abstractPIM: Bridging the Gap Between Processing-In-Memory Technology and Instruction Set Architecture

Adi Eliahu, Rotem Ben-Hur, Ronny Ronen, and Shahar Kvatinsky
Andrew and Erna Viterbi Faculty of Electrical Engineering
Technion - Israel Institute of Technology
Haifa, Israel 3200003
{adieliahu, rotembenhur}@campus.technion.ac.il, ronny.ronen@technion.ac.il, shahar@ee.technion.ac.il

***Abstract*—The von Neumann architecture, in which the memory and the computation units are separated, demands massive data traffic between the memory and the CPU. To reduce data movement, new technologies and computer architectures have been explored. The use of memristors, which are devices with both memory and computation capabilities, has been considered for different processing-in-memory (PIM) solutions, including using memristive stateful logic for a programmable digital PIM system. Nevertheless, all previous work has focused on a specific stateful logic family, and on optimizing the execution for a certain target machine. These solutions require new compiler and compilation when changing the target machine, and provide no backward compatibility with other target machines. In this paper, we present abstractPIM, a new compilation concept and flow which enables executing any function within the memory, using different stateful logic families and different instruction set architectures (ISAs). By separating the code generation into two independent components, intermediate representation of the code using target independent ISA and then microcode generation for a specific target machine, we provide a flexible flow with backward compatibility and lay foundations for a PIM compiler. Using abstractPIM, we explore various logic technologies and ISAs and how they impact each other, and discuss the challenges associated with it, such as the increase in execution time.**

***Index Terms*—Memristor, processing-in-memory, RRAM, stateful logic, ISA**

I. Introduction

In recent years, a plethora of data-intensive applications has been developed. These applications require massive data transfer between the memory and the central processing unit (CPU), and have raised the need for processing-in-memory (PIM) [1, 2]. Various new and emerging memory technologies, *e.g.*, resistive random access memory (RRAM) [3], often referred to as memristors, have been explored lately for the purpose of PIM. By applying voltage across the device, it performs switching between two resistance values, high resistance value (R_{OFF}) and low resistance value (R_{ON}), therefore can function as a binary memory element. In addition to their storage capabilities, memristors can be also used for computation - both application specific and general purpose. Several methods have been proposed to use the memristor as a computation unit for a specific task, *e.g.*, vector-matrix multiplication using analog computation [4]. In this manner, the dual-function memristor can perform efficient computing and reduce data transfer requirements between the CPU and the memory. Numerous accelerators integrating analog memristor-based computations have recently been developed [5].

Together with the approach of using memristors to accelerate application-specific architectures, a different approach, called 'stateful logic', uses memristive memory cells as building blocks to construct logic gates within the memory array. In this paper, we focus on stateful logic rather than analog computation. Stateful logic enables programmable general-purpose architectures since every memristive cell can be used as a storage element, as well as an input, output or a register. Several memristor logic gate families have been designed, including MAGIC [6], IMPLY [7], and resistive majority [8].

Some stateful logic families can be easily integrated within a memristive crossbar array with minor modifications. Designing a functionally complete logic gate set using such a family, *e.g.*, a MAGIC NOR gate, enables in-memory execution of any function. There are many logic gate families which have been explored in the literature, and each of them has different advantages. Previous efforts to execute a function within the memory concentrated on utilizing a specific PIM family and optimizing the latency, area or throughput using this technology, *e.g.*, SAID [9] and SIMPLER [10] for MAGIC technology [6] and K-map based synthesis [11] for IMPLY [7].

While current approaches have substantially improved the latency, area or throughput of a logic function execution, they are strongly dependent on the PIM technique and its basic operations, and therefore are bound to a specific target machine, *i.e.*, the machine on which the logic function is executed. Flexibility in the used PIM technology has many motivations since different logic families have different advantages. For example, MAGIC provides memristive crossbar compatibility and high parallelism, whereas CRS [12] provides flexibility by executing 16 Boolean functions in a single operation.

