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

GERALDO FRANCISCO DE OLIVEIRA JUNIOR

A Generic Processing in Memory Cycle Accurate Simulator under Hybrid Memory Cube Architecture

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

Advisor: Prof. Dr. Luigi Carro

Porto Alegre
September 2017

CIP — CATALOGING-IN-PUBLICATION

de Oliveira Junior, Geraldo Francisco

A Generic Processing in Memory Cycle Accurate Simulator under Hybrid Memory Cube Architecture / Geraldo Francisco de Oliveira Junior. – Porto Alegre: PPGC da UFRGS, 2017.

70 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2017. Advisor: Luigi Carro.

1. In-Memory Processing. 2. Simulators. 3. Hybrid Memory Cube. 4. 3D-Stacked. I. Carro, Luigi. II. Título.

“The fairest thing we can experience
is the mysterious.”

— Albert Einstein

AGRADECIMENTOS

Agradeço aos meus pais pelo suporte. Agradeço ao Paulo pelo apoio durante a elaboração deste trabalho (e por todas as discussões). Agradeço ao Professor Luigi por me ensinar a pensar fora da caixinha. Agradeço aos amigos do Laboratório de Sistemas Embarcados. Em especial ao Marcelo e o Anderson pelas caronas. O Jeckson pela parceria. O Arthur pelas histórias. Agradeço ao Tiago pelas longas discussões sobre Game of Thrones. E ao Leonardo por ser uma enciclopédia. Agradeço ao Professor Caco pelos concelhos. E agradeço a Michele por aguentar minhas bagunças.

ABSTRACT

PIM - a technique which computational elements are added close, or ideally, inside memory devices - was one of the attempts created during the 1990s to try to mitigate the memory wall problem. Nowadays, with the maturation of 3D integration technologies, a new landscape for novel PIM architectures can be investigated. To exploit this new scenario, researchers rely on software simulators to navigate throughout the design evaluation space. Today, most of the works targeting PIM implement in-house simulators to perform their experiments. However, this methodology might hurt overall productivity, while it might also preclude replicability. In this work, we showed the development of a precise, modular and parametrized PIM simulation environment. Our simulator, named CLAPPS, targets the HMC architecture, a popular 3D-stacked memory widely employed in state-of-the-art PIM accelerators. We have designed our mechanism using the SystemC programming language, which allows native parallel simulation. The primary contribution of our work lies in developing a user-friendly interface to allow easy PIM architectures exploitation. To evaluate our system, we have implemented a PIM module that can perform vector operations with different operand sizes using the proposed set of tools.

Keywords: In-Memory Processing. Simulators. Hybrid Memory Cube. 3D-Stacked.

Um Simulador Genérico Ciclo-Acurado para Processamento em Memória baseado na arquitetura da *Hybrid Memory Cube*

RESUMO

PIM - uma técnica onde elementos computacionais são adicionados perto, ou idealmente, dentro de dispositivos de memória - foi uma das tentativas criadas durante os anos 1990 visando mitigar o notório *memory wall problem*. Hoje em dia, com o amadurecimento do processo de integração 3D, um novo horizonte para novas arquiteturas *PIM* pode ser explorado. Para investigar este novo cenário, pesquisadores dependem de simuladores em software para navegar pelo espaço de exploração de projeto. Hoje, a maioria dos trabalhos que focam em *PIM*, implementam simuladores locais para realizar seus experimentos. Porém, esta metodologia pode reduzir a produtividade e reprodutibilidade. Neste trabalho, nós mostramos o desenvolvimento de um simulador de *PIM* preciso, modular e parametrizável. Nosso simulador, chamado CLAPPS, visa a arquitetura de memória *HMC*, uma memória 3D popular, que é amplamente utilizada em aceleradores *PIM* do estado da arte. Nós desenvolvemos nosso mecanismo utilizando a linguagem de programação *SystemC*, o que permite uma simulação paralela nativamente. A principal contribuição do nosso trabalho se baseia em desenvolver a interface amigável que permite a fácil exploração de arquiteturas *PIM*. Para avaliar o nosso sistema, nós implementamos um modulo de *PIM* que pode executar operações vetoriais com diferente tamanhos de operandos utilizando o proposto conjunto de ferramentas.

Palavras-chave: Processamento em Memória, Simuladores, *Hybrid Memory Cube*.

LIST OF ABBREVIATIONS AND ACRONYMS

ACM	Active Memory Cube
AF	Atomic Flag
ALU	Arithmetic Logic Unit
CGRA	Coarse-Grain Reconfigurable Array
CLAPPS	Cycle Accurate Parallel PIM Simulator
ConvNet	Convolutional Neural Network
CRC	Cycle Redundancy Check
DDR	Double Data Rate
DIMM	Dual in-line Memory Module
DINV	Data Invalid
DRAM	Dynamic Random Access Memory
ERRSTAT	Error Status
FRP	Forward Retry Pointer
FSM	Finite State Machine
FU	Functional Unit
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HIVE	HMC Instruction Vector Extensions
HMC	Hybrid Memory Cube
HPC	High Performance Computing
ISA	Instruction Set Architecture
LM	Link Master
LS	Link Slave

NDP	Near-Data Processing
NN	Neural Network
NoC	Network-on-Chip
Pb	Poison bit
PIM	Processor-in-Memory
RRP	Return Retry Pointer
RTC	Return Token Count
RTL	Register Transfer Level
RVU	Reconfigurable Vector Unit
SEQ	Sequence Number
SLID	Source Link ID
SMC	Smart Memory Cube
SoC	System-on-Chip
SSD	Solid-State Drive
SST	Structural Simulation Toolkit
TSV	Through-Silicon Via

LIST OF FIGURES

LIST OF TABLES

CONTENTS

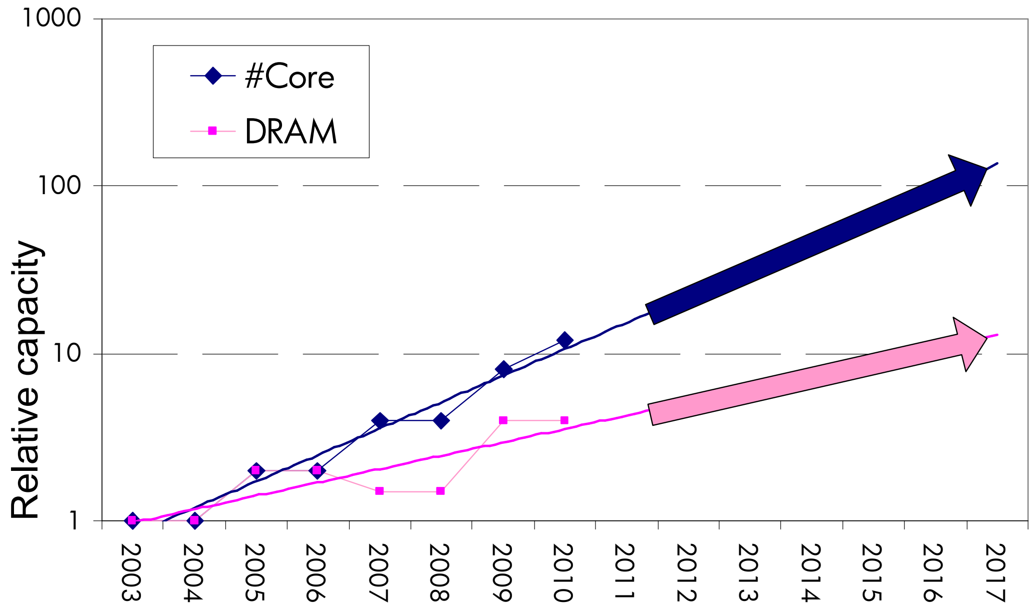
1 INTRODUCTION

For decades, the slow advancements of main memory technology and manufacturing processes have been overshadowed by Moore's law (SCHALLER, 1997). As smaller transistors paved the way for ever-faster processing units, the same could not be done for memory devices, which have different trade-offs and design points. This led to both a performance and a scaling gap between processing units and memory devices (WULF; MCKEE, 1995). Figure 1.1, extracted from (LIM et al., 2009), illustrates the current trend related to memory and CPU scaling. One can notice that the number of CPU cores per socket has been doubling every two years, while the memory capacity doubles every three years. If future systems continue to scale in this fashion, we expect memory capacity to start dropping by 30 % every two years (LIM et al., 2009).

The bottleneck that the memory hierarchy represents to modern computational systems is due to two fundamental reasons:

1. *Technological.* Dynamic Random Access Memorys (DRAMs) are commonly employed as the main memory device because of its reasonable cost per bit since it requires only one transistor to store a single bit (JACOB; NG; WANG, 2010). As detailed in Section 2.1, DRAM devices are charge-based modules, meaning that data is stored using capacitor cells. This technological design choice adds com-

Figure 1.1: Current memory and CPU scaling trends. The processor line depicts the number of cores per socket, while the memory line predicts the memory capacity per socket.



Source: (LIM et al., 2009).

plexity and extra latency to access memory because (i) capacitors leak charge over time, forcing data to be refreshed during fixed time intervals, and (ii) all read operations are destructive; therefore all read commands need to be followed by a write command.

2. *Architectural.* The memory system has always been treated as a slave mechanism that only responds to read/write commands. However, as previous works have shown (LEE et al., 2010; STUECHELI et al., 2010), performance can be improved when the memory system and computational units cooperate sharing run-time information. For instance, it is possible to improve memory throughput by (i) taking advantage of memory parallelism and (ii) building memory controllers that explore data locality.

The gap between memory and CPU technology becomes even more visible in systems that process large volumes of data (MURPHY, 2007; RADULOVIC et al., 2015; LIM et al., 2009), in the so-called Big Data and High Performance Computing (HPC) workloads. For example, in a data-center scenario, a single query may request massive amounts of data to be moved between the whole memory system. Passing through non-volatile to the main memory modules, then by several layers of cache memories, until the requested data finally reaches the processor, where typically just a small amount of computation will be performed (SANTOS et al., 2017). Even though this issue is becoming more problematic in modern workstations, (WULF; MCKEE, 1995) has first foreseen the memory impact for future systems in 1995, calling it the **Memory Wall Problem**. The authors observed that processor speed rises at a range of 75 % per year, while memory speed increases by a factor of 7 % from each generation. This enormous gap between memory and CPU performance implies that the memory system would be the primary source of performance slowdowns. These days, to access main memory, the processor has to wait for around 100 clock cycles to receive the requested data back from memory. This scenario is even more problematic when one considers multi-core and heterogeneous systems, where multiple memory requests are being made simultaneously, thus adding extra latency due to conflicted accesses (MOSCIBRODA; MUTLU, 2007; EBRAHIMI et al., 2010; SUBRAMANIAN et al., 2013). Besides that, current DRAM memories are not able to supply enough bandwidth for HPC and Big Data applications (RADULOVIC et al., 2015). To conclude, what we have today are systems that require significant bandwidth from the memory system, but a memory system that cannot provide such amount of data per seconds.

Endeavoring to reduce the memory wall impact over large workloads, several memory manufacturers have taken advantage of 3D-stacked integration technology to implement 3D-stacked memories that fit better to HPC workloads. In a 3D-stack environment, several layers of memory modules are vertically connected by Through-Silicon Via (TSV), improving data density and also increasing memory bandwidth. Also, 3D integration makes it possible to implement heterogeneous stacks, with layers of memories and logic all together in one single chip. These features help to reduce the latency to access memory by moving memory controllers from CPU units into the memory device itself. Some commercial 3D stacked memories are available in the market nowadays, as Hybrid Memory Cube (HMC) (Hybrid Memory Cube Consortium, 2013), High Bandwidth Memory (HBM) (HONG, 2014), and DiRAM4 (Tezzaron, 2015).

Most important, the possibility to combine memory devices and computation elements into one single chip renews the prospect to explore Processor-in-Memory (PIM) architectures (PUGSLEY et al., 2014). PIM, previously named Near-Data Processing (NDP), was first designed to reduce the execution time of memory bounded applications by placing logic elements inside DRAM modules, aiming to explore their much larger internal bandwidth (PATTERSON et al., 1997). However, besides improving how the system would exploit memory bandwidth, PIM approaches can potentially help reduce overall memory access latency. Moreover, by adding logic modules closer to the memory, the amount of data that needs to navigate throughout the whole memory system can be reduced, therefore diminishing the total number of energy consumed by the entire system - a key design constraint for large-scale data-processing centers and embedded systems.

To understand the impact that memory components impose to a system, either regarding performance or energy consumption, researchers in academia heavily rely on software simulators to navigate throughout the design evaluation space. A lot of effort has been made during the past years by the academia to build memory simulators that could push state-of-the-art designs to the next level of complexity. To list some, the DRAMSim2 simulator (ROSENFELD; COOPER-BALIS; JACOB, 2011), a cycle-accurate Double Data Rate (DDR) memory simulator, is widely employed to estimate DDR2/3 speed and power consumption; the Cacti (SHIVAKUMAR; JOUPPI, 2001) model is a popular tool to estimate cache performance, area, and power consumption; and VSSIM (YOO et al., 2013) is a new Solid-State Drive (SSD) simulator that can mimic today's SSD architectures.

Since PIM has emerged again recently, simulating the industry's new memory de-

Table 1.1: PIM works and their simulators.

PIM Architecture	Platform	Category
Tesseract (AHN et al., 2015a)	In-house	cycle-accurate
IMPICA (HSIEH et al., 2016)	gem5	cycle-accurate
NDA (FARMAHINI-FARAHANI et al., 2015)	gem5	cycle-accurate
SMC (AZARKHISH et al., 2016a)	gem5	cycle-accurate
AMC (SURA et al., 2015)	Mambo Simulator	cycle-accurate
HIVE (ALVES et al., 2016)	SiNuca	functional
GPU+CPU (XU; ZHANG; JAYASENA, 2015)	In-house	analytical
Neurostream (AZARKHISH et al., 2017)	In-house	cycle-accurate
NIM (OLIVEIRA et al., 2017)	SiNuca	functional
RVU(SANTOS et al., 2017)	SiNuca	functional

Source: Provided by the author.

vices or even testing novel PIM mechanisms is a common issue that researchers face. As illustrated in Section 3, most of the works that target PIM implement in-house simulators to support their experiments (ALVES et al., 2016; XU; ZHANG; JAYASENA, 2015; AHN et al., 2015b; OLIVEIRA et al., 2017; SANTOS et al., 2017). Table 1.1 classifies some previous PIM works and the simulator used in each case. Today's PIM simulators can be categorized into three broad groups: *functional*, *cycle-accurate*, and *analytical models*. To illustrate, one could find works as (SANTOS et al., 2017), (OLIVEIRA et al., 2017), and (ALVES et al., 2016) that make use of simulators in the functional category. These simulators can determine the device performance isolated from other system components while modeling none or a limited number of elements from the device's micro-architecture. On the other hand, simulators as the ones employed by (FARMAHINI-FARAHANI et al., 2015) and (AZARKHISH et al., 2016a) perform a cycle-by-cycle evaluation of all system components, from interconnection modules to the operational system. Finally, the simulator used by (XU; ZHANG; JAYASENA, 2015) fits in the third class of simulators. Their simulator is designed using machine learning techniques to estimate the final device performance.

Two problems can be foreseen out of this scenario of multiple PIM simulators. First, a significant part of researching effort is spent building the required simulation environment. Second, it becomes difficult to reproduce another researchers' work. Another well-known problem related to PIM simulators is how to measure three important aspects of embedded system design: area, power, and energy consumption. Evaluating the total design area is important to put boundaries to PIM architectures. Also, dissipated power and energy consumption are two key design parameters in both embedded and HPC environments. Some approaches can be employed to obtain those metrics. First, one could build their design model using Hardware Description Language (HDL) and directly extract the produced circuitry using synthesis tools (AZARKHISH, 2016). Even though the Register Transfer Level (RTL) model may produce the most accurate result, its design flow is extremely time-consuming. Another approach would be adopting tools as McPat (LI et al., 2013) to estimate the model outputs. However, it is possible that the estimation tool may produce imprecise results due the number of variables related to the matter (XI et al., 2015).

Goal. In this work, we aim to build a generic PIM simulator that can aid researcher to build and investigate novel PIM mechanisms. However, there are two major challenges we have to deal with to cope with this task.

Challenge 1. *Which type of 3D-memory organization should our simulator target?* Since the maturation of the 3D manufacturing technology, several stacking memories devices have been proposed both by the academia and by different vendors (HONG, 2014; Hybrid Memory Cube Consortium, 2013; Tezzaron, 2015). However, in most cases, there is not a detailed documentation public available that describes these devices.

Challenge 2. *How to provide a simple mechanism to allow designs to implement new PIM architectures easily?* An ideal PIM simulator would have to provide to the final user two critical capabilities. First, it needs to be accurate as a memory simulator. Second, the simulator needs to be flexible and easy to use. It needs to be flexible to be employed in different PIM designs, which would probably use different types and numbers of Functional Units (FUs), internal storage elements, distinct and specialized Instruction Set Architectures (ISAs), and target various classes of applications. Also, the simulator needs to be easy to use to reduce the amount of effort that is spent understanding and extending the simulator.

To solve the first challenge, we have decided to employ the HMC device as our 3D-stacking memory mechanism. We took this decision based on two major facts. First, even though there is no documentation publicly available describing in details the internals of an HMC module, the HMC Consortium has published an extensive documentation that provides a high-level view of the memory organization and employed communication's protocols. Second, there are a large number of works that have investigated the capabilities and trade-offs related to HMC devices (ROSENFELD, 2014; AZARKHISH et al., 2016a; MATHEW et al., 2017).

To solve the second challenge, we have implemented into our simulator a PIM interface that deals with conventional operations, as *reads* and *writes*, to the user. In this way, the user can focus only on designing the internals of their mechanism. Also, we have extended the HMC communication's protocol to understand a new set of operations without imposing significant modifications to the already proposed mechanism.

Contributions. The major contributions of this work are:

- We built a generic Cycle Accurate Parallel PIM Simulator (CLAPPS) that can be used to create custom PIM architectures. Our framework, described in details in Section 4, has been developed using the SystemC programming language (PANDA, 2001). We have chosen to develop our system using SystemC because it can be easily integrated with already available simulation platforms, for example, with the widely employed gem5 (BINKERT et al., 2011) simulator [1]. Also, a SystemC module can quickly produce a synthesizable RTL model. Moreover, by implementing our simulation using SystemC, we were able to simulate parallel behavior natively. In the current version, we developed an HMC simulator targeting its latest technical specifications (Hybrid Memory Cube Consortium, 2013), including all HMC instructions.
- We validated the memory side of our simulator using previous works that target HMC devices. In special, we have used the work of (ROSENFELD, 2014) as a guide for the results expected of internal and external memory bandwidth. To demonstrate the PIM capabilities of our simulator, we have implemented a simple vectorial processor, similar to the one proposed by (SANTOS et al., 2017), using the proposed PIM interface (Section 6).
- With our simulator, we could observe common challenges that new PIM architectures would possibly have to deal in the future, as the role of a 3D-stacking memory in the system, programmability, and data coherence (Section 7).

[1]Gem5 is a robust and extensible system that can simulate most elements of a computer system, including a number of instruction sets, micro architecture organizations, memory devices, interconnections, and communication protocols.

