Reconfigurable Vector Extensions inside the DRAM

Marco A. Z. Alves, Paulo C. Santos, Matthias Diener, Luigi Carro
Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
Email: {mazalves, pcssjunior, mdiener, carro}@inf.ufrgs.br

***Abstract*—Near-data processing is emerging as a response to the low memory bandwidth and the high energy costs associated with the data transfer between the processor and the main memory. Proposals that move the execution of vector instructions to the DRAM device present good trade-offs in terms of performance, energy consumption and area. Since these previous proposals usually operate at the frequency of the DRAM, they lose opportunities for improvements due to the long clock period of the memory devices. In this paper, we propose Reconfigurable Vector Extensions (RVX), which use Coarse-Grained Reconfigurable Arrays (CGRAs) to execute vector instructions inside the DRAM. Our mechanism reconfigures the functional units inside the memory in order to combine them and thereby reduce the operation time of the near-data instructions, fully leveraging the potential of near-data processing in the main memory. Comparing to previous near-data processing approaches that move vector instructions to the DRAM, our proposal enables performance gains of up to 31% and reduces the energy consumption by up to 76% of the functional units inside the memory device.**

***Keywords*—Near-data processing; Reducing data movement; Vector instructions; Coarse-Grained Reconfigurable Array;**

I. Introduction

Data movement between the main memory and the processor is a major source of inefficiency in terms of performance and energy consumption, especially for applications that present a high spatial locality and low temporal locality of memory accesses, such as stream applications. For these applications, the memory hierarchy is very inefficient, because large amounts of data are transferred from the memory into the processor, exposing the bottleneck in the interconnection between them. Moreover, for such algorithms the cache hierarchy represents a waste of resources. The reason for this waste is that data that is brought into the caches is used only once and removed as soon as possible to make room for new data.

In this scenario, multiple memory channels and multiple memory controllers are being used together with dedicated point-to-point interconnections in order to reduce the memory bottleneck. Despite these advancements, memory accesses still present a large challenge for the performance and energy consumption of modern multi-core processors. In order to reduce the data transfer impact on the performance and energy consumption, the *near-data processing* concept is becoming more important. Placing processing elements close to the data aims to minimize data movement, performing the processing in the most appropriate location of the memory hierarchy [1].

A possible approach to perform near-data processing is to implement vector operations inside the DRAM devices [2], [3]. In this type of mechanism, several functional units are placed inside the memory device, coupled to the sense amplifiers or the row buffers in order to receive and execute operations triggered by the processor. A typical row buffer size provides 8 KB of contiguous data from multiple devices, which can be processed by a single vector instruction. Such a mechanism is able to achieve high performance improvements compared to a traditional multi-core system. However, the operations are performed in the DRAM cycle, which is usually more than 10 times slower than the processor. For this reason, the gains that can be achieved are limited.

In this paper, we propose *Reconfigurable Vector Extensions (RVX)*, which implement a large array of small Coarse-Grained Reconfigurable Arrays (CGRAs) inside the DRAM device in order to execute vector operations inside the main memory. The processor works as a front-end, by fetching, decoding and triggering the reconfiguration and the instructions to be executed inside the RVX. The reconfiguration before the actual execution increases the performance by coupling multiple operations that are performed in the same DRAM clock cycle.

Moving the vector instructions to be executed inside the memory outperforms traditional processor vector instructions for algorithms that present high spatial locality and low temporal locality, such as applications with a stream memory access behavior. This is because such applications cannot benefit from cache memories inside the processors to hide the memory access latency and limitations of the interconnection. By using RVX, these benefits from the near-data processing are increased further. The main contributions of this paper can be summarized as follows:

Multiple system designs: Several setups with varying numbers of functional units per CGRA and read and write ports in the register bank are proposed and evaluated. Our mechanism provides up to 31% (16% on average) of performance improvement over the baseline system that performs near-data processing inside the memory device by using 256 sets of Functional Units (FUs) per device.

Different configuration options: We show that different configuration options become available when increasing the number of resources. Moreover, the experimental evaluation shows that a reduction of the number of instructions sent to the memory correlates with the number of available configurations. Using the possible configurations, the number of instructions triggered by the processor can be reduced by up to 63% (40% on average) with our mechanism.

