely heavily on host-side hardware modifications for integration. This requirement presents a severe limitation, as no current system can natively support such PIM.

The most disruptive approach to overcome the memory wall is to think of the memory as a computational device. This approach uses the memory's analog circuitry to process data by allowing multiple cells to be accessed simultaneously, then computing while transferring data between the memory cells and the sense amplifiers. The computation can happen in either the Memory Array or the Peripheral Circuits of the Memory Core, as shown in Figure 1.2. In the Dynamic Random Access Memory (DRAM), this is accomplished by sharing capacitor charges. (GAO; TZIANTZIOULIS; WENTZLAFF, 2019b; HAJINAZAR et al., 2021; SESHADRI et al., 2013; SESHADRI et al., 2017; DENG et al., 2018). In the same way, new technologies (e.g., memristors, spin-transfer torque cells) use electrical resistance, exploiting Kirchhoff's Law, to compute on stored data (DREBES et al., 2020; Xie; Cai; Yang, 2019; Jain et al., 2018; AGA et al., 2017; CHI et al., 2016; ECKERT et al., 2018; LI et al., 2017; SHAFIEE et al., 2016; SONG et al., 2017; SONG et al., 2018; XIN; ZHANG; YANG, 2020).

1.1 Motivation

Regardless of memory-cell technology, integration technology, or architecture, the PIM device can be adapted to work with a GPP environment as described by Santos et al. (SANTOS; FORLIN; CARRO, 2021b). Integration with a multi-core host processor brings several advantages, including enabling the host hardware. Recently commercial PIM solutions have sprung up, promising out-of-the-box integration solutions for GPP environments (LEE et al., 2021; NIDER et al., 2021). Unlike many academic PIMs presented before (SHAFIEE et al., 2016; LI et al., 2017), these PIMs are presented as general-purpose units, which allow the use of general-purpose threads to complete a given task. Instead of behaving as dataflow engines, where data is received, processed, and forwarded in a neatly task-optimized pipeline, these PIMs allow flexibility in the organization and computation of data. These designs are composed of several PIM units within the same memory module, which allow independent access with thread and data-level parallelism.

In this multi-thread scenario, shared data structures and inter-thread data movement inevitably appear in the application. Communication between threads is handled differently depending on the system's memory. This communication, when happening between multiple independent clients, (e.g., multi-core processors, multi processor systems, networks), requires memory consistency (JACOB; NG; WANG, 2010). In the last 40 years, this consistency was implemented in multiple manners, from software-only to hardware mechanisms (JACOB; NG; WANG, 2010). Whether this consistency is maintained through hardware (e.g., MOESI), software mechanisms (i.e., APIs), or both, the data still has to move between different, often physically separated memory regions. For most current shared-memory systems, such as multi-core processors, multi-processor systems, or GPUs, this movement happens inside the caches or with specific protocols and pathways. History tends to repeat itself, and as it was with processor development 40 years ago, the best solution for data movement depends on the technology available and the hardware limitations.

The cache hierarchy in host processors handles inter-thread data movement inside the chip, with minimal costs for the application. However, this cost is aggravated for PIM solutions as the proposed commercial solutions rely on the host hardware to relay communications between the threads. Using the host for inter-thread communication simplifies the hardware at the PIM side at the cost of rearranging the data on the host processor. This inter-thread data movement is forced to occur off-device, incurring a more significant communication latency, energy costs, and degradation of host performance. Currently, there is a general lack of research on the behavior of PIM systems integrating with out-of-the-shelf host processors in multi-threaded regimes. Together with the lack of studies, the absence of tools to efficiently evaluate and explore the design space for PIM software and hardware. This dissertation strives to improve this scenario.

1.2 Research Goals and Contributions

The objective set for this dissertation is to evaluate the performance and energy costs of using the host for inter-thread data movement in the PIM device and investigate the gains of using a dedicated in-device mechanism to do so. First, we review works in the literature showing some of the history of PIM and highlighting that multi-thread execution has not been a point of focus for most works. We

explore the theoretical limits of multi-threaded PIM execution for the data and thread mapping in the PIM. This analysis is agnostic to the PIM hardware and technology, only caring about data arrangement and access width. Then, we explore a PIM architecture and its integration with the host system.

We evaluate the communication costs of PIMs using the host for communication, as is the case with current commercial solutions (i.e., UPMEM (NIDER et al., 2021) and HBM-PIM (LEE et al., 2021; KWON et al., 2021)). This analysis yields the results we need to develop a better model for multi-thread PIM execution and how the lack of dedicated PIM-thread communication harms PIM efficiency. We propose Inter-PIM, a device that can improve energy and performance efficiency for PIM-thread communications. To experiment on this design, we develop Sim^{2}PIM from the ground up. Sim^{2}PIM is a simulation framework that focuses on the interaction between the host and PIM system by executing host code on the actual processor, only simulating the PIM. The simulator will be released as an open-source tool to improve PIM research and development. As we will demonstrate, the inter-thread communication approach is orthogonal to technology and PIM implementation. It is also transparent to the programmer, as it does not require any extra APIs or other coding mechanisms.

1.3 Dissertation Overview

This dissertation is organized as follows: Chapter 2 dives into more detail on some of the PIM design proposals and commercial products that appeared in the last decade. Chapter 3 presents a theoretical analysis of PIM performance when operating with multiple threads. This chapter also demonstrates the means to integrate the PIM device into a host system and the requirements and implications of inter-thread communication. In Chapter 4, we present Inter-PIM, explaining its integration with commercial and academic solutions. We also demonstrate the efficiency and energy costs of moving data with the host system. Chapter 5 contains the description of our simulator and its design philosophy. The results extracted are presented in Chapter 6. Finally, Chapter 7 presents our conclusions and the future work left by this dissertation.

2 RELATED WORK

PIM architectures are starting to solidify with the advent of commercial products (NIDER et al., 2021; LEE et al., 2021; KWON et al., 2021). However, that was not the case less than a decade ago. After a hiatus of almost 15 years, the idea of processing within the memory was brought back. The technology was not quite there to deliver on the promise of in-memory computing (SANTOS, 2019), starting to re-appear with the advent of 3D-stacked memories (PAWLOWSKI, 2011) and a more mature toolchain. One of the most exciting works to come in these early days was RowClone (SESHADRI et al., 2013). While not precisely proposing to operate on the data in the memory, it demonstrated a way to use the DRAM's internal row buffers and busses to transfer data between two rows. This mechanism allowed for incredible performance and energy gains by saving on expensive data transfers between host and memory.

Soon after, more complex PIM architectures appeared based on the recently released HMC standard (Hybrid Memory Cube Consortium, 2013). One example is Tesseract (AHN et al., 2015a) a PIM architecture focused on accelerating Graph processing workloads. It focused on extracting the vast internal bandwidth available in the HMC, with each vault capable of independent processing, in a system composed of multiple memory cubes. The computing cores in each vault were connected by a crossbar network, which allowed for independent package-like communication. The researchers also wrote the graph applications to extract parallelism from the available data by parallel computation for different vertices. When the data is in a different core than the instruction, Tesseract transfers the computation itself instead of the data. They designed an Application Programming Interface (API) for handling this message passing interface, with a blocking and a non-blocking remote function call.

Still capitalizing on the memory technology provided by the HMC, HIVE (ALVES et al., 2016) strived to increase the efficiency of vector operations by executing them inside the HMC. They leveraged that 8 host cores executing parallel SSE computation could not extract the entire HMC internal bandwidth. HIVE was able to extract this performance by using 8kB vector operations in simple, functional units. Additionally, a more straightforward functional unit design could avoid the large area and power overheads of including entire processing cores inside the HMC. For the streaming applications it was designed to handle, the cores caches and out-of-order

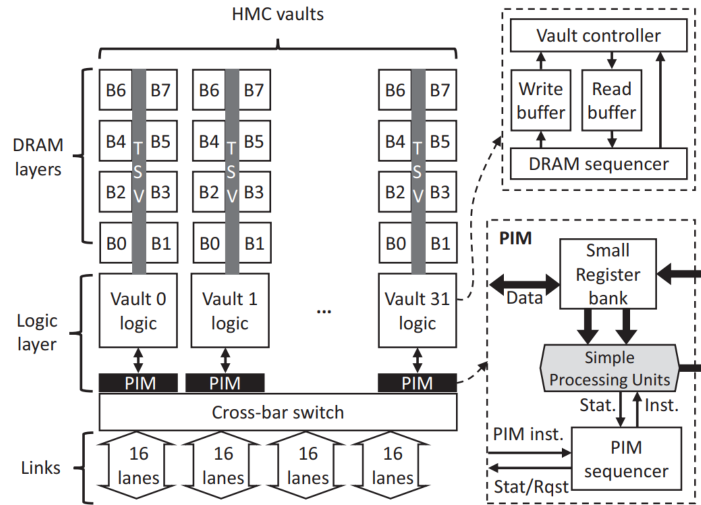
execution increased overhead with little to no performance advantages.(SANTOS et al., 2016; SHAHAB et al., 2018; SANTOS et al., 2017; NAI et al., 2017; GAO; TZIANTZIOULIS; WENTZLAFF, 2019a)

As memristor technology matured, designs also started to become more complex and integrated. ISAAC (SHAFIEE et al., 2016) was arguably the first architecture to deploy a fully-fledged accelerator for Deep Neural Networks (DNNs) inference using memristor crossbars. The architecture adopted a pipelined approach, with each neural network layer receiving its own set of dedicated crossbars. It uses a mix of memristor arrays, eDRAM buffers, networks, and digital-analog converters to create multiple tiles. Utilizing tiling to divide the layers, it can efficiently execute all multiply-accumulate operations in parallel. Each memristor array stores the neuron weights of that layer. The inputs and outputs are stored in the eDRAM, and each layer feeds its output to the following layer. It is interesting to note that as the operations happen in the analog domain, the data must be converted from digital to analog and back multiple times. This operation and the required hardware constitute a significant proportion of the memristor array costs overall.

The Reconfigurable Vector Unit (RVU) (SANTOS et al., 2017) expanded on the concept of HIVE by utilizing the inherent parallelism on the HMC technology. It divided the vector units between the 32 vaults, supporting operand size reconfigurability, from 256B to 8KB data width. This reconfigurability allowed for extended flexibility on the applications that could take advantage of the massive internal bandwidth of the PIM. This flexibility is well represented in Figure 2.1. Each RVU unit is allocated to an HMC vault. They can work in tandem with each other or independently. The issue of instructions and actual work division happens on the host core, where an AVX-512-like compilation divides the workload accordingly. The architecture also supports inter-vault communication through a crossbar switch, allowing the vaults to access each other’s memory space. In this manner, each RVU has access to all of the data stored in the HMC.

The Mondrian Data Engine (DRUMOND et al., 2017) focused on optimizing data analytics workloads with a software-hardware co-design. The authors argue that accessing the large bandwidth available in the HMC with fine-grained random accesses was wasteful and required complex hardware. Not only that, but the nature of data-analytics algorithms, which over time had been heavily optimized for Central Processing Units (CPUs), required an entirely new design both from software and

Figure 2.1 – Reconfigurable Vector Unit (RVU) architecture.



Source: (SANTOS et al., 2017).

the hardware to achieve performance and energy efficiency. From the software perspective, the authors argued that serialized access played well with the nature of data analytics while requiring hardware capable of handling the data stream. Thus, they leverage Single Instruction Multiple Data (SIMD) units to maintain the compute throughput. However, the Mondrian Data Engine takes PIM design a step further. The engine is designed with multiple HMC modules in mind, fully connected through a crossbar network. By using an entire in-order processor core with SIMD capabilities in each HMC vault, and a host processor responsible for orchestrating and synchronizing PIM activity, it manages to extract a great deal of performance.

DRISA (LI et al., 2017) takes a different approach to the PIM design. It proposes to change DRAM cells to make them capable of doing basic binary operations, such as AND, NOR, and Shifts. DRISA is not meant to be used as the main memory, but rather a Convolutional Neural Network (CNN) accelerator. It takes more liberty with the memory cells, reducing the standard DRAM cell density and trading-off area overhead for performance. This trade-off also increases the manufacturing cost for the memory. However, the authors argue that this trade-off is valid for an accelerator. The software design is also highlighted in the solution. To explore DRISA's performance efficiently, the programmer must optimize resource allocation, and the inherent parallelism of the CNN must be coupled with DRISA, which is not a trivial task. Since it is designed as a SIMD architecture, it can be treated as a vector processor, and other applications can be mapped, such as meta-genome data analysis (CHEN; PACHTER, 2005).

The work of Liu et al. (Liu et al., 2018) investigated mechanisms to integrate heterogeneous PIMs, those with fixed-function units and programmable units, using OpenCL. Machine learning frameworks usually rely on middleware to abstract the hardware to the user. However, this increases the burden on the system programmer, which must be aware of several different programming models for different PIMs. Using a 3D stacked PIM architecture, closely integrated with the CPU, they extend the OpenCL API. They demonstrate that a software-hardware co-design technique that leverages the CPU, programmable cores, and fixed-function units, can optimize the training of Neural Networks (NNs). They provide a transparent model to the programmer with efficient run-time scheduling and fit several heterogeneous PIM models.

As the PIM idea reaches maturity, less invasive concepts on the available technologies start to show up. ComputeDRAM (GAO; TZIANTZIOULIS; WENTZLAFF, 2019b) is a PIM architecture that proposes to use off-the-shelf, unmodified Double Data Rate (DDR) modules to realize in-memory operations. It manages to do this by violating the DRAM timing parameters on the Memory Controller. With shorter refresh and open-row phases, this design forces two rows to share the bit-line. With this, the capacitors share charges for a moment, an effect that can be manipulated to operate on the cell charges analogically, with the following refresh operation reverting the charge to a nominal value. The concept is innovative and surprising for working with unmodified DDR modules.

PIM solutions have appeared in multiple different manners, processing paradigms, architectures, and position in memory, as seen in Figure 1.2. Finally, we arrive at the commercial PIM designs of today, where the industry has finally caught up with a decade of PIM development in academia. One of these solutions is the UPMEM system (NIDER et al., 2021), which according to their website, is "the first PIM solution that is fully programmable, scalable, and efficient to address data-intensive applications and without requiring any hardware architecture changes." A UPMEM module is composed of a DDR-like module with 2 ranks, with 8 memory chips each. The chips are not connected, thus inter-DPU communication is only possible by copying data through the host. Each chip contains a DPU, which is composed of a general-purpose processor and SRAM buffers. Memory is copied from the DRAM to the local DPU SRAM via Direct Memory Access (DMA) requests. To hide the latency from the memory reads, each DPU handles up to 24 hardware threads in

an Interleaved Multi-Threading (IMT), which means only one thread advances each cycle. These threads are also limited to the data in the same chip as the processor, so workloads must be mindful of the data layout. Due to the portability to the DDR standard, the host processor reads and writes in bursts of 64 bytes to the entire module, with each chip receiving 1-byte slices of data. This access is divided equally to all the chips; thus, the host must reorder data when loading from the main memory and storing it in the UPMEM. Additionally, this requires that all reads and writes to the UPMEM be synchronous as they happen in all the units. For data to transition from one chip to another, it must be explicitly handled by the host processor and the programmer via the UPMEM API.

The first soon-to-be-available PIM solution from a major memory manufacturer, in this case, Samsung, is an HBM-based PIM (LEE et al., 2021). The authors make an effort to explain why it took so long for the industry to develop a viable PIM. They argue that past PIM architectures all required modifications to the Host hardware, which made adoption challenging enough not to warrant a commercial investment. Their solution sandwiches PIM execution units between the DRAM banks in the High Bandwidth Memory (HBM). Each PIM unit is comprised of command, general, and scalar register files, with floating-point SIMD, add and multiply units. With minimal control logic on the PIM, the host handles memory requests on the PIM's behalf. Thus, PIM execution units access the memory at the same data access granularity as a host processor (LEE et al., 2021). Moreover, as a System in-Package (SiP) can contain hundreds of SIMD logical units, the programming model considers multiple host threads to increase the memory request throughput. This throughput is entirely dependent on the host Instruction Set Architecture (ISA), as the host is integrated into the system, this will most likely propagate to all iterations of the commercial product. The threads are allocated to the same thread group and executed in a lockstep manner, with the same instructions and control-flow path. For their experimental setup, with 4HBM2 cubes, there were 64 pseudo-channels, each requiring 16 host threads to reach peak memory bandwidth, resulting in a total of 1024 threads. To minimize fence overheads between different pseudo-channels, each thread group can only access its DRAM channel. As there seems to be no direct link between each pseudo-channel, it is expected that any data rearrangement must occur through the Host and the PIM API.

Even though the commercial products are a welcome addition to PIM research, these designs are currently being implemented with the PIM units physically separated (i.e., UPMEM DPUs, and HBM-PIM pseudo-channels). Thus, **these PIMs rely heavily on the host processor** to orchestrate the many PIM internal threads, delegating memory management, data rearrangement, and synchronization. Likewise, due to the physical separation, inter-thread communication between different PIM units happens exclusively through the host processor. Ironically, this decision is in clear contrast to the design philosophy of PIM devices since more pressure is applied to the memory channel. This added pressure, in turn, harms the energy and performance benefits of the PIM. Inter-thread communication can be avoided if multi-thread applications are entirely rewritten, the algorithms are changed, or the software is mindful of data layout. However, these are clearly show-stopper situations. And as we will show in the next chapter, the neglect of thread communication can have a severe impact on efficiency.

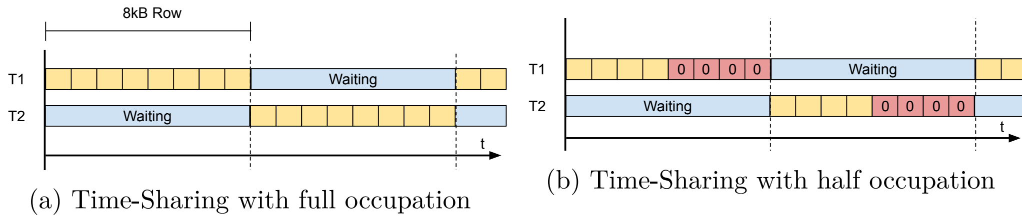
3 UNDERSTANDING MULTI-THREADED PIM

This chapter investigates the advantages and complications of enabling multi-thread support on the PIM device. It also dives in the integration techniques required to connect a multi-thread capable PIM device with an unmodified multi-core host processor. Finally, it discusses the impact that the PIM device can have on inter-thread communication.

3.1 Why Bother with Multi-Thread?

Before we demonstrate why multi-thread PIM requires communication between threads, it is only natural we ask the question: is multi-thread PIM even worth the trouble? The designers from UPMEM (NIDER et al., 2021) and HBM-PIM (LEE et al., 2021; KWON et al., 2021) leverage multiple threads due to the hardware characteristics demanding PIM threads to achieve maximum compute and memory performance, respectively. Their designs rely on Simultaneous Multi-Threading (SMT) concept, where multiple threads are alive simultaneously on the same hardware to exploit memory access latencies. However, we argue that PIMs that allow independent access to parts of the hardware with general-purpose threads are overall more versatile and efficient from the application and data structure points of view by allowing Thread Level Parallelism (TLP).

We can demonstrate this examining two PIM technologies, analog DRAM PIMs (SESHADRI et al., 2017; GAO; TZIANTZIOULIS; WENTZLAFF, 2019a) and memristor devices (Li et al., ; Yu et al., 2018; Xie et al., 2017). Most of the reviewed PIM proposals deal with task-specific hardware. Thus, the concept of general-purpose threads and the optimizations that derive from their flexibility are often overlooked. One such case is how the PIM should handle multiple concurrent host requests, as is the case in a multi-core environment. For task-specific hardware, optimizing for multiple requests defaults to a time-sharing approach, where requests are served in different time-slots, as each task is completed (e.g., cryptographic hardware in a CPU). In general-purpose hardware, more flexibility exists in how requests are served. Figure 3.1 presents two distinct methods to serve multi-thread requests in a PIM device with an operand width of up to 8kB.



(a) Time-Sharing with full occupation

(b) Time-Sharing with half occupation

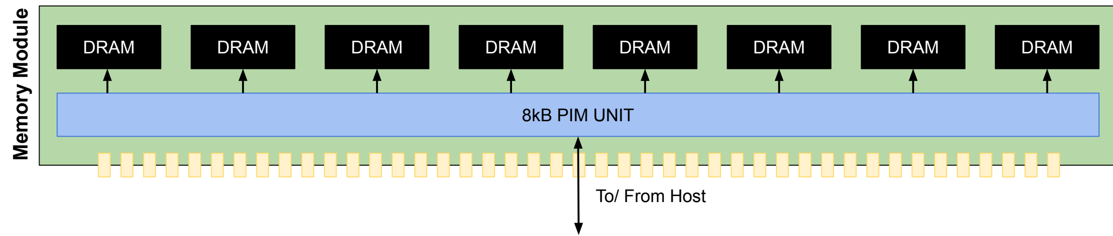
Figure 3.1 – Different modes for threads to share PIM computation

Figure 3.1a demonstrates two threads (T1 and T2) occupying the hardware in a time-shared manner. Although adopting this technique ensures host-thread parallelism, which means parallel request emissions, it does not allow thread-level parallelism at PIM side. Unless the application thread can hide the latency for a *time-slot*, the thread will keep waiting. When a thread is capable of fully exhausting the PIM resources, there is no way around a time-shared approach. However, when the application does not demand enough data locality to occupy the entire 8kB row (i.e., data can not be organized in the memory as to fit the operation size) there is a waste of hardware resources, as shown in Figure 3.1b. For both memristor crossbars[1] and DRAM computations the solution presented in the literature is to pad the rest of the width with zeroes, thus wasting bandwidth and energy (Chu et al., 2020). As these are monolithic structures, the entire crossbar and the entire DRAM row must compute on data. Conceptually, the simplest solution is to increase the granularity of computation.