2 BACKGROUND

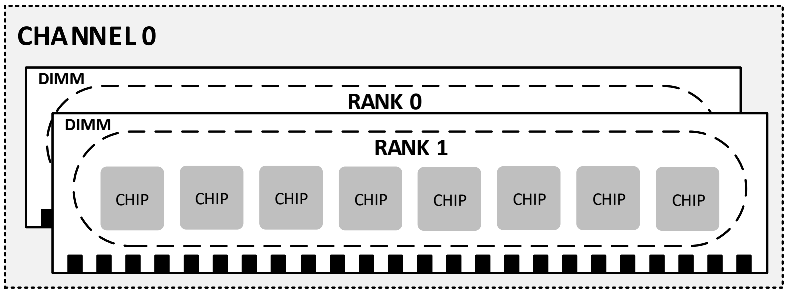
In this section, we introduce some basic concepts that will be used throughout the rest of this work. First, we give a brief explanation about the Dynamic Random Access Memory (DRAM) architecture, organization, and functionalities. For a more detailed overview of DRAM devices, we refer the reader to (JACOB; NG; WANG, 2010; HASSAN et al., 2017; PATEL; KIM; MUTLU, 2017; SESHADRI et al., 2013; LIU et al., 2013; LIU et al., 2012; LEE et al., 2015). Second, we describe in details the internal structure and the communication's protocol of an Hybrid Memory Cube (HMC) module. All information related to HMC devices was obtained from the latest HMC specification (Hybrid Memory Cube Consortium, 2013). Finally, we present a brief description of the SystemC programming model.

2.1 DRAM Basics

2.1.1 DRAM Organization

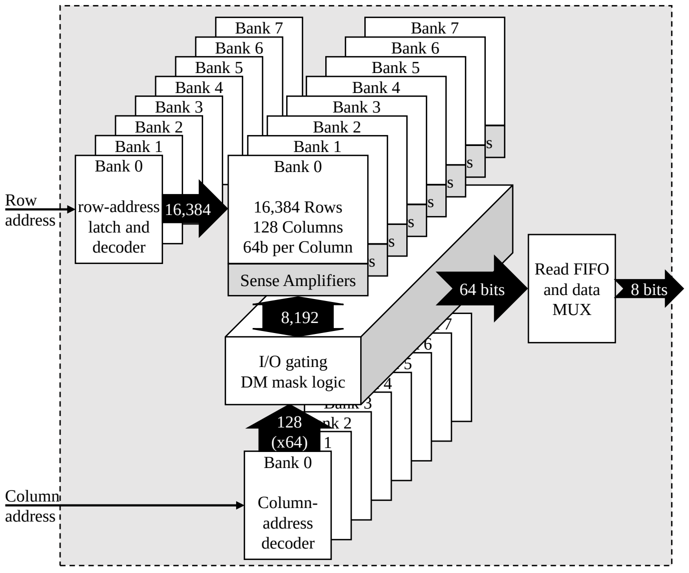
Figure 2.1 illustrates a high-level view of a DRAM module. A single DRAM device is organized hierarchically into *Channels*, *Dual in-line Memory Modules (DIMMs)*, *Ranks*, *Chips*, *Banks*, and *Cells*. A *Channel* receives commands, address, and data via buses connected to the memory controller placed into the CPU socket. Each *Channel* has one or more *DIMMs*. Each *DIMM* is composed of a set of one or more *Ranks*, and operates independently of each other. In today's architectures, a single memory controller can operate over a single *Channel* or over multiple *Channels* using a wider I/O interface. A *Rank* consists of a set of DRAM's *Chips* that share the command signals and works in a lockstep. Typically there are from four to eight DRAM's *Chips* in a *Rank*. Each *Chip*

Figure 2.1: Overview of a DRAM device.



Source: Provided by the author.

Figure 2.2: DRAM's internal chip organization.



Source: (SANTOS et al., 2016).

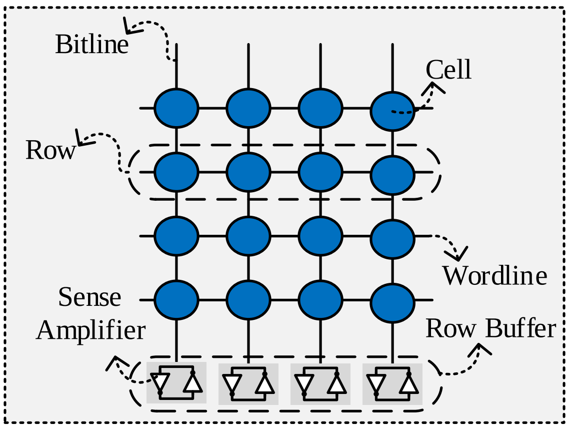
contributes to a portion of the data that flows from/to the request device. For example, in a system with a *Channel* I/O interface of 64 bits, and 8 *Chips*, each *Chip* would output 8 bits of data in parallel.

Figure 2.2 depicts the internals of a single DRAM *Chip*. The primary block in a *Chip* is the *Bank*. Inside a single *Chip* are usually four to eight *Banks* that can be accessed independently and operate in parallel. Besides the *Banks*, there are a set of latches and decoders used to access the *Banks'* rows and columns, and some logic elements used to output data to the bus interface. Figure 2.3a illustrates the internal organization of a DRAM *Bank*. Each *Bank* is composed of a 2D grid of *Cells*, as shown in Figure 2.3b. Each row of *Cells* is connected via a *wordline*, and each column of *Cells* is connected via a *bitline*. Also, each *bitline* is connected to a sense amplifier. The set of sense amplifiers in a bank are called *Row Buffer*. A DRAM *Cell* is the basic element of a DRAM device since it is the actual storage element. Each *Cell* is built using an access transistor, controlled by the *wordline*, connected to a capacitor. When the *worline* is set high, the *bitline* is connected to the capacitor via the access transistor, allowing charge to flow into the sense amplifier.

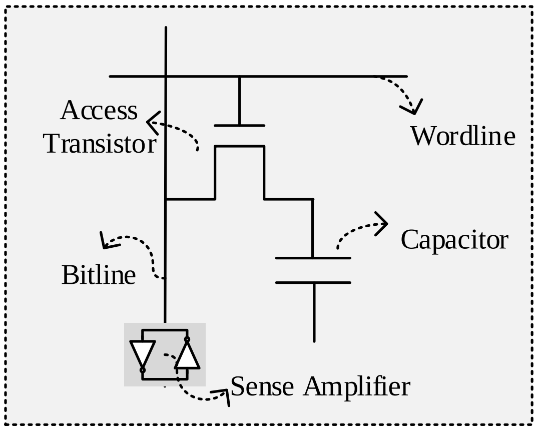
2.1.2 DRAM Operations and Commands

Figure 2.4, extracted from (LEE et al., 2015), illustrates important phases that a DRAM's cell goes through during a request command. Initially, the cell is in the

Figure 2.3: DRAM's Bank and Cell organization.



(a) DRAM Bank



(b) DRAM Cell

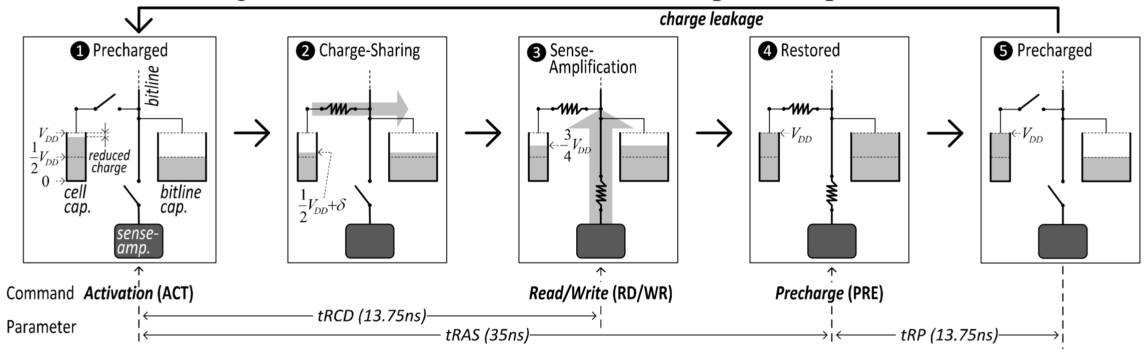
Source: Provided by the author.

precharged state ❶, where the *wordline* is held in low, the *bitline* and the sense amplifier are maintained in an equilibrium of ½ the V_{DD} (maximum) voltage.

Activation. To access data from a particular row, the memory controller will send through the I/O Bus the row address related to that row and the **ACTIVATE** command. The row address is going to be decoded by the row decoder, and then the *wordline* related to the row is going to be set high ❷. After that, all cells in the row are connected to their particular *bitline*, and each cell will lose/gain charge depending on its initial value. The charge will flow from the cell to the *bitline* if the cell is at a higher voltage state that the *bitline*, or from the *bitline* to the cell otherwise. This state is called *chage-sharing*. The sense amplifier will observe a positive or negative voltage difference on the *bitline*, and then fully restore the cell's charge based on that ❸.

Read/Write. When the sense-amplification process is completed, the sense amplifier will have stored the data related to the charge on the cell (either 0V or V_{DD}), and also data will be restored into the cell ❹. The time the memory controller has to wait between

Figure 2.4: Overview of the DRAM's operation protocol.



Source: (LEE et al., 2015).

an **ACTIVATE** and a *READ/WRITE* command and data to be stable on the sense amplifier is called tRCD (row-to-column delay). After this time, the memory controller can send the column address related to the fraction of data it wants to access from the row buffer (a standard DRAM device has a row buffer of 4 Kbits or 8 Kbits). In traditional Double Data Rate (DDR) DRAM modules, new data will be available on the external data bus in both rising and falling edges of the clock. In general, the memory controller would try to maximize the number of consecutive access to the same row buffer (row buffer hit) since data can be obtained in a much smaller latency direct from the row buffer. This mechanism is called *open-row policy*. On the other hand, in a *closed-row policy*, a **PRECHARGE** command would be sent after all *READ/WRITE* commands, therefore cleaning the row buffer.

Precharge. When the memory controller issues a new request to a row different to the one already activated (row buffer miss), or in the case of a *closed-page policy*, the memory controller needs to emit a **PRECHARGE** command ❺. During precharging, the *wordline* will be put in low, disconnecting the cell and the *bitline*. Also, the *bitline* needs to be set again to ½ the V_{DD}. Therefore, the cell is ready to respond to a new request. The time between an **ACTIVATE** and a **PRECHARGE** command is called tRAS (row active time). This time parameter needs to be respected so data can be fully restored back to the cell. Also, after emitting a **PRECHARGE** command, the memory controller has to wait for tRP (row precharge time) between issuing a new command so that the precharging operation can be completed.

Refresh. Since the DRAM cell is a capacitor-based mechanism, it leaks charge over time. Therefore, all rows need to be accessed periodically (typically 64 ms) to restore data back to the cell. This process is called **REFRESH**. During refreshing, a row is activated (opened), so that the sense amplifier can restore the cell's charge. The time parameter related to refresh is called tREFI (refresh interval).

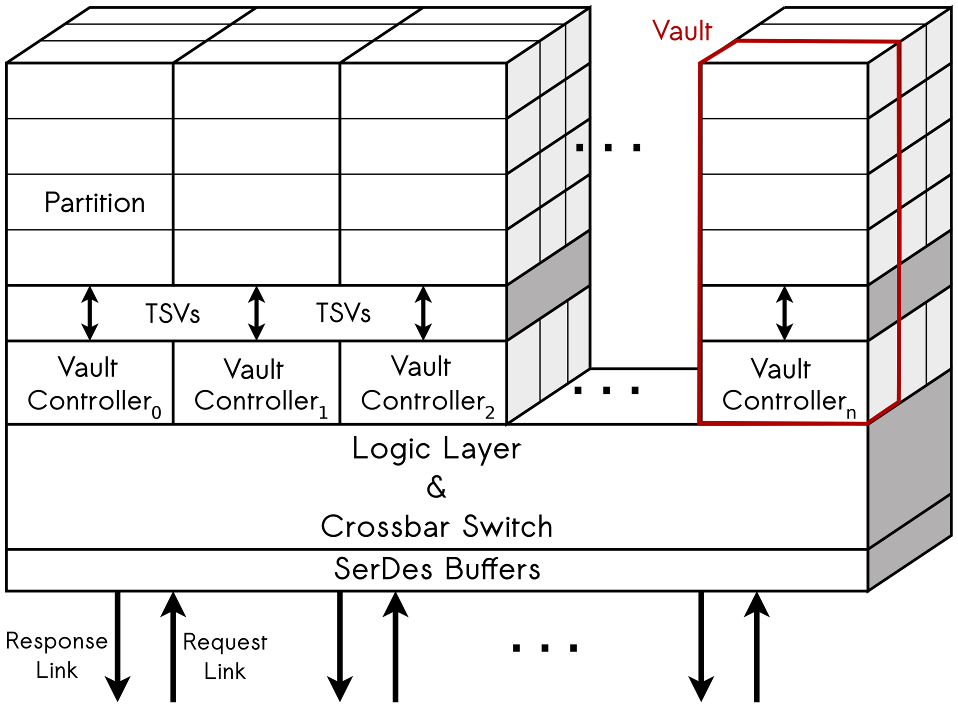
2.1.3 DDR Command Interface

The memory controller issues request to the DDR device via the memory bus. When trafficking through the bus, DDR commands are encrypted into a set of five distinct signals: *CKE*, *CS*, *RAS*, *CAS*, and *WE*. The *CKE* (clock enable) signas tells if the memory is activated (ready to be accessed) or in a low-power mode. The *CS* (chip select) signal enables a given *rank* of chips in the *DIMM*. The *RAS* (row address strobe) signal activates a row of cells, while the *CAS* (column address strobe) signals is used to select a specific column within the row. These signals are combined with the *WE* (write enable) signal to generate read/write requests.

2.2 Hybrid Memory Cube

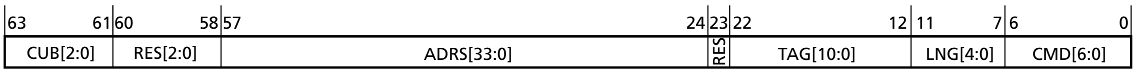
Figure 2.5 depicts the internal organization of an HMC device. HMC is a 3D memory device that targets bandwidth-intensive applications. In the latest HMC specification (Hybrid Memory Cube Consortium, 2013), the device is composed of four high-speed serial links, a logic layer of 32 memory controllers (called *Vault* controllers), and four layers of DRAM memories connected via Through-Silicon Via (TSV) through the *Vault* controller. Each *Vault* controller can operate independently upon 16 memory banks, and also can execute some atomic arithmetic operations. A single HMC device can provide a total bandwidth up to 320 GB/s. With the logic and memory integration provided by HMC, the memory controller can be moved from today's internal CPU package to closer to the memory device, reducing overall system latency and improving performance for

Figure 2.5: Overview of the HMC organization. In this figure, a Partition is equivalent to a DRAM's rank.

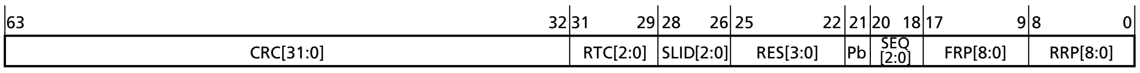


Source: Extracted from (ROSENFELD, 2014).

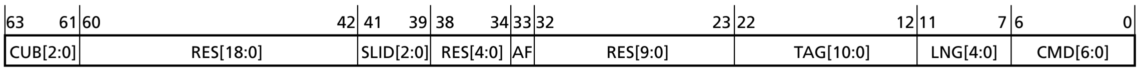
Figure 2.6: Fields for request and response HMC packets.



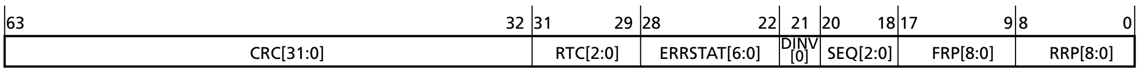
(a) Request header format.



(b) Request tail format.



(c) Response header format.



(d) Response tail format.

Source: HMC Specification 2.1. (Hybrid Memory Cube Consortium, 2013).

multi-core chips. Also, it is now possible to add computation resources alongside the logic layer of the HMC module. The HMC organization can be split into two major blocks. First, the Transceiver layer is responsible for managing request and response packets going between the HMC and the host device. Second, the *Vault* layer operates over these requests, accessing the DRAM layers to read/write data.

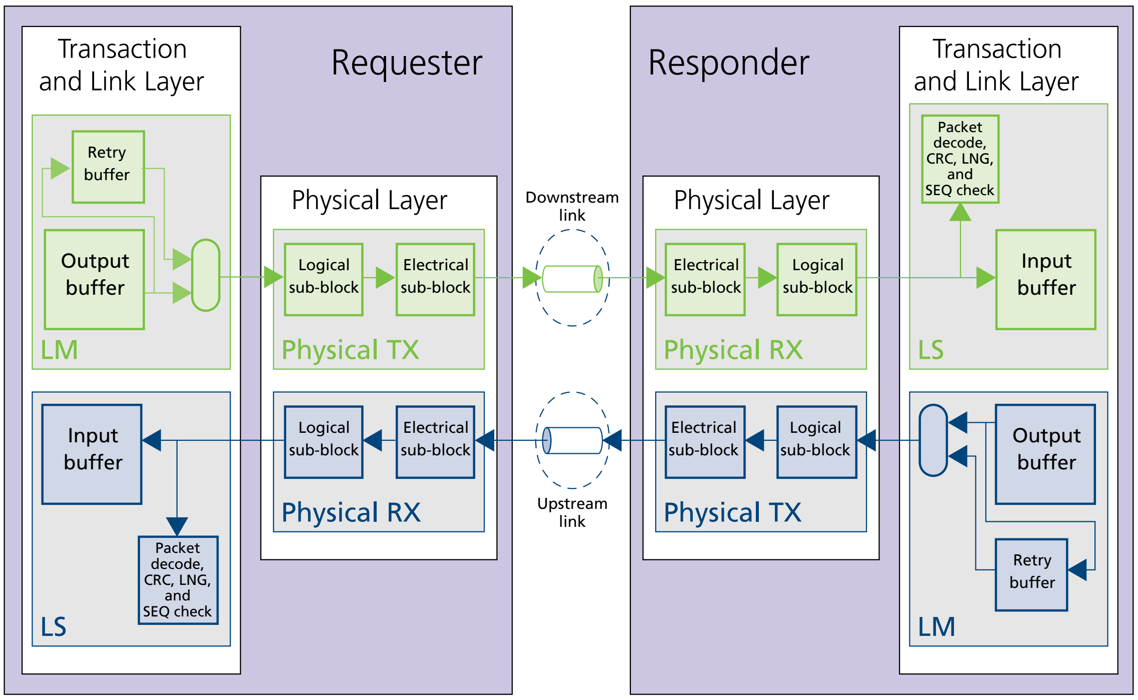
2.2.1 Transceiver Layer

HMC operates using a packet-based protocol. A packet, called FLIT, is 128 bits width and has at least a *header* and *tail* fields, each one being 64 bits width. Figure 2.6a illustrates the fields presented in the request *header* packet. It holds information regarding the requested HMC command (CMD), the number of FLITs in the request (LNG), a unique tag that identifies the request (TAG), the memory address (ADRS), and a module identifier (CUB). The CMD field is seven bits long and stores one opcode from the 54 possible HMC operations. Operations are categorized as READ, POSTED READ, WRITE, POSTED WRITE, and ATOMIC. READ and WRITE operations vary from 16 bytes to 256 bytes, in a 16 bytes step. All these operations return an acknowledge packet to the host. POSTED instructions have the same behavior than standard READ and WRITE operations but do not produce an acknowledge response to the host. The LNG field stores how many FLITs compose that specific request. The minimum number of FLITs per request is one, and the maximum is seventeen (for a 256 bytes request). The TAG field is responsible for identifying each packet. This value is used to ensure ordering between requests and responses throughout the system, and to identify a failing packet that may need to be re-sent. Finally, the CUB field stores an identifier for the particular memory cube, since several HMCs modules can be connected in a network to provide more memory capacity. Figure 2.6b shows the fields presented in the request *tail* packet. Most of the information stored in the request *tail* is used to ensure correctness through the system. Since data traffic back and forth host and memory via high-speed serial lanes (up to 30 Gb/s), errors might happen. Therefore, the HMC protocol provides a mechanism to ensure packet integrity. The Return Retry Pointer (RRP), Forward Retry Pointer (FRP), and Sequence Number (SEQ) fields are used to manage retransmission of invalid packets. The Poison bit (Pb) is set to one when the host wants to mark locations in the DRAM as having poisoned data. Source Link ID (SLID) indicates which one of the fours links the request came from. All response packets must be returned to the same source link. Return Token Count (RTC) indicates the number of free positions available in the request's input buffer. Finally, the Cycle Redundancy Check (CRC) field is used to verify the correctness of the packet.