Fine-grain functional unit selection: Our mechanism reconfigures a large array of small CGRAs to perform the same set of operations. During the reconfiguration, it enables only the necessary functional units. Our evaluation shows that this fine-grain selection is capable of achieving high energy savings.

978-1-4673-7942-7/15/$31.00 ©2015 IEEE

Considering a perfect power gating of unused resources, RVX consumes up to 76% (69% on average) less energy than the baseline system which performs near-data vectorial processing.

II. Reconfigurable Vector Extensions (RVX)

The main focus of this work is to improve the performance of near-data processing approaches that perform vector instruction inside the DRAM. Previous work [3] considers a group of 256 functional units inside the memory device, together with a register bank working on the same width as the row buffer size (8 KB in our case). On the top of this baseline, we propose Reconfigurable Vector Extensions (RVX), which further improve the performance by configuring multiple operations to be executed in the same cycle period of the memory device. RVX can perform load and store operations directly into the register bank. These registers are used to provide data to the large array of small Coarse-Grained Reconfigurable Arrays (CGRAs).

This section introduces RVX in detail, presenting the processor and memory system modifications commonly required by near-data processing mechanisms that execute vector instructions inside the memory. We also present the operations of RVX. For an overview of existing CGRAs, refer e.g. to [4].

A. Execution Overview

Figure 1 depicts the full data-path of a vector instruction sent from the Out-of-Order (OoO) processor to the DRAM. The vector instructions are fetched and decoded by the processor and then sent to the DRAM for execution, avoiding expensive data transfers between the memory and processor.

We assume a memory module formed by 8 DRAM devices, each device containing row buffers of 1 KB. With this configuration, each set of row buffers contains 8 KB of data, which corresponds to 2048 operands of 32 bits (integer or single precision FP). In this way, each vector instruction can use up to 3 internal registers (2 input, 1 output) to perform up to 2048 operations of the same type (compared to e.g. 16 operations in AVX-512).

In order to use the vector instructions, the binary needs to provide such instructions to make usage of the wider operations. To avoid resource conflicts, a sequence of vector instructions needs to be wrapped by a lock, to ensure exclusive access for a specific thread. In addition, RVX requires the configuration bits to be sent by the processor to reconfigure its internal operations. Instead of having one functional unit per operand in the row buffer, we propose to use a small CGRA per operand. Thus, an instruction to reconfigure the CGRAs needs to be sent before the actual operation is performed. We consider that all the CGRAs will be configured to perform the same operations.

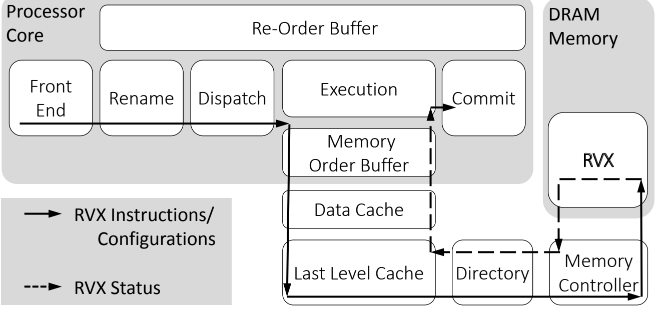


Fig. 1: Data-path illustration of an RVX instruction.

B. Reconfiguration Options

RVX integrates a large array of small CGRAs inside the memory device. One array inside a device consists of 256 cells. Each cell operates over elements of 32 bits. Inside one cell, there is a CGRA that can have multiple functional units. During a normal operation, these functional units are reconfigured in order to act as a single processing element. RVX is designed to select inside the functional units only the specific operation that will be performed. Thus, RVX is capable of disabling the unused logic through power gating [5].