In this paper, we show a new hierarchical compilation method for PIM which is not restricted to a certain PIM technology by separating the code generation into two components: (1) intermediate code generation using target independent instruction set architecture (ISA), (2) microcode generation for a specific target machine and PIM technology, and executing the code using a third component: (3) runtime execution. In the first component, which is independent of the PIM technology, the compiler generates a compiled program that consists of target independent instructions. In the second component, performed by the PIM technology provider, these instructions are translated into an execution sequence of micro-

operations supported by the target machine. In the third component, at runtime, the compiled code instructions are sent from the CPU to the memory controller, which contains the instruction execution sequences from the second component. The controller translates the instructions into micro-operations and sends them to the memory. This third component is similar to an instruction-level opcode being executed using micro-operations in the x86 processors [13].

Figure 1 demonstrates the first and third flow components of a half adder logic for different ISAs and target machines. The first two implementations, shown in Figure 1(a) and 1(b), demonstrate the use of the same target machine while using different ISAs. The code is compiled for a machine that its PIM technology supports only MAGIC NOR logic gates. However, the first example targets a controller which supports only NOR ISA commands, whereas the second example supports all the 2-input and 1-output logic functions as its ISA. In the first component, a netlist and compiled program composed of the ISA commands, dubbed *instructions*, are generated. In Figure 1(a), the netlist is composed of five logic gates that implement the half adder logic, and in Figure 1(b) it is composed of two gates (AND and XOR). The number of gates in the netlist is a representative of both the code size (or number of commands sent from the CPU to the PIM machine), and the control load between the CPU and the memory controller. We will refer to it for the rest of the paper as *code size*. The code size is also a means of estimation of the code abstraction achieved by our flow. In these examples, the code sizes are five and two, respectively. The second component is the microcode generation, where each command is translated to a sequence of MAGIC NOR operations and is embedded in the controller. In the third component, the code is executed. The commands are sent from the CPU to the controller, and then from the controller to the memory; hence, the code size is reduced with minimal changes to the in-memory implementation, namely, adding a few states to the memory controller to support other operations.

Figures 1(b), 1(c) and 1(d) demonstrate the use of the same ISA while using different target machines. These three examples use all 2-input logic functions as their ISA, but the first machine uses MAGIC NOR technology, the second uses MAGIC NAND technology and the third uses all MAGIC 2-input logic functions. This example demonstrates the ISA definition flexibility and command hierarchy enabled by our method, and the possible reduction in code size and reduction in the control load between the CPU and the memory controller. It also demonstrates the backward compatibility feature; in Figures 1(c)-(d), machines with technologies which enable lower execution time are used, and yet the generated intermediate code is backward compatible with other PIM technologies. The separation into two independent code generation components also enables the exploration of the impact of the ISA on the used target machine and vice versa.

This paper makes the following contributions:

1) Development of technology-independent and ISA-flexible flow for executing any logic function to a memristive crossbar array. Our technique, called abstractPIM, presents a hierarchical view and includes three components. It is a solid foundation for implementation of compilers for general-purpose memristive PIM architectures.
2) Examining the impact of the ISA and the target machine on each other using abstractPIM, in terms of flexibility, performance and code size.
3) A 56% reduction of the control load between the CPU and the memory controller as compared to state-of-the-art solutions [10], demonstrated for different benchmarks.

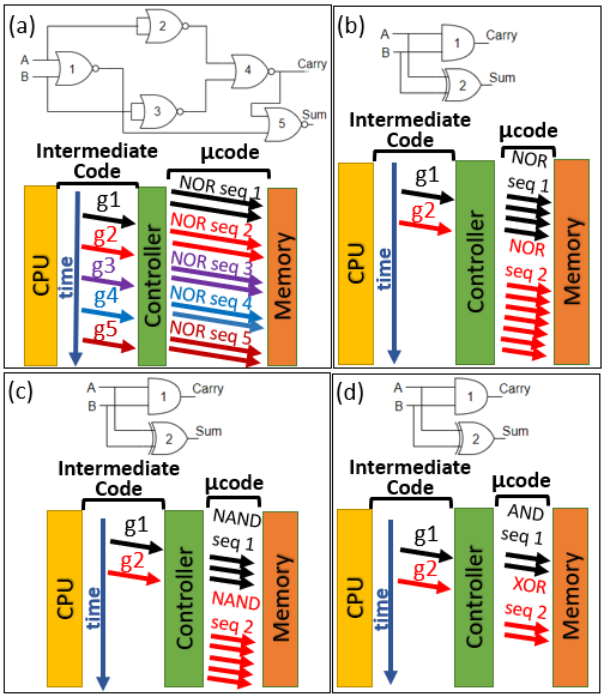

Fig. 1. Compilation example for a half adder using various ISAs and target machines. (a) A NOR ISA and MAGIC NOR target machine. (b) All 2-input and single-output ISA and MAGIC NOR target machine. (c) All 2-input and single-output ISA and MAGIC NAND target machine. (d) All 2-input and single-output ISA and 2-input and single-output MAGIC target machine.

