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

Geraldo F. Oliveira†, Paulo C. Santos†, Marco A. Z. Alves‡, Luigi Carro†
†Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
‡Department of Informatics – Federal University of Paraná – Curitiba, Brazil
Email: †{gfojunior, pcssjunior, carro}@inf.ufrgs.br ‡ mazalves@inf.ufpr.br

***Abstract*—PIM was one of the attempts created during the 1990s to try to mitigate the notorious memory wall problem, where computational elements are added close, or ideally, inside the memory devices. Nowadays, with the maturation of 3D integration technologies, a new landscape for novel PIM architectures can be explored. 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 an in-house simulator to perform their experiments. However, this methodology might hurt the overall productivity, and it can also preclude replicability. In this paper, we show the development of a precise, modular and parametrized PIM simulation environment. Our simulator has been developed using the SystemC allowing native parallel simulation. We have implemented the latest HMC technical specifications, including all HMC instructions. The primary contribution of our work lies on 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.**

***Index Terms*—In-Memory Processing, Simulators, Hybrid Memory Cube, 3D-Stacked**

I. Introduction

During the last three decades, the target goal of computer engineering was to improve computation resources. From adding more logic elements and increasing chip frequency in the later 1990s, to multiplying the number of cores in a single chip during the early 2000s, the primary goal is still the same: maximize the amount of work that can be done by a processor or accelerator while reducing the execution time. However, this metric is affected by a neglected component so far - the memory system. Composed by a hierarchy of modules that varies in technology, cost, power dissipation, and size, a traditional memory system in a modern computer environment has several levels of cache memories, Dynamic Random Access Memory (DRAM) modules, and some non-volatile storage (either Hard-Disk Drive (HDD), Solid-State Drive (SSD), or a mix of both). This extensible number of memory modules distributed throughout the system adds extra latency to a given task [1].

To understand the impact that memory components impose to a system, either regarding performance or energy consumption, researchers 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 [2], a cycle-accurate Double Data Rate (DDR) memory simulator, is widely employed to estimate DDR2/3 speed and power consumption; the Cacti [3] model is a popular tool to estimate cache performance, area, and power consumption; and VSSIM [4] is a new SSD simulator that can mimic today's SSD architectures.

Nowadays, with the advent of Big Data applications and massive data processing workloads, the memory system has become a major performance bottleneck [5]. Even though this issue is becoming more problematic in modern workstations, Wulf [6] 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. Besides that, current DDR memories are not able to supply enough bandwidth for High Performance Computing (HPC) and Big Data applications [5].

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 applications. In a 3D stack environment, several layers of memory modules are vertically connected by Through-Silicon Via (TSV). Therefore, 3D technology improves data density and also increases memory bandwidth. Also, 3D integration makes possible to implement heterogeneous stacks, with layers of memories and logic altogether in one single chip. These features help to reduce the latency to access the 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) [7], High Bandwidth Memory (HBM) [8], and DiRAM4 [9].

Most important, the possibility to combine memory devices and computation elements into one single chip renew the

978-1-5386-3437-0/17/$31.00 ©2017 IEEE

prospect to explore Processor-in-Memory (PIM) architectures [10]. PIM, previously named Near-Data Processing (NDP), was first designed to reduce application's execution time, where logic elements were placed inside DRAM modules to explore their internal bandwidth [11]. However, besides improving how the system would exploit memory bandwidth, PIM approaches can potentially help reduce the 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.

Since PIM has emerged again recently, simulating the industry new memory devices or even testing novel PIM mechanism is a common issue that researchers are facing now. Today, most of the works that target PIM implement in-house simulators to support their experiments ([12], [13], [14], [15]). Two problems can be pointed out in this scenario of multiple PIM simulators. First, a significant part of the research 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 [16]. 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 [17] 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 [18].

In this work, we aim to build a generic Cycle Accurate Parallel PIM Simulator (CLAPPS) that can be used to create custom PIM architectures. Our framework has been developed using SystemC [19]. 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 [20] simulator. Gem5 is a robust and extensible system that can simulate most elements of a computer system, including a number of instruction sets, microarchitecture organizations, memory devices, interconnections, and communication protocols. 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 a HMC simulator targeting its latest technical specifications [7], including all HMC instructions.