The number of configuration options available depends on the number of functional units (per CGRA) as well as the number of read and write ports in the register bank of our mechanism. In this paper, we propose 5 system designs, varying the number of functional units (between 1, 2 and 3), the number of read ports (between 2, 3 and 4), and write ports (between 1 and 2) in the register bank. These design options present a trade-off between the available resources (area usage) and the performance of RVX. Figure 2 illustrates 9 configuration options enabled by each modification in the number of FUs and read/write ports. RVX supports all of these configurations, but it is not limited to those. For each system design with more resources available, the configurations available using a simpler design are still supported, however, more configuration possibilities become available.

C. Processor Modifications

In the processor, we require an Instruction Set Architecture (ISA) extension to provide the vector instructions. These instructions use a new register bank inside the DRAM to perform operations. Such instructions pass through the pipeline in the same way as a memory load operation. The instructions that do not require memory addresses, such as the RVX lock and unlock operations, will bypass the Address Generation Unit (AGU) and wait to be transmitted inside the Memory Order Buffer (MOB).

The vector instructions are sent to the MOB to be delivered to the memory subsystem. These instructions wait inside the MOB for an answer from the memory subsystem, which returns the status of the operation as successful or raises exceptions. The processor uses this status to control execution flags, such as overflow and not-a-number. The new instructions that perform loads and stores work with virtual addresses and therefore have to be translated by the Translation Look-aside Buffer (TLB) and checked for correct permissions to access the given memory address range. After passing through the TLB, the requests follow the cache memory hierarchy, bypassing the memory caches. The cache directory needs to be changed as well, to ensure a write-back and invalidation of all the modified data in the range of addresses at which the specific vector instruction will operate.

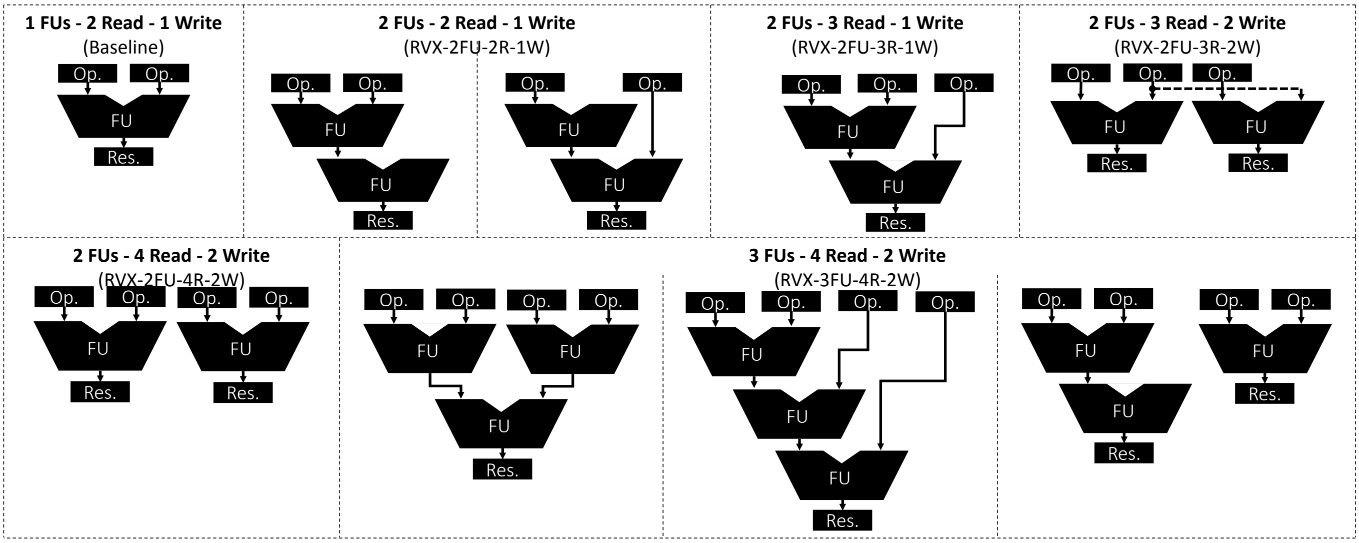


Fig. 2: Illustration of different configurations supported by the RVX FUs.

D. Memory Controller Modifications

In RVX, the memory controller is responsible for handling the instructions and sending them to the DRAM in-order. This reduces the logic required inside the DRAM. Furthermore, the vector instructions use the internal buffers inside the memory controller to wait until they can be executed.