Once the memory has finished processing the request packet, the Transceiver layer will generate a packet response to the host. Similar to the request packet, the response packet has a *header* and *tail* fields, plus any data payload related to the request. Figure 2.6c illustrates the response *header*. The CMD field specifies the request command that relates to the response. The LNG field tells how many FLITs compose the packet. The TAG field is the same identifier that came with the request packet. The Atomic Flag (AF) is set when overflow happens while executing arithmetic instructions. The SLID field indicates the link where the response should return. Finally, the CUB tells which HMC device the packet belongs. Figure 2.6d shows all fields in a response *tail*. The RRP, FRP, SEQ, RTC, and CRC fields serve the same purpose than the ones in the request *tail*. The Data Invalid (DINV) bit tells the validity of the packet payload. The Error Status (ERRSTAT) field is used to codify the type of error that might have happened during the Transceiver process, as warnings, DRAM errors, link errors, protocol errors, *Vault* errors, or fatal errors.

Figure 2.7 depicts the overall organization of the Transceiver module. The Requester and Responder sub-modules compose the Transceiver module. The former one is related to the host or any other HMC module that sends packets in the downstream flow, while the latter relates to the HMC module the sends packets in the upstream flow.

Figure 2.7: Overview of the Transceiver module organization.



Source: HMC Specification 2.1. (Hybrid Memory Cube Consortium, 2013).

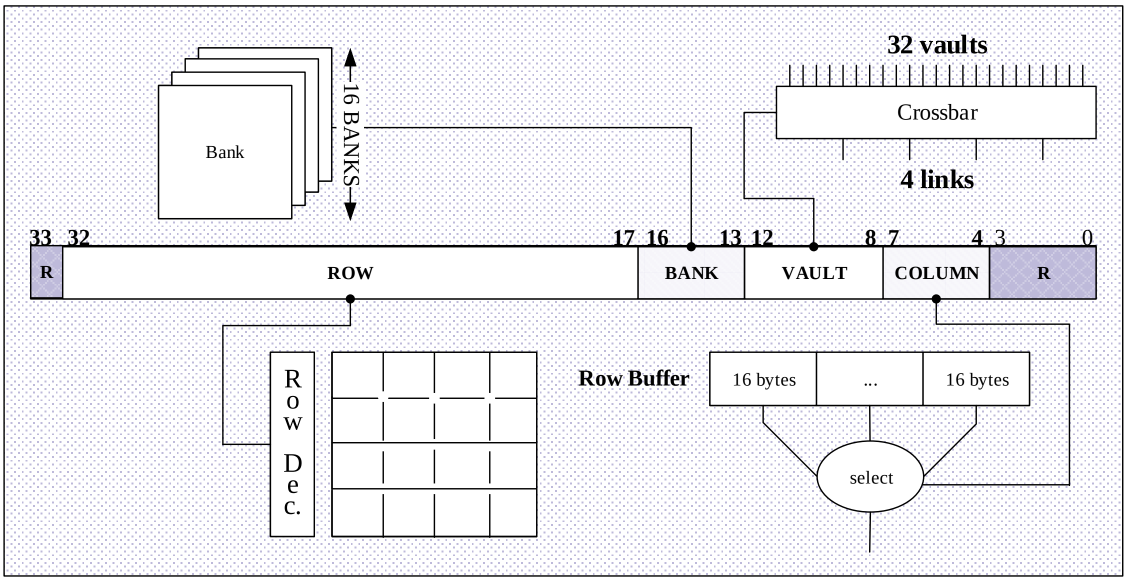
Also, both Requester and Responder sub-modules have a Transceiver and Link Layer and a Physical Layer. The Transceiver and Link Layer are responsible for handling the internal HMC communication protocol, encapsulating the requests and response packets and managing packet integrity. The Link Master (LM) is the module responsible for generating the packet, while the Link Slave (LS) receives and forwards the packet. Meanwhile, the Physical Layer serializes and de-serializes the packets produced by the Transceiver and Link Layer, and also manages transmissions across the high-speed serial links.

2.2.2 Vault Layer

The major modules within the Vault Layer are the *Vault* controller, the memory banks, some request/response buffers, and control logic. An HMC device is composed of 32 independent *Vaults* that can operate in parallel. Each *Vault* provides an aggregated bandwidth of up to 10 GB/s, which sums up to an internal aggregated bandwidth of 320 GB/s.

The *Vault* controller represents the same as the memory controller of traditional DRAM devices. It has five primary functionalities. First, to de-codify the packet coming from the Transceiver layer into DRAM's READ and WRITE commands. Second, to

Figure 2.8: Representation of the low-interleave algorithm used as memory mapping scheme.



Source: Provided by the author.

schedule the memory requests in some order based on the row buffer policy. Third, to control the flow of data that navigates through the TSV to/from the DRAM banks. Fourth, to generate response packets to the host when appropriate. Finally, the *Vault* controller is responsible for preserving the order of all response packets, since even though it is possible to execute the memory requests out-of-order, they must be committed in-order to maintain data coherency. The TSV connects the memory banks to the *Vault* controller via a vertical bi-directional bus of 32 bytes. The bus is used to send the sequence of DDR commands generated by the *Vault* controller, the address related to the request, and data from READ and WRITE requests. Finally, the *Vault* controller has some logic units used to execute arithmetic operations generated by HMC atomic instructions.

The memory banks have the same internal organization as presented in Section 2.1. A single *Vault* controls 16 memory banks. For an 8 GBytes HMC module, a single bank stores a total of 16 MBytes of data. The row buffer size of the memory banks in an HMC device is much smaller than the one from standard DDR memories. While the latter ranges from 4 Kbits from 8 Kbits, the former can be reconfigurated during initialization to be 32 bytes, 64 bytes, 128 bytes, or 256 bytes width. This difference in size is due to the fact HMC was originally designed for High Performance Computing (HPC) applications, where data locality is small at the row buffer level. In consequence, the row buffer policy employed natively by HMC is closed-row.

The row buffer size influences directly the interleave mechanism, i.e., how physical addresses are mapped to the memory modules. The interleave used by HMC is called low-interleave and is illustrated in Figure 2.8. In this interleave algorithm, the four least significant bits of the memory address (34 bits in total) are ignored. The next up to four bits (the forth through up the seventh bits) address the number of 16 bytes segments within the row buffer. Then five bits are used by the crossbar switch between the *Vault* and Transceiver layers to address the target *Vault*. The next four bits address the banks, and finally, the remaining bits are used by the row address decoder within the bank. This algorithm favors subsequent requests since adjacent addresses are stored in sequential *Vaults*, and then in sequential banks within a *Vault*. Therefore, bank conflicts are avoided in requests for sequential addresses.

2.3 The SystemC Programming Model

Hardware Description Languages (HDLs), like Verilog and VHDL, have significantly increased the productivity level for hardware designers by creating a modeling environment that imposes a higher abstraction level than the previously schematic-only view. However, with the growth in complexity in modern hardware systems, the modeling platform employed would also need to be elevated its abstraction level. From modeling individual bits in HDL systems to simulating complex systems consisting of complex IP blocks, a vast number of processor cores, embedded software and other elements that might be included in today's System-on-Chips (SoCs).

Table 2.1 gives an overview of different programming models and whether or not they are appropriated for developing systems at distinct levels of abstractions. One can notice that high-level programming languages are a good modeling choice when only the primary system's behavior needs to be evaluated. It can be used to abstract the complexity involving the model's architecture, and provide an overview of how the mechanism will work conceptually. However, it does not provide detailed information regarding tailored timing parameters and the internal organization of the system. On the other hand, behavioral and Register Transfer Level (RTL) simulations can be obtained by employing HDLs, where each component of the system can be modeled taking into account their architecture, organization, and significant timing constraints. However, it can be a challenge to implement whole systems using these levels of abstraction, due to the scale and complexity of modern hardware systems.

SystemC (GHENASSIA et al., 2005) was created in 1999 by the Open SystemC

Table 2.1: Programming languages categorized targeting distinct levels of abstraction.

Level of Abstraction	**Verilog**	**VHDL**	**C/C++**
System Level	No Suitable	Poor	Very Good
High Level (Behavioral)	Good	Very Good	Good
Medium Level (RTL)	Very Good	Very Good	Poor
Low Level (Gates)	Good	Poor	No Suitable

Source: Adapted from (JAYADEVAPPA; SHANKAR; MAHGOUB, 2004).

Initiative (OSCI, now Accelera) as a modeling platform that aims to elevate the abstraction level targeting to create a more flexible and productive hardware/software co-design environment. SystemC was built atop the C++ programming language. Therefore, it takes advantages of the already presented object-oriented programming paradigm, the native data types, the inheritance and polymorphism capabilities, and all the software infrastructure already available for C++. It also includes functionalities that are specific of hardware designs as concurrency, clocks, and bit-based data types.

Listing 2.1 shows a generic SystemC code with the major components of its programming model. It is composed of the module declaration, a set of input/output signals, a method, a list of sensitivity related to the processes, and potentially one or more submodules. A module is a container class that can be employed hierarchically to build each entity of the systems. A module typically has a set of input/output ports, processes, signals, and local variables. All modules must implement its constructor method, where submodules can be connected, and procedures are tight up with their respective sensitive list. A method defines one or more behaviors produced by the module. Each method is correlated to a sensitivity list of input ports. All methods within a module are executed concurrently and are scheduled to run when any port described in their sensitivity list changes.

Listing 2.1: A generic SystemC code snippet.

```
SC_MODULE(module){
    sc_in     <T>     ....;
    sc_out    <T>     ....;
    SC_CTOR(module){
        SC_METHOD(prc_module);
        sensitive << clk.pos();}
    void prc_module();
    sub_module mod_submodule;};
```

3 RELATED WORK

In this section, we present an overview of past works that aim to employ Processor-in-Memory (PIM) architectures in diverse classes of application. Also, we depict a list of Hybrid Memory Cube (HMC) simulators available as open-source tools.

3.1 Processing-in-Memory Architectures

Several works have proposed to explore internal memory bandwidth and reduce data movement through the memory system by exploring new PIM architectures. During the 1990s, some authors provided mechanisms that integrated logic units inside DRAM chips (PATTERSON et al., 1997; ELLIOTT et al., 1999). However, due to manufacturing restrictions that DRAM devices face, this line of thought was abandoned until 3D-stacked technologies have became a reality. In this section, we list some past works that employ PIM designs targeting to accelerate distinct classes of applications.

First, the Tesseract mechanism (AHN et al., 2015a) aims to accelerate large-scale graph processing workloads by exploring the internal bandwidth of HMC. The device is composed of a set of 16 memory cubes. A simple ARM processor was inserted into each memory vault of all memory cubes (the whole mechanism has 512 ARM processors). Tesseract acts as a side accelerator to the host CPU, which is responsible for offloading the graph operations to the device. The authors also provided a programming interface that helps the accelerator's memory prefetches to explore the available memory bandwidth fully.

Second, in (HSIEH et al., 2016), the authors presented IMPICA, a PIM architecture to accelerate pointer-chasing in memory. The authors observed that structures like linked lists and B-trees have sparse memory access patterns, which harm systems' performance by increasing the number of cache misses. The mechanism integrates two different engines into the vault units inside the memory cube. One mechanism is responsible for translating virtual addresses to physical addresses efficiently, and the second one accesses the memory using the converted address taking advantage of memory parallelism.

Farmahini-Farahani et. al. (FARMAHINI-FARAHANI et al., 2014) proposed an architecture that consists of a Coarse-Grain Reconfigurable Array (CGRA) placed atop a stack of DRAM modules. The mechanism aims to accelerate scientific and embedded applications. The CGRA unit has 32 Functional Units (FUs) and receives 128 bits of data from the DRAM layers bellow. To execute an application, the user needs to create a data flow graph to be placed into the array. Then, data is read and write from memory via memory mapped I/Os.

In (AZARKHISH et al., 2016a), the authors proposed to include an ARM processor between the vault controllers and internal interconnection of an HMC device, targeting to reduce data movement and improve the performance of memory bounded applications. The authors also developed an interconnection network that connects the memory layers to the ARM processor. To execute an application, the processor offloads the required kernel to the processor placed inside the memory cube via a DMA engine. Virtual memory capabilities are provided to the application by a TLB unit inside the ARM core. (AZARKHISH et al., 2016a) was integrated with the gem5 simulator (BINKERT et al., 2011).

The Active Memory Cube (ACM) mechanism (NAIR et al., 2015) is a PIM architecture that can execute up to 1 Exaflops in 20 MW. Their device is composed of 32 processing lanes that are connected to HMC vault interconnection. (NAIR et al., 2015) has a set of vector register file that can efficiently explore the available memory parallelism. The authors provided a full suite of APIs and simulation tools to explore their design, including support for power and reliability calculation.

HMC Instruction Vector Extensions (HIVE) (ALVES et al., 2016) provides vector instruction capabilities to HMC modules. Each vector instruction operates over 8KB of data retrieved from the memory vaults. This architecture target applications whose behavior is similar to stream ones. HIVE is composed of 2048 functional units, capable of executing integer and floating point operations, and a register bank with eight registers of 8192 bits each.

In (XU; ZHANG; JAYASENA, 2015) it is proposed a PIM that targets Convolutional Neural Network (ConvNet). Their design is comprised of a network of four HMC modules and a host CPU. Each HMC is augmented with an accelerator layer of several Graphics Processing Units (GPUs) and CPU. To take advantage of their design, the authors have classified the layers of a ConvNet into two categories: explores data parallelism and explores model parallelism. When data parallelism is used, the input batch of images is partitioned equally between the PIM stacks. Thus each accelerator can execute independently of each other. On the other hand, when model parallelism is employed, the Neural Network (NN) itself is partitioned between the PIMs, allowing much larger ConvNets to execute. The authors do not provide information about the architecture of the employed

GPU and CPU in the PIM layers and how the communication between modules is handled. Also, to evaluate their design, the authors make use of a machine learning-based simulator that only take into account the difference between bandwidth measures.

In (AZARKHISH et al., 2017), the authors presented Neurostream, a ConvNet architecture built as a network of 4 HMC devices. In the logic layer of each memory cube, there is a Neurocluster device, which is composed of four RISC-V processors, 32 streaming co-processors built to accelerate multiply-and-accumulate operations. Their design was developed based on the observation that the most significative operation executed by today's ConvNet is the MAC one. The major drawback to their design is the fact that their accelerator layer is placed outside HMC vaults. Therefore, it is not possible to employ the maximum available bandwidth provided by the memory.

The authors of (OLIVEIRA et al., 2017) proposed NIM, a PIM-based approach that targets Spiking Neural Networks, another class of NNs. In this class of NNs, the primary challenge is that to evaluate each neuron it is necessary to evaluate complex mathematical equations under a strict time window. The authors have observed this application does not take advantage of the cache hierarchy due to the lack of significative data reuse between layers. Also, the time the application spent loading the parameters required to evaluate a single neuron represented was significantly notable when compared to the rest of the execution. In their design, the authors have elaborated tailored vector units placed inside the HMC vaults that were specialized to compute the required mathematical equations for two different neuron models.

Finally, in (SANTOS et al., 2017), the authors have observed that database operations have a similar behavior to streaming applications, i.e., there was not a significant amount of data reuse between different queries execution. Another observation they made was that, for queries that have a composed condition statement (SELECT * FROM table WHERE [a AND b AND c ...]), data requests could be reconfigurable based on the result of the previous conditional statement. With these two notes in mind, the authors built a simple reconfigurable PIM mechanism to accelerate database operation, improving energy efficiency by reducing the total amount of data that needs to travel through the whole memory system.

3.2 Processing-in-Memory Simulators

Since the release of the first 3D-stacked memory, several attempts to build a concise simulator have been made by different researcher groups. However, it is not an easy task to replicate those devices functionalities because their internals are not available as open source information. Besides that, most of the simulators presented in this section have been implemented using sequential high-level programming languages. This methodology faces some issues since memory modules have a highly-parallel behavior. For example, any cycle-accurate simulator implemented using a non-parallel language would need to check at every single clock single the current state of all modules in the simulation, causing long simulation times.

From all currently available HMC simulators, HMC-Sim (LEIDEL; CHEN, 2016) is the most similar to our framework. This simulator was developed using the C++ programming language and provides a cycle-accurate memory simulation. Also, the simulator provides a simple data-structure based interface. Thus one could extend the already presented HMC instructions. Even though their approach can assist one to investigate the PIM capabilities of an HMC device, it has some limitations. First, to implement custom instructions, the authors have taken advantage of all *opcodes* that are not being used by the current HMC specification. However, this methodology adds a scalability problem to their interface due to two factors. First, only seventy (the number of currently free *opcodes*) new instructions can be created by the user, and second, as it had happened from HMC specification 1.0 to 2.0 (Hybrid Memory Cube Consortium, 2013), new native instructions are introduced by the HMC Consortium, making use of reserved *opcodes*. Finally, since shared library objects were used in their framework to provided a friendly interface to include the new HMC instructions, only Unix users can take advantage of the HMC-Sim framework. One significant difference from (LEIDEL; CHEN, 2016) and the presented work is that our PIM interface allows the user to include new architectures into HMC logic layer, rather than extending the already included instructions.

CasHMC (JEON; CHUNG, 2016) is a C++ HMC simulator that provides full HMC capabilities. It is an offline simulator that uses an external memory trace as its input generated by any processor simulator. This simulator aims to implement most of the HMC resources, as packet error detection, link flow control, and HMC instructions, while providing to the user some output files as performance summary, trace logs, and simulation graphs. This approach can lead to longer simulation times since the simulator needs to write information to four different files in each clock since. Besides that, the simulation accuracy can be profoundly affected by the fact that the simulation does not run alongside the program execution. Finally, CasHMC does not provide any PIM extension.