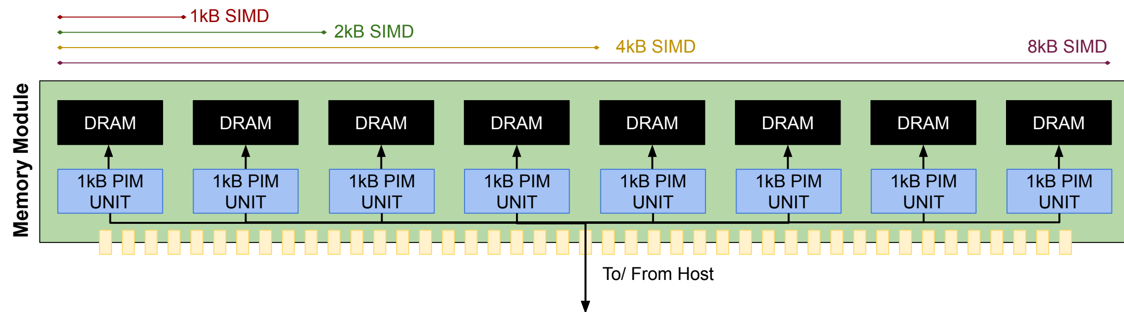
The design presented in (GAO; TZIANTZIOULIS; WENTZLAFF, 2019a) can be abstracted to a monolithic, very large SIMD unit, as shown in Figure 3.2a. If this PIM is paired with an application or data set that can not support such large data locality, it will not be as efficient. PIMs must be able to handle applications that exploit multiple data granularities efficiently. We can envision a PIM capable of operating on different vector sizes, by simply dividing the SIMD unit in smaller segments, as shown in Figure 3.2b. These could operate in unison to provide larger widths, while still being controlled by individual threads. All other implementation variables disconsidered, we can use the number of sequential operations as a proxy for evaluating performance between these two hypothetical PIM hardwares. We model the number of sequential vector operations required (VOP) by taking in to

[1]The occupancy issue for memristor crossbars is similar, but with the wasted resources expanding quadratically and with additional concerns of data arrangement.

Figure 3.2 – Different PIM Vector Engines in an 8kB row DRAM device.
(a) Monolithic 8kB PIM unit.



(b) Granular 1kB PIM units can be arranged to form larger units.

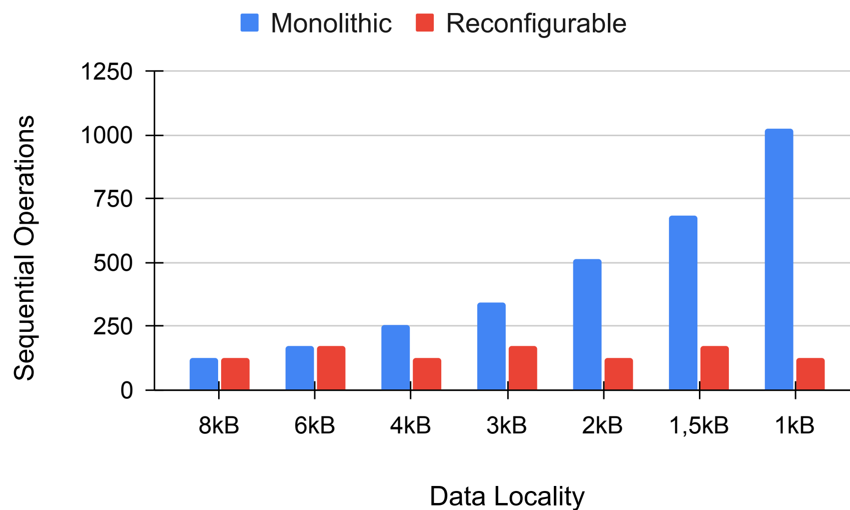


account the data size (D_S), PIM unit width (P_W), PIM unit used (P_U), concurrent threads (T_N), and percentage of PIM unit occupancy (P_O):

$$VOP = \frac{D_S}{P_W P_U T_N P_O} \tag{1}$$

In Figure 3.3, we plot this equation for 1MB of aligned data, with different available data localities, from 1kB to 8kB, and the maximum number of concurrent

Figure 3.3 – Vector operations required to operate over 1MB of data with different spatial data localities for a monolithic SIMD unit and multiple smaller units in parallel.



threads[2] so as to maximize PIM occupancy, we can see that the monolithic PIM requires more sequential operations to compute the data. When the combined width of the PIM units does not match data locality (e.g. data locality equals 1.5kB) or there are left over PIM units (e.g. data locality equals 6kB), the performance is never worse than the monolithic approach. Of course, this distributed approach requires the data to be ordered appropriately in the memory so each PIM unit can process data properly. For the example PIM, this is much like the approach taken by UPMEM, where data for PIM operation is transposed. For memristor crossbars, the issue is similar, but instead of 1D vector operations, the data locality problem expands to 2D. As stated by Chu et al. (Chu et al., 2020), there is a trade-off for memristor crossbar sizes, as a large-scale crossbar benefits from increased capacity and better energy trade-off with peripheral circuits while suffering from a loss of efficiency for unused memristor cells.

3.2 PIM Architecture

As briefly mentioned in Chapter 2, the underlying PIM hardware can direct which multi-thread programming model is more suitable. We can see these distinct approaches in the two commercial PIMs currently available, the UPMEM (NIDER et al., 2021) and the HBM-PIM (LEE et al., 2021). The HBM-PIM has simpler functional units inside the PIM, so it relies on the host for instruction offloading, virtual memory translation, and data coherence. UPMEM on the other hand has full processor cores inside the PIM, operating akin to a coprocessor (e.g. a GPU). However, it can not be used as the system's main memory. For both PIMs, the host must also handle data transfer between the memory and the PIM.

As to avoid unfair comparisons between memory technology and computing hardware in the PIM, we leverage a different architecture, that integrates characteristics from both of them, and could be feasibly implemented. Our test architecture is similar in technology to UPMEM, with regular DRAM chips containing logic. It follows the DRAM specifications and the DDR protocol, fitting inside a Dual In-line Memory Module (DIMM). The PIM device has 8 memory chips, with 8 PIM compute units in total. This logic is similar to the one in the HBM-PIM, with simple SIMD units, that can be combined to work in tandem, as to completely exhaust the

[2] The monolithic hardware only has 1 thread and 1 PIM unit.

internal memory bandwidth. It also serves to avoid the problems of the monolithic PIM device presented in Section 3.1.

The PIM uses an ISA without branch or jump instructions, relying on the host processor for the instruction offload. It contains load, store, and arithmetic instructions. Loads and stores can happen directly in the main memory, or in the internal registers of the PIM unit, or other units of the same PIM. Each PIM unit contains a 1 kB vector unit, with the arithmetic instructions operating in data that is currently in the registers.[3] To coordinate with the host, each PIM unit has a few memory-mapped registers that the PIM can use to communicate with the host. A minimal version of this functionality allows the PIM to set 3 flags: **1** waiting host data, **2** busy, and **3** done. This way, the host can become aware of the current PIM status, by polling these registers directly.

The implemented PIM is presented in Figure 3.4. The following sections will demonstrate its integration with the host system using a novel integration method (SANTOS; FORLIN; CARRO, 2021a).

3.3 Integrating With the Host System

Integrating PIM with the host system comes with its own set of challenges. PIM designs that adopt full processors require no modifications on the host side to provide cache coherence, data consistency, and virtual memory support since they can rely on well-established multi-processing methods (i.e., OpenMP, MPI) (NAIR et al., 2015; ZHANG et al., 2014b; DREBES et al., 2020). Some of the PIM solutions described in Chapter 2 do not have the means to integrate with the host by themselves. Thus, to couple with a general-purpose environment, these PIMs require a series of novel solutions for code offloading, cache coherence, and virtual memory support (SANTOS et al., 2019).

As seen by recent commercial solutions (NIDER et al., 2021; LEE et al., 2021), integration with an unmodified host processor is preferred for a rapid PIM adoption. Few solutions provide seamless integration with the host processor (DRUMOND et al., 2017; Liu et al., 2018; DREBES et al., 2020). All of these problems are tackled in our previous work (SANTOS; FORLIN; CARRO, 2021b), where integration with unmodified host processors is prioritized. The Plug N' PIM strategy accomplishes

[3]The implemented ISA is described in further details on (SANTOS, 2019).

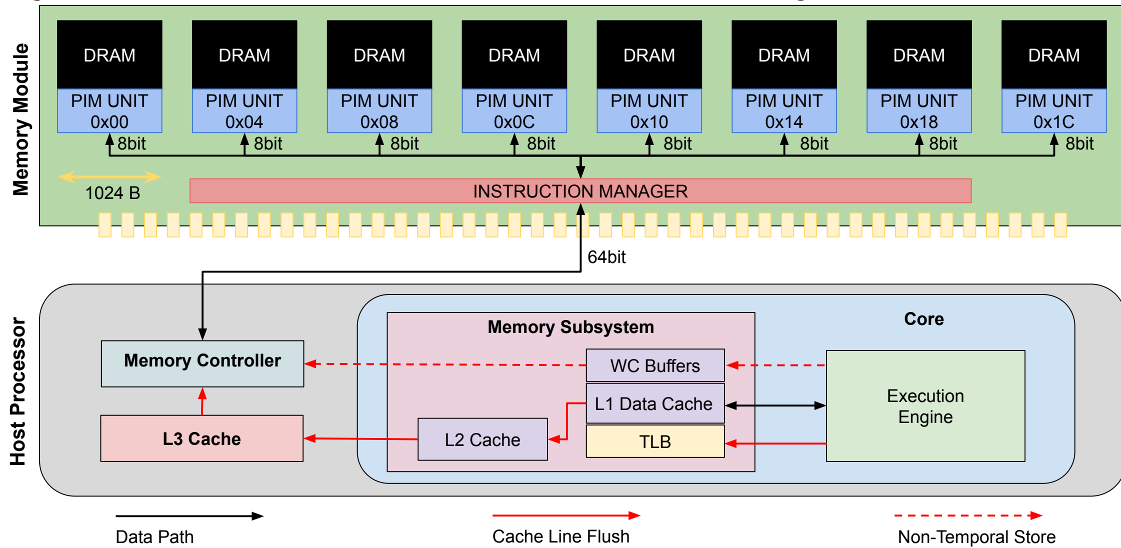
this with the use of native host instructions to deliver the PIM instructions, added to the application executing on the host processor. While the performance onus for the integration falls on the host, the use of similar strategies on commercial products seem to indicate this is preferential to modifying the host processor, at the cost of inserting hardware in the PIM module. This extra hardware, named Instruction Manager (IM) (shown in Figure 3.4), is a small Finit State Machine (FSM) as described in (SANTOS, 2019). It is responsible for decoding PIM instructions and sending them to the correct units. In the next sections, more details and functionalities of the Instruction Manager will be disclosed.

3.3.1 in-Memory Mapped PIM

Plug 'N PIM allows communication between host and PIM units by mapping each PIM unit in memory. That is, the host accesses each PIM unit through a unique address. The mapped devices are within the main memory module instead of coupled to the bus. Moreover, unlike typical memory-mapped devices, the proposed solution only maps PIM devices to addresses as to provide a means for host and accelerator to communicate. Also, the present technique requires no particular type of caching or logical memory type, which means it is fully compliant with any memory region (e.g., uncacheable, write-combining, write-back, write-through). Hence, a host processor can directly access the PIM units via ordinary memory-access instructions *load/store*, with no need for additional host's hardware resources or software/system unique treatments.

Figure 3.4 illustrates the memory-mapped PIM units. Operating System (OS) support is required to allow such mapping, e.g., through a driver. Since PIM instructions are emitted to known addresses, the IM Module can monitor and intercept incoming commands to these addresses and then issue the instructions to the proper PIM device. Therefore, in the case of PIM instructions, the traditional DRAM synchronous communication latency is avoided.

Figure 3.4 – PIM units on DDR DIMM-like module connecting with an unmodified host



3.3.2 Code Offloading

Code offloading can happen in different granularities. Coarser grained implementations dispatch more complex commands to PIM units, such as MPI, OpenMP (NAIR et al., 2015), CUDA-like functions (BOROUMAND; GHOSE, 2018; DREBES et al., 2020), and kernel functions(Liu et al., 2018). The fine-grain approaches rely on individual instructions being offloaded to computing units (AHMED et al., 2019; NAI et al., 2017; LEE et al., 2021).

Commonly, when implementing simple Functional Units (FUs) or exploiting logical resources of modern memory technologies (e.g., ReRAM), the PIM designers avoid implementing all typical stages of a processor, such as instruction and data cache memories, and complex pipelines with intricate fetch and decode stages. This decision occurs mainly due to the area, and power constraints (LIMA et al., 2018). Therefore, the host processor must offload the PIM instructions one by one. This behavior requires both software and hardware to manage the per instruction code offloading.

The compiler naturally manages the software-side in order to adequately select the suitable instructions to be offloaded to the PIM devices (e.g., operating over huge vector) (AHMED et al., 2019). This way, both host and accelerator codes are intrinsically interleaved, being optimized and generated as one. This approach can then generate PIM instructions, as shown in Listing 3.1:

Listing 3.1 – Hybrid Code - x86 and PIM ASM code

```
1  mov $-16384, %rax
2  .LBB0_1:
3  PIM_256B_LOAD_DWORD [%rax+b+16384], %V0_R256B_0
4  PIM_256B_LOAD_DWORD [%rax+c+16384], %V0_R256B_1
5  PIM_256B_VADD_DWORD %V0_R256B_0, %V0_R256B_1, %V0_R256BB_1
6  PIM_256B_STORE_DWORD %V0_R256B_1, [%rax+a+16384]
7  add $4096, %rax
8  jne .LBB0_1
```

However, to deliver these instructions to the PIM device, there must be hardware support from the decoder and execution units. Plug 'N PIM aims at a non-invasive solution, taking advantage of the memory-mapped approach to issue each instruction as a native *store* instruction to the respective address.

Towards this, the compiler embeds each PIM instruction as 16 bytes of data, which can be seen on Listing 3.2. This data is then sent to a memory-mapped PIM address as a *Non-Temporal Store* (i.e., *MOVNTDQ* for x86), natively supported by the host hardware, as shown in Listing 3.2. This solution uses the *Non-Temporal Store* instruction to ensure fast streaming-like writing to the main memory while avoiding polluting the cache memories and their latencies. As illustrated in lines 3 through 8 of Listing 3.2, the compiler generates the PIM memory instruction and places it as an immediate value into a 16 Bytes long register (e.g., *xmm0*).

Listing 3.2 – PIM LOAD embedded into x86 instructions.

```
1  clflush 16384(%rsp,%rax,4)        ;Cache coherence
2  clflush 16448(%rsp,%rax,4)        ;Cache coherence
3  movq $0x002a000000000000, %rbx ;PIM inst. first half
4  movq %rbx, %xmm0
5  movq $0x0000000000000000, %rbx ;PIM inst. second half
6  movlhps %xmm0, %xmm0
7  movq %rbx, %xmm0
8  movntdq %xmm0, PIM0_ADDR(%rip) ;Instruction emission
9  mfence
10 movntdq %xmm0, 16384(%rsp,%rax,4) ;Address emission
11 mfence
```

Then, as shown in line 8, the PIM instruction is emitted to the memory-mapped PIM (*PIM0_ADDR*) via *non-temporal store MOVNTDQ* instruction. The

mfence on lines 9 and 11 ensures that execution will be done in order before other access to those addresses occurs. The non-temporal store (e.g., *MOVNTDQ*) instruction avoids cache hierarchy, which takes between 4 and 7 cycles for the instruction to be committed, according to our measurements for the experimented architectures (Table 6.1). While its throughput is 1 cycle (DEVICES, 2017; CORPORATION, 2016), its total latency is dependent on the main memory performance and its latencies.

Despite the Plug N' PIM technique requiring store operations, it is essential to observe that: 1) Only instructions that effectively access memory (e.g., Load/Store) require two non-temporal stores, as they must emit the address. 2) Other instructions only require a single non-temporal store. In these cases, the non-temporal stores do not touch the memory cells, being captured by the Instruction Manager, as will be shown in Subsection 3.3.4, effectively reducing the instruction latency cost.

3.3.3 Cache Coherence

PIM devices and host may share the same memory address space. Therefore, it is crucial to keep data coherent since the cache memory may contain data that can be requested by PIM units, as illustrated in Listing 3.1. Cache coherence is dealt with, with distinct methods in PIM designs. GraphPIM(NAI et al., 2017) uses a reserved uncacheable memory space for PIM memory, and to guarantee cache coherence, all data allocated to this region bypasses the cache hierarchy. DNN-PIM(Liu et al., 2018) proposes a modification to openCL, inserting an explicit method for host-PIM synchronization. Some designs use flush calls to guarantee cache coherence via high-level APIs (DREBES et al., 2020).

Flush instructions are present in the host processor's ISA (e.g., *CLFLUSH* for x86, *MCR* for ARMv8, *SFENCE.VMA* for RISCV) to be used at no-privilege mode, which makes them suitable for running at user-level application. Similar to (DREBES et al., 2020), Plug 'N PIM maintains cache coherence by adopting *flush* instructions. However, this approach can burden the programmer, and the authors claim not to have experimented with its interaction with an operating system. Instead, Plug 'N PIM lets the **compiler** handle the generation of flush operations by checking whether an instruction accesses or modifies memory. For instance, if a *PIM_LOAD* instructions that access 128 Bytes must be triggered, the compiler will

assure coherence by flushing all related cache lines before sending the instructions, as shown on lines 1 and 2 of Listing 3.2. In this example, as PIM requests 128 Bytes, two *flush* operations of 64 Bytes (cache line size) are emitted to keep cache coherence. In case of *PIM-STORE*, *flush* operations must also be emitted since *host* may request the same address after the PIM modifies it.

Although this is a functional strategy, flushing the cache line for each *PIM-LOAD* or *PIM-STORE* instruction is costly. Thus, our compiler inserts flush instructions only on memory regions that could have been previously modified by the host processor or the *PIM*. This flush operation could also be handled directly by the programmer or at the run time, as is the case with devices with distinct memory partitions such as GPUs.

According to our measurements, the costs of a *flush* operation can achieve between 105 cycles (Intel Kaby Lake/Cascade Lake) to 249 cycles (AMD Ryzen Summit Ridge), depending on cache memories latency and cache line status. For this operation, a throughput of 10 cycles can be achieved (DEVICES, 2017; CORPORATION, 2016). Also, this overhead is dependent on main memory latency; hence it is essential to avoid unnecessary *flush* operations. Moreover, the communication between host and PIM can be harmed by poorly generated PIM code.

3.3.4 Virtual Memory Support

Virtual memory support allows PIM to integrate easily with current programming methods and practices, including the abstraction of memory addresses and ensuring multi-process isolation. This can be accomplished if the PIM replicates the host Translation Look-aside Buffer (TLB) and the Memory Management Unit (MMU) hardware (DRUMOND et al., 2017; AHN et al., 2015b), or the PIM must be able to share or access the host's TLB (SANTOS et al., 2019; NAI et al., 2017).

Plug 'N PIM allows for tightly-coupled PIM devices to support unrestricted memory sharing while providing code offloading and cache coherence through ordinary host's instructions. Thus, virtual to physical addresses translations should occur in the same way. Our approach avoids replication of TLBs (DRUMOND et al., 2017; Liu et al., 2018; BOROUMAND; GHOSE, 2018), by leveraging hardware and software at the host side. The targeted PIM types can adopt the PIM Instruction Manager module, as illustrated in Figure 3.4.

PIM IM module works as a front-end for PIM instructions. Its main role is to identify, *trap*, and compose at running-time *Load* and *Store* PIM instructions. PIM instructions cannot contain memory addresses, since the host emits them as data (Section 3.3.2), and its TLBs cannot translate addresses from data. As aforementioned in Section 3.3.2, the compiler generates a second *Non-Temporal Store* instruction (line 10 of Listing 3.2) to the target address-base of the *Load/Store* instruction. Also, the compiler issues a second *fence* instruction to ensure the execution order (line 11 of Listing 3.2). It is essential to notice that the two *fence* instructions ensure order between the first and second *Non-Temporal Store* operations.

The second *Non-Temporal Store* instruction does not require cache coherence treatment as the memory was previously flushed at this specific address, and it will be *trapped* by the IM not effectively accessing memory. The host's TLBs will translate this address granting the virtual memory support seamlessly. Both non-temporal instructions will be jointly used to compose the *LOAD* or *STORE* PIM instruction, which will then be processed by the correctly mapped PIM. In these *non-temporal store* operations, DRAM cells are not accessed, hence no additional latency is caused. Thus, the presented design allows low latency virtual memory support using the native host's instruction and a module at the PIM side.

3.4 Communicating Between Threads

UPMEM and HBM-PIM are both designed around multiple PIM threads. However, they do not allow for PIM threads to access the memory of one another, specially in different PIM units. Thus, they rely on the host hardware to bounce data between PIM units. This section will discuss the requirements and consequences on the PIM hardware and software model. The execution of this operation in the host system and its costs, will be evaluated in Section 4.1.

3.4.1 Parallel Programming Model

The general-purpose programming paradigm that current commercial PIMs present comes with a series of expected guarantees and mechanisms which programmers are used to. This includes familiarity with the parallel programming

model, memory coherence, and support for higher-level functions and libraries. The parallel programming model adopted by the PIM mainly depends on the hardware support available to handle memory consistency. We can have atomic data access in a scenario where memory coherence and consistency are guaranteed with tight constraints (JACOB; NG; WANG, 2010). The atomic access allows us to use synchronization primitives, including fences, barriers, and release and acquire operations. A strong foundation for these primitives allows the programmer or API to avoid data race conditions in a multi-thread environment. Modern host systems primarily utilize hardware-based memory consistency implementations. Different parallel programming frameworks arise depending on the memory consistency mechanisms present in the system. Two of the most common, OpenMP and MPI, are almost directly translated from the underlying memory available to the threads (KANG; LEE; LEE, 2015). OpenMP relies on the shared memory for synchronization and data transfer between threads. While MPI, a specification rather than a specific implementation, coordinates messages between the memories of different processes that are not required to share memory space.