When the memory controller receives the lock operation, it has to reserve the mechanism inside the DRAM to operate only for the thread that requested the lock. In case the memory is already locked, the lock instruction from the requester waits until the DRAM gets unlocked. Locking the mechanisms avoids that one thread modifies the content of registers that are being used by a different thread. Normal memory access requests (both reads and writes) can still be serviced while the mechanism is locked, such that other threads that do not use our instructions can continue to execute.

Once our mechanism is locked, the memory controller can start to issue instructions to be executed by our mechanism inside the DRAM. The memory controller issues both the reconfiguration and the instructions to be executed by the RVX.

E. DRAM Modifications

To perform vector instructions inside the DRAM, we require two main pieces of logic inside the Double Data Rate (DDR) device, a register bank and the RVX. Figure 3 illustrates our mechanism inside a DRAM device. This figure presents a DDR 3 x8 device, even though our mechanism can be easily adapted for different DDRx device layouts (for example, different row bank sizes and data bursts, among others).

The additional register bank inside the DRAM device can be used to store full row buffers from any bank inside the device. Each register is capable of handling an entire row buffer (8,192 bits per device). Thus, open row signals can be issued to different banks to achieve a higher performance. RVX performs operations sequentially. However, we adopted a loop unrolling technique in order to expose the loads and operations in such a way that using the register bank to store the data, the memory controller can issue the row access signals as early as possible, making better use of the bank parallelism inside the DRAM.

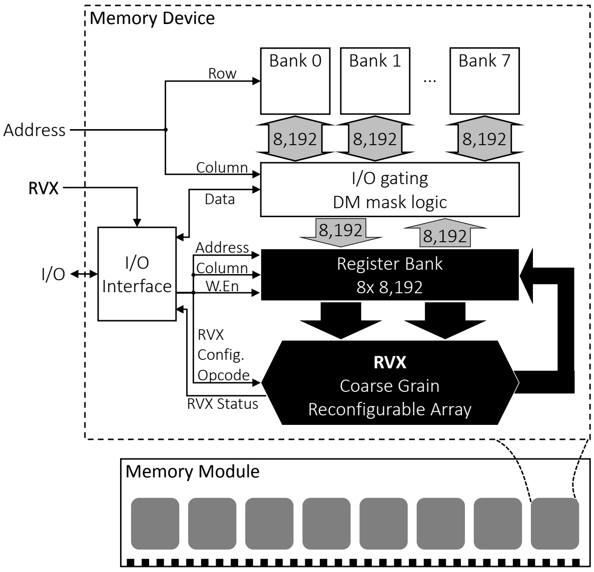


Fig. 3: DDR 3 x8 devices showing the modifications required by our RVX mechanism. All sizes are given in bits.

The vector instructions are executed in-order, however, RVX acts as a restricted data-flow processor. A given operation may start as soon as the registers are ready. To support that data-flow, we provide a flag associated with each register that indicates if the operand is ready. Each vector instruction needs to erase this flag for its destination register, and re-enable it whenever the instruction becomes ready. This system enables the DRAM to open rows from different banks in parallel, and also ensures that once a vector instruction requires operands

TABLE I: Baseline and RVX system configuration.