The Smart Memory Cube (SMC) Simulation Environment (AZARKHISH et al., 2016a) is a complete set of applications built inside the gem5 framework targeting PIM architectures. The simulator makes use of the already present memory modules, interconnection networks, and CPU implementation to create the HMC device. Also, it provides a software stack, including drivers and code annotation, to forward instructions to the processor core inside the memory device. Even though SMC-Sim is a complete and well-developed set of tools that can help the user to investigate the advantages of 3D-stacked memories, it is possible to point out some implementation choices that may limit performance exploration. First, the PIM layer is located between the memory controllers and the interconnection layer. Therefore it is not possible to extract all the bandwidth provided by the memory because a significant portion of bandwidth contention exists at this area. Second, although Azarkhish et al. (2016) have claimed to have validated their design latency and performance parameters using a complete Register Transfer Level (RTL) design, the memory implementation inside the gem5 simulator is based on correlations between Double Data Rate (DDR) and HMC architectures. This approach is valid in most parts, but the HMC architecture presents some particular characteristics that are not present in current DDR controllers, like Through-Silicon Via (TSV) access control. Finally, to obtain significant performance from their simulator, the authors have employed some structural modifications to the current HMC implementation, for example, changing the maximum request size from 256B to 512B, increasing the number of interconnection from 4 to 8, while duplicating its data width from 128b to 256b. Some previous studies, as (ROSENFELD, 2014), have shown that it is possible to extract close to the theoretical HMC bandwidth using the already presented design modules.

Finally, many in-house simulators have been employed during the past few years to obtain information regards PIM advantages. To illustrate, the author of (ALVES et al., 2016) have used a functional HMC simulator to implement different accelerators that target vector operations, database, and deep learning applications. Stelle et. al. (2014) use a Structural Simulation Toolkit (SST) simulator to investigate how data and computational locality impact PIM design. ZHang et. al. (2014) implement an analytic simulator aiming to understand how High Performance Computing (HPC) applications can benefit from 3D-stacked memories. Sura et. al. (2015) present a set of methodologies that can

Table 3.1: Comparison between available PIM simulators. "?" indicates the information could not be obtained.

Simulator	**Simulation Accuracy**	**Native PIM Capabilities**	**Novel PIM Capabilities**	**Peak Bandwidth**	**Deal with Parallelism**	**Input Type**	**Parame-terizable**	**Simulation Time**
HMC-SIM	Cycle-Accurate	Yes	Yes - Limited	320 GB/s	Event queues	Offline Trace	No	?
Cas-HMC	Cycle-Accurate	Yes	No	?	Event queues	Offline Trace	No	28.7s
SMC	Cycle-Accurate	Yes	Yes	205 GB/s	Event queues	Online Trace	Yes	4.5s

Source: Provided by the author.

help reducing access latency of 3D memories. To evaluate their proposal, the authors implemented a timing-accurate simulator.

Today's PIM simulators can be categorized into three broad groups: functional, cycle-accurate, and analytical models. Analytical models provide faster simulations but may not take into account important design metrics. Functional models execute only the behavior of the target architecture, instead of implementing and simulating the internals of the architecture itself. Cycle-accurate models are the most precise but can lead to slow simulations. Our simulator fits the latter category, but since we have used a high-level hardware description language to build our design, the simulation time is significantly lower than previous implementations. Also, our proposed framework has been constructed following the most current HMC specifications.

To summarize, Table 3.1 list the most important open-source HMC simulators available currently. Each simulator is categorized by their simulation' accuracy, if it implements the native HMC 's PIM instructions, if it provides any means to include novel PIM architectures into the simulator, the peak bandwidth obtained by the simulator, its type of input file, whether or not the simulator is parameterizable, and the time it took to simulate 1 M read instructions. As one can notice, all listed simulators claim to be cycle-accurate, implement the native HMC atomic instructions, and deal with parallelism using event queues. While HMC-SIM provides the user the ability to include novel PIM instructions by extending the already present HMC atomic instructions, SMC allows the user to insert a whole architecture within the HMC logic layer. However, as aforementioned, SMC cannot achieve HMC peak bandwidth. SMC is the only online simulator, meaning it can run alongside a CPU simulation. It is the only parameterizable simulator, while also being the fastest.

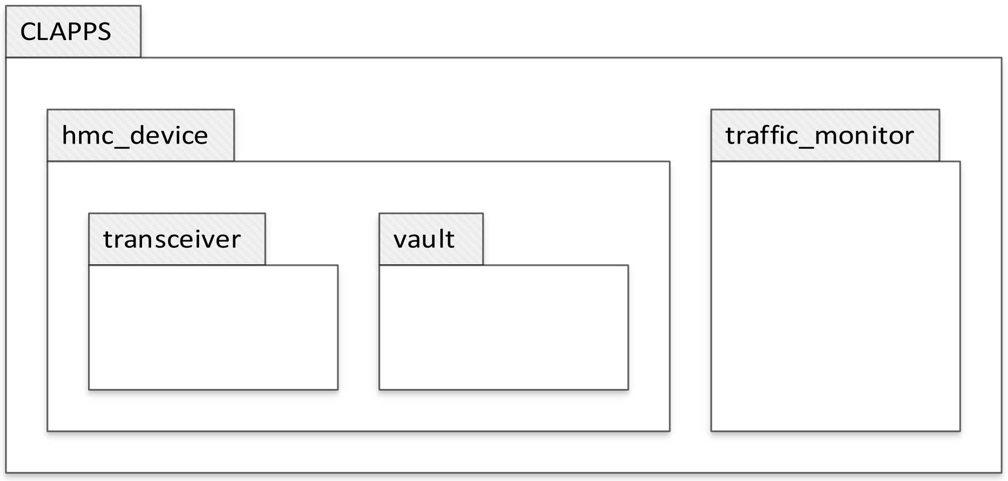
4 SIMULATOR IMPLEMENTATION

This section describes in details our simulator's architecture and discusses some design choices we have made during the implementation process. To implement the simulator, we have carefully studied the current Hybrid Memory Cube (HMC) specification (Hybrid Memory Cube Consortium, 2013) and then elaborated each component described in the documentation.

The presented simulator, called Cycle Accurate Parallel PIM Simulator (CLAPPS), is divided into three major blocks: *Vault*, *Transceiver*, and *Traffic Monitor*. The first one is responsible for receiving a memory request, translating the request into memory commands, scheduling and executing the generated memory operations, and generating the appropriate response packet for either a READ, WRITE, native Processor-in-Memory (PIM), or custom PIM request. The second one is responsible for encapsulating requests coming from the host, sending and controlling the request flow from the host to the *Vault* and vice-versa. Finally, the latter one is a pure C++ model that is responsible for tracking the request/response flow to produce statistics at the end of the simulation.

Figure 4.1 provides a simplified view of CLAPPS's UML packet diagram. Due to the complexity of the system, we broke-down the major modules into sub-modules and give a detailed description of their functionalities and interconnections.

Figure 4.1: Simplified packet diagram of the proposed simulator.



Source: Provided by the author.

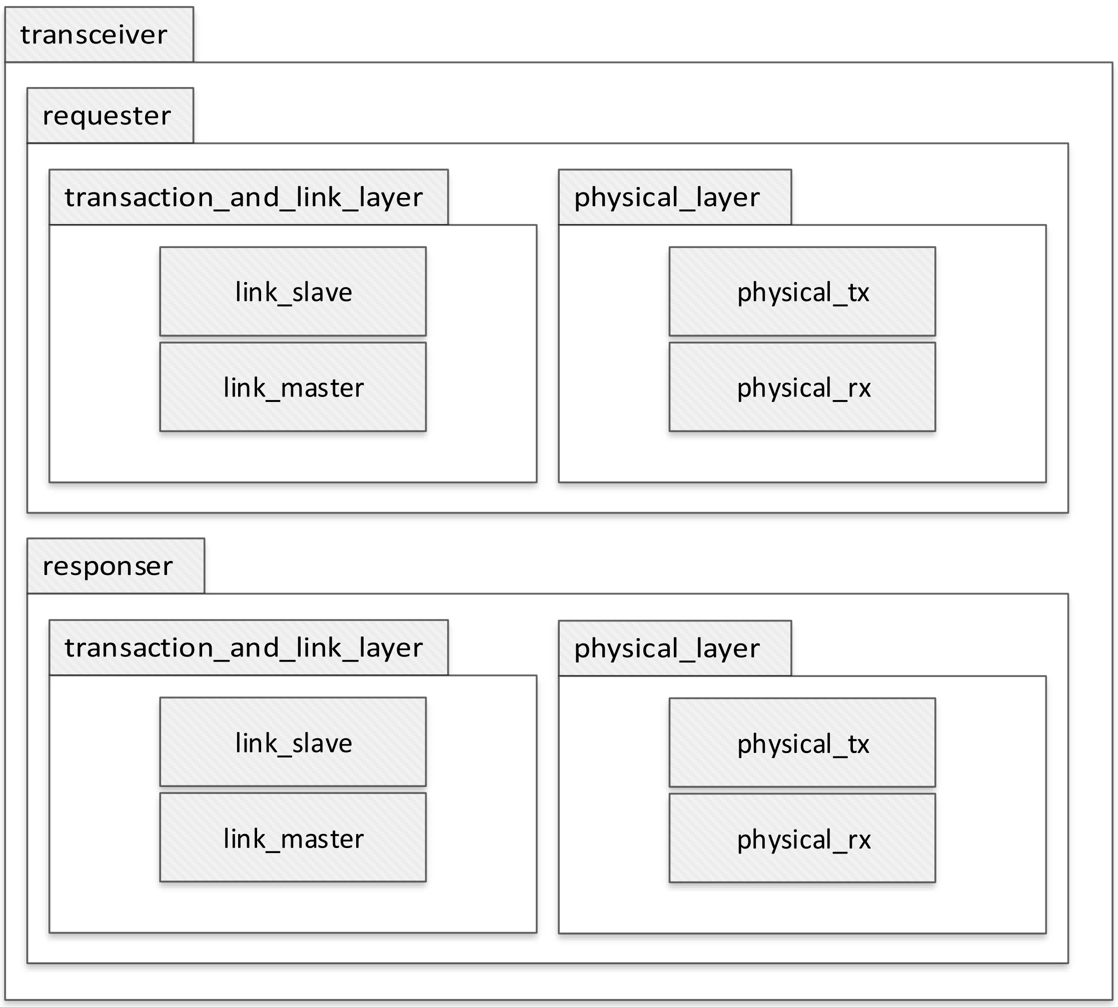
4.1 Transceiver

Figure 4.2 shows the packet diagram of the *Transceiver* module. As aforementioned in Section 2.2, the *Transceiver* is divided into two big blocks: the *Requester* and the *Responser* modules. First, we described the *Requester* block, its internal modules and operation, and then the *Responser* block.

4.1.1 Requester

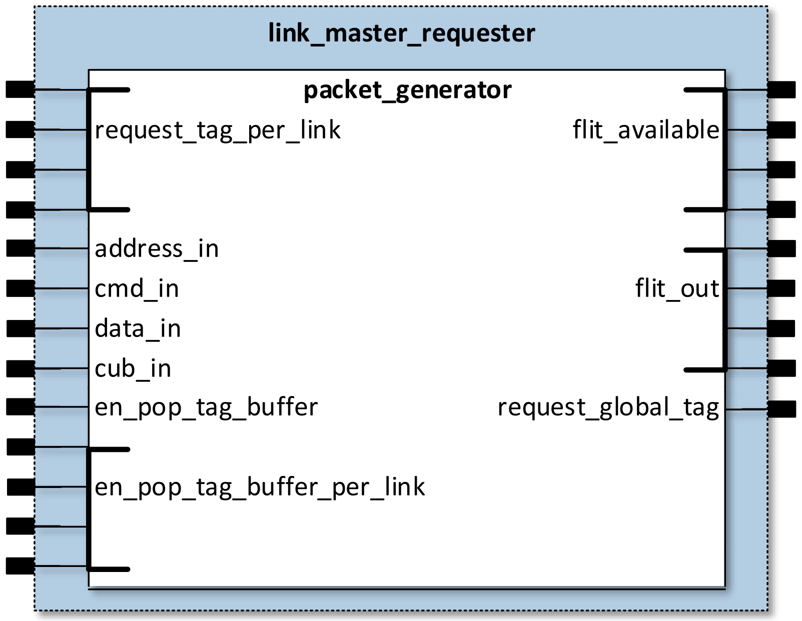
The *Requester* module is composed of the *transaction and link layer* and *physical layer* sub-modules. The former one is responsible for implementing the HMC communication protocol. It provides the definition of FLITs and also ensures that order between

Figure 4.2: Simplified packet diagram of the Transceiver Layer.



Source: Provided by the author.

Figure 4.3: Block diagram of the packet generator module.

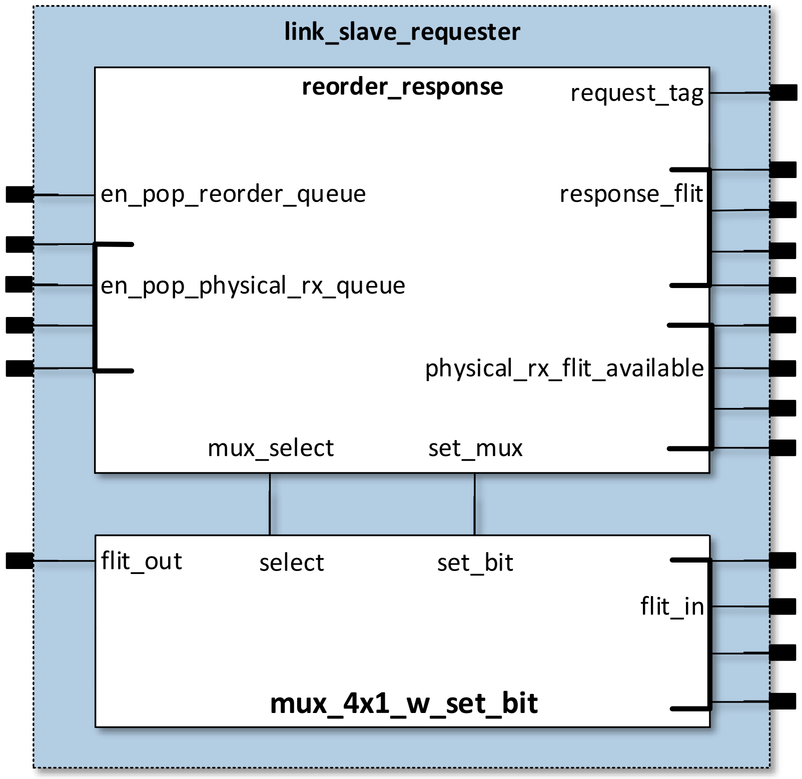


Source: Provided by the author.

requests is maintained. The latter one controls the high-speed serial links by serializing and de-serializing request and response FLITs. The *transaction and link layer* is once again divided into two sub-modules: *link master* and *link slave*. The *link master* encapsulates the request command that comes from the host, stores all generated FLITs at a local queue related to each link and also saves the generated tag identifier of the current request. Meanwhile, the *link slave* receives the response FLITs that come from the *Vault*, and stores them into the target link response buffer. Then, it reads all tag values that come from all link response buffers and based on the identifier stored at the link master queue it allows the appropriate link response buffer to send its FLITs back to the host. The *physical layer* is also split into two other sub-modules: *physical tx* and *physical rx*. While the *physical tx* sub-module reads FLITs stored in each link queue and serializes them into chunks of 16 bits, the *physical rx* sub-module reads chunks of 16 bits of data that come from the *Responser* module and reconstructs them into whole 128-bit width FLITs.

The primary module inside the *link master* sub-layer is the *packet generator*. Figure 4.3 illustrates the *packet generator* internal organization. The module works as follows. First, when a change happens in either the *address*, *cmd*, *data*, or *cub_id* input, the *prc_flit_generator()* method is launched. This method will first create the *header* and *tail* portions of the FLIT related to the new request using the *create_header()* and *create_tail()* functions. In the former one, the *decode_cmd()* function is called, so the number of FLITs necessary by this request can be generated appropriately. After that, the *generate_tag()*

Figure 4.4: Block diagram of the link slave module.



Source: Provided by the author.

function creates a unique tag identifier to the request. This tag is then stored into the global tag buffer. When creating the *tail* part of the packet, the *link_to_send()* function is called to indicate the path the request should follow through one of the four links. This decision is made by selecting the destination link in a round-robin fashion. Finally, in the case of a WRITE request, the appropriate chunks of input data are inserted into the FLITs, and then the packet[1] can be stored in the local buffer related to the chosen transition link. The *prc_send_flit_to_physical_tx()* method runs at all positive edges of the clock, and is responsible for reading the link's request buffer, and writing the request at the top of the buffer in the *flit_out* port, and also rising the *flit_available* signal. To conclude, the *prc_check_tag_ordering()* reads the global tag buffer and writes its value at the *request_global_tag* signal during all positive clock edges.

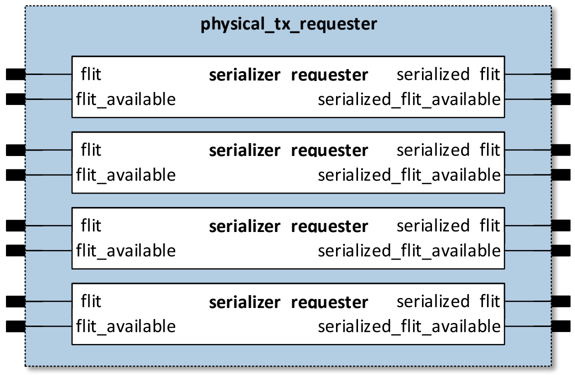
In the other hand, as illustrated by Figure 4.4, the *link slave* module has two submodules, the *reorder_response*, and the *mux_4x1_w_set_bit*. The first one is an Finite State Machine (FSM) that keeps track of the order which the *link master* module generated packets, and based on that, selects the appropriate response packet to send back to the host. To do so, it reads the tag at the top of the global tag buffer at the *link master* module and

[1]We assume during the rest of the text that a packet is equal to the set of FLITs generated.

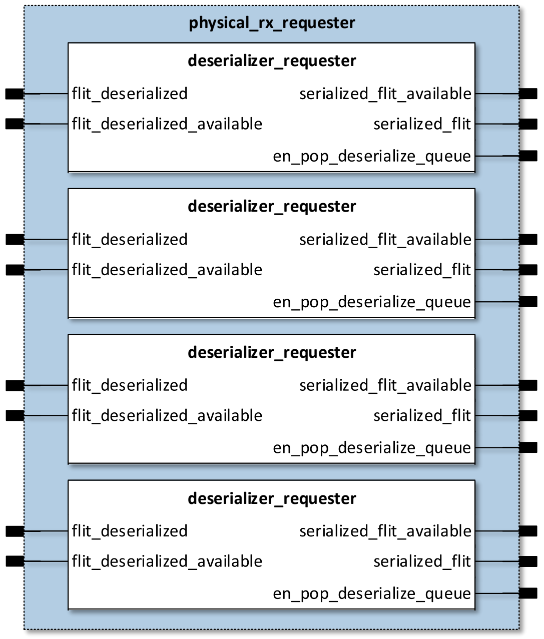
the available FLITs that come from the downstream traffic. If any of the four links have a response packet with a tag field matching the one at the global tag buffer, the multiplexer is set to this link, and the path between link and host is locked until all FLITs from the response packet have been sent. When this process is concluded, the *reorder_request* sends a pop signal to the *link master* module so that the next global tag can be output.