This paper is organized as follows. Section II enumerates current HMC simulators that are related to our work. Section III describes our framework infrastructures in details, showing how we built the memory simulator and how we expose the custom PIM layer to the final user. Section IV depicts our memory simulation results and also presents a case of study to the proposed interface. Finally, in Section V we present future work.

II. Related Work

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 is 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 [21] 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 from 2.0 [7], 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 [21] 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 [22] is 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 SMC Simulation Environment (SMC-Sim) [23] 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 [23] has claimed to validated their design latency and performance parameters using a complete RTL design, the memory implementation inside the gem5 simulator is based on correlations between 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 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 [24] 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 year to obtain information regards PIM advantages. To illustrate, the author of [12] have used a trace-based HMC simulator to implement different accelerators that target vector operations, database, and deep learning applications. [25] uses a Structural Simulation Toolkit (SST) simulator to investigate how data and computational locality impacts PIM design. [26] implements an analytic simulator aiming to understand how HPC applications can benefit from 3D-stacked memories. [27] presents a set of methodologies that can help reducing access latency of 3D memories. To evaluate their proposal, the authors implemented a timing-accurate simulator.

To summarize, today's PIM simulators can be categorized into three broad groups: trace-based, cycle-accurate, and analytical models. Analytical models provide faster simulations but may not take into account important design metrics. Trace-based models execute after the application had finished, and therefore do not consider on-the-fly information. 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

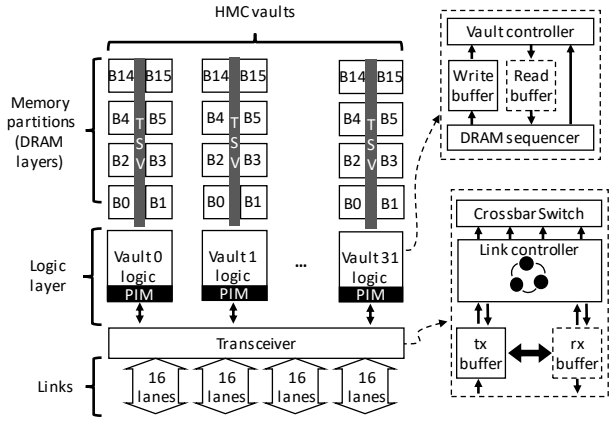

Fig. 1: Overview of the proposed HMC and PIM simulator.

previous implementations. Also, our proposed framework has been constructed using the RTL design flow, while following the most current HMC specifications.

III. Simulator Mechanism

This section describes in details our simulation 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 [7] and then elaborated each component described in the documentation following the Register Transfer Level (RTL) design flow chain. However, since we have chosen to work with the SystemC programming language, the total time spent building the simulator was reduced due to the flexibility provided by the language. In the latest HMC specification [7] the memory device is composed of four high-speed serial links (up to 30 GB/s), a logic layer of 32 memory controllers (called vault controllers), and up to eight layers of Dynamic Random Access Memory (DRAM) memories connected via Through-Silicon Via (TSV) to 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.

A. Hybrid Memory Cube Simulator

Figure 1 depicts Cycle Accurate Parallel PIM Simulator (CLAPPS) architecture. The simulator is divided into independent three modules: *transceiver*, *vault*, and *Processor-in-Memory (PIM) Interface*. The *transceiver* module is composed of the *packet encapsulator*, *serializer*, *link control*, *deserializer*, and *switching network*. All these modules are presented in both *tx* (host to memory) and *rx* (memory to host) directions. The *vault* module is constituted of *data request buffer*, *request command buffer*, *response control*, *memory controllers*, *TSV controller*, and *memory banks*. Finally, the *PIM interface* is comprised of one *data buffer* and *instruction queue*.