While theoretically possible to implement each system on any hardware, some hardware implementations demand a specific paradigm for better performance. Different PIM hardware is more suitable for different models than others. If the PIM threads can share the entire memory space with low-latency, they are more easily represented by a shared memory system with the OpenMP model. Otherwise, if the threads are separated and cannot directly access the entire memory, they are closer to an MPI model. However, the difference between current PIMs and the CPU systems that originated these models is the communication cost between the threads (i.e., shared data structure coherence). Whether coherence should happen with oversight from the hardware (e.g., MOSI) or software (e.g., APIs) is a decision that should take into account the hardware architecture and targeted algorithms. However, the means through which coherence is enforced in the PIM are not necessarily related to the architecture. For a PIM device connected to the memory bus, the only options for data transfer are transferring data internally on the memory module, or bouncing data through the host. If the PIM units depend on the host to communicate between each other (regardless if software or hardware monitoring triggered the coherence request), they must bounce data through the host caches, communicating through a distant bus and busing the host with additional work.

3.4.2 PIM Hardware Support

Since the host is responsible for overseeing PIM communication, it must take active part in the process. As there is no means for a device in the memory bus to directly communicate with the host, the host is required to poll the PIM status. The communication requires 2 additional status flags for the PIM units: **1** data to send and **2** data to receive. This way, the host can wait for the PIM devices to hit a synchronization barrier before exchanging data.

Considering the PIM approach presented in Figure 3.4, if a PIM unit requests data from another PIM unit, the following steps are required : **1)** the host processor must keep polling all PIM units to be aware of each operation being demanded; **2)** after the host processor identifies an inter-PIM request, it needs to check whether the source PIM unit is available; **3)** if so, the host sets the flags to halt the source PIM unit, reads the data from the source PIM memory space, and stores it into its cache memory or registers; **4)** the host sets the destination PIM unit, writes the data into the destination address space.

A PIM unit can access its local physical memory through large internal buses (depending on implementation). However, as highlighted in Figure 3.4, each PIM unit can access external components through a limited 8 bit wide bus. By distributing PIM units across memory devices (e.g., Figure 3.4) or memory partitions (SANTOS et al., 2017; LEE et al., 2021), each PIM device is prevented from fully accessing the entire memory bandwidth, limited by its narrow private external buses.

Although the host processor can read and write 64 bytes of data (i.e., x86 cache line writeback - 8 bursts of 64 bits), this limitation primarily impacts PIM-to-host communication. It requires that the host processor and PIM units filter the incoming and outgoing data to avoid reading from and writing to non-required PIM units. Furthermore, this leads to increasing the number of transmissions to overcome that bus limitation.

4 IMPROVING PIM COMMUNICATION

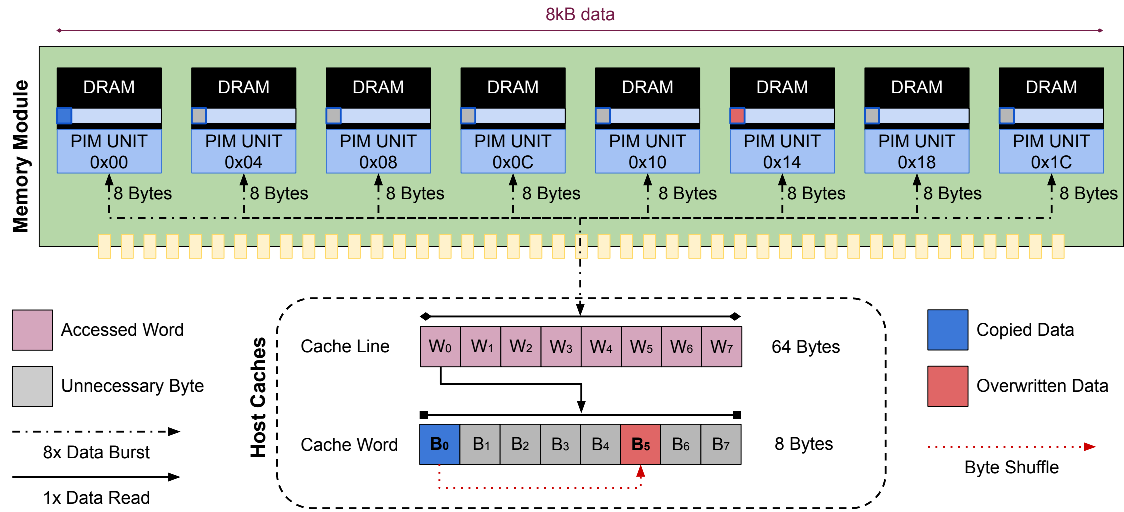
Communication between PIM units is managed by the host happens through a distant bus. As shown in Section 3.4 this spares PIM hardware modifications in exchange for affecting the programmers choices on how to couple PIM hardware and software. This chapter will evaluate the impact this communication can have and means to implement a more efficient hardware solution and the energy benefits of applying it to PIM communication.

4.1 Communication Efficiency

This section investigates the efficiency of using the host's memory hierarchy to provide communication between PIM threads. Given the PIM design presented in Figure 3.4, the entire memory module composes the PIM device, with each PIM unit contained inside a DRAM chip. For the purpose of this dissertation, we evaluate only communication between units, considering each one an application thread. Thus, threads do not share PIM units. Regardless of the memory technology, several bottlenecks and disadvantages arise when adopting the host processor as support for transferring data between. There are two scenarios where the lack of communication capabilities between PIM units can really hurt performance. When PIM threads need to pass data chunks between each other, and when there are alterations in data layout. The first scenario can be as follows: an application may require processed data to join in a single thread to perform another operation that requires the data from the other threads (e.g., layers of a CNN converging data for the fully connected network). The second scenario may happen when a matrix that was split among the multiple PIM units, must now be transposed for another operation.

We perform an analytical evaluation of the performance impact of two corner cases that stress the memory system in different ways. First, a broadcast operation that occurs from one PIM unit to all the other currently active units (one-to-all). Second, multicast operations that occur between all the active threads concurrently (all-to-all). Regardless of the the number of PIM units involved in the communication, the data transfer will still follow the standard main memory access protocol. The smallest data chunk available for the processor to access from the memory is a cache line. The way memory is partitioned in the DRAM ranks and banks optimizes for

Figure 4.1 – Host loading, rearranging, and then storing data back to memory.

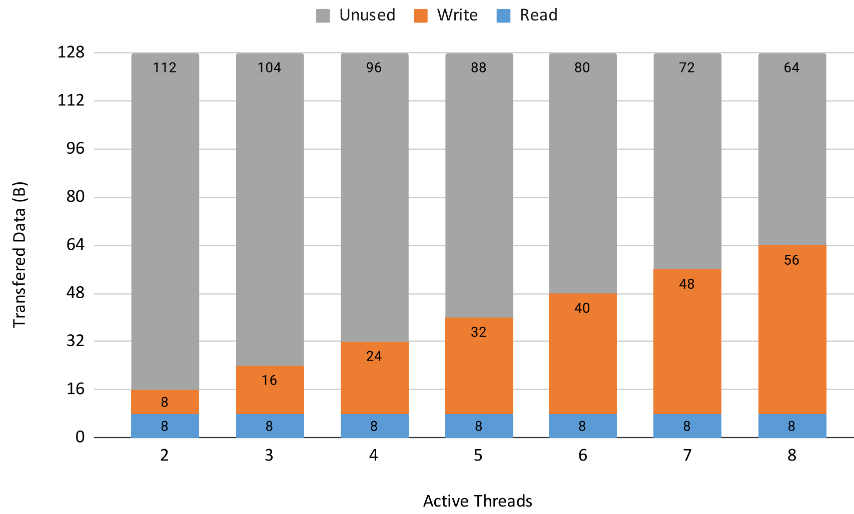


transferring an entire cache line. This results in the separation of consecutive bytes of a word in different DRAM chips. So if the PIM needs to transfer data from one PIM unit to another, the host needs to read an entire cache line, access each individual word, shuffle the data, and then write the cache line back to memory, as shown in Figure 4.1.

There is no in-device physical shared cache memory; hence the data must be copied from the PIM unit source to the host's cache memory and then stored into each PIM destination memory space. If the data can not be transferred to the same row address in every PIM, the host will have to execute a store operation for each PIM destination. DMA is ineffective in this scenario, since data must be operated on by the host. A 64 bytes (8×64-bits) write operation is emitted from the host to PIM, but each PIM unit only receives 8 bytes (8×8-bits). The behavior mentioned above adds to the disadvantages in terms of performance and energy efficiency. A simple optimization is to synchronize the PIM units and align their input buffers in memory. Therefore a single cache line written by the host can carry data for all units at once. In this case, each PIM unit will receive 8×8-bits in parallel, using all 64 bytes available by the host operation.

We can plot this efficiency looking at the number of bytes actually used in each two-way 128 byte transaction (64 bytes each way). For the broadcast operation, we can see the distribution of used bytes for read and write depending on the number of threads involved in the communication in Figure 4.2. The most efficient scenario, where all 8 threads are involved (1 to 7), we can only achieve 50% efficiency in the

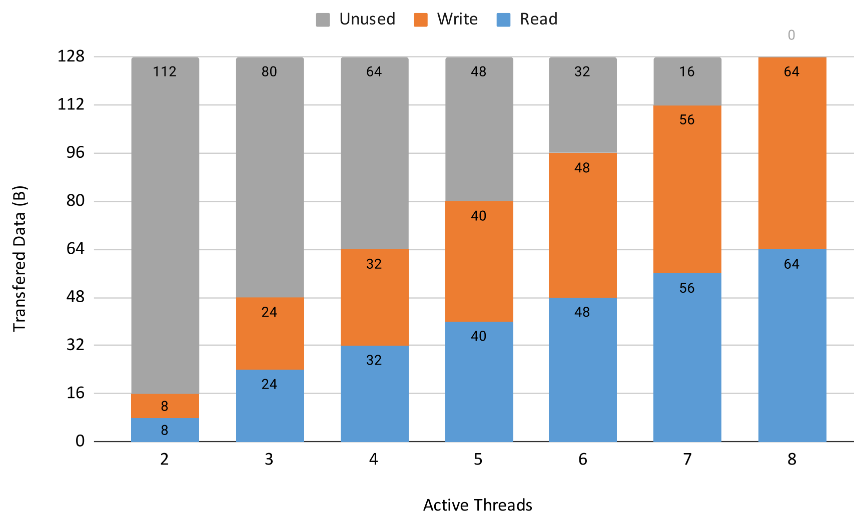
Figure 4.2 – Amount of bytes read and written in a broadcast operation between the active threads in two 64-byte transfers.



transmission. The more threads are involved in receiving the broadcast, the greater the data transfer efficiency due to the data interleaving. However, reading data will never become efficient.

Figure 4.3 shows a more efficient scenario, where all threads are communicating with each other at the same time. In this scenario the cache lines are entirely used for writes and reads. This requires that all threads are synchronized in time (temporal allocation) and all data is correctly aligned in the same DRAM row (spatial allocation). The criteria can happen when threads are involved in computing the same kernel with different data. However, this temporal and spatial allocation might not be possible

Figure 4.3 – Amount of bytes read and written in a multicast operation between the active threads in two 64-byte transfers.



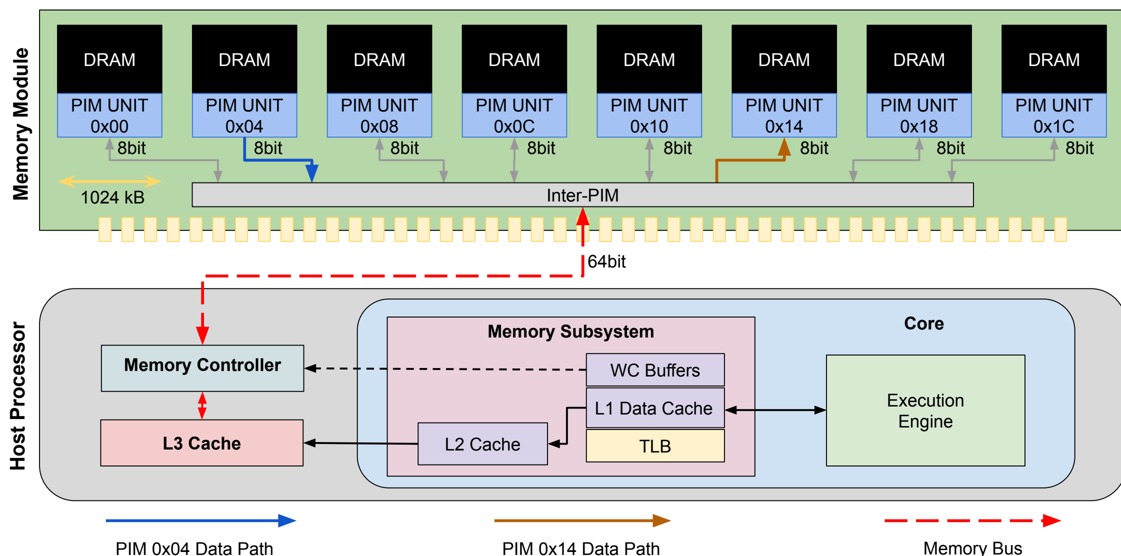
when threads are computing different kernels or handling data of unequal size. It is also noticeable that, as the number of involved threads falls, so does efficiency.

In both scenarios the implication is that for this method of communication to be efficient all threads need to be synchronized, making use of communication at the same time. This also requires that shared data addresses must be aligned in the memory, otherwise communication will have to happen separately. Thus, to improve inter-thread communication, we must be able to access each PIM unit independently, both in time and space.

4.2 Inter-PIM Hardware

This section presents a concept to increase the communication efficiency shown in the previous section when using the host processor as a medium between multiple PIM units. The main objective of this concept is to restrict the data movement between PIM units to the memory module only, hence avoiding the use of the host, and accessing only the PIM units involved in the communication. Figure 4.4 illustrates the *Inter-PIM* placement within a typical DDR memory module.

Figure 4.4 – Placement of the Inter-PIM mechanism in between the data-path of the DIMM.



4.2.1 Functional Requirements

Inter-PIM is designed to reduce data movement with the host, without harming regular memory access. Thus, the Inter-PIM can:

Be transparent to the host: The host is responsible for feeding PIM instructions and copying data from the main memory into PIM memory on the target PIM architectures. The host also needs to monitor the PIM for additional data requests or completion. Thus, Inter-PIM keeps a copy of all PIM status to provide information to the host when required. When handling host load/stores, the Inter-PIM simply forwards the requests. This entire process happens transparently to the host. The only difference is that the host must read the Inter-PIM status addresses instead of the PIM units.

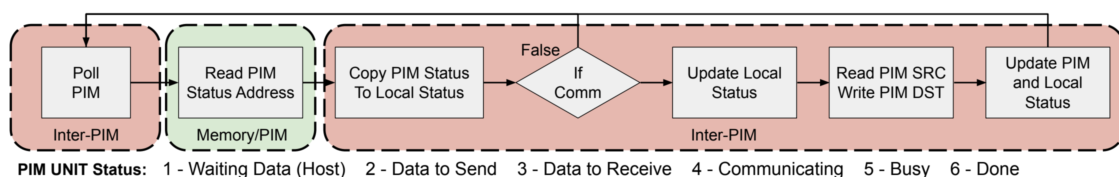
Autonomously poll PIM units status: Since Inter-PIM triggers status polls directly from the PIM's memory, it does not burden the host processor nor pollutes its cache memories. Inter-PIM keeps polling all PIM units, checking whether any request is pending. If desired, this strategy allows it to keep polling PIM in a different frequency than the host.

Manage load from and store to the PIM: Inter-PIM can access a PIM's memory space region, which can be used as a shared buffer between it and each PIM unit. This buffer memory region can be dynamically addressed by the application or fixed by design.

Integrate with the Instruction Manager: Inter-PIM integrates with the IM, as it has the same placement in the memory module. As the IM must write to each PIM unit, it can use the Inter-PIM infrastructure to do so.

Allow parallel communication between PIM units: By removing communication from the host, energy gains are guaranteed. However, for performance gains, the Inter-PIM must be capable of providing multiple concurrent accesses between PIM units asynchronously, as shown in Section 4.1.

Figure 4.5 – Inter-PIM communication mechanism and PIM Unit status.



Inter-PIM obeys a simple flow, as shown in Figure 4.5, indicating each possible PIM status. Status **1**, **5** and **6** are signals that the host interprets. Thus Inter-PIM forwards them without any action. When the PIM status from one of the PIM units changes to **2** (Data Send) or **3** (Data Receive), Inter-PIM interprets which PIM is the destination and which one is the source from the same status registers. Then it waits for the other PIM unit to enter the complementary status (either send or receive) and handles the data transfer. When the handshake is performed, and the threads start transferring data, Inter-PIM updates the local and the PIM unit status to **4** (communicating), so the host is aware of the communication, and the PIM units know when the transmission is occurring. Our design is generic in terms of PIM architecture, which means it can be adopted by PIMs that implement either full processors or simple FUs. It also performs regardless of the memory architecture or technology, being it classical DRAM or memristor based.

4.2.2 Interconnection Device

There are three main types of generic interconnections: buses, crossbars, and Network-on-Chips (NoCs) (PASRICHA; DUTT, 2008).

Bus: these are the simplest types of interconnects, used for decades in industry due to their simplicity. The bus operates on a master-slave control scheme, where only one master has access to the bus at a time. However, as one of the goals of Inter-PIM is to provide simultaneous communication multiple input and outputs, busses are not optimal for the Inter-PIM interconnect. Specially because the bus throughput and bandwidth do not scale well with the number of devices connected to it.

Crossbar: they consist of a matrix of switches and wires, interconnecting inputs and outputs. When there is contention in the destination, an arbiter is considered. Crossbars allow for more scalability and simultaneous communication can happen between threads, as long as there is no contention for the output ports. Crossbars select the inputs through MUTEX trees, and a large centralized arbitration logic. The area and power overheads of these interconnects falls mostly on the long wires connecting all of the inputs to all of the outputs (MATOS, 2014; PASRICHA; DUTT, 2008).

Network-on-Chip: NoCs integrate a series of independent routers, responsible for forwarding packages to one of five I/O ports, as shown in Figure 4.7a. Arbitration happens in a distributed manner at each of the routers, allowing for a more dynamic data flow to adjust to traffic contention (ANDERS et al., 2010). Each router also contains levels of queues and buffers, allowing for communication to be preempted, which is specially useful for allowing priority host access to the memory. NoCs have been used in 3D-stacked memories due to their high bandwidth and capability to deal with multiple transactions simultaneously (HADIDI et al., 2018; MATOS, 2014; Hybrid Memory Cube Consortium, 2013). Furthermore, the Instruction Manager can integrate seamlessly as one of the nodes in the network, and have simple access to all PIM units, without interfering with the memory-host data flow.

Since NoCs have been used in high-bandwidth scenarios, such as the Hybrid Memory Cube (HMC) hardware, it makes sense that a NoC could be efficiently used for Inter-PIM. Thus we selected the NoC as the interconnect to that enables Inter-PIM.

4.2.3 Hardware Topology

For Inter-PIM to be compatible with regular memory accesses and Instruction Manager oversight, it must allow reads from the memory to reach the memory bus directly, having minimal interference in host-memory data transfer, and place the IM in the path between memory and host.

Figure 4.6 – High level depiction of Inter-PIM hardware, with the IM as part of the package.

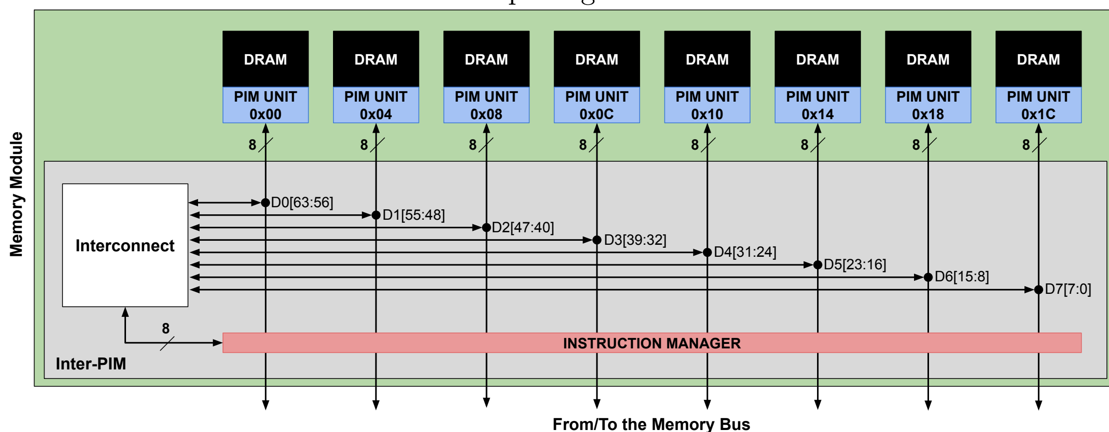
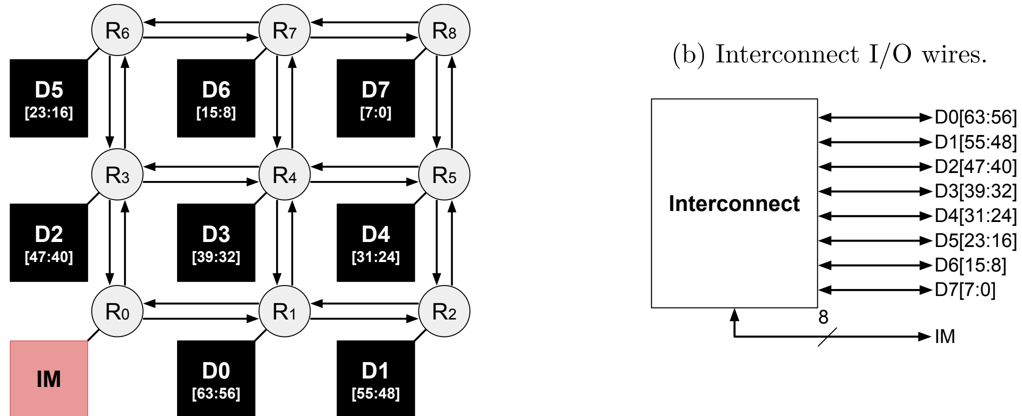


Figure 4.7 – Inter-PIM interconnect description.
(a) Inter-PIM interconnect as a 3x3 mesh NoC.



(b) Interconnect I/O wires.

As shown in Figure 4.6[1], the first criteria is achieved by not having the interconnect in the critical path of the bus. The second criteria is achieved by simply maintaining the IM position in the bus, as would already happen without Inter-PIM (SANTOS; FORLIN; CARRO, 2021b; SANTOS, 2019). Status updates all happen through the interconnect, and data movement not requested by the host can be blocked from reaching the memory bus by the Instruction Manager.