Figure 4.5 depicts the *physical layer* module. This module is built using two sub-modules, a serializer *(physical tx*) and a de-serializer module (*physical rx*). Each link has its own set of serializer/de-serializer modules. The serializer (Figure 4.5a) splits the 128-bit FLIT request into eight chunks of 16 bits and then transmits the fragmented pieces to the high-speed serial lanes. All four serializers can work in parallel, i.e., sending different requests at the same time, but only one request can navigate through a link per time. Therefore, one link must conclude to service (transmit) all FLITs of a request before switching to another request. On the other hand, the de-serializer (Figure 4.5b) receives chunks of 16 bits of data and reconstructs the FLIT so that it can be sent to the *transaction and link layer*.

Figure 4.5: Block diagram of the physical layer.



(a) Physical tx module.



(b) Physical rx module.

Source: Provided by the author.

4.1.2 Responser

Similarly to the *Requester* module, the *Responser* module has two internal sub-modules, the *physical and transaction* and *link layers*. The *physical layer* has the same functionality as the *Requester*'s one. It serializes FLITs that come from the *Vaults* and de-serialize the ones received from the *Requester*.

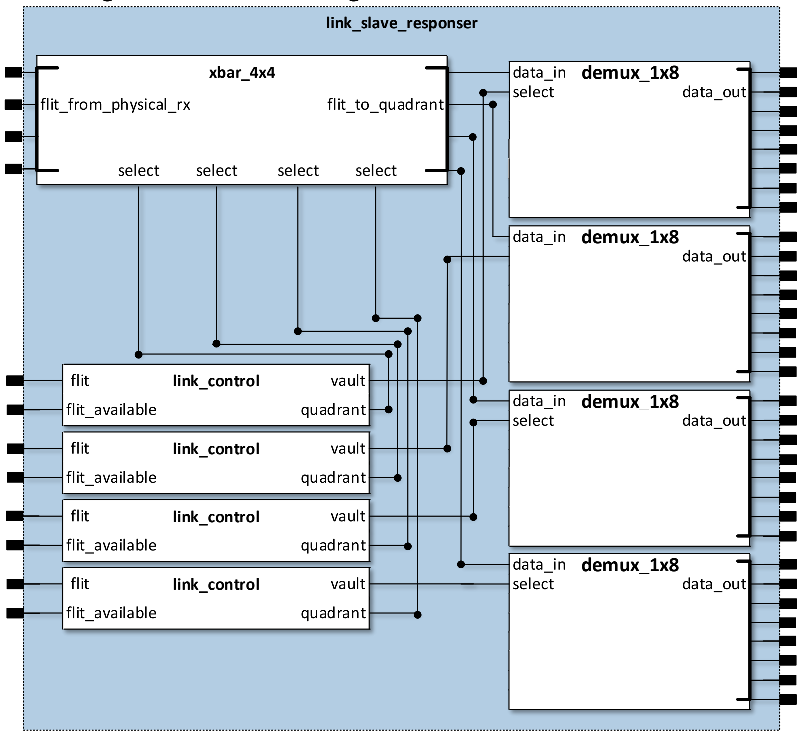
Figure 4.6 illustrates the *link slave* organization. The module is composed of a four by four crossbar network, four one by eight de-multiplexers, and the link control. The interconnection between the four serial links and the 32 *Vaults* is a critical performance bottleneck for an HMC device. Potentially, all four links will be sending data for all the 32 *Vaults*, and vice-versa, creating a producer–consumer scenario. Ideally, this interconnection would have low-latency to transmit packets, would be no-blocking, and also would be scalable in size. However, it is not possible to always address these three characteristics at the same time. For example, a non-blocking interconnection network, as a crossbar network, does not scale well in size [2]. Meanwhile, a smaller and scalable interconnection network, as a butterfly networks, will potentially generate conflicts between accesses. Azarkhish et. al. (2014) have addressed this problem by proposing to employ a Network-on-Chip (NoC) as the interconnection network between *Vaults* and Links. The proposed NoC was based on the AMBA AXI-4.0 standard (AMBA, 2003). Even though this solution provided a non-blocking and scalable interface, it added extra latency to the *Transceiver* mechanism.

To cope with this problem, we have decided to implement a two-level interconnection network. The first level connects the four links to four quadrants using the 4x4 crossbar switch. As specified by (Hybrid Memory Cube Consortium, 2013), a quadrant is a group of eight *Vaults*. Then, each quadrant is connected to the *Vaults* using either an 8x1 multiplexer (in the downstream flow) or a 1x8 de-multiplexer (in the upstream flow). Consequentially, we can archive a low-latency and non-blocking communication at the first level, and a low-latency and blocking communication at the second level. Regarding scalability, the crossbar switch now grows with the number of quadrants, in contrast to a fully-connected 4x32 crossbar that would grow in terms of the number of *Vaults*.

The *link control* is responsible for managing the interconnection network and works as follows. At the *link slave*, when the *physical layer* sends a new FLIT, the *link control* reads the packet's header to obtain the address, and the number o FLITs related to

[2]In fact, a crossbar network with N inputs and M outputs will require $O(NM)$ switches.

Figure 4.6: Block diagram of the link slave module



Source: Provided by the author.

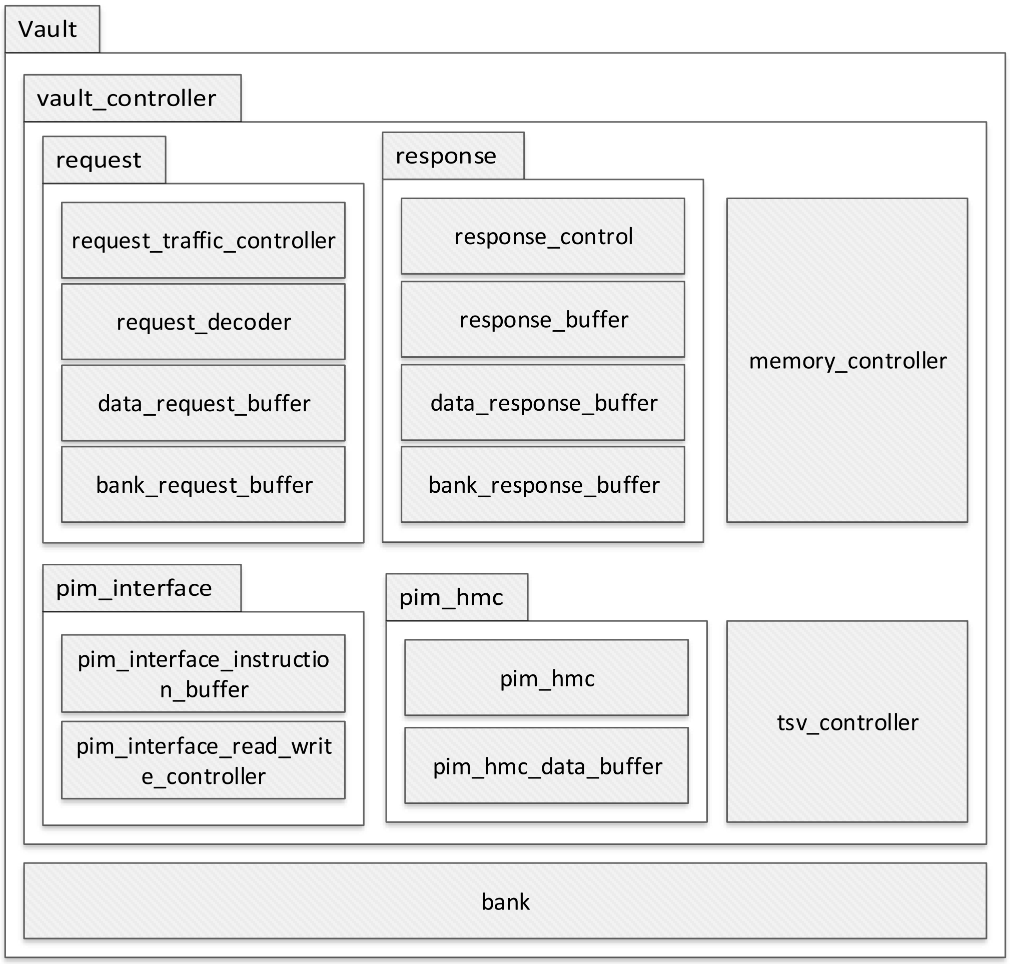
the request. The address is important because it tells the destination *Vault*, based on the interleaving being employed. The number of FLITs is required because the *link control* must ensure that all FLITs from the request are sent prior servicing a new request. Then, it sets the crossbar network and the de-multiplexer based on the obtained *Vault* address. Besides that, it stores in the *tag_per_link* queue the tag related to the request.

At the *link master* (downstream flow), the *link control* reads the FLITs available from the *Vaults*, checking if there is a response tag that matches the tag at the top of the *tag_per_link* queue, therefore ensuring ordering between response packets within a quadrant. If the required response packet is ready to be sent, the *link control* sets the multiplexer and the crossbar to close the path between the *Vault* and the *physical layer*. At the end of the process, the *link control* sends a pop signal to remove the current tag at the top of the *tag_per_link* queue, allowing the next request can be serviced.

4.2 Vault

Figure 4.7 shows the UML packet diagram of the *Vault* module. The *Vault* module is similar in functionality to Double Data Rate (DDR) Dynamic Random Access Memory (DRAM)'s memory controller. However, due to intrinsic characteristics of 3D-

Figure 4.7: Simplified packet diagram of the Vault Layer.

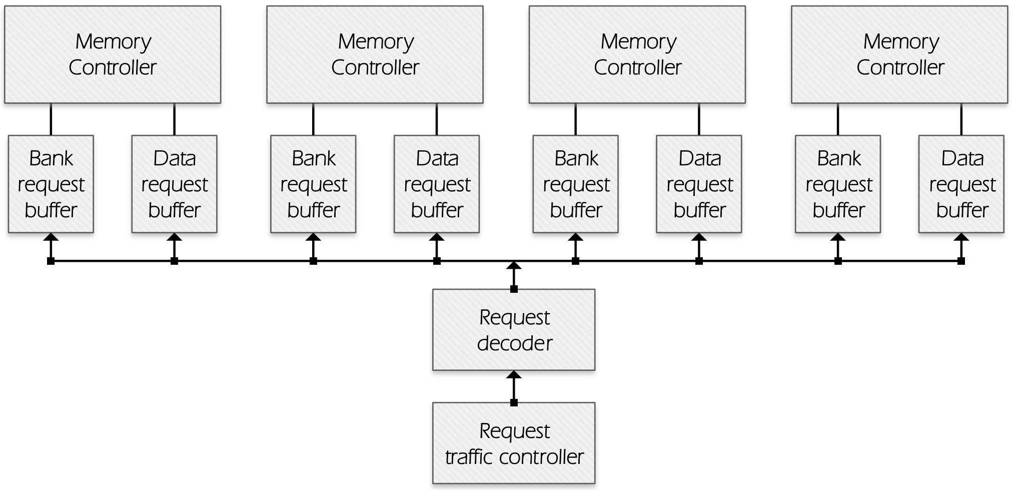


Source: Provided by the author.

stacked memories and also to respect the HMC protocol, diverse modifications to a simple DRAM's memory controller must be employed. The *Vault* is divided into *Vault Controller* and *Bank*. The first one has the following sub-modules: *request*, *response*, *memory controller*, *Through-Silicon Via (TSV) controller*, *PIM HMC*, and *PIM interface*. The *PIM interface* is the principal contribution of this work. The last one implements the functionalities of a DRAM bank.

In the following sections we discuss all the sub-modules present in the *Vault*. In contrast to the *Transceiver* module, the HMC Consortium (Hybrid Memory Cube Consortium, 2013) does not give information related to the *Vault* internal organization, TSV width and operational frequency, the number of banks that can be opened per time, and native PIM instructions' implementation. To overcome this issue, we have gathered information related to the *Vault* implementation from previously published works, in special from (AZARKHISH et al., 2016b), (MATHEW et al., 2017), (ROSENFELD, 2014), and (LEE et al., 2016). From (AZARKHISH et al., 2016b), we could obtain a general view

Figure 4.8: Block diagram of the Vault Request module.



Source: Provided by the author.

of the *Vault* organization, and how the elements of the module correlate to traditional memory controllers. From (MATHEW et al., 2017), we gather the timing parameters and operational frequency for the DDR memories, and also the *Vault* operational frequency. From (ROSENFELD, 2014), we obtained information about the expected memory bandwidth for diverse set of *Vault* configurations. Finally, from (LEE et al., 2016), we extracted information about how the TSV is expected to work.

4.2.1 Request

Figure 4.8 shows the block diagram of the *Vault* Request module. This module is composed of the *request_traffic_controller*, *request_decoder*, *bank_request_buffer*, and *data_request_buffer* sub-modules. The *request_traffic_controller* is responsible for managing the stream of FLITs coming from the host and from the PIM Interface (we give more information about the PIM Interface in Section 4.2.5). Inside the module, an arbiter decides which request's flow can send its FLITs to the *Vault* in a round-robin fashion. Therefore, if only requests from the host-side or only from the PIM Interface-side of the flow are coming to the *Vault*, the *request_traffic_controller* just forwards the FLITs to the *Vault* without checking the arbiter mechanism. On the other hand, if requests are coming from both host and PIM Interface, the *request_traffic_controller* checks the round-robing counter and forwards the request of the flow that has priority. At the end of the traffic, the module increments the round-robin counter and serves the next requests.

The *request_decoder* module is the primary component of the *Vault*'s request

Table 4.1: Description of th request_decoder module, its input/output ports, methods, and functions.

Inputs	Outputs		Methods	Functions
start_decoding	data_from_packet	en_bank_data_buffer	prc_decode	header_decoder
request	tag	en_bank_response_buffer		tail_decoder
	link	en_response_buffer		decode_address
	cub_id	packet_to_pim_interface		decode_cmd
	response_lng	en_pim_instruction_buffer		bank_policy
	cmd	address_offset		operation_size
	pim_instruction	alu_operation		
	operation	bank_id		
	operation	address		
	response_dst	en_bank_operation_buffer		

Source: Provided by the author.

block. It reads the upcoming FLITs and translates the request into READ/WRITE commands, which the memory controller can understand. The module also schedules the request to the appropriate memory controller, aiming to explore the maximum bank-level parallelism by avoiding bank conflicts. Also, it stores upcoming data from the request into the bank's data request buffer. Finally, it sends important package-information that comes with the request to the response module.

Table 4.1 specifies the internal organization of the *request_decoder* module, its set of input and output ports, methods, and functions. The module works as follows. During all rising edges of the clock, the *prc_decode()* method checks if the *start_decoding* signal was raised. When the signal is set, the module starts decoding the upcoming flow of FLITs. To keep track of the number of FLITs related to the request, the module has an internal counter, which is initialized with zero during reset. If the counter is equal to zero during decoding, it means that the module is dealing with the first FLIT of the request. Then, the *header_decoder()* function is called to split the header's fields into command opcode, tag, the number of FLITs within the request, address, and cube. With the command opcode, the module can call the *decode_cmd()* function to translate the HMC instruction into simple READ/WRITE/ATOMIC commands, obtain the size of the operation, and the number of response FLITs that the response module will generate when the request process concludes. Also, the *header_decode()* calls the *decode_address()* function to obtain the bank address targeted by the current request, the address within the bank, and starting position data should be read/write in the row buffer.

When the *header_decode()* function returns, the *bank_policy()* function is called to decide which memory controller should manage the current request. This decision is made based on the target bank from the request, and the banks that are currently being serviced by the memory controllers. If any memory controller is servicing the same bank targeted by the current request, the request is sent to this memory controller. Otherwise, the request is sent to the next memory controller in a round-robin fashion. To simplify our simulator's design, we decided to correlate the number of memory controllers that can be operating in parallel with the tFAW DRAM timing parameters. This parameter specifies the number of rows that can be kept open at the same time, without overheating the chip. Since the HMC row policy is defined as close-row, there is no row buffer parallelism to explore. Therefore, it is possible to implement several memory controllers working in different banks in parallel, instead of a single memory controller managing several banks without losing performance because the memory controllers do not need to share row-level information. In today's DDRx modules, the tFAW is up to four. Therefore we implemented four memory controllers into the *Vault*. However, one could potentially keep all 16-row buffers (there are sixteen banks per *Vault*) open at the same time to explore the available bank level parallelism dramatically[3].

Following the request decoding process, after decoding the header and deciding the destination memory controller, it is required to determine if the upper part of the current FLIT represents a portion of data or it is the tail field. If the number of FLITs is one, the higher part of the current FLIT has the tail payload, where the return link id can be obtained. Otherwise, it will be data coming from a WRITE or ATOMIC request. In this case, the *request_decoder* module stores all following data chunks in the respective *data_request_buffer* (each memory controller has it private set of bank operations and data request buffers) until the final FLIT arrives. When this happens, the returning link id can be extracted, and finally, all pieces of information related to the request were obtained. Then, the request command is stored in the *bank_request_buffer*, and the memory controller can start working.

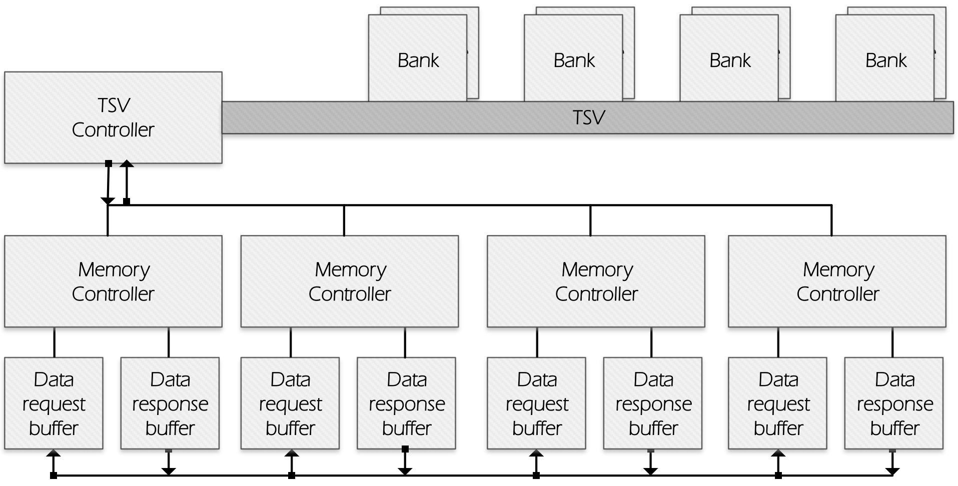
[3]For a more detailed exploration of the available bank level parallelism in HMC devices, we conduct the reader to (ROSENFELD, 2014).