OoO execution cores:	2 GHz; 16 cores; Front-end 2-wide; 14 stages (3-fetch, 3-decode, 3-rename, 2-dispatch, 3-commit); 24-entry fetch buffer, 32-entry decode buffer, 32-entry ROB; 2-alu, 1-mul. and 1-div. integer units (1-3-20 cycle); AVX-512 capable; 1-alu, 1-mul. and 1-div. floating-point units (5-5-20 cycle); 1-load and 1-store functional units (1-1 cycle); 10 R/W MOB entries;
Branch predictor:	1 branch per fetch; 8 parallel in-flight branches; 4 K-entry 4-way set-associative, LRU policy BTB; Two-Level PAs predictor; 16 K-entry BHT, 2-bits prediction;
L1 data + inst. cache:	32 KB, 8-way, 2-cycle; 64 bytes line; LRU policy; MSHR entries: 8-request, 8-write-back, 1-prefetch; Stride Prefetcher;
L2 cache:	1 MB shared for every 2 cores, 16-way, 4-cycle; 64 bytes line; LRU policy; MSHR entries: 16-request, 8-write-back, 2-prefetch; Inclusive LLC; MOESI coherence protocol; Stream Prefetcher;
Memory controller and interconnection:	Bi-directional ring; Single memory channel; On-chip DRAM controller, Open-row first policy;
DRAM module:	DDR3-1333, 8 burst length at 3:1 bus frequency ratio; 8 GB; Device@166 MHz, 8 DRAM banks, 1 KB per device row buffer; CAS, RP, RCD, RAS and CWD latency (9-9-9-24-7 cycles);
RVX processing logic:	Operation freq.: 166 MHz; Array of 256 CGRAs; Each CGRA with 2 or 3 sets of functional units (int. + fp); Latency (ns): 0.5-alu, 1.5-mul. and 10-div. integer units; Latency (ns): 2.5-alu, 2.5-mul. and 10-div. floating-point units; 1 register bank / device, with 16 registers of 8,192 bits each;

that are not ready yet, execution will stall.

Upon the source registers being ready, the functional units inside the array of CGRAs operate in several steps to process the entire row buffer. All functional units operate at the same frequency as the DRAM device. After completion, every RVX instruction sends a status to the processor. These acknowledgment signals provide important information for the processor regarding the status of each operation, such as overflow, division-by-zero and other exceptions. For instance, in the Intel AVX instruction set, 17 bits are enough to provide the information regarding the operation status [6]. During our mechanism evaluation, we considered an acknowledgment of 64 bits in order to correctly simulate the impact of this transmission on the final performance.

III. Experimental Evaluation of RVX

This section presents our evaluation of RVX, detailing the simulation environment, the application kernels, as well as the performance and energy consumption results of RVX.

A. Configuration Parameters and Baseline

To evaluate our RVX mechanism, we used an in-house cycle accurate microarchitecture simulator. [7], [8]. The simulation parameters are inspired by Intel's Atom OoO processor with the Silvermont microarchitecture [9]. Table I shows the simulation parameters used for our tests. As the baseline for our comparisons, we use a previous near-data computing approach [3] which implements vector instructions inside the memory device (described in Section II). Additionally, performance comparisons for a multi-core processor without near-data processing are presented as well.

B. Application Kernels

For our evaluations, we used four different floating point applications kernels: A **vector sum** using three vectors of 64 MB each. A **stencil** with 5-points over a matrix of 64 MB, adding up the 5 neighboring elements, multiplying the result by two and then storing every result in an output matrix. A **matrix multiplication** using three square matrices (2 source and 1 result) of 9 MB each. A **data search** application that searches a vector of 64 MB for a specific value. To make the amount of operations comparable between different mechanisms, we always put the searched value in the last element of the vector. All applications use floating-point operands.

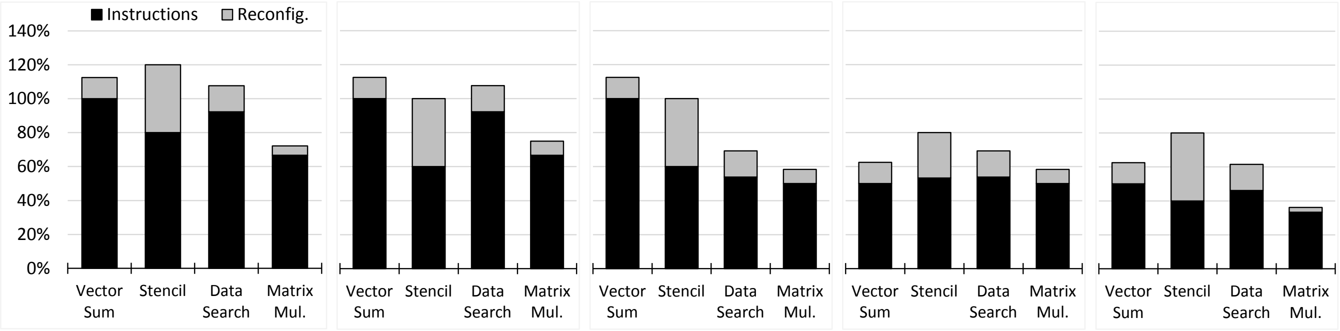
The vector sum and data search applications represent the most favorable case for near-data computing since they do not reuse data and only perform a stream over contiguous vector elements. The stencil benchmark presents some data reuse and can make use of the cache memories. The matrix multiplication application has a high amount of data reuse, thus benefiting greatly from the cache memories. However, for the RVX mechanism, the most favorable applications are those that perform a sequence of operations over the same data. It means that the matrix multiplication and the stencil computation represent the most favorable case for our mechanism.