In Figure 4.7a we can see the NoC topology. The NoC is inspired by the high-bandwidth NoC presented in (ANDERS et al., 2010). It consists of 9 routers with internal crossbars, global arbitration, I/O buffers, and internal queue slots for dynamic traffic pattern optimization. The NoC design is well suited for any multicast configuration. A broadcast can happen by copying the packets and forwarding them to all routers.

4.2.4 Overheads

To calculate the overheads for the Inter-PIM device with a NoC interconnect, we can take the router design from a high-bandwidth NoC (ANDERS et al., 2010) and scale the arbiter logic to the appropriate data width. Their 8x8 NoC design implemented in a 45nm technology node resulted in a device with a die area of 6.25 mm^2 with 64 routers. The router power consumption is dependent on the network saturation, with a fully saturated NoC consuming 74 mW/router, while with 50% saturation the consumption falls to 21 mW/router. The power consumption of the

[1]The control and address wires are not shown in the figures for simplification.

arbitration logic also varies with the saturation, resulting in 17% and 10% of total router consumption, respectively.

In (ANDERS et al., 2010), the authors provide the relation of arbiter-width to power consumption, and not to area. So for the area scaling, we simply recalculate the number of routers in the network from the 8x8 grid to a 3x3 (64 routers to only 9). This results in a reduction of more than 7× in the total interconnect area, resulting in 0.87 mm^2. As this interconnect is an off-chip (but still inside the memory module) device, the area cost is difficult to compare. The original design used 512 bit-wide arbiter logic in the routers and each router in Inter-PIM has a data-width of 8 bits, a reduction of 64×. Thus, we scale the router power consumption due to the reduction in arbiter logic size as:

$$P_8^{Sat} = \frac{P_{512}^{Sat} \cdot 0.17}{64} + P_{512}^{Sat} \cdot 0.83 = 61.6\ mW/router \tag{2}$$

$$P_8^{50\%} = \frac{P_{512}^{50\%} \cdot 0.1}{64} + P_{512}^{50\%} \cdot 0.9 = 18.93\ mW/router \tag{3}$$

For the total power consumption of the Inter-PIM interconnect we simply multiply the number of routers for each scenario. The results are presented in Table 4.1.

Table 4.1 – Area and power overheads for the 3x3 NoC interconnect @45 nm.

Traffic	**Total Power (W)**	**Total Area (mm²)**	**Latency (ns)**
Saturated	0.55	0.87	1.3
50%	0.17		

Finally, we calculate the extra latency added to the system. Without entering in the details of the routing algorithm, Quality of Service (QoS), and device positioning, we can only have a rough, worst case estimate of the actual costs. We do this by taking the result from the original NoC proposed by Anders et al. and scaling it down to our NoC. The original NoC could transmit a burst of 512 bits from corner to corner in 11ns. As our NoC is smaller both in number of hops from corner to corner (33% smaller) and in burst length (512 to 64 bits), we consider a simple scaling that takes in to account the reduction of packet size as a first order approximation of the worst case. Thus, we consider the delay added by Inter-PIM to be 1.3 ns for 64-bit transmissions.

4.3 Memory Access Power

To evaluate the power consumption overheads of the Inter-PIM solution, we first must understand the costs of moving data without it. Many works have evaluated power performance of DRAM chips (SESHADRI et al., 2013; GHOSH; LEE, 2007; GHOSE et al., 2018; AHN et al., 2012). DRAM power consumption can be divided in two parts, static and dynamic power. Static power comes mostly from peripheral circuits, transistor leakage and refresh operations. Dynamic power is a two step process, activate-precharge and read-write operations. Activate-precharge power comes from decoding addresses, and opening and closing rows inside the chip. Read-write power comes from reading or updating data in the column-level operations, this includes transferring control and data signals through the memory bus and chip-to-chip I/O (AHN et al., 2012). We modify the highly detailed power consumption model for a DDR4 memory from a Micron datasheet (MICRON, 2017) to suit our analysis. As communication between PIM units will still have to access the DRAM chips and the per-chip data transfer is still the same, we modify the formula so that the number of active devices D_A stays in evidence. So the total memory power P_M is given as:

$$P_M = P_{static} \cdot D_T + D_A \cdot (P_{WR} + P_{RD} + P_{ACT}) + P_{IO} \cdot B_w \quad (4)$$

Where P_{static} is the accumulated power demand of keeping the chips on, the memory bus connected and the memory refreshing, D_T is the total number of memory chips, P_{WR} is the power required for writing to the DRAM columns, P_{RD} is the power required for reading from the DRAM columns, P_{ACT} is the power required to activate the rows in the DRAM chip, P_{IO} is the power consumed by data moving through the bus and through the pins, it is dependent on bus width B_w. As stated before, the internal power consumption of each memory chip remains the same, thus we can take this values directly from the specification provided by Micron, as shown in Table 4.2.

The Inter-PIM device coupled to a DIMM shares electrical characteristics to a Load Reduced Dual In-line Memory Module (LRDIMM). This means the load in the bus is less dependent on the module implementation and the number of devices connected to the bus. Data is driven through the bus from the memory controller on writes and from the Inter-PIM device on reads. This means the system behaves

Table 4.2 – Power consumption for each component, calculated with data from Micron's 8GB DDR4-2666 Data Sheet (MICRON, 2017).

Power Component	**Power (mW)**
ACT	153.9
RD	64.6
WR	35.3
$Static$	85.5

as a point-to-point system. The energy for driving the signals through the bus (P_{IO}) is system-dependent, and there are different manners to implement the DDR4 JEDEC standard (JEDEC, 2012). For this analysis, the bus impedance is composed of the driver impedance and the termination impedance, where $R_{drvr} = 20\ \Omega$ and $R_T = 47\ \Omega$, thus $R_{bus} = 67\ \Omega$ (Values referenced from (NXP, 2016) for an SDDR4 memory interface).

$$R_{bus} = R_{drvr} + R_T \tag{5}$$

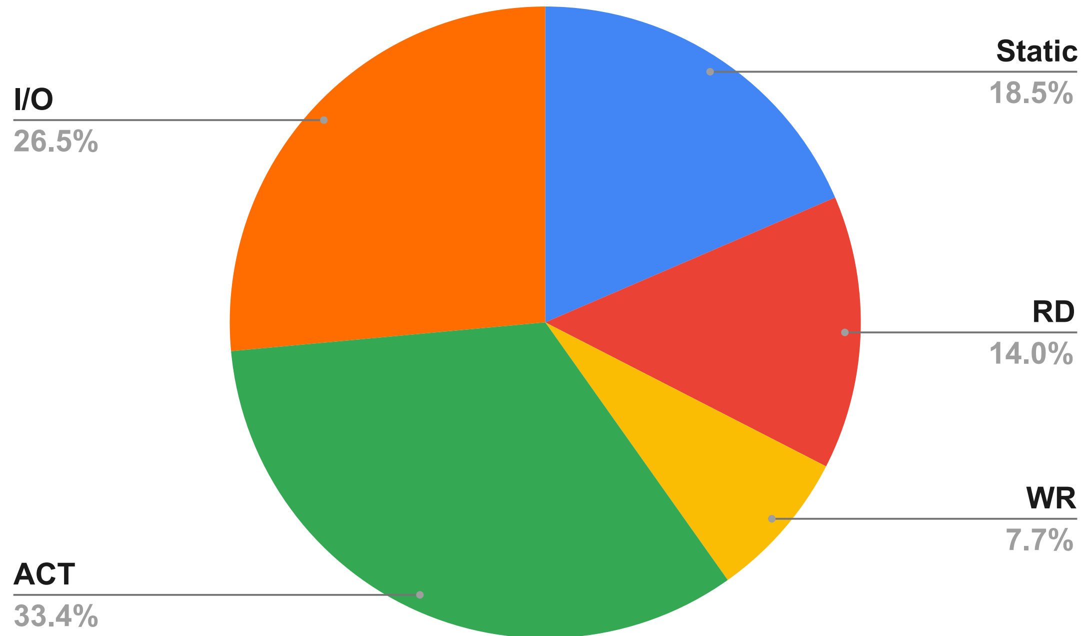
Two methods can be used to calculate the power consumed by the driver sinking and sourcing current (MICRON, 2017). The first is an accurate simulation of the data bus components on SPICE, with a sufficiently long pattern of pseudo-random data. The second, is to calculate the DC power of the driver against termination. This method is simpler and yields smaller results than the worst-case (MICRON, 2017; NXP, 2016). We use the values presented in a technical document presented by a DDR memory interface manufacturer (NXP, 2016). From this document, we get that the source(I_{RD}) and sink(I_{WR}) currents for a DDR4 interface are 10 mA and 11.3 mA, respectively. We can calculate the read and write power consumption on each bus lane by using the following equations:

$$P_R = I_{RD}^2 \cdot R_{bus} = 10\ mA^2 \cdot\ 67\ \Omega = 6.7\ mW \tag{6}$$

$$P_W = I_{WR}^2 \cdot R_{bus} = 11.3\ mA^2 \cdot\ 67\ \Omega = 8.56\ mW \tag{7}$$

$$P_{IO} = P_R + P_W = 15.26\ mW \tag{8}$$

Figure 4.8 – Breakdown of the total memory power consumption by its components.



Now, using these values and the values from Table 4.2 on Equation 4, assuming $D_T = D_A = 8$ and $B_w = 64$:

$$P_M = 85.5 \cdot 8 + 8 \cdot (35.3 + 64.6 + 153.9) + 15.26 \cdot 64 = 3,690\ mW \tag{9}$$

We are interested in evaluating the power drawn from the entire system, independently if the memory module generated the consumption. Figure 4.8 shows a breakdown of the power consumption by the individual components of Equation 4. More than a quarter of the total power is dedicated to I/O, this value is well above the calculated in the Micron reference material (MICRON, 2017). This increase is mostly due to the inclusion of memory controller driver in the P_{IO} calculation.

There are some notable changes when the data copy does not need to happen through the memory bus. First and foremost, due to avoiding moving data off-device, P_{IO} is absent from the Inter-PIM power consumption, which immediately yields a 26.5% decrease in power consumption. Thus we are no longer calculating power consumption for accessing a different device, but two chips in the same module. Also,

the number of active DRAM chips (D_A) is reduced to the number of active threads involved in communication. Thus, we can rearrange Equation 4 as:

$$P_{MP}^{Scaled} = P_{static} \cdot D_T + D_A \cdot (P_{WR} + P_{RD} + P_{ACT}) \tag{10}$$

We can exemplify with a scenario where only two threads are communicating, which improves in 4× the dynamic energy from accessing memory (RD + WR + ACT) . This result is inline with the one found in (AHN et al., 2012), where DRAM chips could be accessed in smaller groups. If we recalculate this partial memory power consumption with these parameters (P_{MP}^2), we arrive at only 1.19 W. This totals a 2.5 W power saving, or a reduction of 3.1× in power dissipation on the DRAM alone.

$$P_{MP}^2 = 85.5 \cdot 8 + 2 \cdot (35.3 + 64.62 + 153.9) = 1,190\ mW \tag{11}$$

This estimate only serves as a gross approximation of a real-world device power consumption, as it is known that DRAM devices power specifications are not reliable (GHOSE et al., 2018). Furthermore, this analysis ignores the consumption of sending DRAM commands through the bus, as well as ignoring the costs of the memory controller internal logic.

4.4 Processor and Cache Energy

Inter-PIM will not save energy only on DRAM data transfers, but on the processor caches and logic as well. As shown in Section 4.1, for the host to access a DIMM-like PIM unit, it must read from the entire PIM device. The host needs to read an entire cache line, access each individual word, shuffle the data, and then write the cache line back to memory, as shown in Figure 4.1.

We can measure the impact this procedure has on the host by implementing a simple test, as shown in Appendix A. The test code loads data (DRAM rows) from the main memory in chunks of 64 bytes. For this device with 8 DRAM chips, each chip contributes with 1 kB of data to the row. By accessing an entire row, 1 kB of data moves from one chip to another. Data is flushed out of the caches before being loaded, this operation is not accounted. We use Streaming SIMD Extensions (SSE) intrinsics to perform the temporal load operations more efficiently,

using non-temporal stores to avoid the caches on the way back to the main memory. The code is instrumented using the host Hardware Performance Counters (HPC) to evaluate performance metrics. The host system used is shown in Table 4.3. With it, we collect hardware metrics to show the impact of data movement inside the host.

Table 4.3 – Host System used in the data movement test.

OS: Ubuntu 18.04.4 LTS
Baseline/Host Intel i5 - 7600 @ 3.5GHz
Cache per Core L1 = 32kB; L2 = 1024kB; Last Level Cache = 6MB;
Main Memory DDR4 1x16GB 2400MHz CL16;

The results from this test are shown in Table 4.4. From the hit/miss counters alone it seems that most of the impact of the data transfer stays located to the L1 cache. There are very few misses on the L1 and L2 caches, even fewer hits on the L2 cache, and no hits or misses in the L3. Non-temporal stores guarantee that the L3 cache is not touched on the writes. Thus we can deduce that the loads are also skipping the L3 cache. This behavior is expected, as this processor contains a non-inclusive cache, that behaves as a victim cache on this application. The L2 hit and miss metrics refer to a cache line, so we can approximate the data requests by multiplying the counter results by 64 bytes, as shown in Table 4.4. We can see that these metrics do not seem to scale with the data.

There are two extra components to cache access, the L1 and L2 prefetcher requests. These can be accessed via the ***PF_REQSTS*** and ***L2_RQSTS_PF_HIT*** counters. *PF_REQSTS* stands for the total prefetch requests originating from the L1 and L2. The *L2_RQSTS_PF_HIT* counter represents the amount of L1 prefetcher requests that hit the L2 cache. With these counters, we can extract the number of requests each prefetcher made. These values are also shown in Table 4.4 scaled for the cache line size (64 bytes).

Table 4.4 – Collected metrics from executing data transfer between DRAM chips with 10^6 repetitions

Data Size	Cycles	L1 hit	L1 Miss	L2 Hit	L2 Miss	L1 Pref.	L2 Pref.	L3 Miss/Hit
8kB	31,229	9,971	320	192	320	11,456	7,552	0
16kB	61,901	19,946	576	256	384	22,080	15,040	0
24kB	96,031	29,921	768	256	576	32,384	22,720	0
32kB	125,366	39,895	832	256	704	44,736	30,272	0
40kB	158,611	50,025	1,216	256	832	53,888	37,824	0
48kB	188,141	59,835	1,216	448	1,024	65,152	45,568	0
56kB	216,274	69,823	1,280	448	1,088	75,328	52,928	0
		Bytes						

By plotting the total cache access for different data transfer sizes (Figure 4.9), we see that the L1 hits and Prefetchers compose most of the data movement in the caches. This total access also scales linearly with the size of the transfer and the amount of cycles spent. This indicates that the L1 and L2 prefetchers identified the access pattern and are bringing data to the cache. Thus, we can deduce that the L2 prefetcher brings data into the L2, which then the L1 prefetcher brings back to the L1.

With this data in hands, we can see that even though misses in L1 and L2 are not counted in the counters, data is being brought in by the prefetchers. Thus, we can create an equation that uses the most significant cache access components for this type of data movement inside the processor:

$$E_{HC} = E_{Hit_L1} \cdot N_{Hit_L1} + E_{PF_L1} \cdot N_{PF_L1} + E_{PF_L2} \cdot N_{PF_L2} \tag{12}$$

Where E_{HC} is the total host data movement energy, E_{Hit_L1} is the energy cost of accessing and moving a word from the L1 to the registers, N_{Hit_L1} is the number of L1 hits. E_{PF_L1} and E_{PF_L2} represent the energy of prefetching (hardware + data movement + cache access) a cache line to the L1 and L2, respectively. N_{PF_L1} and N_{PF_L2} are the number of cache lines prefetched. We can extract the number of accesses from Table 4.4. These energy costs are not documented and experimentation to retrieve them is cumbersome. Thus, we rely on the experiments made by (KESTOR et al., 2013) to extract these values. Their experiments were made for a different

Figure 4.9 – Size of each type of cache access for different data transfers.

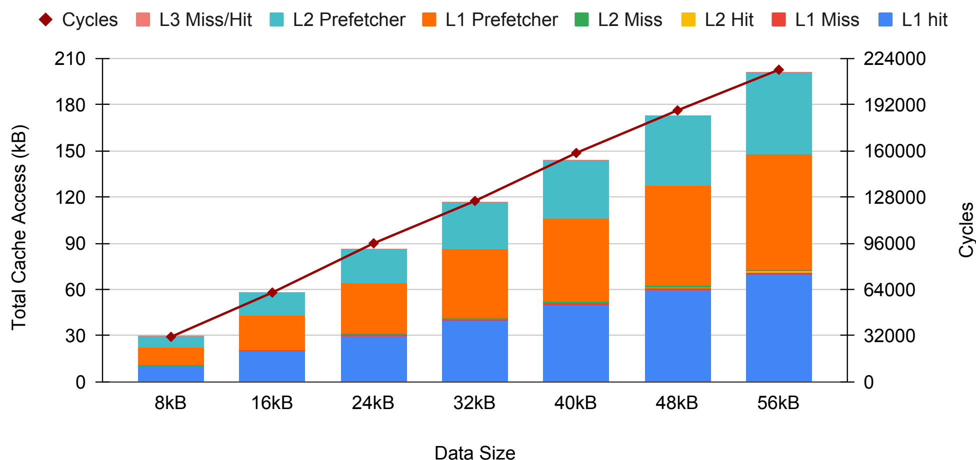


Table 4.5 – Energy consumption for data access and movement. Extracted from (KESTOR et al., 2013).

	Energy (nJ)
Hit_L1	1.11
PF_L1	3.65
PF_L2	11.24

processor, on a different technology and microarchitecture. We consider these values as a first order approximation for the actual energy costs, as shown in Table 4.5.

Using these values with Equation 12 for 1 kB data transfer between DRAM chips (8 kB accessed data), we arrive at a per core energy cost of $\mathbf{13.5}\ \mu J$. This analysis does not take in to account the energy spent on the SIMD units to shuffle data, nor does it take in to account stalled core cycles. This also fails to take in to account other secondary drain of energy in the core. The objective here is to show a lower bound energy cost of a processor core transferring data between PIM units. This extra energy will not be spent if the PIM can transfer data directly. If the host is required to execute other tasks concurrently, the extra memory accesses will pollute the caches, busy the prefetchers, and lower overall efficiency for the host application.

To provide a counterpart upper bound for this analysis, we measure the Model-Specific Registers (MSR) from the Running Average Power Limit (RAPL) interface. These registers provides a clean interface for energy measurements in the CPU package. However, they are not actually measurements, but rather an approximation made on the fly by an on-chip model. While accurate on the CPU measurements, they tend to present energy offsets on the DRAM measurements (DESROCHERS; PARADIS; WEAVER, 2016). By executing the same tests with the RAPL interface, we arrive at an estimated per core energy cost of $\mathbf{73.24}\ \mu J$ for 1 kB of transferred data between DRAM chips. As we made little effort to minimize background processes on the measured core, this estimate acts as an upper bound for a busy system.

Taking the results from Equations 4, 11, and the per core energy cost, the total energy cost for the host system to handle 1kB of inter-thread communication (E_{Comm}) is given as:

$$E_{Comm} = E_{HC} + P_M \cdot \frac{Cycles_{Transfer}}{f} \tag{13}$$

Where we multiply the memory power consumption by the number of cycles over frequency spent transferring data. For $31,299$ Cycles at 3.5 GHz we get

the lower bound communication energy $E^{l}_{Comm} = \mathbf{46.5}\ \mu J$, and the upper bound communication energy $E^{u}_{Comm} = \mathbf{106.24}\ \mu J$. In Section 6.2 we will demonstrate the total energy gains of using Inter-PIM compared to the host for different number of threads and in Sections 6.3 and 6.4 the energy consumption for different applications.

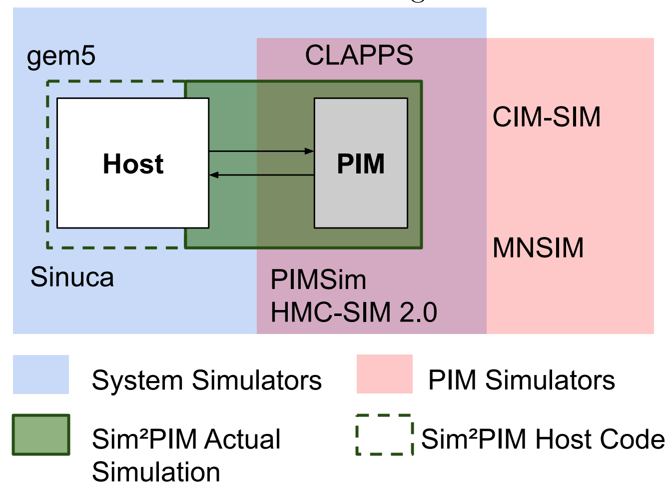
5 BUILDING A SIMULATOR

Not all PIM designs are created equally, and most simulators available can handle only a tiny subset of these (Oliveira et al., 2017; Xia et al., 2018; Leidel; Chen, 2016). The simulation must be aware of the OS and the underlying hardware to handle a multi-thread application in a multi-core environment accurately. Some simulators include the hardware and a virtualized operating system, while others simulate only the PIM device and use the complete host system. Simulators such as gem5 (BINKERT et al., 2011) and SiNUCA (Alves et al., 2015) can simulate entire micro-architectures with an elevated level of accuracy. SiNUCA (Alves et al., 2015) is a trace-based simulator, which uses traces generated on a real machine. The simulator has accurate descriptions of the hardware components down to the processor's pipeline. However, it can not simulate the interactions with the operating system and other processes. The simulator also suffers from the flaws of other trace-based simulators in that the benchmark can not interact with simulated hardware, as the traces have already been collected.

Researchers have used the gem5 (BINKERT et al., 2011) simulator to evaluate a set of applications under different hardware configurations, making this setup perfect for hardware-software co-design. The simulator is divided into several independent modules, coupled and decoupled to test different combinations. However, this modularity and broad configuration options create a notoriously steep learning curve for using the gem5 environment. While the code maintainers strive to improve usability, testing disruptive new hardware such as PIM units on the simulator can prove a hurdle, even for simplistic experiments.