4.2.2 Memory and TSV Controllers

Figure 4.9 illustrates the memory controller and the TSV controller modules. The memory controller reads the current request at the top of the *bank_request_buffer* and generates a sequence of DDR commands that are sent to the memory banks via the TSV bus. The module is implemented as an FSM. When a new request is available, the memory controller reads the target bank address, the operation (READ/WRITE/ATOMIC), the operation size, the memory address, and the memory offset from the buffer into local registers. The memory controller follows the same path as presented in Section 2.1.3. Then, it decodes the bank id and sends the CS command to the destination bank. After that, the memory controller issues a RAS command followed by the row address to bring data from the rows to the row buffer at the selected bank. After the row-to-column delay (tRCD) has passed, the memory controller can start reading/writing data from/to the banks. Then, the memory controller first selects the portion of data within the row buffer that is related to the request by issuing a CAS command, following the column address. At this point, the memory controller can start transferring data to the bank, in the case of a WRITE request (issuing a WE command), or from the bank in the event of a READ request (issuing an OE command). The memory offset value is used to indicate the start position within the row buffer the operation should start being performed. This value is incremented by one until the total request size has been performed. Since the TSV width is limited, potentially several chunks of data need to be transmitted between the memory banks and the memory controller's queues (up to 8 transmissions to move 256 bytes of data). Data from a READ request is stored into the *data_response_buffer*, while data from a WRITE request is read from the *data_request_buffer*. The memory controller controls both buffers.

The TSV is a bi-directional vertical bus with a width of 32 bytes that connects the memory banks to the *Vault Controller*. Since the memory controllers work in parallel and do not share requests' information between them, there must be a module that controls the flow of data and commands through the bus. In consequence, the TSV controller is responsible for monitoring the TSV traffic. The module is connected to all memory controllers, which share information regarding the requested bank address, the size of the request, and the operation being performed. The module implements a round-robin mechanism to select which memory controller has the ownership of the bus. Also, based on the current memory operation being executed, the controller selects the appropriate direction of the bus, allowing data to navigate to the bank or from the bank.

Figure 4.9: Block diagram of the Memory and TSV controller modules.

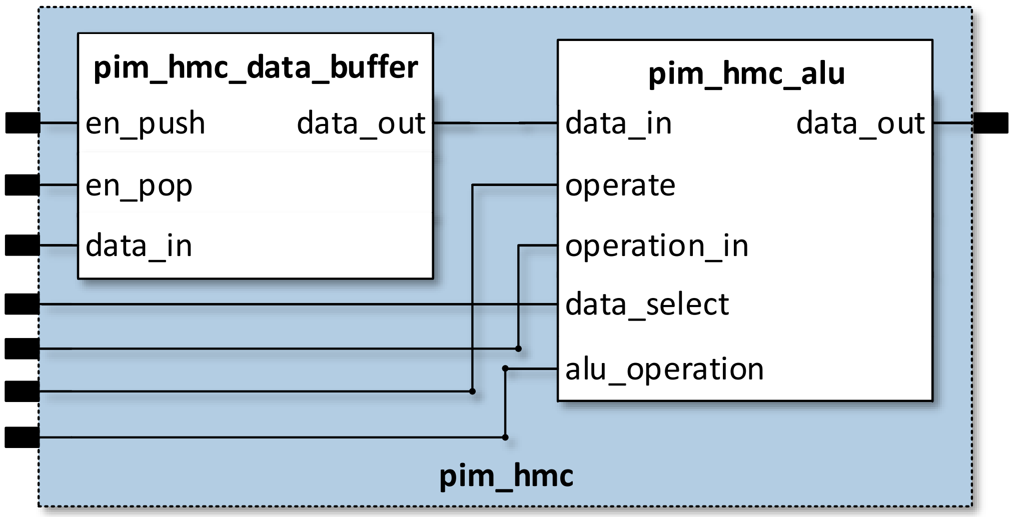


Source: Provided by the author.

4.2.3 Native PIM

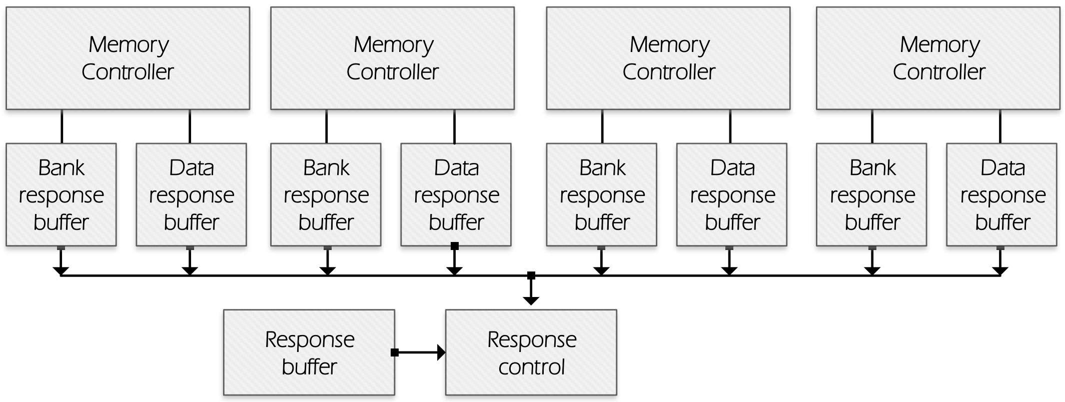
As previously mentioned, the HMC device is natively capable of executing some arithmetic operations, like addition and bitwise logic operations. All these operations are performed over 8 bytes or 16 bytes of data. Also, they are atomic read-modify-operations, meaning that they operate over a single memory address. To provide these capabilities, we implemented a simple Arithmetic Logic Unit (ALU), which is controlled by one of the memory controllers. Therefore, all native PIM requests are sent to the same memory controller by the *request_decoder* module.

Figure 4.10: Block diagram of native PIM module.



Source: Provided by the author.

Figure 4.11: Block diagram of the Vault Response module.



Source: Provided by the author.

Figure 4.10 illustrates the *pim_hmc* module, which is composed of a local data buffer and a simple ALU. Any native PIM instruction follows three steps during their execution. First, when the instruction arrives at the memory controller, the module issues a READ command to the bank. However, instead of storing the read data into the *data_bank_buffer*, the data is sent to the *pim_hmc_data_buffer*. Since the maximum size of a native PIM instruction is 16 bytes, the *pim_hmc_data_buffer* has this same length. Second, the memory controller issues an execute command to the *pim_hmc module* by setting the *operate* signal to high. With this signal, the memory controller also sends the required ALU operation that will be performed, the portion of the data from the *data_buffer* that should be read (in a case of an eight bytes request), and any immediate value that came with the request package. Finally, after executing the atomic instruction, the memory controller sends a WRITE command bank to the bank, with the result data from the operation. Different from a standard WRITE request, instead of sending data from the *bank_request_data_buffer* to the bank, the memory controller sends the resulted computation that the *pim_hmc* module outputs back to the memory bank.

4.2.4 Response

Figure 4.11 illustrates the *Vault* response mechanism and its internal modules. The *response_control* module is the principal component of the response mechanism, being responsible for scheduling response packets based on the request order, encapsulating the response packet into FLITs, and sending the FLITs back to the *Transceiver* layer.

Table 4.2 describes the internal set of input/output ports, methods, and functions present at the *response_control* module. The module works as follows. First, the *prc_response_control()* method is called during all positive edges of the clock. This method implements the FSM that controls the scheduling response mechanism. The FSM works as follows. First, at the IDLE state, the method reads the the *tag_from_rsp_buffer* and the *is_pim_instruction* fields at the top of the response buffer. The first signal tells the current tag that must be serviced by the response mechanism, and the second indicates if this tag is related to a custom PIM instruction. All this information was stored at the *response_buffer* by the *request_decoder* module at the moment the request packet arrived at the *Vault Controller*. If the current response tag is related to a custom PIM instruction, the FSM can follow to the next state. Otherwise, it needs to check the tag at the top of each *bank_response_buffer* to identify which memory controller is servicing the request related to the current tag. After that, the FSM needs to wait for this particular memory controller to raise the *bank_status* signal, indicating that it has finished servicing the request. Then, the FSM reads relevant pieces of information from the specific memory controller *bank_response_buffer*, as the number of FLITs within the response packet, the link the response must travel, the cub id of the response, and the command that came with the request packet. After that, the FSM can raise the *rsp_will_send* signal indicating to the memory controller that its request was chosen to be serviced by the response module. Finally, the FSM transitions from IDLE to the SENDING state. At this state, the FSM starts encapsulating the response packet. First, it creates the header portion of the response FLITs. If the number of response FLITs is one, it also creates the tail portion of the FLIT and stores the produced FLIT into the local *host_response_buffer*. Otherwise, it needs to read the response data from the specific *data_response_buffer*, encapsulate it into the FLIT, and store the produced FLIT into the *host_response_buffer*. The process of reading data from the *data_response_buffer* is repeated until the number of response FLITs of the response was obtained. When that happens, the FSM can go to the FINISHED state, when it sends a pop signal to the response buffer, and the next response tag can be serviced. This described process is performed when the *response_dst* input signal is equal to zero, indicating that the request packet came (and must return) to the host. If this signal is equal to one, the request came (and must return) from the PIM Interface. In this case, the buffer that the FSM stores the produced FLITs would be the *pim_response_buffer*. At the end of the FINISHED state, the FSM return to the IDLE state.

The *prc_response_control_read_host()* method runs at all positive edges of the

Table 4.2: Description of th response_control module, its input/output ports, methods, and functions.

Inputs	Outputs	Methods	Functions
bank_status	packet_out_host	prc_response_control	creat_header
en_pop	packet_out_pim	prc_response_control_read_host	creat_tail
en_pop_from_pim	packet_available_host	prc_response_control_read_pim	creat_data
is_pim_instruction	packet_available_pim		creat_response_packet
tag_from_bank_rsp_buffer	en_pop_bank_response_buffer		
tag_from_rsp_buffer	rsp_will_send		
data	en_pop_data_buffer		
link_from_pim_rqst	en_pop_response_buffer		
response_dst			
link			
cub			
response_lng			
cmd			

Source: Provided by the author.

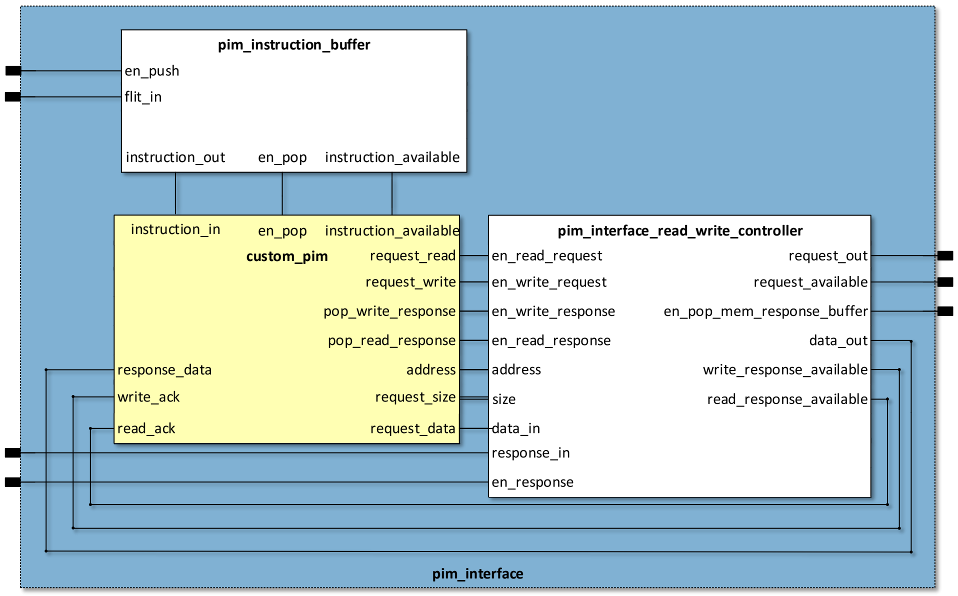
Transceiver clock. This method is responsible for reading the FLIT at the top of the *host_response_buffer*, and writing its value to the *packet_out_host* output signal. It also raises the *packet_available_host* signal, indicating that there is a new response FLIT available to the *Transceiver* Layer. On the other hand, the *prc_response_control_read_pim()* method runs at all positive edges of the PIM Interface clock. It has the same functionality of the *prc_response_control_read_host()* module, but it reads the FLITs from the *pim_response_buffer*, and sets the *packet_out_pim* and *packet_available_pim* output ports.

4.2.5 PIM Interface

The primary goal of this work is to provide a simple yet concise infrastructure that allows new PIM architecture exploration. To do so, we have implemented the HMC design as previously described, and studied a way to insert a custom processing module into HMC without having to worry about HMC organization. We came to a simple PIM Interface, placed into the *Vault Controller*, that receives custom PIM instructions, and also provides a mechanism to perform read/write operations.

To implement the PIM Interface, we decided not to modify dramatically the HMC data-path and its communication protocol. However, we still had to perform four minor modifications to the HMC design. First, we have included a new opcode command to the set of HMC command operations. The opcode included was the PIM_INSTRUCTION.

Figure 4.12: Block diagram of the proposed PIM Interface.



Source: Provided by the author.

All custom PIM instructions will make use of this same opcode. A single PIM request is seen by our simulator similarly to a 16 bytes write request. All information related to the custom PIM instruction will be placed in the data sections of the request packet. Therefore, the user does not need to worry about violating already reserved opcodes or being limited by the number of free available HMC opcodes when designing his or her instruction set. Second, we have included the already mentioned *traffic_request_controller* module. This module is required to allow a request flow to come from both the host and from the PIM Interface. Third, we have made use of a reserved bit from the request header field (bit 23), to indicate to the *request_controller* whether the current READ/WRITE request should return to the host or the PIM Interface. Finally, we have included a path between the request decoder and the PIM Interface, therefore FLITs related to a PIM request can be sent to the PIM Interface.

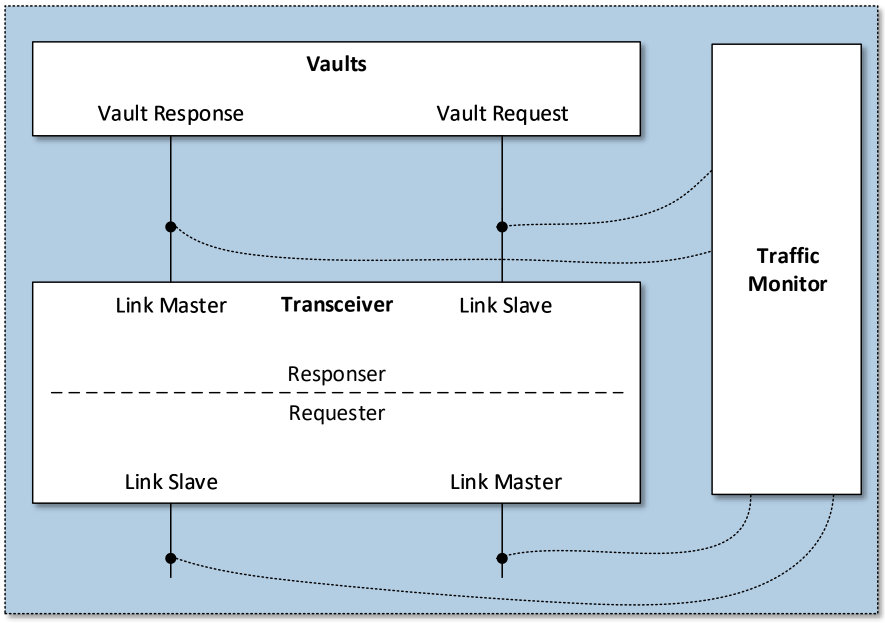
Figure 4.12 depicts the proposed PIM user interface. The module is composed of the *pim_instruction_buffer* and the *pim_interface_read_write_controller*. The process of performing custom PIM operations works as follows. First, when issuing a new request to the *Transceiver* module, the user must specify the PIM_INSTRUCTION opcode, the address of the request, and then a 16 bytes of data that represents their custom instruction. The *packet_generator* will encapsulate this request the same way it makes a write request. After that, the produced FLITs will travel through the same path as a standard request until arriving at the *request_decoder*. At this point, when the *request_decoder* de-codifies the command portion of the request, it will not recognize the opcode, and then will send the upcoming FLITs to the *pim_instruction_buffer* at the PIM Interface. The *pim_instruction_buffer* will then remove additional payloads of the FLITs (as header and tail fields), and will raise the *instruction_available* output signal to notify the *custom_pim* module that there is a new instruction currently available to it. The *custom_pim* does not need to have any information about how the packet was encapsulated since the upcoming PIM instructions have only information related to its own instruction set.

When the *custom_pim* instruction decides to perform a READ or WRITE operation to the HMC module, it sets the *request_read* or *request_write* signal, informing to the *pim_interface_read_write_controller* that a new HMC request must be generated. The *custom_pim* also provides the address it wants to access, the size of the request, and data values in the case of a WRITE operation. The controller will then encapsulate the PIM request into FLITs, creating a header and tail field in accord with the HMC protocol, and then transmitting the generated request back to the *Vault Controller*. Also, it will set bit 23 of the header field to one, indicating that the response package must return to the PIM Interface. The request will be sent to the *traffic_request_monitor*, and when appropriated, it will be sent to the *request_decoder*. Once at the *Vault Controller*, the request will follow the same path as any standard host request. However, once the request goes through the *response_control*, the module will check that the *rsp_destination* signal is set to one (recall this signal stores the value of bit 23 of the header), and then all response FLITs will be sent back to the PIM Interface. Finally, when the PIM Interface receives the response packets from the *Vault Controller*, it will notify the *custom_pim* module that its READ/WRITE request is completed by raising the *write_ack* or *read_ack* signals. In the case of a READ request, the PIM Interface removes the header and tail payloads of the response package, and send the requested data to the *custom_pim*.

4.2.6 Bank

The Bank module replicates the behavior of a DRAM bank, as described in Section 2.1. The module receives the CS, RAS, CAS, OE, WE signal from the memory controller. The memory controller also sends the request address and the row buffer offset to the bank. Finally, data can traffic through *the Vault Controller* and the banks, and vice-versa.

Figure 4.13: Block diagram of the Traffic Monitor class.



Source: Provided by the author.

Each HMC bank stores 16 Mbytes of data (there are 16 banks per *Vaults*, and 32 *Vaults*, summing up 8 Gbytes of memory). To reduce the amount of data that the simulator has to allocate to each bank, we have implemented the 2D grid of DRAM cells using the C++ standard hash data structure. In this way, we could reduce the amount of memory that our simulator will allocate during simulation. In other words, our simulator will only allocate all 8 Gbytes of memory when all bank addresses, from all banks, in all *Vaults* are currently being accessed. To perform a READ/WRITE operation, the bank module goes through the following states. First, when a CS signal arrives, the bank will decode this signal to check if the upcoming request targets itself. If it does, the bank will go to the activated state. Second, when the RAS signal is received, the bank loads the data stored in the hash in the request address into the row buffer. Third, when the CAS signal arrives, the bank will select the section of the row buffer related to the received column address. Fourth, when receiving a WE signal, the bank reads data from the TSV bus and writes it back to the row buffer. If it receives an OE signal, it writes the row buffer value to the TSV bus. Finally, at the arrival of a DONE signal, the bank writes back the row buffer value at the hash.

4.3 Traffic Monitor

The Traffic Monitor class is responsible for tracking the request and response flux of packets in both *Transceiver* and *Vault* modules. Figure 4.13 illustrates how the traffic monitor is connected to the system. The class has internal variable counters that store the number of packets transmitted, the total number of bytes written/read by the host, the total number of READ/WRITE instructions, and the total time spent during run-time.[4] To connect the traffic monitor to the rest of the HMC design, a pointer to the class is stored in some key *hmc_device* modules. These modules are the *link_master* and *link_slave* modules of both Requester and Responder Layers. Also, we connected each *Vault*'s request and response modules to the traffic monitor class. In this way, we can check the external Link bandwidth and internal *Vault* bandwidth.