II. Background and Related Work

A. Stateful Logic

In stateful logic families [14], the logic gate inputs and outputs are represented by memristor resistance. We demonstrate the stateful logic operation using MAGIC [6] gates, which are used as a baseline in this paper. Figure 2(a) depicts a MAGIC NOR logic gate; the gate inputs and output are represented as memristor resistance. The two input memristors are connected to an operating voltage, V_g, and the output memristor is connected to the ground. The output memristor is initialized at R_{ON} and the input memristors are set with the input values. During the execution, the resistance of the output memristor changes according to the ratio between the input values and the initialized value at the output. For example, when one or two inputs of the gate are logical '1', according to the voltage divider rule, the voltage across the output memristor is higher than $\frac{V_g}{2}$. This causes the output memristor to switch from R_{ON} to R_{OFF}, matching the NOR function truth table. The MAGIC NOR gate can be integrated in a memristive crossbar array row,

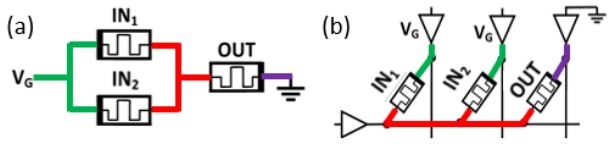

Fig. 2. MAGIC NOR gates. (a) MAGIC NOR gate schematic. (b) MAGIC NOR gate in a crossbar array configuration.

as shown in Figure 2(b). This enables massive parallelism in executing gates in different rows in the same clock cycle.

B. Logic Execution within a Memristive Crossbar Array

Unlike CMOS logic, execution of an arbitrary logic function with stateful logic is performed by a sequence of operations and takes several clock cycles. In each clock cycle, one operation can be performed on a single row, or on multiple rows concurrently. A valid logic execution is defined by mapping of every gate in the desired function to several cells in the crossbar array, and operating it in a specific clock cycle.

Many tools to generate the sequence of operations and map them into the memristive crossbar array cells have been discussed in the literature, *e.g.*, YADAV [15] and SIMPLER [10]. However, the gap between target machine constraints and architectural design choices, *e.g.*, ISA, has never been addressed. Attempts have been made in existing mapping tools to support complex operations in the in-memory execution, *e.g.*, 4-input LUT function [9]. However, their flexibility is limited and they do not completely separate the intermediate code generation and microcode generation, therefore they impose target machine and ISA dependency and do not provide backward compatibility with other target machines.

III. abstractPIM: Three-component Code Execution Flow for PIM

The abstractPIM flow includes two code generation components and one execution component. In the first component, *intermediate representation generation*, the program is compiled into a sequence of target independent instructions based on a defined ISA. In the second component, *microcode generation*, each instruction is translated into micro-operations that are supported by the target machine. The translation is performed once per instruction, and is embedded in the controller design. We adopt an existing mapping flow and modify it to support different ISAs and PIM technologies. In the third component, *runtime execution*, the instructions in the compiled code are sent from the CPU to the controller, which translates them into micro-operations and sends them to the memory.

Existing logic execution methods use a set of basic logic operations to implement a logic function. They rely on a memory controller which is configured to perform these operations by applying voltages on the rows and columns of the memory array. In this paper, we assume that the memory controller is configured to perform several logic operations, dubbed *instructions*. Their execution sequence is determined according to a specific target machine and the PIM technology it supports. For example, if the ISA includes an AND instruction and the used technology is MAGIC NOR, 3 computation operations and 1 initialization cycle will be executed one after the other to run the AND instruction, as demonstrated in Figure 1(b), gate 1. An alternative PIM technology that consists of NAND gates will perform the same AND instruction using two NAND computations and one initialization cycle (Figure 1(c)). The instruction execution using different PIM technologies may differ in the execution time and cell usage. Our approach raises the system abstraction level and reduces the flow dependency of the specific PIM technology. It also moves one step closer towards defining a general instruction set to a memristor-based PIM architecture and designing its compiler.