When the host generates a new request to the main memory, the simulation processing works as follows. First, the host outputs to the system the correspond HMC instruction, memory

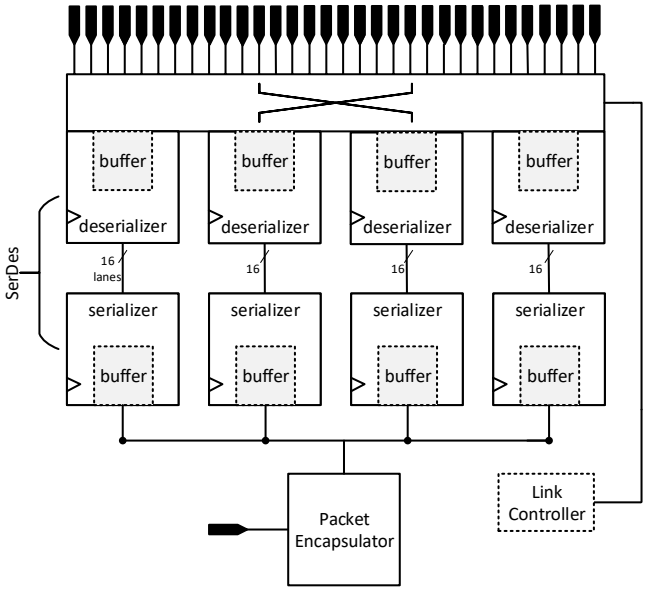

Fig. 2: Transceiver host-to-memory overview.

address, and data blocks (maximum of 2 Kb), in the case of a store or atomic instructions, activating the currently available link controller. This controller will encapsulate the memory request into packets, or FLITs, of 128b. A single FLIT is divided into header and tail field of 64b each. The header provides information about the requested command, the total number of FLITs that constitutes the request, a unique tag that identifies the request, and the target address. The tail provides information to handle potential transaction failures, as retry pointer and CRC checker. The maximum number of FLITs that one single request can generate is seventeen. After the request has been encapsulated, the serializer module transmits the FLITs through high-speed lanes of 16 bits. The path that the request will travel until arrive at the vault controller depends upon the memory interleaving being adopted. Currently, we have implemented the standard HMC interleaving: varying the vault address first, followed by the bank, and the row. Figure 2 illustrates the *tx* datapath.

Second, once at the vault controller, the packet will be decoded into simpler operations that the memory controller can execute. Potentially, the number of memory controllers can be equal to the number of banks available in one single vault (up to 16 banks). To schedule the incoming request to one free memory controller, we have used a simple policy based on the bank address targeted by request, which also reduces bank conflict. If there is already one memory controller that has later worked on the same bank address of the request, the request will be passed to this memory controller command queue; otherwise, the memory controller will be chosen in a round-robin fashion. After that, the memory controller will be responsible for accessing the memory dies following standard Double Data Rate (DDR) command sequence. We have extracted DDR timing parameters from the work of [28].

The memory controller sends and receives data from/to the memories dies through TSV. However, the number of TSV is limited due to physic constraints. Thereby, there is only one path from memory dies to memory controller per vault. The host can request load/store command that ranges from 16 to 256B. However, only 32B of data can navigate through the TSV, forcing the memory controller to split its requests over TSV size multiples. Besides that, since there is more than one bank that potentially will need to transmit/receive data at the same time, the vault controller needs to manage the TSV ownership. Therefore, we have implemented a fair round-robin mechanism that allows each bank to send or received data from/to the vault controller. Once the memory controller has completed processing the request, the response packet is generated and sent back to the transceiver layer. Figure 3 illustrates the implemented vault architecture.

Finally, according to the HMC specification, packet ordering must be maintained during all operations. To cope with this