The monitoring process works as follows. When creating a new request packet, the Requester Layer informs the Traffic Monitor the size of the request in bytes and its type (READ/WRITE). If this is the first package being created, the Traffic Monitor sets its timing counter. Then, every time the request passes through one of the modules that have a Traffic Monitor pointer, it writes the current timing stamp back to the Traffic Monitor. The timing stamp depends on the clock of the module. Also, it can depend on specific timing parameters (as the DDR timings). To end the simulation, the Traffic Monitor also checks if and *request_global_tag* buffer at the Request Layer is empty, what indicates that all request packets have already been returned to the host. Also, it checks if the *custom_pim* is not executing any instruction. If both conditions are true, the Traffic Monitor can end the simulation and finish to count the execution run-time.

[4]It is important to differentiate the total simulation time and the final run-time. The former one tells the time the simulator spent running on the user machine. The latter informs the time the proposed HMC model took to service all host requests.

5 SIMULATION MECHANISM

In this section, we explain the steps to produce a simulation using the proposed framework. There are eight important files related to the simulation process. First, since our simulator is trace-based, we provided some scripts that can produce valid trace files to feed the simulator. There are four distinct trace scripts available. The *trace-generator-seq.py* script generates READ/WRITE requests that accesses memory sequentially. The *trace-generator-pim-seq.py* script also generates a trace file that access memory sequentially, but creates native PIM HMC requests. On the other hand, both *trace-generator-rand.py* and *trace-generator-pim-rand.py* scripts generate READ/WRITE and PIM HMC requests, respectively, that access memory randomly. The trace file generated has the memory address, HMC command, cub_id, and in the case of a WRITE request, the input data. Besides that, the defines configuration file specifies some parameterizable HMC values. The user can modify the number of Vaults, the number of links, the row buffer size, the TSV width, the number of memory controllers, the DRAM's timing parameters, and the Vault and Transceiver clock rate in the defines file. Meanwhile, the *hmc-testbench* file drives and monitors the *hmc-device* module. As previously mentioned, the *traffic-monitor* gathers run-time statistics. The *hmc-device* is the implementation of the HMC architecture. Finally, the simulator outputs the simulation results and simulation statistics files.

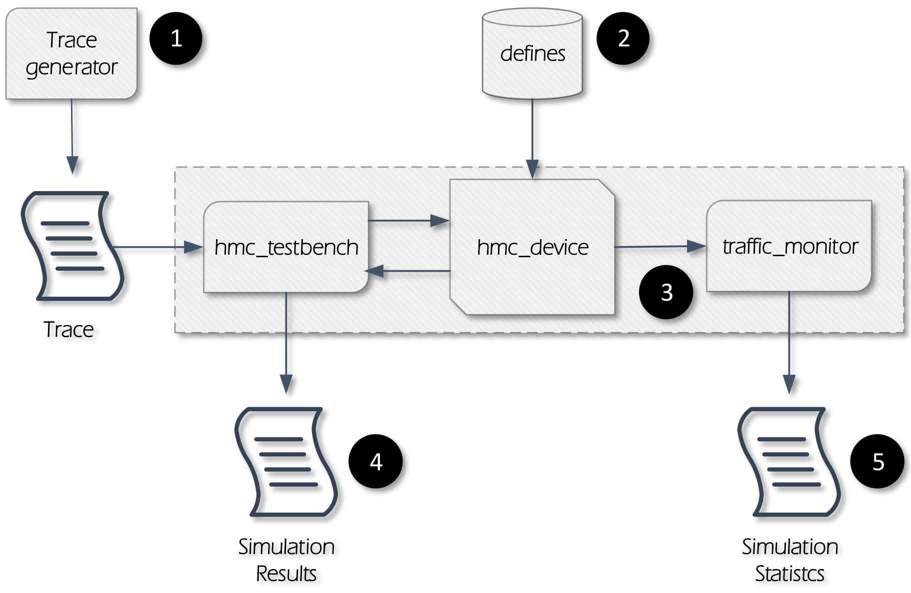
Figure 5.1 illustrates the simulating steps that CLAPPS goes through during execution. In ❶, the user generates the trace file that will be used during simulation. The trace file can be produced using any of the available trace generator scripts. However, if the user decides to evaluate the memory execution of a real benchmark, he or she needs to convert the benchmark's memory access pattern to the simulator's trace format. Each line of the trace file has the HMC opcode, the memory address, the number of data FLITs in the request, and the list of data itself. For example, in the following line of a trace file,

0001001 0000000000000000000000000000000000 2 7 9

the first seven bits (0001001) indicate the operation is a WRITE request of 32 bytes, the following 34 bits indicate the target memory address, then the third portion indicates that the list of input data has two elements, and then 7 and 9 are the input data elements themselves.

In ❷, the user can modify the standard HMC structure by changing the define file. Then, he or she needs to compile the simulator. Since SystemC is based on C++, the

Figure 5.1: Simulation steps our mechanism travels through execution.



Source: Provided by the author.

only library the user needs to provide and link during compilation is the SystemC 2.3.1. one. When the compiling and linking process has finished, the user can run the simulator. 3 describes the run-time process. The *hmc-testbench* will read the trace file, driving the *hmc-device* with all requests in the trace file. The requests are sent to the *hmc-device* one per clock. Besides that, the *hmc-testbench* will monitor the output port of the *hmc-device*, writing all response packets generated by the latter into the simulation results file 4. To check the correctness of the simulation, the user can compare the simulation result file with the gold file produced by each one of the four trace generator scripts. Finally, at the end of the simulation, the *traffic-monitor* writes the produced statistics to the simulation statistics file 5. This file is divided into per Vault and per Link statistics. In both cases, it is shown the total number of bytes and written, the total time (in ns) the *hmc-device* took to perform all requests, the obtained individual and global bandwidth (in GB/s), and the number of packets produced.

6 EXPERIMENTAL SETUP AND RESULTS

In this section, we will show the simulation potential provided by our mechanism. All our experiments target the entire vault and link bandwidth that we were able to extract during simulation. To simplify the analysis, we have divided our results in three categories: HMC Memory Validation, HMC Atomic Requests, and PIM Implementation. Table 6.1 describes all configured parameters used in all presented results. All these parameters can be modified as needed.

6.1 Memory Validation

Our first concern when building the simulator was to create an architectural HMC design that would also work as a simple memory simulator. However, since there is no public information about the HMC internals or even about the DDRx memories that composes the design, we have used as an implementation guide published related work, in special the work presented by (ROSENFELD, 2014).

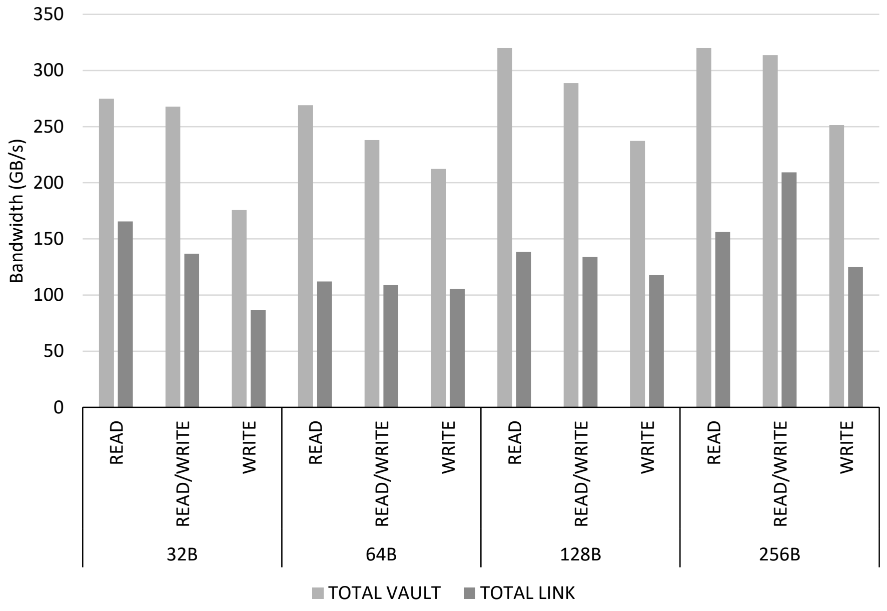
First, we were interested in investigating if there was a significant performance different when comparing read, write, and read+write requests, as cited in (ROSENFELD, 2014). Therefore, we have run 8K read and write requests for various row buffer sizes. All the requests match the row buffer size. Therefore we could extract maximum performance from the current simulation. We have used a 50% read/write ratio in our testbench.

Table 6.1: HMC configuration.

Number of Vaults	32
Number of Links	4
Banks/Vault	16
Lane Bandwidth	30 Gbs
Memory Size	8 GB
Burst Width	8B
Number of DRAM Dies	8
RCD Latency	10.4 ns
DRAM Frequency	166 MHz
Row Buffer	32, 64, 128, 256B
Row Policy	Close-row

Source: Provided by the author.

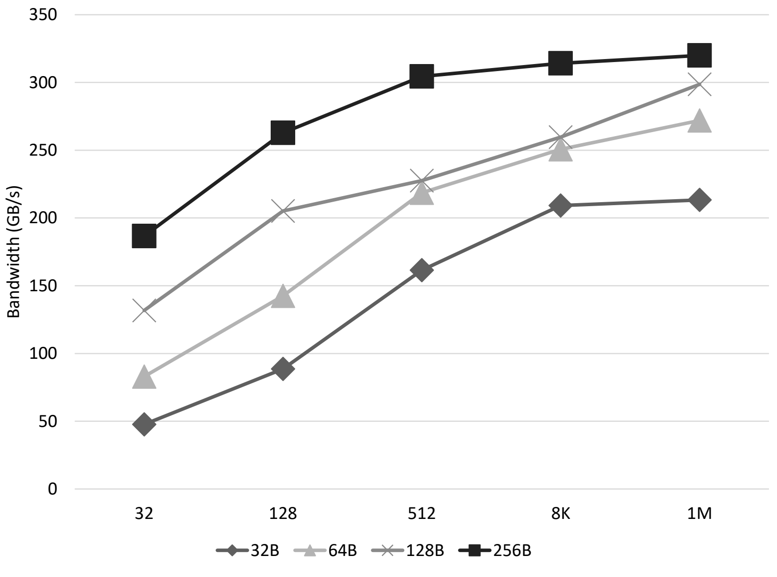
Figure 6.1: Link and Vault total bandwidth for sequential read and write requests.



Source: Provided by the author.

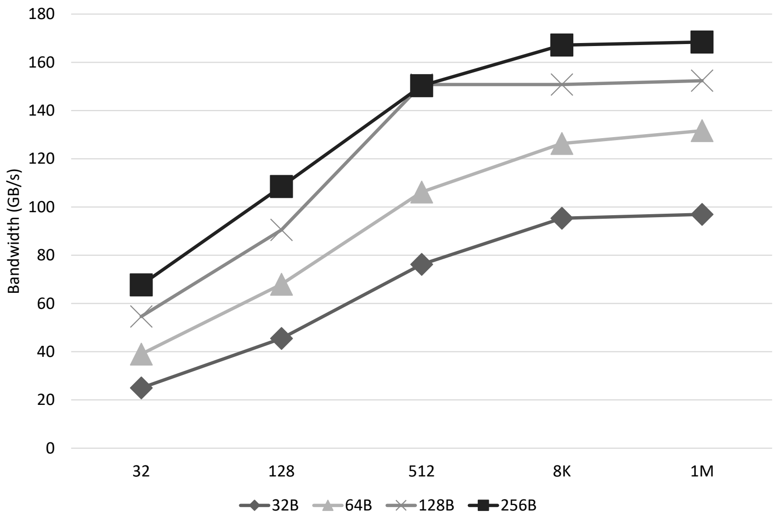
Besides that, the request addresses were generated aiming to evict link/vault/bank conflicts. Figure 6.1 shows the results of the simulation. One can notice that read requests are faster than write requests when considering the total vault bandwidth. That happens because, from the DDR point of view, a read request loads data from the memory cells, stores the data into the row buffer, to then be read out by the memory controller. On the other hand, write requests will generate the same process, and also it will need to write user data to the row buffer, to then, be stored back into the memory cells. To summarize, a

Figure 6.2: Vault total bandwidth for sequential read requests.



Source: Provided by the author.

Figure 6.3: Link total bandwidth for sequential read requests.

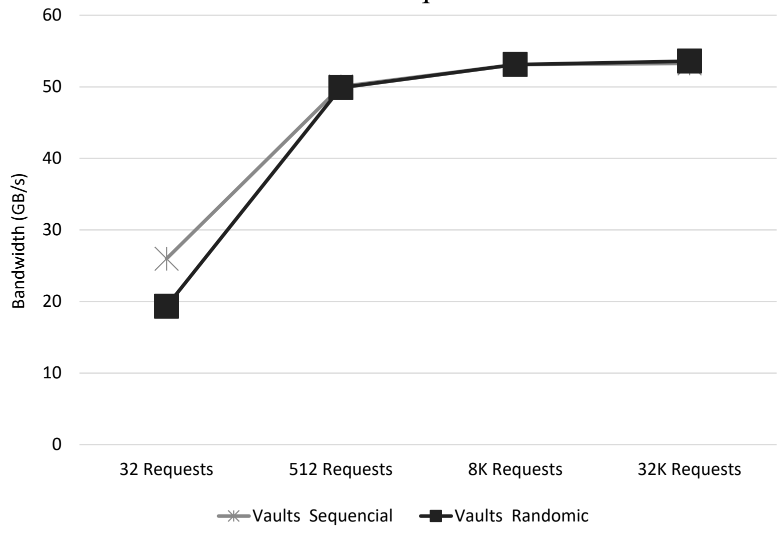


Source: Provided by the author.

write request is slower than a read request because a write command generates a sequence of reading and writing requests. However, this observation is not valid for the links. In general, the maximum link bandwidth will be achieved when a mix up of reading and writing request occurs. That behavior is caused by the fact that the overhead to send the read request to the memory or to send the acknowledge packet back to the host, in the case of a write request. This observation agrees with the one presented in (ROSENFELD, 2014).

Next, we have focused on obtaining the maximum vault bandwidth that our simulator can provide. To do so, we have simulated five different scenarios with 32 to 1M read instructions while varying the row buffer size. This number of requests was chosen because with 32 requests one could measure the bandwidth in the case of only one request per vault; with 512 requests one can understand the available vault parallelism when all banks of all vaults receive a single request; with 1M requests, one could saturate the memory performance. 128 and 8K requests were used as intermediate points. Figure 6.2 and Figure 6.3 show the vault and link performance results respectively. It is evident that the biggest the row buffer size, the better the achieved bandwidth. This result is explained because the vault controller works in a pipeline fashion. Once a bank row buffer has been opened, the only performance limitation would be receiving or sending data from/to the bank unit through the TSV. However, since DDRx memories reply to requests in bursts of data, this extra latency to access the TSV is reduced. The maximum bandwidth CLAPPS could achieve 312 GB/s, with row buffer size of 256B.

Figure 6.4: Vault total bandwidth for sequential and random atomic requests.

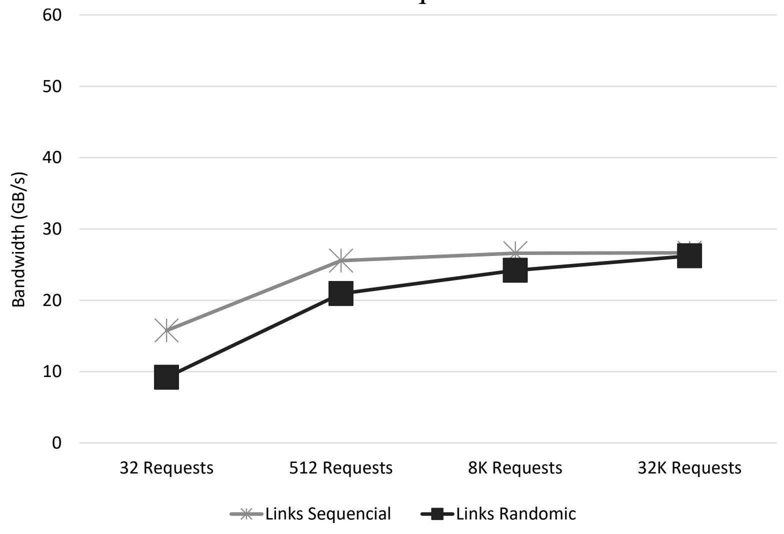


Source: Provided by the author.

6.2 Atomic Requests

With our memory architecture validated, we needed to experiment with the HMC atomic instructions. Since all atomic instructions follow the same data path, they all have the same execution latency. Therefore, all our simulation were based on the add dual 8B instruction. Also, we have generated testbenches with random request addresses. Experimenting with a random set of requests is important because HMC devices were first designed to target applications with sparse data accesses. Figure 6.4 and Figure 6.5 show our simulation results. The results for the random-based requests were obtained

Figure 6.5: Link total bandwidth for sequential and random atomic requests.



Source: Provided by the author.

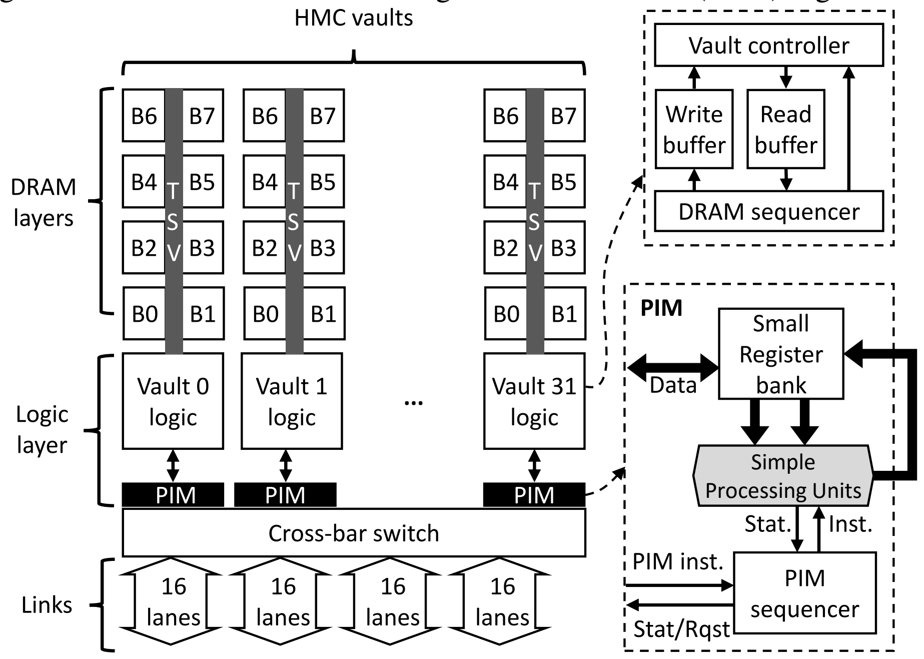
with three different set of inputs. Some observations can be pointed out by these results. The first observation one could make is that atomic operations provide significative lower bandwidth than reading or writing requests. The bandwidth reduction happens because a single atomic request will generate a sequence of reading and write requests. Besides that, only one ALU is available to execute the instruction, therefore limiting the vault parallelism. Secondly, it is possible to notice that unpredicted access patterns do not severely prejudice the bandwidth for random requests.