The controller support of complex instructions also reduces the code size and hence the code transfer between the CPU and the memory controller. However, there is a code size and execution time trade-off; the reduction in the code size may cause an execution time penalty. For example, in Figure 1, the first NOR-based implementation takes $5T_{NOR}$ clock cycles to operate, where T_{op} is the number of clock cycles required for execution of an *op* operation. The second implementation, however, takes a total of $T_{AND} + T_{XOR}$ clock cycles. In a machine which supports MAGIC NOR operations, the first implementation takes 10 clock cycles (2 cycles per NOR), and the second implementation takes 11 clock cycles (4 for the AND2 gate and 7 for the XOR2 gate, according to Table I). Some execution cycles are computation cycles and some are initialization cycles, as further elaborated is Section V.

The instruction hierarchy in abstractPIM improves the flexibility of the compilation flow, as demonstrated in Figures 1(b)-(d). This is similar to high-level programming compared to assembly coding, which can improve flexibility at the cost of execution time penalty. While we demonstrate it using MAGIC-based logic families, the flow can be easily used for other target machines and stateful logic families. In our study, we choose different groups of ISAs, and different target machines that support different logic families. We demonstrate how they can be used to execute different benchmarks, and analyze the code size and execution time of the configurations.

IV. Case Study: Vector-Matrix Multiplication

We showcase our flow with a vector-matrix multiplication (VMM) benchmark (a 5 element vector and a 5×5 matrix with 8-bit elements), which is useful in many applications, *e.g.*, neural networks. The benchmark is tested over a target machine with 1024-sized memristive memory row that supports the MAGIC NOR logic family. The supported set of operations (NOT, NOR2) by the target machine is called *TS0*. Other logic families are discussed in Section VI. We first compile the benchmark for a basic case, where the ISA is also the technology set, *i.e.*, *TS0*. The selection of this ISA enabled a fair comparison between abstractPIM and existing logic execution methods, such as SIMPLER [10], which do not use a two-component code generation process. The used technology sets supported by the target machines we use and their instruction parameters are listed in Table I. Each instruction has three parameters: the number of inputs (I), the number of outputs (O) and the number of execution cycles (T). The first two parameters are technology independent, whereas

TABLE I
INSTRUCTION EXECUTION CHARACTERISTICS FOR MAGIC FAMILIES

Instruction	I	O	T_0	T_1	$TS0$	$TS1$	$IS2$	$IS3$
NOT	1	1	1+1	1+1	✓	✓	✓	✓
NOR2	2	1	1+1	2+1	✓	✓	✓	✓
NOR3	3	1	3+1	3+1	-	-	-	✓
NOR4	4	1	5+1	4+1	-	-	-	✓
OR2	2	1	2+1	1+1	-	✓	✓	✓
OR3	3	1	4+1	2+1	-	-	-	✓
OR4	4	1	6+1	3+1	-	-	-	✓
AND2	2	1	3+1	1+1	-	✓	✓	✓
AND3	3	1	6+1	2+1	-	-	-	✓
AND4	4	1	9+1	3+1	-	-	-	✓
NAND2	2	1	4+1	2+1	-	-	✓	✓
NAND3	3	1	7+1	3+1	-	-	-	✓
NAND4	4	1	10+1	4+1	-	-	-	✓
XOR2	2	1	6+1	5+1	-	-	✓	✓
XOR3	3	1	11+1	9+1	-	-	-	✓
XOR4	4	1	16+1	15+1	-	-	-	✓
XNOR2	2	1	5+1	5+1	-	-	✓	✓
XNOR3	3	1	11+1	6+1	-	-	-	✓
XNOR4	4	1	16+1	8+1	-	-	-	✓
IMPLIES	2	1	2+1	2+1	-	-	✓	✓
!IMPLIES	2	1	2+1	2+1	-	-	✓	✓
MUX	3	1	7+1	4+1	-	-	✓	✓
HA	2	2	7+1	6+1	-	-	✓	✓
HS	2	2	6+1	5+1	-	-	✓	✓

The execution time format is $T_c + T_i$, where T_c is the number of computation cycles and T_i is the number of initialization cycles.

the last parameter is technology dependent. The parameter corresponding to technology set N is T_N. For example, the OR instruction has two inputs and a single output ($I = 2$, $O = 1$), and requires, when using a target machine that supports *TS0*, three clock cycles for execution ($T_0 = 3$, two computation cycles and one initialization cycle). Using ISA=*TS0* for the VMM benchmark, there are 25470 execution cycles, out of which, half are initialization cycles and half are computation cycles. Therefore, the code size is 12735 instructions.

