Processing in 3D memories to speed up operations on complex data structures

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

***Abstract*—Pointer chasing has been, for years, the kernel operation employed by diverse data structures, from graphs to hash tables and dictionaries. However, due to the bewildering growth in the volume of data that current applications have to deal with, performing pointer chasing operations have become a major source of performance and energy bottleneck, due to its sparse memory access behavior. In this work, we aim to tackle this problem by taking advantage of the already available parallelism present in today's 3D-stacked memories. We present a simple mechanism that can accelerate pointer chasing operations by making use of a state-of-the-art PIM design that executes in-memory vector operations. The key idea behind our design is to run speculative loads, in parallel, based on a given memory address in a reconfigurable window of addresses. Our design can perform pointer-chasing operations on b+tree 4.9× faster when compared to modern baseline systems. Besides that, since our device avoids data movement, we can also reduce energy consumption by 85% when compared to the baseline.**

***Keywords*—In-Memory Processing, Pointer-Chasing, Big Data, Reconfigurable, Vector Instructions, Hybrid Memory Cube**

I. Introduction

In this paper, we aim to take advantage of 3D-stacked memories to track the pointer-chasing problem. Previous works, as [1], [2], [3], [4] have already design Processor-in-Memory (PIM) mechanisms that seek to solve the pointer-chasing problem. However, the fundamental differences in our mechanism are twofold. First, we observe that one can use of the vast parallelism available in 3D memories, in special the Hybrid Memory Cube (HMC) device, to induce spacial locality via speculation. Second, we show that when spacial locality cannot be exploited due to sparse memory access, one can still take advantage of PIM by including a cache-like mechanism near memory. Our mechanism was designed inside the logic layer of a HMC device.

The major contributions of this work are the following. First, we propose a simple PIM architecture, called Pointer-Chasing Engine (PCE), to accelerate pointer-chasing operations. Our mechanism takes advantage of the available parallelism in 3D-stacking memories to execute speculative loads without the interference of the host processor. Second, we evaluation of our design using classic data structures that the pointer-chasing behavior dominates execution time. Besides that, we make a design space exploration of our mechanism to understand the benefits it could provide depending on the address pattern produced by the algorithm. Finally, our results show that we are up to 4.9× faster than the baseline, reducing energy consumption to only 13% of baseline's consumption.

The authors gratefully acknowledge the support of CNPq and CAPES.

II. Mechanism

Our mechanism explores speculative loads in memory by taking advantage of a state-of-the-art PIM design that includes reconfigurable vector units and vector register files inside HMC, called Reconfigurable Vector Unit (RVU) [5], [6]. We expand the Instruction Set Architecture (ISA) of the RVU accelerator to include a single *FIND* instruction that is responsible for triggering the traversing along the targeted data structure. The major component of our design is the PCE unit, a simple finite-state machine that decouples the *FIND* instruction into the required operations.

A. PCE

Figure 1 illustrates the distribution of this engine along all *vaults* inside the HMC module. Each instance of our mechanism is basically composed by a simplified version the RVU architecture and a specialized Finite State Machine (FSM). The Functional Units (FUs) are capable of executing scalar and vector operations such as *addition*, *comparison* and *gather/scatter* micro-instructions, which are the main computations required by our design. The FSM is responsible for managing requests and triggering operations according to the information decoded from the *FIND* instruction. Four main operations are managed by the FSM:

- ***Load Generation*** - The FSM generates a *LOAD* operation with the address provided by a previous load or by a *FIND* instruction.
- ***Check Data*** - FSM checks if the data being searched has been reached. It selects the correct operand from the current vector register.
- ***Address Translation*** - Virtual address are translated directly by the PCE using the available FUs. The virtual-to-physical address translation is presented in more details on Section II-D.
- ***Internal Find*** - A PCE instance forward the *FIND* instruction to another PCE when the required data lies on another *vault*.