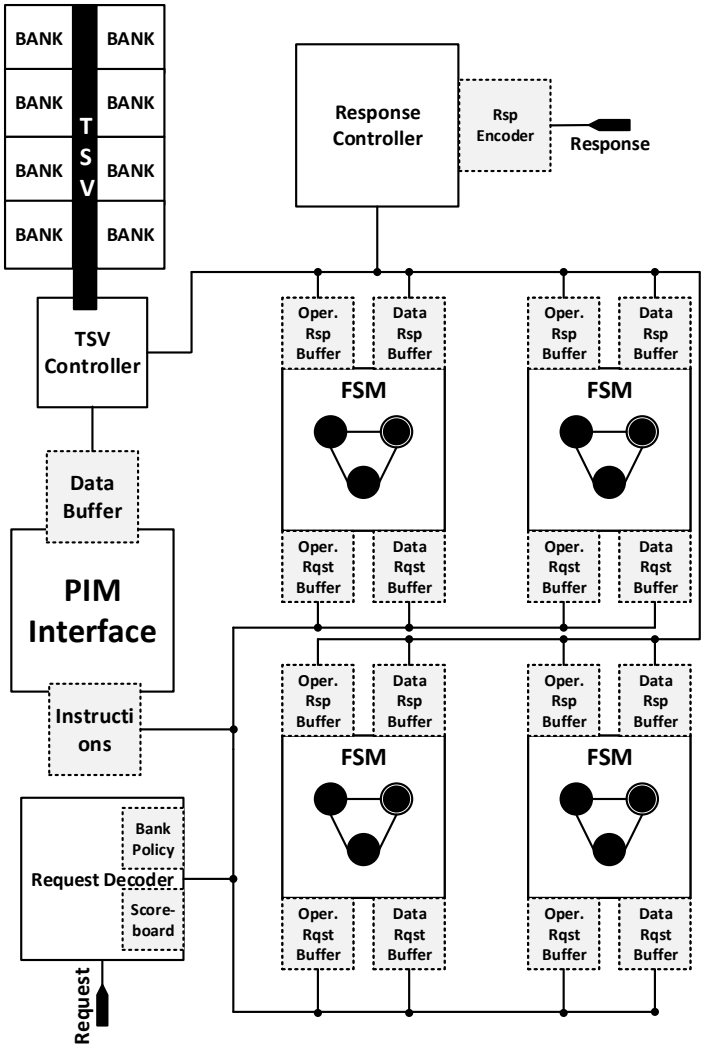

Fig. 3: Vault Internals overview.

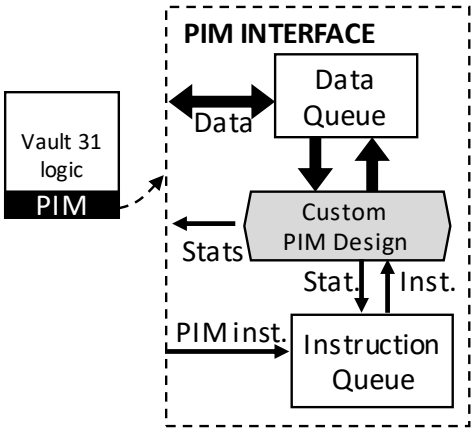

Fig. 4: Overview of the proposed custom PIM interface.

constraint, the link controller will compare the tag stored at the top of its request queue with the output of all vault response buffers. When a matching between request and response occurs, the link controller will allow the respective vault to send its packets, locking the *rx* switching network with the required response addresses. Then the response FLITs travel in the opposite flow of the request packet, passing through the serializer and deserializer modules, until finally being unpacked and delivered back to the host device. The *rx* process in similar to the one described in Figure 2.

One of the main features of HMC devices is the native support to execute *read-modify-write* operations with only one memory request. The latest HMC specification defines arithmetic, boolean, comparison, and bitwise atomic operations of 8 and 16 bytes. To implement these operations, we have included to the vault controller module a simple combinational Arithmetic Logic Unit (ALU) that receives its operand from the memory controllers buffer. Since all operations must be executed atomically, the vault controller cannot perform any other operation until the *read-modify-write* process has finished. Therefore, each vault controller only has a simple ALU that controls are connected with only one memory controller. According to [24], since HMC modules target primarily High Performance Computing (HPC) applications, which most of the time generate data requests with little spatial locality, a close-row policy is employed. Also, different from DDR memories that have large row buffer (8KB in traditional memories), HMC memories have a maximum row buffer of 256B per vault. Thus, sub-sequential requests that reuse row buffer data is unlike to happen. However, by following the close-row policy, the performance of the atomic operations is impaired, since a single atomic operation has to obey a complete read-write request flow.

B. PIM Interface Protocol

Figure 4 depicts the proposed PIM user interface. A single PIM request is seen by our simulator similar to a 16B write request. To send instructions to the custom PIM architecture inside the vault, the user needs to provide the reserved command opcode *PIM_INSTRUCTION*, the target memory address, and the instruction to be executed by the PIM module. The PIM instruction is related to the user PIM unique design and its instruction set. Figure 5 illustrations the generated request FLITs to this transaction. After the request has been made, it will follow the same path of any regular HMC instruction. However, when it reaches the vault request decoder, instead of treating the most significant 8B and least significant 8B from the current and subsequent FLIT as store data, it will send these fields to the PIM Interface Instruction Queue. Besides that, we have provided to the user basic load/store instructions that send/gather data from the memory to the PIM interface. In this way, the user does not need to modify the already presented memory data path, thereby focusing only on implementing the PIM design that would fit their needs.