In attempt to reduce the code size, we used *IS2*, which contains all the functions with 1 or 2 inputs and 1 output, excluding trivial functions, *e.g.*, constant '0' and identity functions[1]. The set also includes common combinational functions with more than 2 inputs or more than 1 output. Since the number of such functions is large, even for a small number of inputs, we chose three functions which, according to experiments we conducted, were useful in certain benchmarks: half adder [HA], multiplexer [MUX] and half subtractor [HS]. Because of the the circular dependency limitation of our flow, which is further elaborated in Section V, some useful instructions, *e.g.*, 4-bit adder, could not be used. Using *IS2*, code size is reduced by 52%, but execution time is increased by 16%.

To demonstrate the benefit of a larger number of instruction inputs and reduce the execution time, *IS3* was defined. It contains the *IS2* instructions, and the 2-input and single output symmetric functions from *IS2* extended to 3 and 4 inputs. Using *IS3*, lower execution time and code size, as compared to *IS2*, are achieved. The execution time is increased by only 8%, and the code size is reduced by 57%, as compared to *TS0*.

V. ABSTRACTPIM FLOW AND METHODOLOGY

The flow of abstractPIM is composed of three components, as shown in Figure 3. In the first component, the *intermediate representation generation*, the input is a Verilog program. The program is synthesized using the Synopsys DC synthesis tool [17], where the synthesis standard cell library includes the ISA in .lib format. The Synopsys DC synthesis tool was chosen since it supports multi-output cell synthesis. Then, a compiled program is generated using a modified and extended version of the SIMPLER mapping tool [10]. This tool builds a directed acyclic graph (DAG). In its original form, every node represents a NOR gate in the netlist, since SIMPLER was designed specifically for the MAGIC NOR family [6]. In the modified mapping tool, each node represents a wider variety of instructions based on the ISA. Using the DAG, the inputs and outputs of the instructions are mapped to row cells in the memristive array, and a compiled program is generated. The *I* and *O* parameters are used to build the DAG and are technology-independent. The *T* parameters (see Table I), which are technology-dependent and determined in the second component, are not used for compilation. Therefore, a complete separation between the code generation components and backward compatibility with other target machines is achieved.

The second component of the abstractPIM flow is *microcode generation*. For each instruction, a microcode is generated by synthesizing the instruction to a micro-operation netlist and then to an execution sequence, which includes mapping to the memristive crossbar array and intermediate computation cell allocation based on specific PIM technology. The second component input is the instruction implemented in Verilog. The instruction is synthesized using the Synopsys DC synthesis tool for a specific PIM technology, described in the synthesis standard cell library. In this paper, we demonstrate the flow with the MAGIC [6] family, and therefore we extended the SIMPLER [10] mapping tool to support different MAGIC operations instead of only MAGIC NOR. The execution times, listed in Table I, were calculated using this flow. The second component of abstractPIM can be replaced by handcrafted execution sequences or other mapping tools, depending on the PIM technology in use, which may produce even faster execution sequences. In the third component, *runtime execution*, the two components outputs are used for full program execution. Instructions are sent from the CPU to the controller, and micro-operations are sent from the controller to the memory.

The SIMPLER mapping tool [10] traces the number of available cells, and when they are all occupied, adds a cycle which initializes several unused cells in parallel. However, not all stateful logic families use initialization, therefore initialization cycles should not be part of the first component of the flow so we remove them. In the second component, since the flow is demonstrated using the MAGIC [6] family, we perform initialization. As opposed to SIMPLER, the second component is not aware of the full program and instruction dependencies, therefore optimized parallel initialization cannot be performed. Instead, output and intermediate computation cell initialization is performed at the first cycle of each instruction execution (if needed, additional initialization cycles can be added to the instruction execution sequence). Overall, the component separation enables flexibility and backward compatibility at the cost of execution time penalty.