A faster alternative is to use PinTools (LUK et al., 2005). Utilizing trace files containing cycles, memory access, and data as input for basic processor and memory hierarchy models, acting much like SiNUCA, the tool can interpret each issued instruction. PinTools's huge instrumentation overheads prohibit direct code measurements and gem5's extensive simulation times and barrier of entry. None of these simulators can handle threaded applications with native system calls. Baremetal simulators as (Alves et al., 2015) can not simulate OS-level thread scheduling and system calls, while gem5 based simulators still face long simulation times and added virtualization overheads.

Figure 5.1 – Simulators scope when considering system integration. Many more examples exist in all categories.



However, even with limited support from simulators, multi-thread applications are a majority in high-performance computing applications. Thus, the need arose for simulators capable of handling multiple memory stacks at the host and PIM sides. Developed explicitly for this purpose, MultiPIM (Yu; Liu; Khan, 2021), based on two other simulators (SANCHEZ; KOZYRAKIS, 2013; Kim; Yang; Mutlu, 2016), can simulate a multi-stacked-memory PIM device. The simulator offloads *POSIX* and *OpenMP* threads, mapping them to the PIM hardware, maintaining coherence between cores, and a PIM-side task scheduler. It can therefore handle multi-thread applications on the PIM side. However, the simulator utilizes Intel's PinTool (LUK et al., 2005) based instruction feeding mechanism, which interprets each instruction in the virtual environment at run-time. The program must then simulate all the metrics in a virtual environment, bearing a long simulation time. Figure 5.1 summarizes the current academic simulation ecosystem. Current PIM architecture simulators are either architecture-specific (Leidel; Chen, 2016) or rely on system simulators (Oliveira et al., 2017; BINKERT et al., 2011; Alves et al., 2015) or other tools with a heavy overhead (XU et al., 2018). Finally, simulators for newer technologies are incomplete, as they do not try to simulate a fully connected system (Xia et al., 2018), and again rely on tools presenting a heavy overhead (BANAGOZAR et al., 2019). PIM's current simulation ecosystem lacks a low-overhead solution capable of providing system integration with coherence and code offloading mechanisms, together with virtual memory capabilities, which does not rely on full-fledged system simulators.

5.1 Sim^{2}PIM Framework

The main focus of development during this dissertation, Sim^{2}PIM (SANTOS; FORLIN; CARRO, 2021c) evolved from a single-thread PIM simulator, to a full-fledged multi-thread capable PIM simulation and instrumentation framework. The Sim^{2}PIM framework presents a high accuracy, low overhead, low execution time, and quick to implement simulation. These qualities place the Sim^{2}PIM framework in stark contrast to other current simulation methodologies like full-system (BINKERT et al., 2011; Alves et al., 2015) and trace-based simulators (Alves et al., 2015; XU et al., 2018; Oliveira et al., 2017; BANAGOZAR et al., 2019).

With Sim^{2}PIM, the application is integrated and controlled by the framework, in an inversion of control, to become a single executable that can run natively on the host terminal. This allows Sim^{2}PIM to use host hardware for executing host instructions and the PIM-simulator for PIM instructions. Thus, Sim^{2}PIM execution consists of a mix of instrumented host code and simulated PIM hardware, where the entire stack of the host is available to the application and can be used natively (e.g., OS, libraries, drivers). Moreover, the framework is designed with multi-threaded code execution in mind, allowing for parallel applications to actually access multi-core hardware and software resources. This support is built around the *pthread* library, allowing for the library's complete functionality, including synchronization capabilities. Therefore, Sim^{2}PIM does not have to replicate or emulate this functionality. Other current simulators presented in the literature must simulate performance metrics, while Sim^{2}PIM delivers the host's HPC as the most accurate baseline possible for real hardware. The simulator makes smart use of the host HPC to integrate code instrumentation directly with application code. This allows Sim^{2}PIM to add functionality to the application environment and control even fine-grained PIM interaction with host hardware. Furthermore, by running natively, Sim^{2}PIM makes use of the OS, with its libraries, kernel calls and any other native element. This integration between native application and the PIM can be added automatically by the instrumentation tool, or manually by invoking Sim^{2}PIM as an API. The Sim^{2}PIM framework provides:

- **Hardware Prototyping Flexibility** - The PIM-simulator modularity allows the developer to deal with PIM hardware and its design independently from the application and instrumentation. The framework allows the designer to

experiment different PIM designs and their interaction with different host resources, including multiple cores and multiple processors.

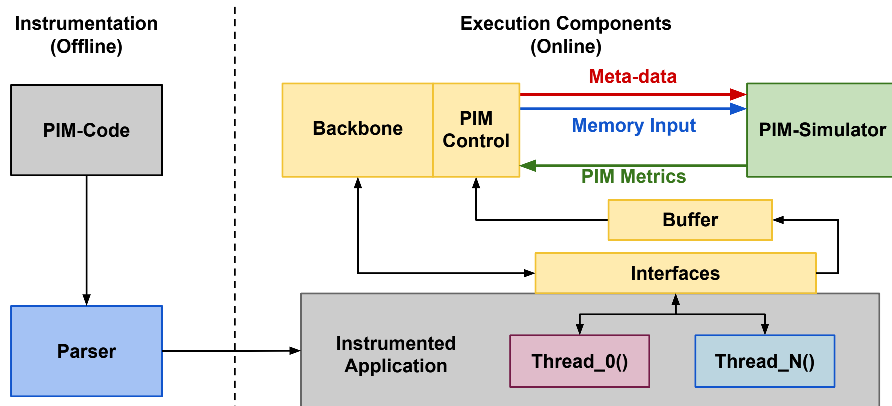
- **Fast PIM Prototyping** - The framework provides a flexible abstraction level, allowing the developer to decide the simulation level of detail during development, including hardware description language, PIM technology, and architecture. Since host integration is guaranteed, this leads to a fast implementation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

Sim^{2}PIM minimizes overheads when not simulating PIM instructions. As will be shown in Section 5.8, it achieves execution speeds similar to performance profiling tools such as *perf* on host code, with as little as 10% run-time overhead and less than 2% metrics difference for most applications. Additionally, utilizing the host hardware and OS resources allows Sim^{2}PIM to simulate multiple PIM threads concurrently, exploring natural parallelism for the tested applications, achieving more than 8$\times$ simulation speedup compared to a sequential simulation and orders of magnitude compared to other simulators.

The Sim^{2}PIM framework operation is composed of two different phases, offline instrumentation and online execution. The offline phase contains the instrumentation parser, responsible for inserting instructions in the PIM's application assembly code. This is accomplished with a low, and more importantly, known overhead, as is shown in Section 5.2. The online phase is composed of several functionally separate modules.

It integrates the offline phase output, the backbone, and the PIM-simulator interface. An overview of the modules is shown in Figure 5.2. We also present Sim²PIM call graphs in Appendix B, in Figures B.1, B.2, B.3, and B.4.

Figure 5.2 – Overview of Sim²PIM modular components and execution phases.



Sim²PIM strives in software-hardware co-design by not hampering software development and providing a clean interface for any level of PIM simulation. For example, if the PIM-simulator is implemented as a timing-aware functional simulation written in *C* language, the use flow of Sim²PIM is as follows:

1. Generate PIM code assembly (e.g., using a PIM compiler (AHMED et al., 2019)).
2. Parse the assembly with the instrumentation tool (in blue Figure 5.2).
3. Compile the Sim²PIM backbone (in yellow Figure 5.2).
4. Compile the PIM-simulator (in green Figure 5.2).
5. Link the instrumented assembly with the backbone and PIM-simulator.
6. Execute the binary.

If the application software needs to be rewritten, only the instrumented assembly must be generated again. When a hardware detail must be changed, the PIM-simulator can be modified and re-linked to the rest of the framework. For the scenario described, Sim²PIM usage does not differ from compiling a program with an ordinary compiler to run on the terminal, as the entire simulation becomes a binary. Sim²PIM supports *pthread*, as it is the lowest level API for multi-threading. It could also be extended to other multi-processing paradigms.

5.2 Instrumentation

Sim^{2}PIM leverages a static instrumentation tool, namely a parser. The role of the parser is to replace the original explicit PIM instruction, code annotated section, or block call, with a representative code offloading method according to the PIM code offloading design (SANTOS; FORLIN; CARRO, 2021a; SANTOS et al., 2019; ALVES et al., 2016; NIDER et al., 2021; LEE et al., 2021). This arrangement allows for the user to determine and measure the impact of the experimented code offloading. It also replaces *pthread*, with the Sim^{2}PIM interface API to allow simulation support for multi-threaded applications.

Lines 3 and 11, in Listings 5.1 and 5.2 respectively, exemplify a parsed PIM instruction, adopting a simple *store* as code offloading method. The *PIM_LOAD* in this operation represents the original PIM instruction/macro/basic block. This register is then saved to a memory position (e.g., a memory-mapped PIM unit), which the simulator knows and has access to. Like Pin (LUK et al., 2005), the parser saves and restores registers according to calling conventions and adjusts stack pointers to avoid overlapping data addresses in the stack, as illustrated in Listing 5.2.

Listing 5.1 –

Original x86+PIM Code Snippet - Annotated or Compiled

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

Contrary to Pin's dynamic instrumentation, the instrumentation is static, so it can be done beforehand, avoiding Pin's severe Just-In-Time (JIT) execution overheads (LUK et al., 2005; XU et al., 2018). Due to the modular nature of the framework and the application being instrumented separately, Sim^{2}PIM avoids linkage errors. The interface API calls can be inserted automatically by the parser or manually by the programmer in the original *C* code. For basic tests inserting the interface API directly might prove faster. However, as the application code grows

larger, it becomes easier and more reliable to use the parser. The currently available interfaces represent the most basic functionality of Sim²PIM.

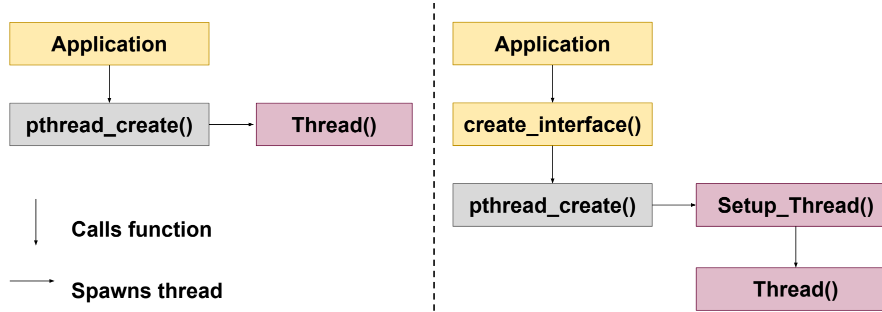
Listing 5.2 –
Parsed x86+PIM Code Snippet

```
movq %rax, %r14
. . . . . .
subq $120, %rsp ;Adjusting Stack Pointer
pushq %rax ;Saving registers
pushq %rbx ;Saving registers
pushq %rcx ;Saving registers
pushq %rdx ;Saving registers
pushq %rdi ;Saving registers
pushq %rsi ;Saving registers
. . . . . .
movq PIM_LOAD_OPCODE, %rcx ;read PIM instruction as a string
movq %rcx, (GLOBAL_VAR_PIM_INST) ;PIM Instruction is emitted
    to the simulator
leaq 32512(%rbx, %rax ), %rcx ;PIM memory access calculation
    in case of LOAD/STORE
movq %rcx, (GLOBAL_VAR_PIM_INST_ADDR) ;PIM memory access
    address is emitted to the simulator
callq PIM_interface ;PIM interface call (Section 5.2)
. . . . . .
popq %rsi;Recovering registers
popq %rdi;Recovering registers
popq %rdx;Recovering registers
popq %rcx;Recovering registers
popq %rbx;Recovering registers
popq %rax;Recovering registers
addq $120, %rsp ;Adjusting Stack Pointer
. . . . . .
addq $2048, %rax
```

5.3 Interfaces

The role of the interfaces is to add functionality to the application code with the least amount of interference as possible. This is accomplished in two ways: first,

Figure 5.3 – Creation of threads before instrumentation (left) and after (right).



code and memory accesses inside the interfaces are kept to a minimum, avoiding too much interference with the caches. Second, as will be discussed in Section 5.4.1, the interfaces efficiently use the HPC to avoid measuring their overhead.

PIM_interface: This interface is inserted right after PIM instruction offloads or annotated PIM blocks. The PIM_interface serves as the output of the application environment. It contains the logic required to retrieve the PIM instructions and memory access addresses. This information is offloaded from the application to the backbone through a non-blocking software FIFO buffer, discussed in Section 5.4.3. The only scenario where this interface presents blocking behavior happens when the host and PIM need to synchronize, discussed in Section 5.6.

create_interface: We encapsulate the original *pthread_create* calls in application code by directly changing the function call, with the same inputs. As will be shown in Figure 5.5, this wrapper dynamically allocates a new physical core for the thread during the execution, and then measures the *pthread_create* with the performance counters. It substitutes the thread function with a dedicated thread launcher function, which is responsible for executing the performance counter setup for the new thread before it is launched (Setup_Thread), as shown in Figure 5.3.

join_interface: Much like the *create_interface*, this function encapsulates calls to *pthread_join* function. The *pthread_join* function is a blocking interface that awaits the end of the issued thread. Due to its blocking behavior, measuring the performance counters when the thread stays blocked is pointless, as simulated and executed metrics are distinct. Thus, this wrapper's role is to avoid measuring the blocking behavior, as will be shown in Figure 5.5.

5.4 Backbone

Sim^{2}PIM online phase tries to isolate the application, backbone, and PIM-simulation in different physical cores for higher simulation speed and precision. Not only does this provide more accurate metrics for the application isolated from the simulation environment, but it also allows for any application threads to actually execute in parallel. The backbone contains Sim^{2}PIM entry-point. It is responsible for generating the multi-thread infrastructure in which the entire simulation will execute. There are inherent advantages to using multiple threads in the simulation environment. Since the application, the backbone, and the PIM-simulator are all threads in the same program, they can easily share the same memory space. Thus it is trivial for the PIM to operate over the target application data as if both PIM and data were physically on the same memory device. This allows simulating PIMs that reside on the system's main memory or accelerators in a specific memory device (by adjusting the data movement overheads).

Additionally, the host's cache hierarchy guarantees cache coherence for the simulation data. The process described in (SANTOS; FORLIN; CARRO, 2021a) can be used to implement the coherence and virtual memory support in unmodified hardware, or the simulator could be coupled with other methods of cache coherence (BOROUMAND; GHOSE, 2016). The backbone is responsible for several housekeeping tasks and interfaces, including environment setup, application thread management, instruction and data buffers, and PIM-simulator and application interfaces. It also contains the low-overhead assembly functions responsible for calling the HPC, and the output functions responsible for delivering the final metrics.

5.4.1 Precise measurements

The HPC are overflow counters. This means one must acquire the difference between two consecutive measures instead of an absolute value. Sim^{2}PIM makes use of inline functions to call the *rdpmc* instructions directly. These instructions take as input the configured HPC register and an output register. Multiple threads can share a core, so the return values of the *rdpmc* instructions are stored in a per-thread data structure. The usage of the counters is straightforward. At the beginning of a backend code segment, the counter values are collected with a **STOP**

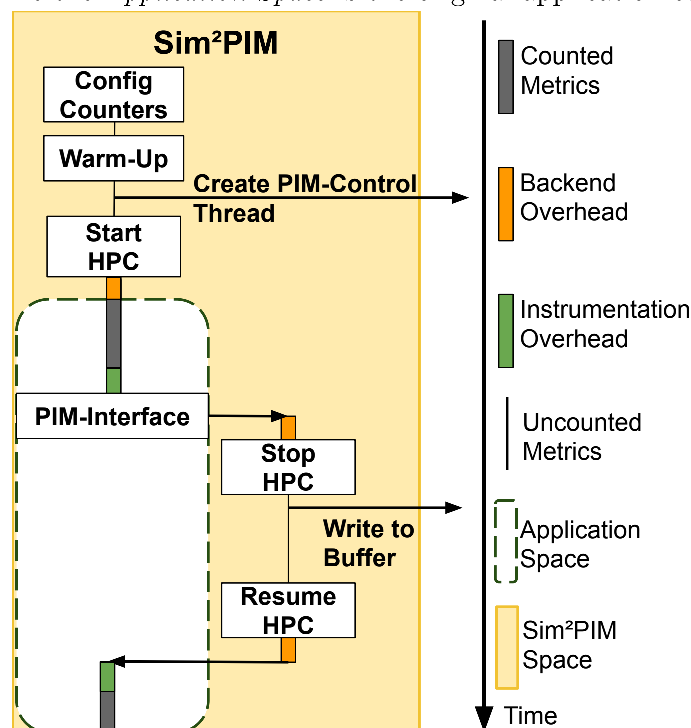
COUNTERS call. When the backend code ends, the counter values are collected with a **RESUME COUNTERS** call. The values subtracted from subsequent resume-stop calls represent the measured code, the *application space*. Backend code effectively runs in a blind spot of the measurement functions, which we call the *Sim^{2}PIM space*.

There are multiple HPC available for any given host, such as *retired instructions*, *unhalted cycles*, *cache misses/hits*, among others. Sim^{2}PIM avoids over-engineering a solution to this multitude of possibilities by providing a clean, uniform interface through a user-defined configuration file. The counters themselves can be configured beforehand for any host architecture.

5.4.2 Environment Setup

Sim^{2}PIM flow for a single-threaded application occurs as in Figure 5.4. We can see that the *application space* (dashed lines) contains the original application code, while the rest of the framework resides in *Sim^{2}PIM space*. The following steps are executed in this part of execution:

Figure 5.4 – Interfaces and overheads of offloading data from the application to PIM-simulation. The functionality encapsulated by *Sim^{2}PIM space* is the executable, while the *Application Space* is the original application code.



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

WARM-UP: Different hosts and compilers may result in different measurements and generated code. Considering the relevance of precise metrics, this module executes several back-to-back HPC calls to collect the instrumentation overhead. These overheads include the serialization instructions inserted on each HPC call (as is shown in Section 5.2) and the instruction overhead required to read the HPCs. In Figure 5.4 the measured overheads for instrumentation are represented before and after the PIM-interface (green in Figure 5.4), and the HPC calls overheads before each counter invocation (orange in Figure 5.4). The values extracted here are subtracted after each call in *Sim²PIM space*.

CREATE ENVIRONMENT: Sim²PIM tries to isolate the simulation from the user application to provide more accuracy for the hardware counters by reserving a physical core for the PIM-Control and another one for the PIM-simulator. The rest of the host's cores remain free for the user's application. This approach reduces the interference in the data and instruction caches, hence allowing more precise measurements.

5.4.3 Communication Buffer

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

When a single-thread application interacts with the PIM-simulator, the instruction offload throughput from the host might not be enough to keep the PIM-simulator fully occupied. However, as more threads offload to the PIM-simulator,

there will be contention on the input-side and a bottleneck on the output. This can happen because one core is responsible for simulating PIM instructions delivered from many cores. Although this is an exclusive simulation bottleneck, it may cause contention on the host side, artificially reducing the PIM instruction offload throughput and the host's native instruction execution.

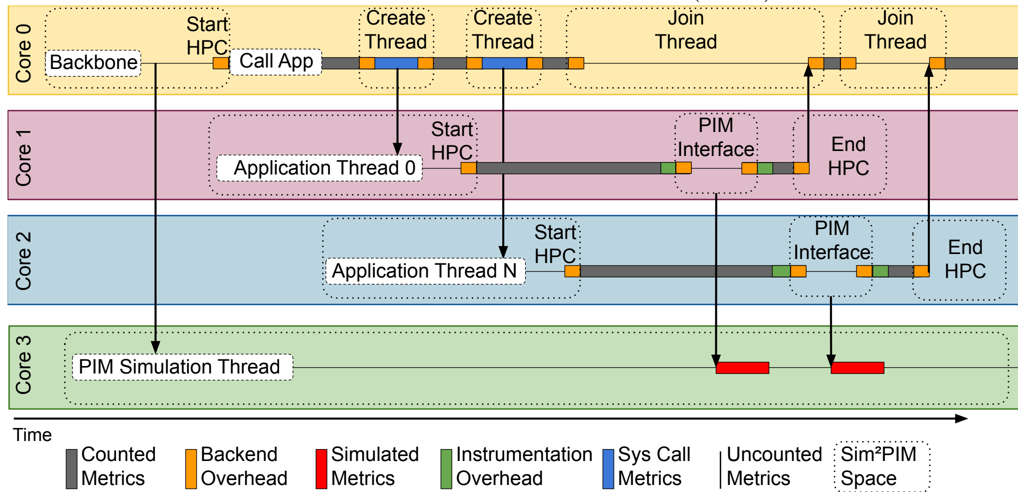
To mitigate this effect in the multi-threaded architecture of Sim^{2}PIM, the PIM-interface writes a data structure to a non-blocking FIFO buffer. Each thread has its row in the buffer, so they do not contend between themselves. The data structure written contains instructions, data addresses, and any other meta-data Sim^{2}PIM has access to, including current performance counter values. The non-blocking buffer is implemented with atomic primitives to avoid costly system calls. On the other side of the buffer, the PIM-Control retrieves the data structure from all the threads and offloads them to the PIM-simulator as needed.

As shown in Figure 5.4, when the application code calls the PIM-interface, the HPCs are sampled, and the code is now in *Sim^{2}PIM space*. Thus, it can interact with the instruction buffer without the risk of interfering with host metrics. When the PIM-interface is done with the buffer, the counters are resumed, and the code returns to *application space*.

5.4.4 PIM-Control Interface

The PIM-Control Interface is responsible for reading the data buffer, invoking the PIM-simulator, and collecting its metrics. It is worth noting that any PIM-simulator that fulfills the interface requirements can be coupled to this interface, especially those that take trace-files as inputs (XU et al., 2018; Oliveira et al., 2017; Alves et al., 2015; BANAGOZAR et al., 2019; Xia et al., 2018; Leidel; Chen, 2016). As shown in Figure 5.2, the PIM-Control Interface feeds three types of input to the PIM-simulator, meta-data, PIM instruction, and data addresses. Meta-data outputs consist of those originating from the instrumentation. These can include any metrics, including the cycle in which the PIM instruction was issued or the number of cache misses. The instruction and address outputs are the data that would typically exit the host processor core and go through the memory bus to the PIM.