B. Find in Memory Instruction and Algorithm

Aiming to generalize the use of our mechanism, we gathered all necessary information to traverse representative linked

978-3-9819263-0-9/DATE18/©2018 EDAA

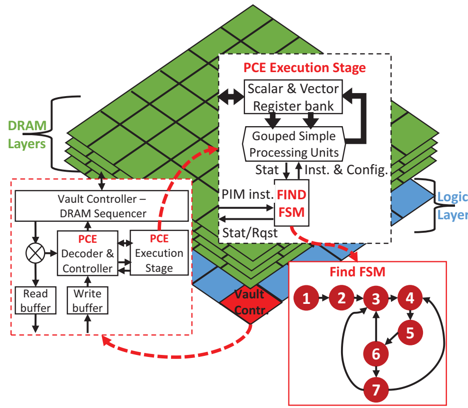

Fig. 1: Overview of our Pointer-Chasing Engine placed into the HMC device.

data structures on a *FIND* instruction. In this way, different data structures can take advantage of our design.

The *FIND* instruction is composed by:

- ***structure type*** - This field informs the PCE the type of data structure to traverse. It can be linked list, hash, or b+tree.
- ***base address*** - The address of the node where the search begins. It is the start point and the first node to be loaded by PCE.
- ***data offset*** - The distance between the data of interest and the beginning of its node. It is the position of the data itself in linked lists and hash tables, and it is also the offset of an array of keys in b+trees.
- ***data size*** - The size of data in bytes. Again, it represents the size of data in linked lists and hash tables and the number of pages/key in b+trees.
- ***next address offset*** - The distance between the pointer to the next node and the beginning of its node for linked list and hash tables. For b+trees, it contains the offset to the start of an array of pointers to its children.
- ***gold*** - The value used as a key in the search.
- ***structure size*** - The size of a single element in the data structure, generally referred as a node.
- ***operand size*** - Our mechanism can work with different operand sizes, allowing to speculate through different amounts of memory address. In this way, this parameter is used to configure the PCEs size, by grouping them in order to form speculative operands from 64 bytes to 8192 bytes.

When an instruction is received by the HMC, it is directed to a corresponding *vault* based on the *base address* field and the HMC interleaving. In this way, the algorithm used in our design to traverse the target data structures are explained as follows:

❶ PCE receives the *FIND* instruction and translates the *base address* using the address translation explained on Section II-D;

❷ PCE checks the address range of the request. Since each PCE can operate only on its *vault* address range, we can use the interleaving mapping scheme to trivially calculate whether or not the request belongs to the current *vault*. If it does, the process continues to step ❸. Otherwise, PCE encapsulates the request as an *Internal Find* operation, and sends it to the correct *vault*, following to the step ❸. The *Internal Find* operation replaces the *base address* field present in the virtual address in the request with its corresponding physical address. From this point, PCE only operates on physical addresses.

❸ PCE decodes the instruction, and reads data from memory. According to the *operand size*, the data chunk can range from 32 Bytes to 8192 Bytes. Once HMC memory controller returns the requested data, PCE stores it to one of its internal vector registers. By loading a large amount of data from memory at once, PCE can take advantage of speculative load behavior. Besides that, in this step, the physical address and the operand size that originates the read operation are noted on specific registers, *RA* and *RS* respectively.

❹ PCE compares the *gold* present in the instruction with the loaded data (*data offset* is needed to determine its position): If the *gold* value matches the loaded data, PCE finishes the operation by flagging a particular memory address where the host keeps polling to evaluate the result. If not, the operation continues to the step ❺. Although PCE takes advantage of vector operations, the comparison between the *gold* data and the current data is a scalar comparison, since we are sure about just one node, and the remaining data is speculation for next operations.

❺ PCE translates all virtual *next address* fields that lies on the active vector register by applying the address translation techniques from Section II-D as vector operations. The *next address offset* is needed to determine the position of virtual addresses.

❻ PCE checks if the *next address* belong to the current *vault*. If the *next address* does not belong, the *Internal Find* operation is encapsulated and sent to the correct *vault*, continuing to the step ❸. Else, the address belongs to the current vault and it continues on step ❼.