Since the user can perform any operation inside their PIM mechanism, data integrity becomes a potential problem to CLAPPS. To illustrate, suppose that the user has decided to implement a MIPS architecture into the PIM interface, and then the following sequence of commands are sent to the memory:

```
PIM_INSTRUCTION 0x00 PIM_RD32 $r5
PIM_INSTRUCTION 0x00 add_fp $f3, $f3, $r5
PIM_INSTRUCTION 0x01 PIM_WR32 $f3
RD32 0x01
```

In this sequence of instructions, the simulator will first read 32B from memory and send the data to be stored in the register five inside the MIPS processor. Then, the processor will receive the add_fp operation, a floating-point operation that depended on the register $r5, and stores it in register f3. After that, the processor will store the resulting computation back at address 0x01 in the memory. Finally, a standard read request will send right after the PIM store instruction to the vault controller. However, the *add_fp* instruction might take a variable number of clock cycles to complete, and then an additional time to store back the data. Therefore, when the *RD32* operation is executed, it will potentially read old data from the *0x01* address. To cope with this READ-AFTER-WRITE issue, we have implemented a simple address scoreboard inside the vault request decoder. When a PIM write request is received, the target address is placed into the scoreboard, and all next incoming read requests that target that same address will be delayed and released only when the marked PIM store instruction be completed.

Fig. 5: FLITs generated by a PIM instruction.

TABLE I: Baseline, HMC and PIM configurations.

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

IV. 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 dived our results in three categories: HMC Memory Validation, HMC Atomic Requests, and PIM Implementation. Table I describes all configured parameters used in all presented results. All these parameters are parameterized and can be modified as needed.

A. 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 [24].

First, we were interested in investigating if there was a significant performance different when comparing read, write, and read+write requests, as cited in [24]. 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

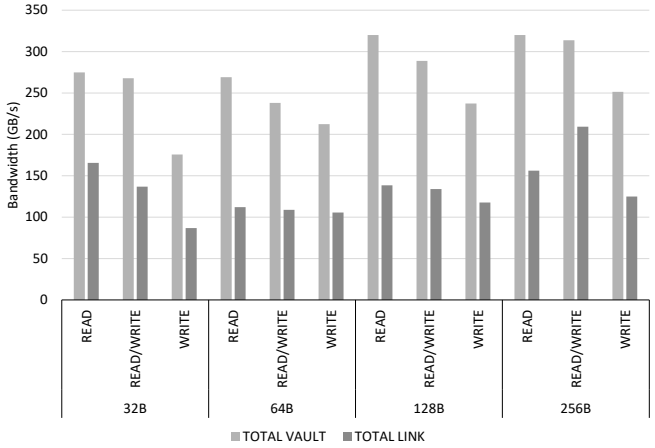

Fig. 6: Link and Vault total bandwidth for sequential read and write requests.

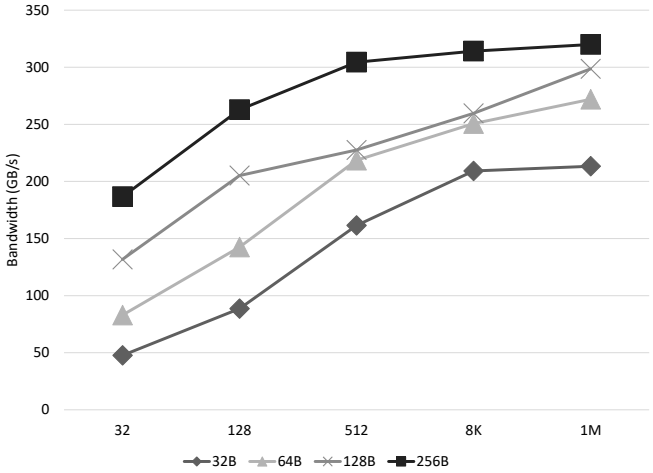

Fig. 7: Vault total bandwidth for sequential read requests.

extract maximum performance from the current simulation. We have used a 50% read/write ratio in our testbench. Besides that, the request addresses were generated aiming to evict link/vault/bank conflicts. Figure 6 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 will read data from memory cells and then store in the row buffer to be then, be read 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 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 [24].