C. Performance Results

Figure 4 presents the number of arithmetic instructions sent to the RVX for the different systems. As expected, an increasing number of functional units and read/write ports in the register file enables more arithmetic operations to be executed by a single RVX instruction, reducing thus the total number of instructions sent to the RVX. Those applications with a small number of operations over data, such as vector sum and data search, will have an overhead of reconfiguration bits that need to be sent by the processor. However, such a reconfiguration overhead can be hidden by the memory load latency, where the functional units can be configured while data are fetched.

Figure 5 evaluates the performance gains when the number of resources inside the RVX increases. The results show that vector sum and data search require a higher number of read ports in the register file to perform parallel operations. Moreover, we show that despite the reconfiguration time, these applications can benefit from our technique, where vector sum achieved 6% improvements and data search 14%. As expected, the stencil and matrix multiplication benefited more from the reconfiguration, achieving 15% and 31% of performance gains respectively. This is because of the higher amount of operations over the same data. These performance results also give us insight regarding a project decision considering that our mechanism could be implemented using a variable number of functional units and read/write ports in the register bank.

Figure 6 presents a comparison of a multi-core system executing the applications with 16 threads. This system is compared to the near-data processing performed by vector instructions and RVX with different design alternatives. With these results, we can observe that despite the high gains achieved by the near-data processing with vector instructions, further gains can be obtained by using RVX.



(a) RVX-2FU-2R-1W (b) RVX-2FU-3R-1W (c) RVX-2FU-3R-2W (d) RVX-2FU-4R-2W (e) RVX-3FU-4R-2W

Fig. 4: Number of arithmetic instructions and configurations performed for each system compared to the baseline.

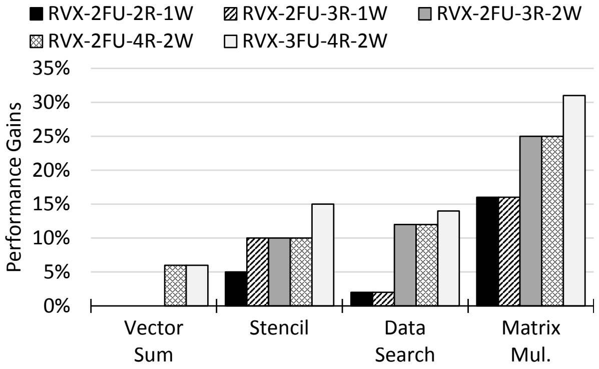


Fig. 5: Performance improvements in percent compared to the baseline without reconfigurable functional units.

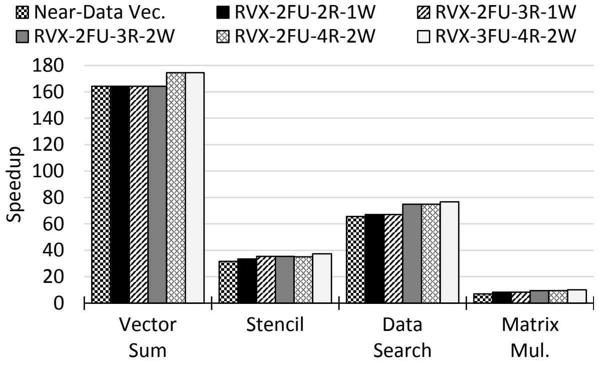


Fig. 6: Overall speedup (in ×) compared to the 16 core processor system.