In both code generation components, each standard library

[1]identity functions, which are in fact copy operations, can be useful in other mapping methods [9, 16], but not in our row-based flow.

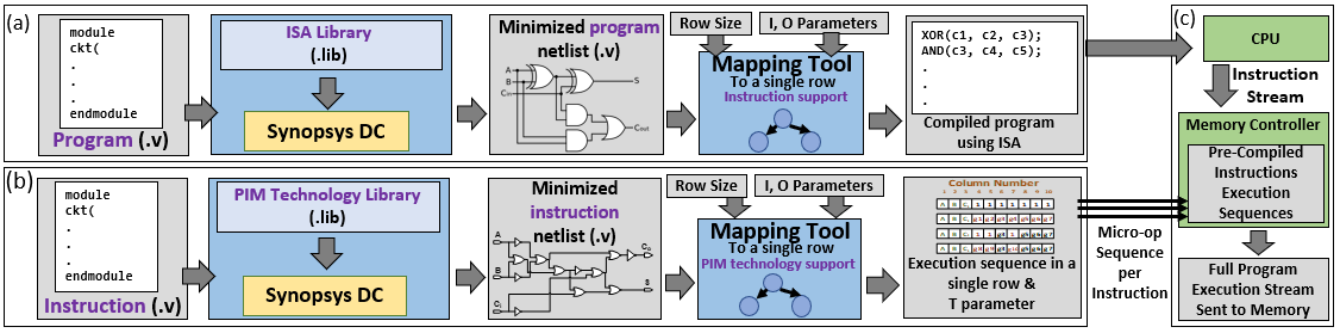

Fig. 3. abstractPIM general flow is composed of three components, two components are for code generation (differences between them are marked with purple.), and the last component is for execution. (a) Intermediate representation generation. (b) Microcode generation. (c) Runtime execution.

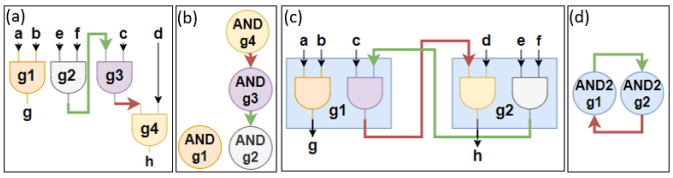

Fig. 4. Compilation with multi-output instructions which creates a circular dependency. (a) Generated netlist using single output gate synthesis. (b) Generated netlist using multi-output gate synthesis. (c) The graph that represents netlist (a), which is a DAG and can be used for the mapping algorithm. (d) The graph that represents netlist (b), which includes a cyclic dependency.

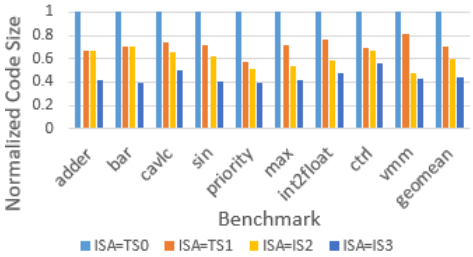

Fig. 5. Normalized code size with respect to TS0 for different ISAs.

cell includes several parameters. Since existing commercial synthesis tools are CMOS-oriented, we set these parameters differently and according to our memristor synthesis flow. Propagation delays, which are relevant for propagating signals in CMOS logic, are irrelevant in the context of memristor logic, where the execution time of each logic operation is a single clock cycle, and are set to 0. The area parameter is set equal for all the library cells, thus the synthesis does not prefer any particular cell, and minimizes the number of cells in the netlist, *i.e.*, minimizes the code size.

AbstractPIM supports multi-output instructions, but not all kinds of multi-output instructions can be used in it, since some may lead to *bogus dependencies* that hinder the execution mapping. Figure 4 shows an example of such bogus dependencies, in which, the input is the function code: $g = ab$, $h = cdef$. In Figure 4(a), the code is compiled using single-output instructions (AND2 instruction), and in Figure 4(b), it is compiled using multi-output instructions (an instruction which computes two AND2 operations, marked in blue). Figures 4(c) and 4(d) show the graphs corresponding to the netlists in Figures 4(a) and 4(b), respectively. While there is no combinational loop in both netlists, a circular dependency was created between the two 2-output AND2 cells. Since abstractPIM relies on the graph acyclic structure, instructions which might cause cyclic dependency cannot be used. A sufficient condition that guarantees no such loops will be created, is to use only cells in which all the outputs depend on all the inputs, *e.g.*, half adder. Future work will ensure support of any multi-output instruction, thus enabling more flexibility in planning the ISA.