Figure 5.5 – Overhead diagram for a multi-thread application on the Sim^2PIM with the Hardware Performance Counters (HPC).



5.5 Application Thread Management

As shown in Figure 5.5, the application itself is a function to be called in the backbone between instrumentation sections. For the simulation of a multi-thread application, each thread will perform its backend calls. As each core has its performance counters, we must ensure that the threads do not migrate cores, which would result in a wrong calculation of the metrics.

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

As shown in Section 5.3 *pthread_create()* and *pthread_join()* functions are replaced by the interface functions, *create_interface* and *join_interface* respectively. These interfaces wrap around the *pthread* call functionality, adding the capability of setting the core affinity, marking the physical core as in-use (if using the strategy mentioned above **2)**), and acquiring metrics before the start of the thread function itself. Inside the *create_interface* function, the only measurement made is around the original *pthread_create()* call. We make sure to measure the thread's launch, as

this can pose a significant overhead in multi-thread applications, shown in Figure 5.5 as *Sys Call Metrics*.

Each thread counts its metrics, and we assume all the threads are alive simultaneously. For all metrics, core/thread-wise metric counts are available. Only the largest value is considered to the final count for the total program *elapsed-cycles* and *elapsed-time* metrics, as it is the bottleneck for program conclusion. The interface function around *pthread_join()* guarantees the waiting time for the simulations to end is not counted on the benchmark's main thread metrics, as shown in Figure 5.5.

5.6 Thread Synchronization

When using *pthreads*, the programmer primarily handles synchronization between threads, with mutexes, semaphores, and other atomic operations. This behavior does not change on Sim^{2}PIM, as the host memory caches are still used for shared memory space between host threads. Synchronization between PIM-threads still happens at the host side, as the PIM depends on the host for memory accesses and instruction offloading.

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

5.7 PIM-Simulator

As described in Section 5.4.4, the PIM-simulator module is fed from the PIM-Control interface. PIM instruction and data addresses are used to trigger specific operations (e.g., PIM arithmetic, PIM memory access), while the meta-data involves metrics that might be useful to the simulation, such as the cycle in which

the PIM instruction was emitted. This module can take the complexity level the designer needs, from a cycle-accurate to an instruction-level look-up table. Moreover, due to the modular nature of the Sim^{2}PIM, for this module, any language can be adopted, as well as connected to different tools (e.g., specialized memory or PIM simulators (BANAGOZAR et al., 2019)). Therefore, the designer can use hardware description languages (e.g., SystemVerilog, SystemC, VHDL) with an inter process communication, or high-level abstract languages (e.g., C, C++, Python), where the C languages could be integrated as simple function calls.

5.8 Validating the Simulator

As shown in Figure 5.1, most simulators focus on PIM architecture experimentation. However, they lack connections between actual hosts and simulated architectures. Thus, their metrics need to be *virtualized*, as no real hardware is in play. Furthermore, this behavior prevents the utilization of existing host resources, such as multiple cores, integration with the memory system, OS support, and the HPC. While these features can be simulated, they make implementation more complex, more costly, and less accurate if not done carefully.

Table 5.1 – Baselines and Case Study PIM Parameters

Baseline/Host Intel i5-7600 @ 3.5GHz; Cache per Core L1 = 32kB; L2 = 256kB; Last Level Cache = 6MB; Main Memory DDR4 1x16GB 2400MHz CL18;
Baseline/Host Intel Xeon Silver-4214 @ 2.2GHz; Cache per Core L1 = 32kB; L2 = 1024kB; Last Level Cache = 16MB; Main Memory DDR4 2x32GB 2400MHz CL16;
Baseline/Host AMD R5-1600 @ 3.2GHz; Cache per Core L1 = 32kB; L2 = 512kB; Last Level Cache = 8MB; Main Memory DDR4 2x8GB 2666MHz CL16;
RVU Processing Logic (SANTOS et al., 2017; SANTOS; FORLIN; CARRO, 2021a) Operation frequency: 1 GHz; Up to 32x 64 functional units (integer + floating-point); Vector sizes (bytes): 32x 256, 16x 512, 8x 1024, 4x 2048, 2x 4096, 1x 8192 Latency (cycles): 1-alu, 3-mul. and 20-div. int. units; Latency (cycles): 5-alu, 5-mul. and 20-div. fp. units; Register bank: 8 sets of 32 composable registers of 256 bytes each;

This Section evaluates the Sim^{2}PIM framework, showing the benefits of simulating only the design of interest. To assess the framework, we implemented a common PIM approach that implements FUs within a 3D-stacked memory (Hybrid Memory Cube Consortium, 2013; LEE et al., 2021; SANTOS et al., 2017). The case study is based on the RVU architecture (SANTOS et al., 2017) and (SANTOS; FORLIN; CARRO, 2021a), and Table 5.1 summarizes the hosts' systems and the case study PIM parameters.

5.8.1 Overhead Evaluation

Two main types of overheads are inserted in the application code by Sim^{2}PIM: *PIM_interface* insertion and the *create_interface*. The former interface is inserted before each PIM instruction, while the last one replaced the original *pthread_create()* in case of a multi-threaded application. As mentioned in Section 5.4.2, the warm-up phase is required to remove the overheads from the HPC and PIM_interface calls. To exemplify this, we collected these overheads for two different host processors for the instructions and cycles metrics. These overheads are directly dependent on the host processor, as illustrated in Table 5.2.

Table 5.2 – Average overheads for two different HPCs, unhalted cycles and retired instructions. Measured with 10,000 repetitions in the warm-up phase of two different processors.

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

In the case of multi-threaded applications, the multi-thread overheads happen in the *create_interface* function, as aforementioned in Section 5.5. To evaluate the impact of these overheads on PIM multi-thread simulation, we compared the same set of applications executing with and without the simulator. We selected the well-established *perf* as an easy to deploy, low-overhead, and high-accuracy performance profiling tool for the comparison. We used some of the algorithms on the PolyBench benchmark suite (POUCHET, 2012). The results are shown in Table 5.3.

For both the single thread and multi-thread results, we can see that for most benchmarks, Sim^{2}PIM and *perf* show similar results, with a trend towards more

Table 5.3 – Simulated Cycles vs. Simulation time for Sim^{2}PIM and *perf* on the AMD processor. Values represent a single thread and the average of 4 threads.

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

minor results in Sim^{2}PIM. We leverage this trend is due to the instrumentation provided by Sim^{2}PIM to be more accurate due to the use of hard-coded HPC, not depending on slower system calls. The cycles metric is influenced by several factors, including the congestion of the memory subsystem and frequency fluctuations. Thus some applications may present more variation than others. Splitting the application between threads might also affect this behavior, increasing the traffic in the processor caches. We can also see the execution time collected with the *time* command for Sim^{2}PIM and *perf* itself. Although Sim^{2}PIM adds execution time, this effect is smaller as the execution gets longer. We can see that Sim^{2}PIM's performance for host code is very competitive, especially if we consider the usual run-times of other simulators.

5.8.2 Simulation Time Evaluation

Sim^{2}PIM merits lay on top of its high simulation speeds, high-accuracy host hardware metrics, and the backbone's structure high modularity. These characteristics make Sim^{2}PIM especially suited to evaluate the interactions between host hardware and the PIM device, including the system's memory hierarchy and technology. We set out to showcase the speedup of simulating multiple threads in Sim^{2}PIMs truly multi-core environment. We test the multi-thread PIM simulation on a simple embarrassingly parallel kernel that performs the vector sum (*vecsum*) over 64MB data, varying from 1 to 8 perfectly balanced threads. This way, we avoid complications dealing with complex vector algorithms and inter-thread communication.

In Figure 5.6 the bars represent the number of simulated cycles for the application with varying numbers of active threads. To evaluate the effectiveness of isolating simulation and application threads in different physical cores, the lines in Figure 5.6 represent the execution speed (ms) of three different Sim^{2}PIM configurations: a single-core execution, a dual-core execution (simulator + application), and

the standard Sim^{2}PIM with dedicated cores. The *Sim^{2}PIM 1-Core* line represents the single-core execution, meaning the entire framework and the application threads share a single core. While the OS can dynamically schedule them, most applications effectively run sequentially with the PIM-simulator. This is similar to many other simulators (BINKERT et al., 2011; XU et al., 2018) that do not support parallel simulation execution readily. The results clarify that forcing the PIM-simulator and backbone to share a core with the application severely harms the framework performance. This is directly proportional to the number of threads disputing for core time, which results in an increased number of context switches between the application threads and between framework and application threads.

Thus, when we remove the framework from this dispute (by placing it in an exclusive core), as shown in line *Sim^{2}PIM 2-Cores* in Figure 5.6, there is a significant release in pressure for the application core. For this example, the *vecsum* application is lightweight and straightforward on the host side, there is still contention between application threads disputing the same core, but it is significantly smaller. Finally, we achieve the most efficient execution when we execute Sim^{2}PIM with dedicated cores (line *Sim^{2}PIM N-Cores* in Figure 5.6) for each application thread and the framework. This way, the application parallelism works in favor of the framework, as long as there are enough free cores to operate in all threads in parallel. This impact might be more meaningful for larger applications on the host side when deciding how to test the application.

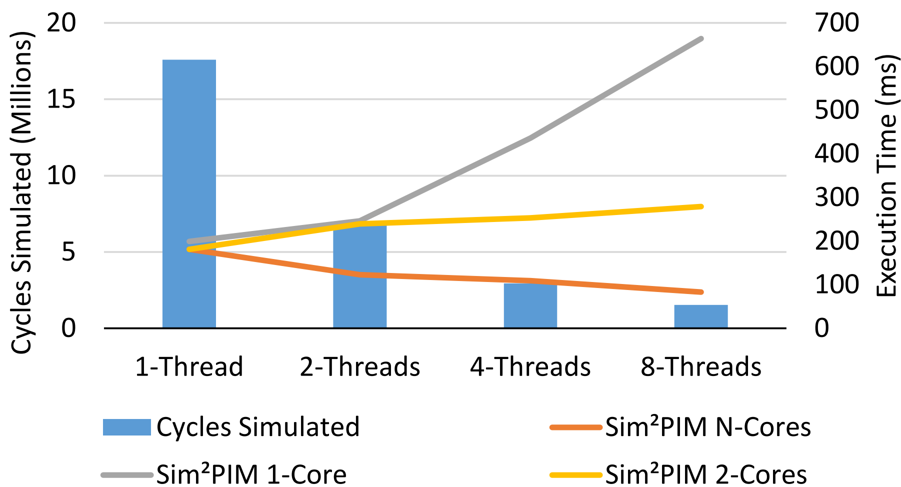


Figure 5.6 – Simulated Cycles and Simulation Time for a 64MB *vecsum* application offloaded by the Xeon CPU to the PIM device using three different simulation configurations.

Our experience with simulators like Gem5 (BINKERT et al., 2011) is that although they can simulate very different architectures and enable the design of new architectures, the designer is forced to simulate the host architecture. If the designer desires to couple the PIM with a different host, there is a need to implement the new host and its features. Quickly changing a software or hardware parameter and rerunning the simulation is not an option as the simulation can take hours, even in the most straightforward modes. Tools based on Intel's Pin (LUK et al., 2005) offer very fine control over the code currently executing, allowing the user to follow branches and see accessed virtual memory addresses without recompiling code. However, the JIT model of instrumentation and execution makes the simulation speed slow and makes quick testing a nuisance.

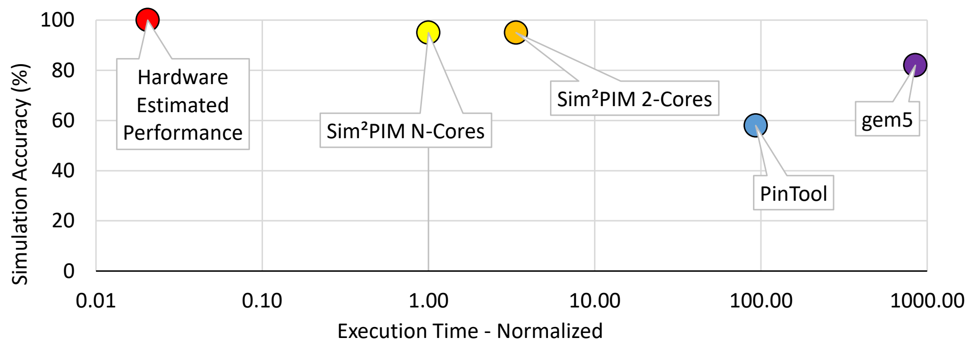


Figure 5.7 – Execution time and accuracy for a *vecsum* application with eight threads in different simulators.

Besides, trace-based simulators, which require application traces, suffer from another challenge: changes in application code require the traces to be reacquired. Figure 5.7 presents a comparison between our previous experiments with these tools showing the accuracy of the metrics concerning simulation estimates of the actual hardware performance (SANTOS et al., 2017; SANTOS; FORLIN; CARRO, 2021a) (y-axis). The simulation speeds (x-axis) are normalized to the run-time of the execution speed of Sim^{2}PIM. In case of a lack of available cores, executing applications serially in Sim^{2}PIM as shown in line *Sim^{2}PIM 2-Cores* in Figure 5.6, slows down the simulation and prevents us from evaluating the interaction of these multiple requests on shared resources, such as memory access bandwidth. Even then, Sim^{2}PIM is orders of magnitudes faster than the other base simulation options, either trace-based or full-system simulators.

6 EVALUATING COMMUNICATION STRATEGIES

There are a lot of variables when it comes to multi-threaded applications executing on a PIM device integrated with an unmodified host processor. The integration between PIM and host, the interaction between threads sharing hardware resources, the data movement in the memory hierarchy, among many other subtle details (e.g., virtual memory paging). These characteristics are inherently linked to how the data parallelism in the application is exploited, and how well the algorithm maps to available hardware resources. Thus, the best communication strategy is very much dependant on the hardware system and software requirements. As discussions on optimal programming paradigms for different algorithms are well beyond the scope of this dissertation, we focus the evaluation on data transfer corner cases.

6.1 Experiment Setup

The parameters for the host processor, memory system details, and the adopted PIM device are shown in Table 6.1. The application was compiled with a PIM compiler (AHMED et al., 2019) and integrated with Sim^{2}PIM (SANTOS; FORLIN; CARRO, 2021c) to simulate the PIM device coupled to a general-purpose system. We implemented the PIM device discussed in Section 3.2, which is capable of handling 8 concurrent threads as it has 8 independent PIM units. In these tests we do not evaluate the efficiency of increasing the number of threads past the number of chips, as this incurs in a different set of optimizations. Each PIM thread is coupled to an exclusive host core for all the tests, and the threads do not migrate between PIM units. This coupling guarantees that the PIM can achieve maximum data transfer bandwidth when multiple requests coincide. It mirrors the HBM-PIM (LEE et al., 2021), where multiple host threads are needed to achieve the desired efficiency. For this experiment, the operations occur over 1024 bytes. Hence, in a memory module comprising eight memory devices (Figure 4.4), each PIM unit can process over the entire local row buffer per operation. The PIM units are also independent of each other.

Each application is tested with our Inter-PIM model and through the host. The Inter-PIM model was implemented within Sim^{2}PIM, which considers the host's memory access costs, and the hardware estimations described in Table 6.1. Host

Table 6.1 – Baselines and PIM Parameters

OS: Ubuntu 18.04.4 LTS **Baseline/Host Intel Xeon Silver-4114** @ 2.2GHz Cache per Core L1 = 32kB; L2 = 1024kB; Last Level Cache = 16MB; Main Memory DDR4 2x32GB 2400MHz CL16;
PIM Processing Logic based on (SANTOS et al., 2017; SANTOS; FORLIN; CARRO, 2021b) @ 300 MHz; Vector Operand size (bytes): 1024 Latency Int/FP. (cycles): 1/5-alu, 3/5-mul. Register bank: 8 sets of 1024 bytes each;
Inter-PIM Mechanism @ 2 GHz; Network-on-Chip - 3x3 8 bit; Corner-to-Corner Latency: 1.3 ns; Input Buffer, Output Buffer: 8 positions; Estimated Power: Saturated Traffic = 0.55W - 50% Traffic = 0.17W Estimated Area: 0.87mm^2@45nm;

communication was measured directly from the host system's access to and from memory, using hardware performance counters. We calculate communication energy consumption separately with the methods described in Section 4, as the nature of Sim^{2}PIM prohibits its integration with the RAPL interface[1].

6.2 Energy Efficiency

In Section 4.4, we calculated the energy costs for the memory module configuration and for a host processor handling communication. As we use a different processor in the following tests, the upper and lower bound energy costs are recalculated. The DRAM power parameters remain the same, requiring recalculation of the frequency and cycles values, as shown in Equation 13. The E_{HC} upper bound must be recalculated with the RAPL interface (96.67 μJ). Collecting these metrics for the tested processor ($48,450$ Cycles at 2.2 GHz), we get that the lower bound communication energy is $E^{l}_{Comm} = \mathbf{94.76}\ \mu J$, and the upper bound communication energy is $E^{u}_{Comm} = \mathbf{177.93}\ \mu J$. We use the mean value between these two scenarios as a fair representation of cost, where the host caches are not the only element active in the transfer, but we also leverage that the host is also executing other tasks concurrently with the transfer, increasing its efficiency. Therefore, we use $E_{HC} = 136.35\ \mu J$ as the energy cost to move 1 kB of data between PIM threads.

[1]The RAPL interface uses register files and system calls, which are too costly for the fine-grained integration that Sim^{2}PIM requires.

For the energy costs of the Inter-PIM mechanism, we scale the worst-case 64 bits transmission delay from corner to corner (Section 4.2.4) to the transmission of 1 kB. We get 166 ns total transmission time in the Inter-PIM NoC. For the DRAM access times, in this case the CAS latency of 12 ns for a DDR4-2666 device, we get a latency of 12,288 ns for 1 kB of data. Inter-PIM power consumption is dependent on the network saturation, so we scale consumption linearly from 50% traffic to 100% saturation ranging from 2 to 8 threads (P_{IP}^{Scaled}). As discussed in Section 4.3, DRAM power consumption with Inter-PIM access scales with the number of active threads, as shown in Equation 10. We calculate the energy consumption of the Inter-PIM to move 1 kB of data between threads as:

$$E_{IP} = P_{IP}^{Scaled} \cdot 166\ ns + P_{MP}^{Scaled} \cdot 12,288\ ns \tag{14}$$

Table 6.2 shows these values for a different number of active threads. Comparing the energy costs for a single 1 kB data transfer between PIM threads yields a difference of **9.29**× in energy consumption. It is important to note that although there is a large difference in energy costs, the cost for Inter-PIM transfers is individual, while the host can perform all transfers with the same cost (i.e., broadcast). So when one of the PIM threads uses a broadcast, Inter-PIM must write to 7 DRAM chips, yielding a gain of **4.08**×.

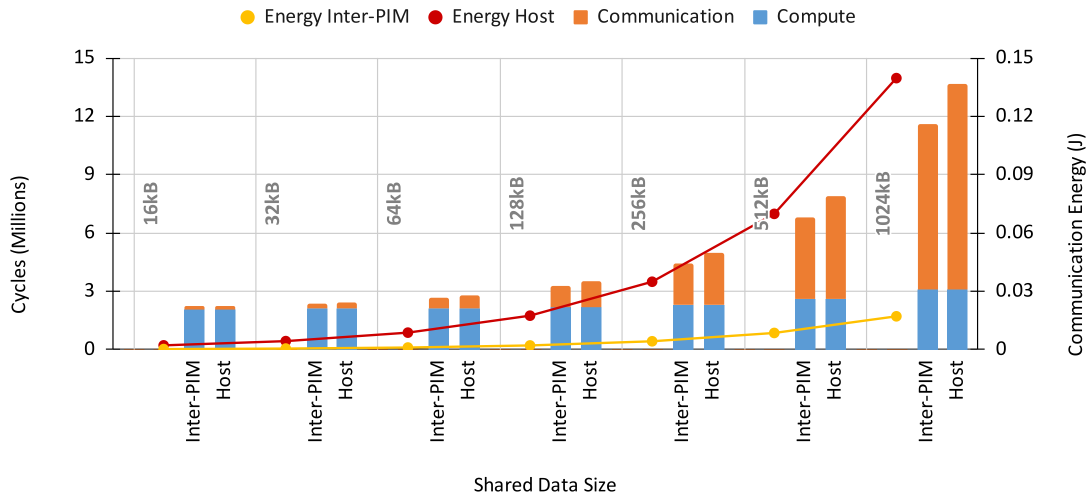
Table 6.2 – DRAM, Inter-PIM, and Host energy and power results scaled according to the number of threads.

Threads	P_{MP}^{Scaled} **(W)**	P_{IP}^{Scaled} **(W)**	**Inter-PIM** (μJ)	**Host** (μJ)	**Difference** (μJ)	**Energy Gains**
2	1.19	0.17	14.67	136.35	121.68	9.29 ×
3	1.45	0.23	17.80	136.35	118.55	7.66 ×
4	1.70	0.30	20.93	136.35	115.42	6.51 ×
5	1.95	0.36	24.06	136.35	112.29	5.67 ×
6	2.21	0.42	27.19	136.35	109.16	5.02 ×
7	2.46	0.49	30.32	136.35	106.03	4.50 ×
8	2.71	0.55	33.45	136.35	102.90	4.08 ×

6.3 Communication Patterns

As corner cases, we selected two communication patterns (broadcast and multicast) with threads arranged into producer and consumer threads (a sending and a receiving end). All tests execute the same PIM algorithm, as the focus here is to observe the impact of the communication on total run-time. The computation is divided into an individual workload that every thread executes and a unique

Figure 6.1 – Energy and Cycle results for 8 threads of vecsum kernel with a broadcast access pattern.



workload that only the consumer thread runs. The first part of the computation (**C1**) is a simple vector sum of 3 arrays, each of 8MB each, totaling 24MB per PIM thread. The second part (**C2**) is an accumulation of the previous result with the received data. So the computation time is expected to increase as the data size increases. For all evaluations, Inter-PIM communication performance is estimated on the simulator and host performance is measured. Energy values are derived from the results found in Section 6.2.