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 7 and Figure 8 show the vault and link performance results respectively. It is evident that the biggest the row buffer size, the better, 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 response 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.

B. 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 test benches 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 9 and Figure 10 show our simulation results. The results for the random-based requests were obtained 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 request. 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.

C. 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 [29].

Reconfigurable Vector Unit (RVU) is a reconfigurable accelerator that targets vector operations with varied operand sizes. [29] as an improvement over the work proposed by citeHIVE 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 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.

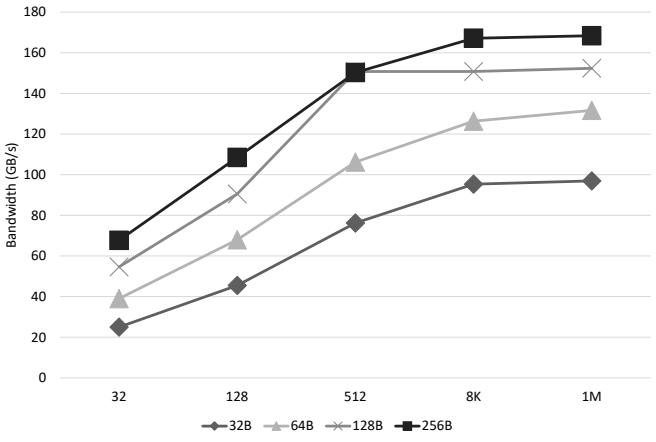

Fig. 8: Link total bandwidth for sequential read requests.

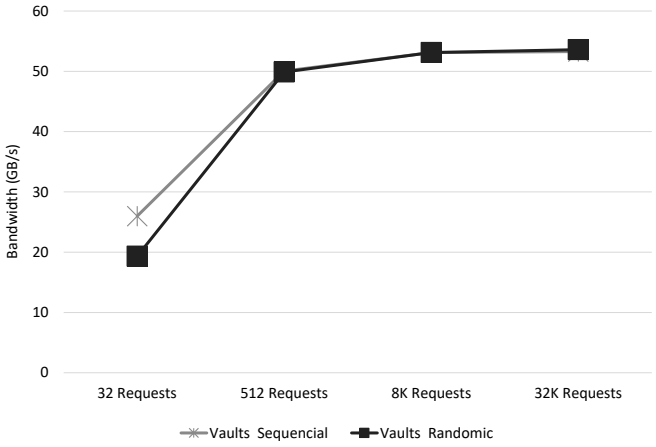

Fig. 9: Vault total bandwidth for sequential and randomic atomic requests.

According to [29], when the user configures their mechanism to run over its maximum operation size, it will achieve similar results to the work of [12]. Besides that, from all benchmarks executed by [12], 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 [12], the total vault bandwidth obtained for a *vec-sum* operation was 315.9 GB/s. There is no information in [12] 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.

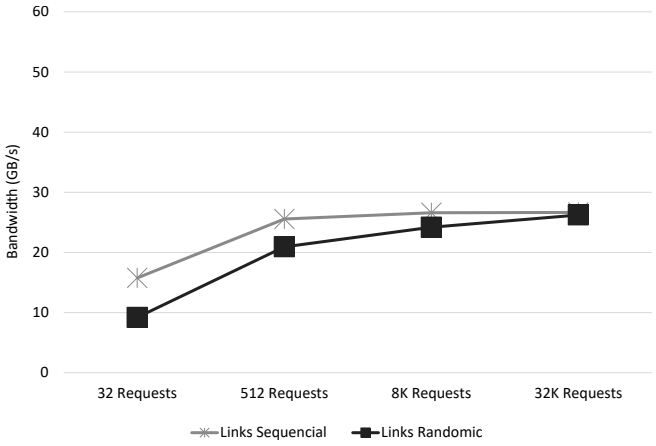

Fig. 10: Link and Vault total bandwidth for sequential and randomic atomic requests.

V. 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. In future works, we aim to include statistics about power and energy consumption into our design. Also, we are working on integrating our simulator into the gem5[20] infrastructure.