❼ PCE checks if the current register contains the required *next address*. This procedure is made by using the previously mentionated *RA* and *RS* registers. In this case, a simple calculus that comprises of checking if the *next address* fits between the range of *RA* and *RA+RS* is performed. If it fits, the least significant bits of *next address* are used to determine its position on the current vector register. Then, it continues on step ❹. Else, as it is guaranteed that the *next address* belongs to the currently *vault*, a load operation is generated for the current *vault*, and it goes to step ❸.

Although the *base address* that comes in the *FIND* instruction emitted by the host processor contains a virtual address, we adopted the idea that if the virtual address does not match with the correct *vault* it will be treated on steps ❶ and ❷. Also, considering that the instruction will trigger several operations, the first *vault* compulsory miss will be mitigated by other operations. Moreover, it is important to note that if we consider one vector register per *vault*, when a second request needs to be written to the same register, a replacement

must occur. Considering more vector registers per *vault*, the replacement can be delayed, increasing the chance of a PCE *hit* to happen.

C. Reconfigurable PCEs and Scalability

As each *vault* controller has access to different Dynamic Random Access Memory (DRAM) regions, a custom engine could manage those independent *vault* controllers to increase or reduce the size of contiguous data chunks loaded in parallel in HMC. Given that *intervault* communication allow us to request registers from different PCEs, we take advantage of this internal communication path to group PCEs and operate on a wider register. For instance, grouping every two *physical PCEs* to provide a single *logical PCEs* enable us to speculate loads in a window of 512 bytes and operate on a register of 512 bytes, which provides 16 *logical PCEs* of 512 bytes. In the same way, larger groups can be unified providing from 32 logical blocks of 256 bytes up to a single block of 8192 bytes of contiguous data. All those different configurations allow us to explore the influence of a wider data chunk in different data structures.

D. Virtual address translation

As big-memory workloads avoid features of page-based virtual, such as swapping and per-page protection, and allocate large chunks of memory with uniform access permission at start-up to prevent the high cost of page-based virtual memory, mapping part of a process' virtual address region with a direct segment rather than pages was proposed by [7]. Direct segments allow efficient mapping of large ranges of contiguous virtual memory to contiguous physical memory address using small, fixed hardware based only on three registers for each core: *base*, *limit* and *offset*. If a virtual address V is between the *base* and *limit* ($base \leqslant V < limit$), it is translated to a physical address $V + offset$ without a Translation Look-aside Buffer (TLB) miss.

III. Experimental Setup and Results

Table I describes the configuration of the baseline systems, an also the internal configuration of the PCE. Our baseline is an ARM-A57 + 1MB of Last-Level Cache (LLC). In our analysis, we explored different configurations for the PCEs, ranging its operand size capabilities from 64 bytes to the limit of 8192 bytes. We extrapolate the LLC size for the baseline to 2M Bytes, aiming to evaluate the benefits of a larger cache memory for the pointer-chasing operation. In our experiment, we used HMC as main memory to all systems. To implement our design, we used a cycle-accurate HMC simulator [8], which allows us to build our custom device into the HMC logic-layer. This simulator can be connected to the popular gem5 [9] system to provide an accurate way to compare our mechanism with the baselines. We estimated energy by synthesizing a Hardware Description Language of PCE using Cadence RTL Compiler Tool with a technology node of 32nm. We also consider the power consumption of the host processor connected to our system. For the baseline ARM processor and cache memories, we are based on McPat [10] coupled with Cacti [11] tools extrapolated to a technology of 22nm. The HMC power and energy consumption results relies on [12]. We have evaluated our design using the same

TABLE I: Baseline and Design system configuration.