6.3 Case of Study: PIM Interface

To test the effectiveness and usability of our PIM Interface, we included into CLAPPS, using the provided set of resources, the PIM architecture developed by (SANTOS et al., 2017).

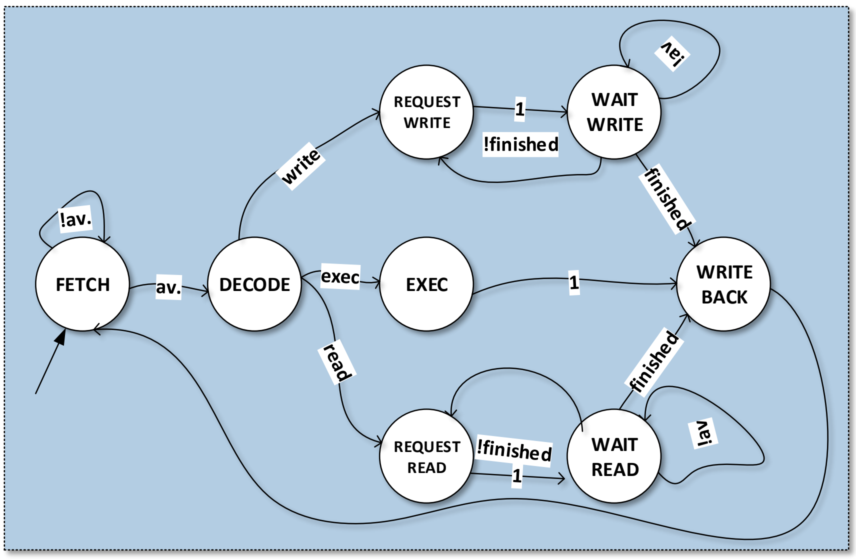
RVU is a reconfigurable accelerator that targets vector operations with varied operand sizes. Figure 6.6 illustrates the internal organization of (SANTOS et al., 2017). It is an improvement over the work proposed by (ALVES et al., 2016) that aims to reduce unnecessary data movement from memory to the accelerator device. One RVU device was inferred inside each vault module. A single accelerator has a set of 8 registers with up to 256B each, 32x64 integer, and floating-points Functional Units (FUs), and operates

Figure 6.6: Overview of the Reconfigurable Vector Unit (RVU) organization.



Source: (SANTOS et al., 2017).

Figure 6.7: RVU's FSM



Source: Provided by the author.

at the same frequency as the vault controller. Besides that, each device can operate over a maximum of 256B at the time; therefore all accelerators together being able to execute an 8KB vector operation at a given time.

To implement the (SANTOS et al., 2017) architecture into our simulator, we included two new files: the *rvu_defines* and the *rvu_implementation*. The first one defines the accelerator's Instruction Set Architecture (ISA), the number of FUs, and the number and width of the register file. The second one is the actual implementation of the proposed RVU architecture. We have developed RVU as a simple FSM, which is illustrated in Figure 6.7. While in the initial state, FETCH, RVU waits until there is a new instruction available. When this happens, RVU reads and stores the instruction into the instruction register, then sending a pop signal to the *pim_interface* to pop the instruction out the *instruction_queue*. Then the FSM goes to the DECODE state, where it decodes the instruction in accord to its ISA defined at the *rvu_defines* file. Once the instruction has been decoded, the FSM can go to either the REQUEST_READ, REQUEST_WRITE, or EXEC states, depending on the instruction type. If it goes to the REQUEST_READ, it means data needs to be loaded from memory. To do so, it writes the required memory address, the size of the request, and rises the *en_read_request* signal, indicating to the *pim_interface* that a new request needs to be serviced. Then the FSM goes to the WAIT_READ state, where it waits until the *pim_interface* raises the *read_response_available* signal, indicating that the data from memory is available. When this happens, the machine goes to the

WRITE_BACK state, where it stores the returned data into the register indicated by the instruction, and sends an acknowledge signal back to the *pim_instruction*, indicating it can pop the data out its data queue. This process is repeated until all requested data has been returned from memory and stored in the vectorial register. Then, the FSM goes back to the FETCH state to wait for new instructions. In the case of a WRITE instruction, the FSM passes through the same states. However, it provides the data from a register to be written back to memory. In the EXEC state, RVU performs actual computational, in accord to the instruction received.

According to (SANTOS et al., 2017), when the user configures their mechanism to run over its maximum operation size, it will achieve similar results to the work of (ALVES et al., 2016). Besides that, from all benchmarks executed by (ALVES et al., 2016), the *vec-sum* algorithm provided a maximum bandwidth exploration from memory. Thereby, since our evaluation metric is vaults and links total bandwidth, we have decided to perform this same benchmark in our experiment.

In the results provided by (ALVES et al., 2016), the total vault bandwidth obtained for a *vec-sum* operation was 315.9 GB/s. There is no information in (ALVES et al., 2016) regarding of the total link bandwidth. In our simulation, we have obtained similar results. In total, the vault performance was 317.8 GB/s, and the link performance was 213.36 GB/s.

7 CONCLUSIONS AND FUTURE WORK

In this work, we presented CLAPPS, a generic Cycle Accurate Parallel Processing In Memory Simulator. Our simulator provides an interface to implement PIM architectures targeting new 3D-stacked memories devices, in particular, the HMC architecture. CLAPPS has been built using the SystemC programming language since it can provide the flexibility of a high-level programming language while generating a final design description similar to the one an HDL language would produce. We have demonstrated that our memory model can achieve closer to the total amount of bandwidth cited by the HMC consortium. Moreover, we have shown with a case of study, how our PIM interface can be useful to the final user.

Table 7.1 compares the simulators discussed in Section 3 with the proposed simulator. First of all, it is important to point out that, similar to the other simulators, CLAPPS is also cycle-accurate and implements the native HMC atomic instructions. It provides a simple mechanism to include novel PIM architectures into the simulator while achieving the theoretical HMC peak bandwidth of 320 GB/s. Besides that, CLAPPS deals with parallelism natively by making use of SystemC's concurrent execution model. Finally, our simulator is fully parameterizable.

There are some known limitations of our simulator. First, it is the fact that our simulator is trace-based. It means that the memory footprint generated by an application needs to be recorded during run-time and then read back into our simulator off-line. Even though this is a common practice of memory simulators, it might include some error to the simulation results, since online behaviors, as changes in context or interrupts, are

Table 7.1: Comparison between available PIM simulators and CLAPPS. "?" indicates the information could not be obtained.

Simulator	**Simulation Accuracy**	**Native PIM Capabilities**	**Novel PIM Capabilities**	**Peak Bandwidth**	**Deal with Parallelism**	**Input Type**	**Parame-trizable**	**Simulation Time**
HMC-SIM	Cycle-Accurate	Yes	Yes - Limited	320 GB/s	Event queues	Offline Trace	No	?
Cas-HMC	Cycle-Accurate	Yes	No	?	Event queues	Offline Trace	No	28.7s
SMC	Cycle-Accurate	Yes	Yes	205 GB/s	Event queues	Online Trace	Yes	4.5s
CLAPPS	Cycle-Accurate	Yes	Yes	320 GB/s	Natively	Offline Trace	Yes	20m

Source: Provided by the author.

not replicated during the off-line simulation. To cope with this problem, we expect to integrate our simulator into the gem5 (BINKERT et al., 2011) infrastructure, since it supports SystemC-based designs. Second, the custom PIM programmability can impose a challenge for novel architectures. Despite the fact that this is not a problem intrinsic of the presented simulator but indeed an issue related to state-of-the-art PIM architectures, this can be a problem when implementing a cycle-accurate detailed model of the proposed PIM design, since probably there will be no compiler to generate code for the design. Some previous works, as (AHN et al., 2015b), have offered to solve this problem by developing PIM architectures with the same ISA as the host device. However, there is a trade-off between flexibility and programmability in this case. Third, data coherence is not guaranteed by our simulator. This problem occurs when both host and PIM are working in parallel, and the PIM generates a READ request, and in the sequence, the host sends a WRITE request to the same memory address. It might happen in this case that the WRITE request to be serviced before the READ request. Then the data read by the PIM will be the newest one, instead of the that was stored in memory at the time the PIM received the READ request. Fourth, CLAPPS's simulation time is orders of magnitude larger than the other simulators. This happens for two reasons. The first one is because the HMC device is highly parallel in its organization. For example, during simulation, only considering the HMC Vaults, 32 distinct modules need to be launched and executed concurrently on a single CPU core. This is similar to simulate 32 CPU cores, which is even a challenge for the most popular architectural simulator, gem5. The second one, and most important, is the fact our simulator implements HMC in a granularity of bits, which is different from the other simulators that use a behavior simulation. With this, our design can be easily converted to an RTL description, allowing to perform logic synthesis directly from our code. Finally, our simulator does not implement the HMC engine that deals with errors correction. We have chosen not to design this part of the HMC architecture because the primary source of errors in a real HMC device is the one generated by the high-speed serial links. Since we do not run the link lanes at the same high frequency as the actual hardware, an error coming from the links would not happen. Therefore, the error correction mechanism would only slow down the simulation process. In future works, we aim to include statistics about power and energy consumption into our design.

REFERENCES

AHN, J. et al. A scalable processing-in-memory accelerator for parallel graph processing. In: 42ND ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA), 2015, Portland, USA. **Proceedings...** [S.l.]: IEEE, 2015. p. 105–117.

AHN, J. et al. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In: 42ND ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA), 2015, Portland, USA. **Proceedings...** [S.l.]: IEEE, 2015. p. 336–348.

ALVES, M. A. Z. et al. Large vector extensions inside the hmc. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE (DATE), 2016. **Proceedings...** [S.l.]: IEEE, 2016. p. 1249–1254.

AMBA, A. Protocol specification v2. 0. **ARM Holdings plc Std**, 2003.

AZARKHISH, E. **Memory Hierarchy Design for Next Generation Scalable Many-core Platforms**. Thesis (PhD) — alma, 2016.

AZARKHISH, E. et al. A case for near memory computation inside the smart memory cube. In: INTERNATIONAL WORKSHOP ON EMERGING MEMORY SOLUTIONS, 2016. **Proceedings...** [S.l.:s.n.], 2016.

AZARKHISH, E. et al. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In: INTERNATIONAL CONFERENCE ON ARCHITECTURE OF COMPUTING SYSTEMS, 2016. **Proceedings...** [S.l.]: Springer International Publishing, 2016. p. 19–31.

AZARKHISH, E. et al. Neurostream: Scalable and energy efficient deep learning with smart memory cubes. In: ARXIV PREPRINT ARXIV:1701.06420, 2017. **Proceedings...** [S.l.:s.n.], 2017.

BINKERT, N. et al. The gem5 simulator. In: SIGARCH COMPUT. ARCHIT. NEWS, 2011. **Proceedings...** New York, NY, USA: ACM, 2011. p. 1–7.

EBRAHIMI, E. et al. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In: ACM SIGPLAN NOTICES, 2010. **Proceedings...** New York, USA: ACM, 2010. p. 335–346.

ELLIOTT, D. G. et al. Computational ram: Implementing processors in memory. In: IEEE DESIGN AND TEST OF COMPUTERS, 1999. **Proceedings...** [S.l.]: IEEE, 1999. p. 32–41.

FARMAHINI-FARAHANI, A. et al. Drama: an architecture for accelerated processing near memory. In: IEEE COMPUTER ARCHITECTURE LETTERS, 2014. **Proceedings...** [S.l.]: IEEE, 2014. p. 26–29.

FARMAHINI-FARAHANI, A. et al. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In: 21ST INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA), 2015. **Proceedings...** [S.l.]: IEEE, 2015. p. 283–295.

GHENASSIA, F. et al. **Transaction-level modeling with SystemC**. [S.l.]: Springer, 2005.

HASSAN, H. et al. Softmc: A flexible and practical open-source infrastructure for enabling experimental dram studies. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA), 2017. **Proceedings...** [S.l.]: IEEE, 2017. p. 241–252.

HONG, D. U. L. et. al. S. 25.2 a 1.2v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv. In: IEEE INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE DIGEST OF TECHNICAL PAPERS (ISSCC), 2014. **Proceedings...** [S.l.]: IEEE, 2014. p. 432–433.

HSIEH, K. et al. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In: IEEE 34TH INTERNATIONAL CONFERENCE ON COMPUTER DESIGN (ICCD), 2016. **Proceedings...** [S.l.]: IEEE, 2016. p. 25–32.

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

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

JAYADEVAPPA, S.; SHANKAR, R.; MAHGOUB, I. A comparative study of modelling at different levels of abstraction in system on chip designs: a case study. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, 2004. **Proceedings...** [S.l.]: IEEE, 2004. p. 52–58.

JEON, D. I.; CHUNG, K. S. Cashmc: A cycle-accurate simulator for hybrid memory cube. In: IEEE COMPUTER ARCHITECTURE LETTERS, 2016. **Proceedings...** [S.l.]: IEEE, 2016. p. 1–1.

LEE, C. J. et al. Dram-aware last-level cache writeback: Reducing write-caused interference in memory systems. HPS Technical Report, TR-HPS-2010-002, 2010.

LEE, D. et al. Simultaneous multi-layer access: Improving 3d-stacked memory bandwidth at low cost. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM, v. 12, n. 4, p. 63, 2016.

LEE, D. et al. Adaptive-latency dram: Optimizing dram timing for the common-case. In: 21ST INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA), 2015. **Proceedings...** [S.l.]: IEEE, 2015. p. 489–501.

LEIDEL, J. D.; CHEN, Y. Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM WORKSHOPS (IPDPSW), 2016. **Proceedings...** [S.l.]: IEEE, 2016. p. 621–630.

LI, S. et al. The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing. **Transactions on Architecture and Code Optimization**, v. 10, n. 1, p. 5, 2013.

LIM, K. et al. Disaggregated memory for expansion and sharing in blade servers. In: ACM SIGARCH COMPUTER ARCHITECTURE NEWS, 2009. **Proceedings...** New York, USA: ACM, 2009. p. 267–278.

LIU, J. et al. Disaggregated memory for expansion and sharing in blade servers. In: ACM SIGARCH COMPUTER ARCHITECTURE NEWS, 2013. **Proceedings...** New York, USA: ACM, 2013. p. 60–71.

LIU, J. et al. Raidr: Retention-aware intelligent dram refresh. In: ACM SIGARCH COMPUTER ARCHITECTURE NEWS, 2012. **Proceedings...** New York, USA: ACM, 2012. p. 1–12.

MATHEW, D. et al. A bank-wise dram power model for system simulations. In: WORKSHOP ON: RAPID SIMULATION SIMULATION AND PERFORMANCE EVALUATION: METHODS AND TOOLS (RAPIDO), 2017, Stockholm, Sweden. **Proceedings...** [S.l.:s.n], 2017.

MOSCIBRODA, T.; MUTLU, O. Memory performance attacks: Denial of memory service in multi-core systems. In: 16TH USENIX SECURITY SYMPOSIUM ON USENIX SECURITY SYMPOSIUM, 2007. **Proceedings...** [S.l.]: USENIX Association, 2007.

MURPHY, R. On the effects of memory latency and bandwidth on supercomputer application performance. In: 10TH INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION (IISWC), 2007. **Proceedings...** [S.l.]: IEEE, 2007. p. 35–43.

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

OLIVEIRA, G. F. et al. Nim: An hmc-based machine for neuron computation. In: INTERNATIONAL SYMPOSIUM ON APPLIED RECONFIGURABLE COMPUTING, 2017, Delft, Netherlands. **Proceedings...** Cham, Switzerland: Springer, 2017. p. 28–35.

PANDA, P. R. Systemc-a modeling platform supporting multiple design abstractions. In: 14TH INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, 2001. **Proceedings...** [S.l.]: IEEE, 2001. p. 75–80.

PATEL, M.; KIM, J. S.; MUTLU, O. The reach profiler (reaper): Enabling the mitigation of dram retention failures via profiling at aggressive conditions. In: 44TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 2017. **Proceedings...** New York, USA: ACM, 2017. p. 255–268.

PATTERSON, D. et al. A case for intelligent ram. **IEEE Micro**, v. 17, n. 2, p. 34–44, Mar 1997. ISSN 0272-1732.

PUGSLEY, S. et al. Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads. **IEEE Micro**, v. 34, n. 4, p. 44–52, July 2014.

RADULOVIC, M. et al. Another trip to the wall: How much will stacked dram benefit hpc? In: INTERNATIONAL SYMPOSIUM ON MEMORY SYSTEMS, 2015. **Proceedings...** [New York, USA]: ACM, 2015. p. 31–36.

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

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

SANTOS, P. C. et al. Exploring cache size and core count tradeoffs in systems with reduced memory access latency. In: 24TH EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED, AND NETWORK-BASED PROCESSING (PDP), 2016. **Proceedings...** [S.l.]: IEEE, 2016. p. 388–392.

SANTOS, P. C. et al. Operand size reconfiguration for big data processing in memory. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE (DATE), 2017, Lausanne, Switzerland. **Proceedings...** [S.l.]: IEEE, 2017.

SCHALLER, R. R. Moore's law: past, present and future. **IEEE Spectrum**, v. 34, n. 6, p. 52–59, Jun 1997. ISSN 0018-9235.

SESHADRI, V. et al. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In: 46TH ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE (MICRO), 2013. **Proceedings...** [S.l.]: IEEE, 2013. p. 185–197.

SHIVAKUMAR, P.; JOUPPI, N. P. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2001/2, Compaq Computer Corporation, 2001.

STUECHELI, J. et al. The virtual write queue: Coordinating dram and last-level cache policies. In: ACM. **ACM SIGARCH Computer Architecture News**. [S.l.], 2010. v. 38, n. 3, p. 72–82.

SUBRAMANIAN, L. et al. Mise: Providing performance predictability and improving fairness in shared main memory systems. In: IEEE 19TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA), 2013. **Proceedings...** [S.l.]: IEEE, 2013. p. 639–650.

SURA, Z. et al. Data access optimization in a processing-in-memory system. In: PROCEEDINGS OF THE 12TH ACM INTERNATIONAL CONFERENCE ON COMPUTING FRONTIERS, 2015. **Proceedings...** New York, USA: ACM, 2015.

Tezzaron. **DiRAM4 - 3D Memory**. 2015. Https://tezzaron.com/products/diram4-3d-memory/.

WULF, W. A.; MCKEE, S. A. Hitting the memory wall: Implications of the obvious. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 23, n. 1, p. 20–24, mar. 1995. ISSN 0163-5964. Available from Internet: <http://doi.acm.org/10.1145/216585.216588>.

XI, S. L. et al. Quantifying sources of error in mcpat and potential impacts on architectural studies. In: 21ST INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA), 2015. **Proceedings...** [S.l.]: IEEE, 2015. p. 577–589.

XU, L.; ZHANG, D. P.; JAYASENA, N. Scaling deep learning on multiple in-memory processors. In: WONDP: 3RD WORKSHOP ON NEAR-DATA PROCESSING IN CONJUNCTION WITH MICRO-48, 2015. **Proceedings...** [S.l.:s.n], 2015.

YOO, J. et al. Vssim: Virtual machine based ssd simulator. In: 29TH SYMPOSIUM ON MASS STORAGE SYSTEMS AND TECHNOLOGIES (MSST), 2013. **Proceedings...** [S.l.]: IEEE, 2013. p. 1–14.