References

[1] P. C. Santos, M. A. Alves, M. Diener, L. Carro, and P. O. Navaux, "Exploring cache size and core count tradeoffs in systems with reduced memory access latency," in *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*. IEEE, 2016, pp. 388–392.

[2] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[3] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," 2001.

[4] J. Yoo, Y. Won, J. Hwang, S. Kang, J. Choil, S. Yoon, and J. Cha, "Vssim: Virtual machine based ssd simulator," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*. IEEE, 2013, pp. 1–14.

[5] M. Radulovic, D. Zivanovic, D. Ruiz, B. R. de Supinski, S. A. McKee, P. Radojković, and E. Ayguadé, "Another trip to the wall: How much will stacked dram benefit hpc?" in *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 2015, pp. 31–36.

[6] "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[7] Hybrid Memory Cube Consortium, "Hybrid memory cube specification rev. 2.0," 2013, http://www.hybridmemorycube.org/.

[8] D. U. L. et. al. S. Hong, "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 *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 432–433.

[9] Tezzaron, "Diram4 - 3d memory," 2015, https://tezzaron.com/products/diram4-3d-memory/.

[10] S. Pugsley, J. Jestes, R. Balasubramonian *et al.*, "Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads," *IEEE Micro*, vol. 34, no. 4, pp. 44–52, July 2014.

[11] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Mar 1997.

[12] M. A. Z. Alves, M. Diener, P. C. Santos, and L. Carro, "Large vector extensions inside the HMC," in *Conf. on Design, Automation & Test in Europe*, 2016.

[13] L. Xu, D. P. Zhang, and N. Jayasena, "Scaling deep learning on multiple in-memory processors," 2015.

[14] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture," in *Int. Symp. on Computer Architecture*, 2015.

[15] G. F. Oliveira, P. C. Santos, M. A. Alves, and L. Carro, "Nim: An hmc-based machine for neuron computation," in *International Symposium on Applied Reconfigurable Computing*. Springer, Cham, 2017, pp. 28–35.

[16] E. Azarkhish, "Memory hierarchy design for next generation scalable many-core platforms," Ph.D. dissertation, alma, 2016.

[17] S. Li, J. H. Ahn, R. D. Strong *et al.*, "The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing," *Transactions on Architecture and Code Optimization*, vol. 10, no. 1, p. 5, 2013.

[18] S. L. Xi, H. Jacobson, P. Bose, G. Y. Wei, and D. Brooks, "Quantifying sources of error in mcpat and potential impacts on architectural studies," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 577–589.

[19] P. R. Panda, "Systemc-a modeling platform supporting multiple design abstractions," in *System Synthesis, 2001. Proceedings. The 14th International Symposium on*. IEEE, 2001, pp. 75–80.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[21] J. D. Leidel and Y. Chen, "Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 621–630.

[22] D. I. Jeon and K. S. Chung, "Cashmc: A cycle-accurate simulator for hybrid memory cube," *IEEE Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2016.

[23] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "A case for near memory computation inside the smart memory cube," 2016.

[24] P. Rosenfeld, "Performance exploration of the hybrid memory cube," Ph.D. dissertation, University of Maryland, 2014.

[25] G. Stelle, S. L. Olivier, D. Stark, A. F. Rodrigues, and K. S. Hemmert, "Using a complementary emulation-simulation co-design approach to assess application readiness for processing-in-memory systems," in *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*. IEEE, 2014, pp. 64–71.

[26] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "Top-pim: throughput-oriented programmable processing in memory," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014, pp. 85–98.

[27] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien *et al.*, "Data access optimization in a processing-in-memory system," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM, 2015, p. 6.

[28] D. Mathew, É. Zulian, S. Kannoth, M. Jung, C. Weis, and N. Wehn, "A bank-wise dram power model for system simulations," in *Workshop on: Rapid Simulation Simulation and Performance Evaluation: Methods and Tools (RAPIDO), Stockholm, Sweden.*, 2017.

[29] P. C. Santos, G. F. Oliveira, D. G. Tome, M. A. Z. Alves, E. C. Almeida, and L. Carro, "Operand size reconfiguration for big data processing in memory," in *2017 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2017.