ARM A57: 2.5 GHz; 4 cores; NEON Instruction Set Capable;
I/D 64KB L1 Cache 2 Cycles + 16-way L2 Cache 1MB 20 Cycles;
Power Consumption - 7W;

PCEs: 1.25 GHz; 32 Independent Vector Units;
Vectorial Operations up to 256Bytes per Unit;
Vector Register Bank of 8x256Bytes each;
Scalar Register Bank of 8x32 bits each;
Latency (cycles): 1-alu, 3-mul. and 20-div. integer units;
Interconnection between vaults: 5 cycles latency;
Host Processor - 1.2GHz ARM Cortex A8; IL1 64KB + DL1 64KB;
Power Consumption - PCE Logic - 3.1W estimated;
Power Consumption - Host Processor - 0.6W;

HMC: Version 2.0 - 8GB - 32 Vaults - 16 Banks per Vault - 4 Links;
Power Consumption - 11W;

three data-intensive micro-benchmarks employed in [1], [3]. The micro-benchmarks are a Linked Lists (varying the address space from continuous to 25%, 50% and 100% random), the Hash Table proposed by [13], and the B+tree implementation of DBx1000 [14].

Figure 2 presents the speedup and energy result for the *FIND* instruction traversing a *Linked List* of 1 M nodes using both the ARM processor with extra LLC memory and our mechanism limited to 64k bytes, as originally proposed by [5]. In the first set (orange bars), the *Linked List* lies contiguously on memory, which means that techniques such as cache lines, streaming and next line prefetches can be exploited. In this case, as the application presents a pure streaming-like behavior and no data-reuse is present, our mechanism can avoid cache latencies, thus accelerating the application even when it uses equal data chunk size of the cache memories (64 bytes). Also, it is possible to note that when the cache size is increased, performance remains the same, mainly because only spatial locality is available in this case. Moreover, as PCE can enlarge the accessed data chunk, it is capable of increasing the performance by speculatively reading up to 8192 bytes of data from memory at once taking advantage of spatial locality behavior. Despite the sequential characteristics of the algorithm presented in Section II-B, the speculative loads mitigate the DRAM access latencies, thus increasing the overall performance by 2.7× when speculating over 64k bytes of data in a window of 8k bytes. On the other

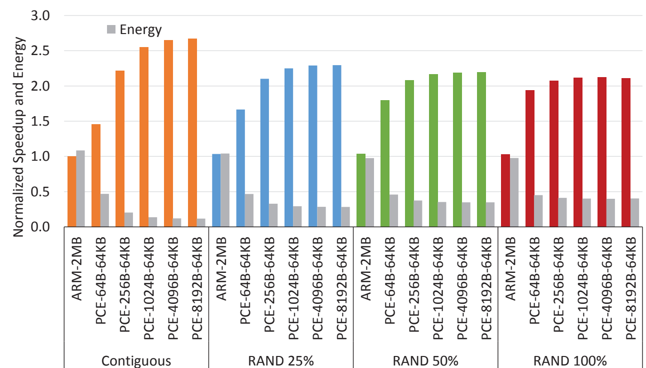

Fig. 2: Linked List 1M nodes
Performance and Energy Normalized to ARM 1MB Cache

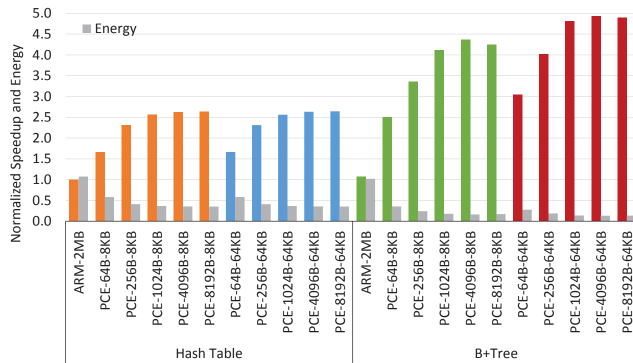

Fig. 3: Hash 1.5M nodes and B+Tree 3M nodes Performance and Energy Normalized to ARM 1MB Cache

hand, when the entire data is placed randomly in memory (red bars in Figure 2), the amount of data reuse increases, leveraging the relevance of the cache size presented in the system. However, even when doubling the cache size of the baseline, the performance increment achieved by the ARM processor is limited to 5%. In contrast, PCE can speedup 2.15× when taking advantage of a speculative window of 4k bytes and making use of 64k bytes of registers. In this way, our design reduces energy consumption by 60% of the energy consumed by the baseline. Also, one can notice that when speculating over 8192 bytes of data, the achieved speedup is of 2.05×, showing the benefits reconfigurable speculation can bring to our evaluation.