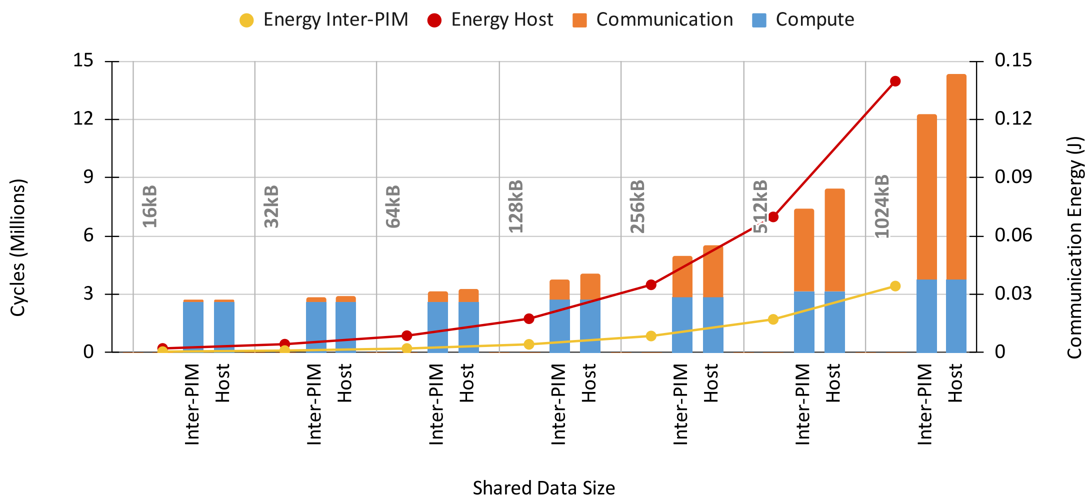
Broadcast - One-to-All communication: In this application, all 8 threads execute computation **C1**. After the end of the first execution block, all threads except the one executing the broadcast lock, changing their status to **3** (Figure 4.5) and waiting for data. This thread sets its status to **4**, and the communication happens according to the tested mechanism (host or Inter-PIM). After communication is complete, all threads except the communicating thread compute **C2** on the received data. As we can see in Figure 6.1, the energy improvement scales with the size of the data transfer. As discussed in Section 3.4 for host communication, the broadcast operation can never achieve 100% efficiency. This results in the communication through the host performing reads in PIM units that are not necessary. Inter-PIM can be precise in reading data from the correct PIM unit, avoiding energy waste.

Multicast - Multiple One-to-one communications:

This application divides threads into two groups, even and odd threads. Each pair of odd and even threads operates similarly to a 2-thread broadcast application. If shared data is aligned in memory between all threads, the host can optimize data access by synchronizing data transfers. Figure 6.2 demonstrates the performance

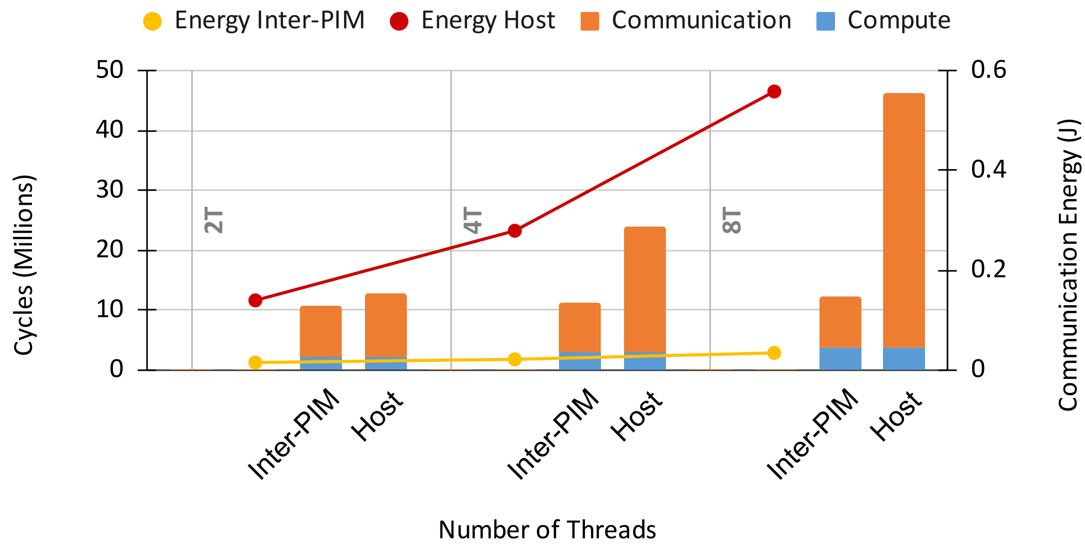
and energy consumption for the solutions in this operation. In this scenario, host energy consumption and performance are identical to that of a broadcast operation, as it performs the same amount of reads and writes. Inter-PIM performs more reads compared to the broadcast operation. Thus its energy consumption increases, but as they happen in parallel as well, they do not affect performance.

Figure 6.2 – Energy and Cycle results for 8 threads of vecsum kernel with a multicast access pattern.



In both communication patterns, performance between host and Inter-PIM communication vary by a small amount, 20% in average. This is reasonable, as the main source of latency from memory access on the host is the DRAM itself (LI; REDDY; JACOB, 2018; CHATTERJEE et al., 2012), where CAS latency limits the physical speed at which memory cells can be accessed. In this manner, Inter-PIM is still mostly limited by the memory access latency. However, there can still be large performance gains in some applications. In a scenario where shared data is not aligned in memory, the host will not be able to use chip parallelism to its advantage. It will have to perform multiple accesses to transfer data between threads. This can also occur if threads operate on vastly different kernels or data sizes, not permitting that transfer-synchronization occurs. As shown in Figure 6.3, Inter-PIM will not suffer from this problem, as it can handle independent concurrent accesses. Besides the performance inefficiency, the host also suffers with excess energy consumption.

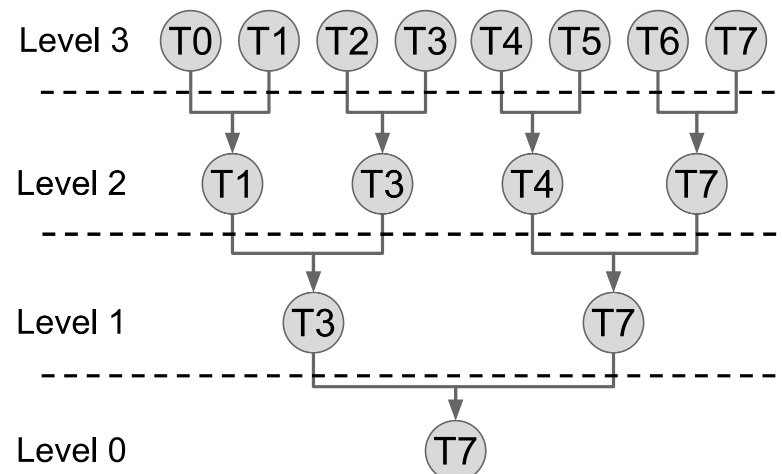
Figure 6.3 – Energy and Cycles for 1MB of non-aligned shared data movement between threads.



6.4 Experimenting with a Complex Application

In more complex applications, the boundary between computation and communication overheads may vary drastically. Thus the performance impact of communication will also vary. We implemented an application that mimics a filter operation on the PIM with multiple *GEMM*-kernel executions to measure this impact. The threads are arranged in a tree-like dependency, where each level operates on 1MB matrices and forwards the resulting 1MB matrix to the next level, as shown in Figure 6.4.

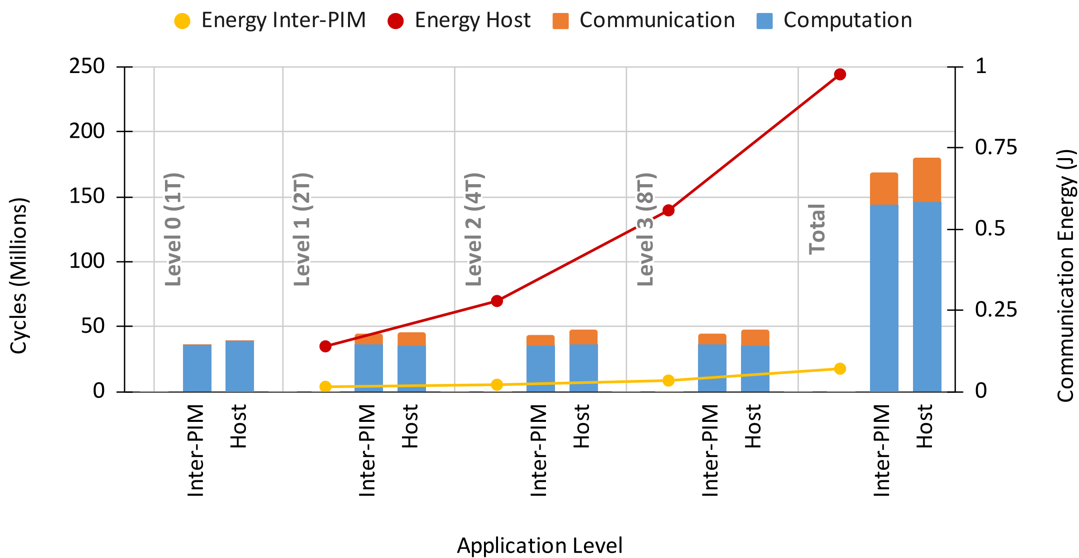
Figure 6.4 – Shared data movement in the *GEMM* application.



Even though the application is essentially compute-bound when the data is present on the correct PIM units, it can still impact performance due to the extensive data sizes being processed. This pattern can be imagined as if the filtering kernels are fixed in each PIM unit, and the resulting data is sent to the next set of threads. The first level has eight threads, the second has four, the third has two, and the last only has one thread. The communication pattern for each level is identical to the *multicast* pattern described above. The resulting application spends most of its time computing and less with data transfers.

As can be seen in Figure 6.5, this results in a minor speedup relative to the total run-time, around 5% incremental improvement. However, the communication energy impact is still significant, reducing from 0.97 J via the host to 0.07 J via the inter-PIM. Regardless of the benchmark used, the key point is to maintain data inside the memory module and perform precise accesses to the PIM units that require them, by removing the host from supervising the communication and replacing it with an efficient hardware implementation.

Figure 6.5 – Execution cycles and Energy for *GEMM* kernels



As discussed in Section 4.4, this will reduce energy consumption by increasing memory access efficiency, by not using host resources, and avoiding cache pollution.

7 CONCLUSIONS

It is more apparent than ever that the architectural community must propose new paradigms to keep up with computing trends. Processing-in-Memory (PIM) is a clear contender to take the top spot of energy efficiency and acceleration. However, there must be leaps in architectural designs and the support environment to accelerate this development, including simulators. The Sim^{2}PIM framework brings a fast and accurate simulation tool for single and multi-threaded applications to the hands of researchers and designers. Sim^{2}PIM makes few compromises, guaranteeing fast simulation speeds and high accuracy, as long as the host hardware is available for testing. Furthermore, the developed simulation framework presents new opportunities to evaluate hardware-software co-design in PIM applications.

This work also demonstrated that using the host as a data relay between the threads in different PIM units can impact the application's performance and energy efficiency. This impact is directly related to how the application orchestrates thread communication, the frequency of this communication, and the transferred data size. We exposed the hardware requirements for a dedicated communication mechanism between PIM units that can mitigate this impact. Inter-PIM provides performance improvements proportional to the amount of data and complexity of the communication involved, and it also significantly lowers the energy footprint of communication between PIM units.

Of course, there are still a lot of improvements that could be made in the work presented here. Although the power and energy models presented are based on previous works and fit in estimates for the evaluated hardwares, they still should be validated by themselves. The performance evaluation of Inter-PIM also leaves a lot of complexity out, including the impact of real-world communication patterns in the NoC, and the impacts of parallelism and pipelining in the memory accesses. Perhaps more critically, a more complete and relevant benchmark suite could help validate the need for Inter-PIM in real world scenarios.

7.1 Future Work

As future work on Inter-PIM, we aim to investigate how the reduction of host cache pollution affects complex applications that divide the load between PIM

and host. The arguments in Section 4.2.2 also beg the question of how would a Network-on-Chip (NoC) inside the memory module change the dynamic of memory access. By following a closer implementation to that of HMC, we might discover that the mechanism can even benefit regular memory operations (AHN et al., 2012).

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

REFERENCES

AGA, S. et al. Compute caches. In: **2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2017. p. 481–492.

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

AHN, J. et al. A scalable processing-in-memory accelerator for parallel graph processing. In: **2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2015. p. 105–117.

AHN, J. et al. A scalable processing-in-memory accelerator for parallel graph processing. In: IEEE. **Int. Symp. on Computer Architecture (ISCA)**. [S.l.], 2015.

AHN, J. et al. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 43, n. 3S, p. 336–348, jun. 2015. ISSN 0163-5964. Disponível em: <https://doi.org/10.1145/2872887.2750385>.

AHN, J. H. et al. Improving system energy efficiency with memory rank subsetting. **ACM Trans. Archit. Code Optim.**, Association for Computing Machinery, New York, NY, USA, v. 9, n. 1, mar 2012. ISSN 1544-3566. Disponível em: <https://doi.org/10.1145/2133382.2133386>.

ALVES, M. A. Z. et al. Large vector extensions inside the hmc. In: **Proceedings of the 2016 Conference on Design, Automation amp; Test in Europe**. San Jose, CA, USA: EDA Consortium, 2016. (DATE '16), p. 1249–1254. ISBN 9783981537062.

Alves, M. A. Z. et al. Sinuca: A validated micro-architecture simulator. In: **2015, 17th Int. Conf. on High Performance Computing and Communications**. [S.l.: s.n.], 2015.

ANDERS, M. A. et al. A 4.1tb/s bisection-bandwidth 560gb/s/w streaming circuit-switched 8×8 mesh network-on-chip in 45nm cmos. In: **2010 IEEE International Solid-State Circuits Conference - (ISSCC)**. [S.l.: s.n.], 2010. p. 110–111.

BANAGOZAR, A. et al. Cim-sim: Computation in memory simulator. In: **22nd Int. Workshop on Software and Compilers for Embedded Systems**. [S.l.: s.n.], 2019. (SCOPES '19).

BINKERT, N. et al. The gem5 simulator. **SIGARCH Comput. Archit. News**, 2011.

BOROUMAND, A.; GHOSE, e. a. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. **IEEE Computer Architecture Letters**, 2016.

BOROUMAND, A.; GHOSE, e. a. Google workloads for consumer devices: Mitigating data movement bottlenecks. In: **Int. Conf. on Architectural Support for Programming Languages and Operating Systems**. [S.l.: s.n.], 2018. (ASPLOS), p. 316–331. ISBN 9781450349116.

CALI, D. S. et al. Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In: **2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2020. p. 951–966.

CHATTERJEE, N. et al. Leveraging heterogeneity in dram main memories to accelerate critical word access. In: **2012 45th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.: s.n.], 2012. p. 13–24.

CHEN, K.; PACHTER, L. Bioinformatics for whole-genome shotgun sequencing of microbial communities. **PLOS Computational Biology**, Public Library of Science, v. 1, n. 2, p. null, 07 2005. Disponível em: <https://doi.org/10.1371/journal.pcbi.0010024>.

CHI, P. et al. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In: **2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2016. p. 27–39.

Chu, C. et al. Pim-prune: Fine-grain dcnn pruning for crossbar-based process-in-memory architecture. In: **2020 57th ACM/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2020. p. 1–6.

CORPORATION, I. **Intel 64 and IA-32 Architectures Optimization Reference Manual**. 2016. Disponível em: <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>.

DENG, Q. et al. Dracc: a dram based accelerator for accurate cnn inference. In: **2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2018. p. 1–6.

DENNARD, R. et al. Design of ion-implanted mosfet's with very small physical dimensions. **IEEE Journal of Solid-State Circuits**, v. 9, n. 5, p. 256–268, 1974.

DESROCHERS, S.; PARADIS, C.; WEAVER, V. M. A validation of dram rapl power measurements. In: **Proceedings of the Second International Symposium on Memory Systems**. New York, NY, USA: Association for Computing Machinery, 2016. (MEMSYS '16), p. 455–470. ISBN 9781450343053. Disponível em: <https://doi.org/10.1145/2989081.2989088>.

DEVAUX, F. The true processing in memory accelerator. In: **2019 IEEE Hot Chips 31 Symposium (HCS)**. [S.l.: s.n.], 2019. p. 1–24.

DEVICES, A. M. **Software Optimization Guide for AMD Family 17h Processors**. 2017. Disponível em: <https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf>.

DREBES, A. et al. Tc-cim: Empowering tensor comprehensions for computing-in-memory. In: **IMPACT 2020 workshop (associated with HIPEAC 2020)**. [S.l.: s.n.], 2020. Informal proceedings.

DRUMOND, M. et al. The mondrian data engine. In: ACM. **Int. Symp. on Computer Architecture**. [S.l.], 2017.

ECKERT, C. et al. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In: **2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2018. p. 383–396.

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

GAO, F.; TZIANTZIOULIS, G.; WENTZLAFF, D. Computedram: In-memory compute using off-the-shelf drams. In: **Int. Symp. on Microarchitecture**. New York, NY, USA: [s.n.], 2019. (MICRO '52). ISBN 9781450369381.

GAO, F.; TZIANTZIOULIS, G.; WENTZLAFF, D. Computedram: In-memory compute using off-the-shelf drams. In: **Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture**. New York, NY, USA: Association for Computing Machinery, 2019. (MICRO '52), p. 100–113. ISBN 9781450369381. Disponível em: <https://doi.org/10.1145/3352460.3358260>.

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

GAO, M. et al. Tetris: Scalable and efficient neural network acceleration with 3d memory. In: . New York, NY, USA: Association for Computing Machinery, 2017. (ASPLOS '17), p. 751–764. ISBN 9781450344654. Disponível em: <https://doi.org/10.1145/3037697.3037702>.

GHOSE, S. et al. What your dram power models are not telling you: Lessons from a detailed experimental study. **Proc. ACM Meas. Anal. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 2, n. 3, dec 2018. Disponível em: <https://doi.org/10.1145/3224419>.

GHOSH, M.; LEE, H.-H. S. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams. In: **40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)**. [S.l.: s.n.], 2007. p. 134–145.

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

HAJINAZAR, N. et al. Simdram: A framework for bit-serial simd processing using dram. In: ______. **Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: Association for Computing Machinery, 2021. p. 329–345. ISBN 9781450383172. Disponível em: <https://doi.org/10.1145/3445814.3446749>.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Fifth Edition: A Quantitative Approach**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 012383872X.

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

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

Jain, S. et al. Computing in memory with spin-transfer torque magnetic ram. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, 2018.

JEDEC. **DDR4 SDRAM specification. JESD79-4C**. 2012. Disponível em: <https://www.jedec.org/standards-documents/docs/jesd79-4a>.

KANG, S. J.; LEE, S. Y.; LEE, K. M. Performance comparison of openmp, mpi, and mapreduce in practical problems. **Advances in Multimedia**, v. 2015, p. 1–9, 2015.

KESTOR, G. et al. Quantifying the energy cost of data movement in scientific applications. In: **2013 IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.: s.n.], 2013. p. 56–65.

KIM, D. et al. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In: **2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2016. p. 380–392.

Kim, Y.; Yang, W.; Mutlu, O. Ramulator: A fast and extensible dram simulator. **IEEE Computer Architecture Letters**, 2016.

KWON, Y.-C. et al. 25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications. In: **2021 IEEE International Solid- State Circuits Conference (ISSCC)**. [S.l.: s.n.], 2021. v. 64, p. 350–352.

LEE, S. et al. Hardware architecture and software stack for PIM based on commercial dram technology : Industrial product. In: **Int. Symp. on Computer Architecture (ISCA)**. [S.l.: s.n.], 2021.

Lee, V. T. et al. Application codesign of near-data processing for similarity search. In: **2018 IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)**. [S.l.: s.n.], 2018.

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

LEISERSON, C. E. et al. There’s plenty of room at the top: What will drive computer performance after moore’s law? **Science**, v. 368, n. 6495, p. eaam9744, 2020.

LI, S. et al. Drisa: A dram-based reconfigurable in-situ accelerator. In: **2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2017. p. 288–301.

LI, S.; REDDY, D.; JACOB, B. A performance amp; power comparison of modern high-speed dram architectures. In: **Proceedings of the International Symposium on Memory Systems**. New York, NY, USA: Association for Computing Machinery, 2018. (MEMSYS '18), p. 341–353. ISBN 9781450364751. Disponível em: <https://doi.org/10.1145/3240302.3240315>.

Li, S. et al. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In: **Design Automation Conference (DAC)**. [S.l.: s.n.].

LIMA, J. a. P. et al. Design space exploration for pim architectures in 3d-stacked memories. In: ACM. **Proceedings of the Computing Frontiers Conference**. [S.l.], 2018.

Liu, J. et al. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In: **2018 51st Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO)**. [S.l.: s.n.], 2018.

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

LUK, C.-K. et al. Pin: Building customized program analysis tools with dynamic instrumentation. In: . [S.l.]: Association for Computing Machinery, 2005.

MATOS, D. d. S. M. **Exploring Hierarchy, Adaptability and 3D in NoCs for the Next Generation of MPSoCs**. Tese (Doutorado) — Programa de Pos Graduação em Computação, UFRGS, Porto Alegre, RS, Brazil, 2014.

MCKEE, S. et al. Experimental implementation of dynamic access ordering. In: **1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences**. [S.l.: s.n.], 1994. v. 1, p. 431–440.

MICRON. **Technical Note: Calculating Memory Power for DDR4 SDRAM**. 2017. Disponível em: <https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf>.

Moore, G. E. Progress in digital integrated electronics. **IEEE Solid-State Circuits Society Newsletter**, 2006.

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

NAIR, R. et al. Active memory cube: A processing-in-memory architecture for exascale systems. **IBM Journal of Research and Development**, IBM, v. 59, 2015.

NGUYEN, H. A. D. et al. A classification of memory-centric computing. **J. Emerg. Technol. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 16, n. 2, jan. 2020. ISSN 1550-4832. Disponível em: <https://doi.org/10.1145/3365837>.