D. Energy Results

In order to obtain energy results for our mechanism, we describe the functional units in the VHDL language, modeled with power gating techniques. The descriptions were synthesized using the Cadence RTL Compiler tool in order to extract the power components. To obtain the energy of the baseline system (near-data processing with vector instructions), we considered the same functional unit power consumption. However, the baseline is not capable of disabling the unused functional units during the instructions' execution. Since RVX turns off the unused parts of functional units, it is capable of reducing the energy consumption by 64% for vector sum, 71% for stencil computation, 64% for data search and 76% for matrix multiplication compared to previous work.

IV. Related Work

Since off-chip data movement is a major bottleneck for computer systems [10], the main goal of near-data processing is to increase performance and energy efficiency by reducing the data transfer between the processor and DRAM. The Intelligent RAM (IRAM) [11] aims to increase the accessible data width by implementing more memory ports and data buses. To efficiently use this large bandwidth, the authors propose to implement a vector processor inside the DRAM module, where this processor is able to access the vector operands directly from RAM through the extra ports. IRAM extrapolates the system with up to 16 ports of 1024 bits, claiming that it is possible to accelerate the execution and simultaneously reduce energy consumption. However, IRAM requires a large amount of logic, as well as extra buses and memory ports, and could become obsolete for faster processors [12].

Elliot et al. [2] present the Computational-RAM (C-RAM), where multiple functional units are inserted together with the sense amplifiers, computing at the bit level. This implementation works in a bit serial fashion, requiring that data be stored orthogonally to the memory rows. In this way, a radical change in the memory organization is required. The DIVA proposal [13], [14] integrates a 32 bit RISC-like processor within the memory device. Despite the reduced communication between main memory and the host processor provided by this approach, the control overhead is substantial.

In [15], the authors present the Near Memory Processor (NMP), implementing an in-order 2-issue wide co-processor between processor/cache and main memory. Although this approach is limited by memory controller bandwidth, NMP has its own local large width scratchpad memory which enables data accesses with a high bandwidth. Despite the higher performance, the data width managed between main memory and NMP is limited to the original memory bus, maintaining compatibility with current architectures. The performance is also limited by the fact that the proposed architecture must fill the scratchpad memory before processing, an operation which depends on the memory and interconnection performance, which are well-known bottlenecks. Moreover, the programmer must carefully control the content of the scratchpad memory.

Taking advantage of die-stack technology, Zhu et al. [16] present a 3D-DRAM customized logic layer capable of accelerating application-specific data intensive computation. FFT and SpGEMM Application Specific Integrated Circuits (ASICs) were implemented as a 3D-DRAM layer with communication using Through-Silicon Via (TSV). The authors claim bandwidths of up to 668.4 GB/s when 16 banks and 1024 TSVs per bank are used to connect the proposed ASIC to the row buffers. The proposed mechanism does not support general purpose computations, which is the focus of our work.

In [17], the authors present a solution where CGRAs are implemented on top of the memory devices using TSV technology. This approach accesses the data from global I/O (that is, the bus after the DDR I/O gating) using TSV, being able to execute instructions over these data. The benefits emerge from the efficient data transmission system from memory devices to the proposed buffers and CGRA. However, this approach does not use the full data bandwidth inside the DRAM devices. Pugsley et al. [18] insert processor cores in the DRAM memory module, one low-power quad-core processor for each DRAM device, with the logic outside the DDR device. Similar to [17], each processor can access the data width present at columns of each DRAM device, which represents less than 1% of the row buffer size. This small bandwidth limits the amount of data that can processed in parallel. This work provides a comparison with multi-core processors, presenting significant performance gains and energy savings.

Our proposal resembles two previous mechanisms [2], [17]. Compared to previous integrations of functional units inside the DRAM [2], we present a balanced design, reducing the number of processing elements by coupling the functional units to the row buffers. We also introduce a register file which enables a parallelism between the memory load and store and the RVX operations. Compared to the related work which uses CGRA, we propose the usage of such functional units with a larger parallelism and coordinated by the processor to execute vector instructions. Moreover, our proposal is able to select only the configured operations inside the functional units, this enables the mechanism to power gate unused logic.