Figure 3 presents results for Hash Table and B+Tree operating over 1.5M nodes and 3M nodes respectively. By making use of the proposed *Find* operation through Hash Table structures leads to a speedup of 2.7×, using 64k bytes of registers with 8192 bytes of speculative load operand (blue bars), while the baseline with additional cache memory (2MB) improves performance by only 7%. One can notice that due to the reduced temporal locality presented in the Hash Table benchmark, our mechanism is not able to take advantage of its 64k bytes of registers, since when only 8k bytes of total registers are available (orange bars), the acceleration is slightly lower, achieving 2.55× of speedup. Besides that, PCEs reduces energy consumption by 65%, while improving performance by up to 2.7× on the Hash Table structure.

The B+Tree application presents a more intensive temporal locality, which can be observed on green and orange bars of the Figure 3. The additional cache size available for the baseline can provide only 7% of performance improvement, showing that the reduced spatial locality dictates an important rule on overall performance. On the other hand, PCE can take advantage of its configurable operands and vector FUs. The operand size set to 4096 bytes represents the best compromise between temporal and spatial locality for this application. Also, as the temporal locality is considerable, the 64k bytes of registers are better exploited. Furthermore, the vector operations and simple TLB design facilitate the calculus of the next addresses (16 per node on B+Tree), which increases overall performance. In this way, our design can achieve a speedup of 4.94× when compared to the baseline, while the energy reduction is near to 87%.

IV. Conclusions and Future Work

In this paper, we present an efficient approach to process complex data structure by tackling the bottleneck present in pointer chasing operations. Our mechanism takes advantage of the parallelism present in 3D-stacked memories and a state-of-the-art PIM design that includes reconfigurable vector units. Our mechanism, named PCE, make use of speculation to perform pointer-chasing operations with reduced energy consumption and improved performance. In a majority of cases, a wider cache line can benefit applications because operands are contiguously allocated in memory. Our device is capable of accelerating linked list traversal and hash tables by a factor of 2.5, and b+tree by 4.9 while consuming 13% of the baseline's energy. In future works, we expect to evaluate how our mechanism behaves in whole applications, as for database and large-scale graphs. Also, we plan to explore popular algorithms that use linked data structures, like PageRank, to be performed in memory.

References

[1] K. Hsieh, S. Khan, N. Vijaykumar *et al.*, "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation," in *Int. Conf. on Computer Design (ICCD)*, 2016.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Int. Symp. on Computer Architecture*, 2015.

[3] B. Hong, G. Kim, J. H. Ahn, Y. Kwon, H. Kim, and J. Kim, "Accelerating Linked-list Traversal Through Near-Data Processing," in *Int. Conf. on Parallel Architectures and Compilation - PACT*, 2016.

[4] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu, "Concurrent data structures for near-memory computing," in *29th ACM Symp. on Parallelism in Algorithms and Architectures*, 2017.

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

[6] G. F. Oliveira, P. C. Santos, M. A. Alves, and L. Carro, "Nim: An hmc-based machine for neuron computation," in *Int. Symp. on Applied Reconfigurable Computing*, 2017.

[7] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ACM SIGARCH Computer Architecture News*, 2013.

[8] G. F. Oliveira, P. C. Santos, M. A. Alves, and L. Carro, "A generic processing in memory cycle accurate simulator under hybrid memory cube architecture," in *Int. Conf. on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2017.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, 2011.

[10] 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.

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

[12] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Symp. on VLSI Technology*, 2012.

[13] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[14] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *Proceedings of the VLDB Endowment*, 2014.