NIDER, J. et al. A case study of processing-in-memory in off-the-shelf systems. In: **Annual Technical Conference (USENIX ATC 21)**. [S.l.: s.n.], 2021. ISBN 978-1-939133-23-6.

NXP. **Hardware and Layout Design Considerations for DDR4 SDRAM Memory Interfaces**. 2016. Disponível em: <https://www.nxp.com/docs/en/application-note/AN2582.pdf>.

Oliveira, G. F. et al. A generic processing in memory cycle accurate simulator under hybrid memory cube architecture. In: **Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)**. [S.l.: s.n.], 2017.

PASRICHA, S.; DUTT, N. **On-Chip Communication Architectures: System on Chip Interconnect**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 012373892X.

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

POUCHET, L.-N. Polybench: The polyhedral benchmark suite. **URL: http://www.cs.ucla.edu/pouchet/software/polybench**, 2012. Acessado em 10/02/2022.

SANCHEZ, D.; KOZYRAKIS, C. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In: **Int. Symp. on Computer Architecture**. [S.l.: s.n.], 2013.

SANTOS, P. C. **Improving Efficiency of General Purpose Computer Systems by adopting Processing-in-Memory Architecture**. Tese (Doutorado) — Programa de Pos Graduação em Computação, UFRGS, Porto Alegre, RS, Brazil, 2019.

SANTOS, P. C. et al. Exploring cache size and core count tradeoffs in systems with reduced memory access latency. In: IEEE. **Int. Conf. Parallel, Distributed, and Network-Based Processing (PDP)**. [S.l.], 2016.

SANTOS, P. C.; FORLIN, B. E.; CARRO, L. Providing plug n' play for processing-in-memory accelerators. In: **Asia and South Pacific Design Automation Conference (ASPDAC)**. [S.l.: s.n.], 2021.

SANTOS, P. C.; FORLIN, B. E.; CARRO, L. Providing plug n' play for processing-in-memory accelerators. In: **2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)**. [S.l.: s.n.], 2021. p. 651–656.

SANTOS, P. C.; FORLIN, B. E.; CARRO, L. Sim²pim: A fast method for simulating host independent pim agnostic designs. In: . [S.l.: s.n.], 2021. (DATE '21).

SANTOS, P. C. et al. Solving datapath issues on near-data accelerators. In: **IFIP WG10.2 Working Conference: Int. Embedded Systems Symp. (IESS)**. [S.l.: s.n.], 2019. (IESS '19).

SANTOS, P. C. et al. Operand size reconfiguration for big data processing in memory. In: IEEE. **Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2017.

SCHALLER, R. R. Moore's law: past, present and future. **IEEE Spectrum**, 1997.

SESHADRI, V. et al. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In: **2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2013. p. 185–197.

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

SHAFIEE, A. et al. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In: **2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2016. p. 14–26.

SHAHAB, A. et al. Farewell my shared llc! a case for private die-stacked dram caches for servers. In: **2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.: s.n.], 2018. p. 559–572.

SONG, L. et al. Pipelayer: A pipelined reram-based accelerator for deep learning. In: **2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2017. p. 541–552.

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

TAYLOR, M. B. Is dark silicon useful? harnessing the four horesemen of the coming dark silicon apocalypse. In: **Design Automation Conference**. [S.l.: s.n.], 2012.

Xia, L. et al. Mnsim: Simulation platform for memristor-based neuromorphic computing system. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, 2018.

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

Xie, L. et al. Scouting logic: A novel memristor-based logic design for resistive computing. In: **2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2017. p. 176–181.

XIN, X.; ZHANG, Y.; YANG, J. Elp2im: Efficient and low power bitwise operation processing in dram. In: **2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2020. p. 303–314.

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

Yu, C.; Liu, S.; Khan, S. Multipim: A detailed and configurable multi-stack processing-in-memory simulator. **IEEE Computer Architecture Letters**, 2021.

Yu, J. et al. Memristive devices for computation-in-memory. In: **2018 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2018. p. 1646–1651.

ZHANG, D. et al. Top-pim: Throughput-oriented programmable processing in memory. In: **Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing**. New York, NY, USA: Association for Computing Machinery, 2014. (HPDC '14), p. 85–98. ISBN 9781450327497. Disponível em: <https://doi.org/10.1145/2600212.2600213>.

ZHANG, D. et al. Top-pim: throughput-oriented programmable processing in memory. In: ACM. **Int. Symp. on High-performance Parallel and Distributed Computing**. [S.l.], 2014.

APPENDIX A — HOST TRANSMISSION CODE

This is the code for the host transmission test.

```
1 #define _GNU_SOURCE
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <stdint.h>
7 #include <sched.h>
8 #include <immintrin.h>
9 #include "HPCounters.h"
10
11 #define CACHELINE_SIZE 64
12 #define ROW_SIZE 8192*1
13
14 /**
15  * @brief Flush pointer *p cache line
16  *
17  * @param p
18  */
19 void flush(void* p) {
20     asm volatile ("clflush (%0)\n"
21         :
22         : "c" (p)
23         : "rax");
24 }
25
26 /**
27  * @brief Print 2 64 bit variables from 128 bit vector
28  *
29  * @param var
30  */
31 void print128_num(__m128i var)
32 {
33     int64_t v64val[2];
34     memcpy(v64val, &var, sizeof(v64val));
35     printf("%.16lx %.16lx\n", v64val[1], v64val[0]);
36 }
```

```
37
38 /**
39  * @brief Flush the content in address [start+backward, start+
       forward) from cache
40  *
41  * @param start
42  * @param backward
43  * @param forward
44  * @modify
45  */
46 void flushAll(void* start, int backward, int forward){
47     for(int i= backward; i< forward; i+=CACHELINE_SIZE){
48         void * address = start + (i);
49         flush(address);
50     }
51 }
52
53 void LoadStoreTest(){
54
55     // Malloc Page aligned region
56     __m128i * page_start;
57     posix_memalign((void *)&page_start,4096,ROW_SIZE);
58
59     // Init data
60     for (size_t i = 0; i < ROW_SIZE/16; i = i + 16)
61     {
62         __m128i index;
63
64         // create vector with indexes with 2 64bit values
65         index = _mm_setr_epi64((__m64)i, (__m64)i+1);
66         _mm_store_si128(page_start+i, index);
67     }
68
69
70     // Flush all data
71     flushAll((void *)page_start, 0, ROW_SIZE);
72     mfence();
73     int repeat_times = 100000;
74
75     // Repeat test
76     for(int i=0; i<repeat_times; i++){
```

```
77
78          __m128i shuffle_buffer [4];
79
80          HPCStart(); // start performance counters
81
82          for (size_t j = 0; j < ROW_SIZE; j = j + CACHELINE_SIZE)
83          {
84              // Load 64 bytes
85              for (size_t k = 0; k < 4; k++)
86                  shuffle_buffer[k] = _mm_loadu_si128(page_start+j
    /16+k);
87
88              // Shuffle data
89              __m128i shuffled = _mm_castpd_si128(_mm_shuffle_pd(
    _mm_castsi128_pd(shuffle_buffer[0]),_mm_castsi128_pd(
    shuffle_buffer[2]),3));
90
91              // Overwrite stored data
92              shuffle_buffer[2] = shuffled;
93
94              // Store 64 bytes
95              for (size_t k = 0; k < 4; k++)
96                  _mm_stream_si128(page_start + j/16+k,
    shuffle_buffer[k]);
97
98              // Flush NT-buffers
99              _mm_sfence();
100
101         }
102
103         HPCStop(); // stop performance counters
104         HPCAccum();
105
106         //Make sure data is flushed
107         flushAll((void *)page_start, 0, ROW_SIZE);
108         mfence();
109
110     }
111
112     printf("Counter total == %ld \n", HPC_measure.accum/
    repeat_times);
```

```
113     free(page_start);
114     return;
115 }
116
117
118 int main(int argc, char *argv[]){
119
120     // Select Performance Counter
121     int counter = 0;
122     if(argc > 1){
123         counter = atoi(argv[1]);
124     }
125
126     //set affanity so that the program will run on one core
127     cpu_set_t cpu_set;
128     CPU_ZERO(&cpu_set);
129     CPU_SET(0, &cpu_set);
130     if (sched_setaffinity(0, sizeof(cpu_set), &cpu_set) < 0) {
131         printf("set affanity error!");
132         exit(EXIT_FAILURE);
133     }
134
135     // Init Counter
136     InitCounters(counter);
137
138     // Run test
139     LoadStoreTest();
140
141     return 0;
142 }
```

Listing A.1 – Host data transfer utilizing SSE instructions.

APPENDIX B — SIM²PIM

Sim²PIM graphs generated with Doxygen documentation.

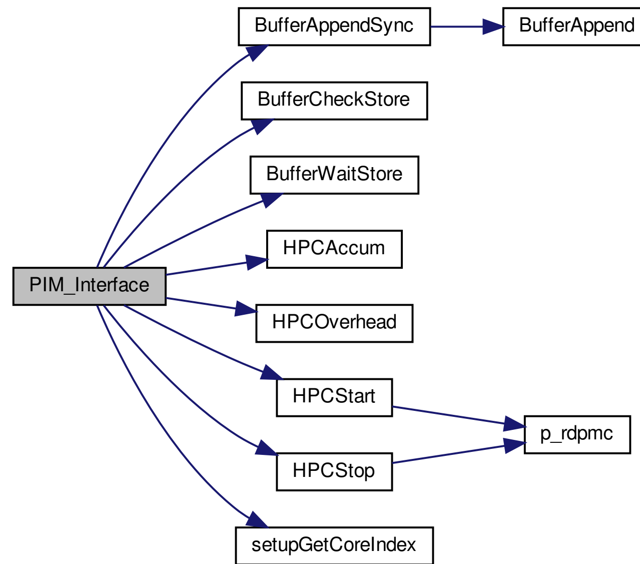


Figure B.1 – Sim²PIM PIM_interface() call graph.

Figure B.2 – Sim²PIM join_interface() call graph.

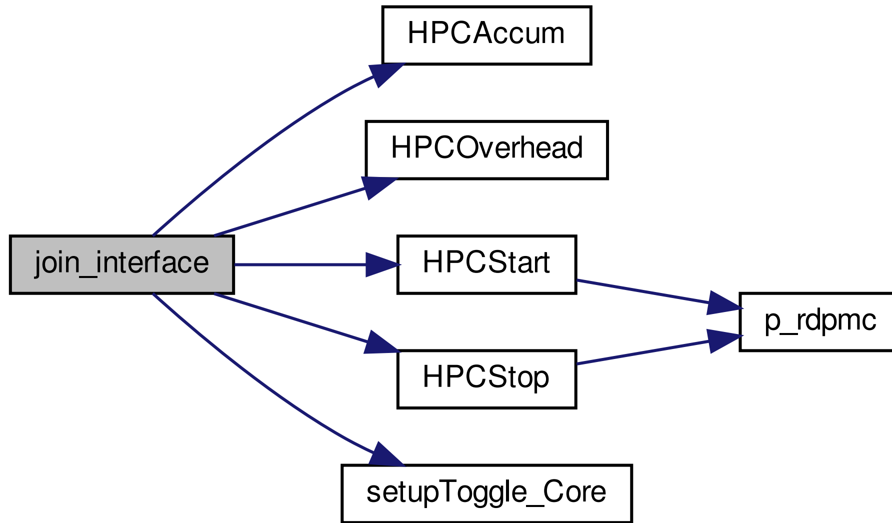
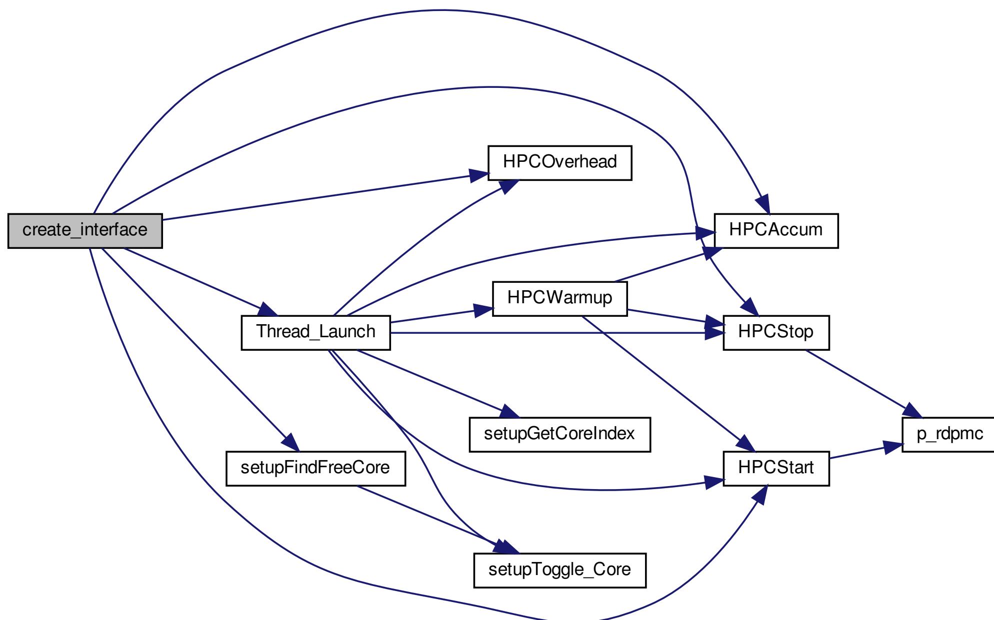


Figure B.3 – Sim²PIM create_interface() call graph.



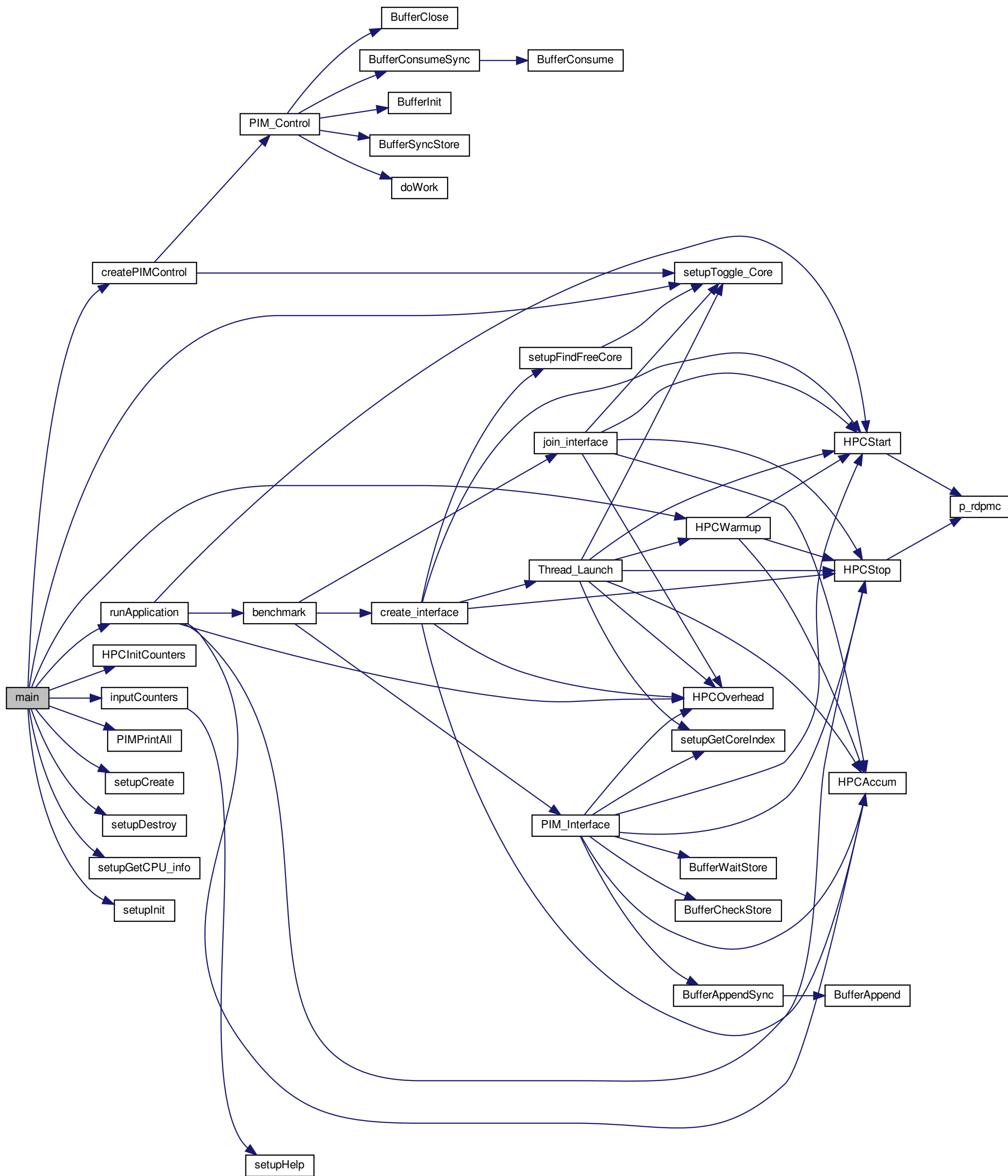


Figure B.4 – Sim^{2}PIM main() call graph.

APPENDIX C — RESUMO EXPANDIDO

Processamento em memória (PIM), com a ajuda de modernas tecnologias de integração, emergiu como uma solução prática para o *memory wall* enquanto melhora a performance e efciciência energética de aplicações contemporâneas. Novas tecnologias de memória juntamente com o surgimento de técnicas de integração 3D proveram os meios para computar dados na memória. Seja explorando as capacidades analógicas ou integrando lógica e memória. Dispositivos PIM tem o objetivo de explorar toda a banda da memória, usando o paralelismo de dados das aplicações de diferentes formas. Com modelos de programação genéricos, e dispositivos de *hardware* que podem ser acessados independentes, é natural que programadores tentem explorar paralelismo a nível de *thread*.

Com *threads* de propósito geral, estruturas de dados compartilhados inevitavelmente surgem, as quais devem ser lidadas corretamente para garantir consistência na memória. Independentemente da maneira como essa consistência é mantida, dados devem ser transmitidos entre diferentes regiões de memória. Os atuais dispositivos comerciais PIM ignoram esse aspectos em seus designs e deixam a transferencia de dados à cargo do processador. Enviando dados através do *bus* de memória para as *caches*, onde eles serão rearranjados e enviados de volta para a memória. Nós argumentamos que esse processo vai contra os princípios de *design* PIM, aumentando os movimentos de dados entre o PIM e o processador. Nós demonstramos essa ineficiência analiticamente e experimentalmente, desenvolvendo um modelo de consumo de potência que consegue extrair limites superiores e inferiores para a comunicação via o processador. Dependendo do processador usado, retransmitir dados através das *caches* pode custar 3 $\times$ mais energia, salientando os altos custos energéticos em usar o processador para esta tarefa.

Para rodar corretamente esses experimentos, nós precisamos executar *benchmarks* muito integrados com o processador, enquanto extraímos suas métricas. Existe uma falta de ferramentas capazes de rapidamente simular diferentes designs PIM e suas integrações com múltiplos processadores *multi-core*. Logo, essa dissertação apresenta *Sim*^{2}*PIM* um simples simulador para dispositivos PIM que integra qualquer arquitetura PIM com um processador *multi-core* e a hierarquia de memória. Analisando casos de compartilhamento de dados, esse trabalho mostra que essa comunicação, se executada pelo processador, pode minar os benefícios de dispositivos

PIM. Nós usamos esse simulador para demonstrar que se o dispositivo PIM depende do processador para compartilhamento de dados, o custo de comunicação entre *threads* escala mais rápido com o tamanho dos dados do que o custo da computação, em alguns casos podendo custar 86% do tempo total de execução.

C.1 Contribuições e Objetivos Alcançados

É mais aparente do que nunca que a comunidade de arquitetura de sistemas deve propôr novos paradigmas para acompanhar as tendências computacionais. *PIM* está na disputa para se tornar um dos principais meios de aumentar a aceleração e eficiência energética. Contudo, devem ocorrer pulos largos nos *designs* arquiteturais e nos ambientes de suporte para acelerar esse desenvolvimento, inclusive com simuladores. O *framework* Sim^{2}PIM traz uma ferramenta de simulação precisa e veloz para aplicações *single* e *multi-thread* para as mãos de pesquisadores e desenvolvedores. Sim^{2}PIM permite novas oportunidades de análise, incluindo estratégias de *co-design* para *hardware* e *software*.

Nós propomos uma solução interna para o PIM que reduz os custos de performance e energia de compartilhamento de dados, mantendo a comunicação dentro do módulo de memória. Esse mecanismo pode acessar unidades PIM e seus espaços de memória independentemente, se desacoplando do padrão de acesso à memória *DDR*, enquanto opera sem supervisão do processador. Com baixos custos de área e potência, podemos atingir ganhos de performance e energia em transferências de dados entre unidades PIM. A solução *Inter-PIM* reduz o custo de performance de movimento de dados entre *threads* em 20% quando os dados estão alinhados na memória e em mais de 4× quando não estão.

C.2 Trabalho Futuro

Ainda existem muitas melhoras possíveis neste trabalho. Os modelos de energia aqui apresentados ainda devem ser validados por si só. A avaliação de performance do *Inter-PIM* também requer melhoras, adicionando mais complexidade. Talvez, de maneira mais crítica, um *suite* de *benchmarks* mais complexo poderia ajudar a validar a necessidade do *Inter-PIM* em cenários reais. Como trabalho futuro

no *Inter-PIM*, nós planejamos investigar a redução da poluição das *caches* do *host*. Também desejamos investigar o impacto uma *NoC* dentro do módulo de memória mudaria a dinamica do acesso, inclusive trazendo benefícios para o *host* (AHN et al., 2012).

O Sim^{2}PIM já tem uma alta complexidade, contudo, ainda há espaço para adicionar mais funcionalidades. Como trabalho futuro, estamos testando maneiras de implementar o Sim^{2}PIM como uma ferramenta de instrumentação para *PIMs* comerciais. Também buscamos implementar um *scheduler* e como ele impactaria a performance das *threads*. Como trabalho futuro, também resta a integração com outros simuladores *PIM*.