V. Conclusions and Future Work

In this paper, we introduced the Reconfigurable Vector Extensions (RVX), a new approach to perform near-data processing that implements a large array of small Coarse-Grained Reconfigurable Arrays (CGRAs). RVX is capable of achieving high performance gains by better utilization of the DRAM clock cycle period. Our mechanism reconfigures the functional units inside a CGRA in order to execute multiple operations in a single memory device cycle. Our mechanism also reduces the energy consumption by selecting the specific logic necessary to execute a given instruction, power gating the unused logic.

RVX executes applications up to 174 times faster than a 16-core processor. Our mechanism is up to 31% faster compared to a near-data processing approach without reconfiguration. Energy consumption results show that our mechanism is able to achieve high savings compared to the baseline. For the future, we plan to implement RVX in a Hybrid Memory Cube (HMC) environment, which presents different trade-offs between area and energy consumption. We will also evaluate more benchmarks with different memory access behaviors.

Acknowledgments

The authors gratefully acknowledge the support of CNPq.

References

[1] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, July 2014.

[2] D. Elliott, M. Stumm, W. Snelgrove, C. Cojocaru, and R. McKenzie, "Computational ram: Implementing processors in memory," *Design and Test of Computers*, vol. 16, 1999.

[3] M. A. Z. Alves, P. C. Santos, F. B. Moreira, M. Diener, and P. O. A. Navaux, "Saving memory movements through vector processing in the dram," in *Int. Conf. on Compilers, Architectures and Synthesis of Embedded Systems*, 2015.

[4] Y. Kim, I. Park, K. Choi, and Y. Paek, "Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture," in *Int. Symp. on Low power electronics and design*, 2006.

[5] Y. Saito, T. Shirai, T. Nakamura, T. Nishimura, Y. Hasegawa, S. Tsutsumi, T. Kashima, M. Nakata, S. Takeda, K. Usami, and H. Amano, "Leakage power reduction for coarse grained dynamically reconfigurable processor arrays with fine grained power gating technique," in *Int. Conf. on Engineering and Computer Education*, 2008.

[6] Intel, "Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual," Tech. Rep., 2012.

[7] M. Alves, "Increasing energy efficiency of processor caches via line usage predictors," Ph.D. dissertation, Universidade Federal do Rio Grande do Sul, May 2014.

[8] M. A. Z. Alves, M. Diener, F. B. Moreira, C. Villavieja, and P. O. A. Navaux, "Sinuca: A validated micro-architecture simulator," in *High Performance Computation Conference*, 2015.

[9] Intel, "Intel Atom Processor E3800 Product Family," Tech. Rep., 2015.

[10] D. P. Zhang, N. Jayasena, A. Lyashevsky *et al.*, "A new perspective on processing-in-memory architecture design," in *Workshop on Memory Systems Performance and Correctness*, 2013, p. 71–73.

[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] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang, "Evaluation of existing architectures in iram systems," in *Workshop on Mixing Logic and DRAM*, 1997.

[13] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The architecture of the diva processing-in-memory chip," in *Int. Conf. on Supercomputing*, 2002.

[14] T.-J. Kwon, J.-S. Moon, J. Sondeen, and J. Draper, "A 0.18 μm implementation of a floating-point unit for a processing-in-memory system," in *Int. Symposium on Circuits and Systems*, 2004.

[15] M. Wei, M. Snir, J. Torrellas, and R. B. Tremaine, "A near-memory processor for vector, streaming and bit manipulation workloads," University of Illinois at Urbana-Champaign, Dept. of Computer Science, Tech. Rep., 02 2005.

[16] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," in *Int. 3D Systems Integration Conf.*, 2013.

[17] A. Farmahini-Farahani, J. Ahn, K. Compton, and N. Kim, "Drama: An architecture for accelerated processing near memory," *Computer Architecture Letters*, no. 99, 2014.

[18] S. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Comparing implementations of near-data computing with in-memory mapreduce workloads," *IEEE Micro*, vol. 34, no. 4, pp. 44–52, July 2014.