After developing abstractPIM and composing the ISAs, the code size and execution time were explored. We show the two metrics separately, due to the absence of a natural metric that combines both of them[2]. We used the EPFL benchmark suite [18]. Each benchmark was tested with different technology sets and ISAs, listed in Table I, within a 512-sized row. One benchmark, *max*, could not be mapped to a 512-sized row and was therefore tested with a 1024-sized row.

VI. Results

The abstraction achieved by our flow using different ISAs reduces the code size as compared to an implementation based on a specific PIM technology. Figure 5 shows the code size needed for the execution of each benchmark using different ISAs: *TS0*, *TS1* (used as ISAs and not as technology sets), *IS2* and *IS3*. The code size is determined only by the ISA, and is independent of any target machine. Since the chosen sets are subsets of each other, *i.e.*, $TS0 \subset TS1 \subset IS2 \subset IS3$, then $CS_{TS0} > CS_{TS1} > CS_{IS2} > CS_{IS3}$, where CS_{set} is the code size of *set*. Using *TS1*, *IS2* and *IS3* reduced the code size by 30%, 40% and 56% compared to *TS0*, respectively.

For execution time evaluation, we compiled the benchmarks with the different ISAs and for the different target machines to demonstrate the flexibility and PIM technology independence achieved by our flow. We used two "native" configurations: *TS0/TS0*, *TS1/TS1*, and four "abstract" configurations: *TS0/IS2*, *TS0/IS3*, *TS1/IS2*, and *TS1/IS3*, where the notation is target-machine/ISA. We also compare the results to a single-component target-specific flow, SIMPLER [10].

The results are shown in Figure 6. When comparing *TS0/TS0* with SIMPLER, the execution time is approximately doubled, since in our flow, every NOR or NOT operation takes an additional cycle for initialization. In SIMPLER, which operates at full program context and not at single instruction context, multiple initialization cycles can be combined and therefore the number of initialization cycles is negligible.

[2]Weighted product of code-size and execution-time were found misleading.

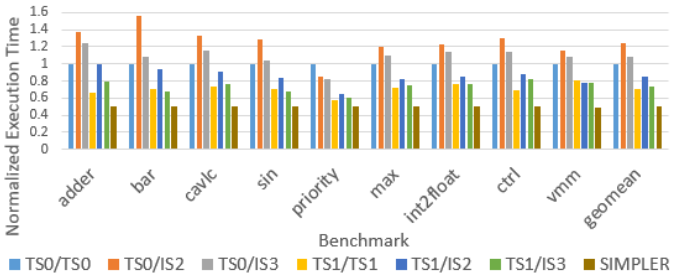

Fig. 6. Normalized execution time with respect to *TS0/TS0* for the different target machines and ISAs.

When comparing target machines that use native configurations (*TS0/TS0* vs. *TS1/TS1*) we observe that the target machine which is more capable (*TS1*) runs faster (30%). When comparing target machines that use the same abstract configuration (*TS0/IS2* vs. *TS1/IS2* and *TS0/IS3* vs. *TS1/IS3*) we also observe that the target machine which is more capable runs faster (32% and 33%, respectively). When comparing the execution time of a native configuration (*TS0/TS0* and *TS1/TS1*) with that of an abstract configuration using the same target machine, we see that the abstract configuration is slower. *TS0/IS2* and *TS0/IS3* are 24% and 8% slower than *TS0/TS0*, respectively. Comparing the native *TS1/TS1* configuration with the relevant abstract configurations exhibits similar results.

The above observations are quite expected. An important but less obvious benefit of abstractPIM is shown when changing a target machine. For example, when the target machine is upgraded from *TS0* to *TS1*, a program that has been compiled natively (*TS0/TS0*) executes the same number of cycles when running on *TS1* (if $TS0 \subset TS1$, otherwise even slower). However, a program that has been compiled in the first place using *IS3* (*IS2*) runs 27% (16%) faster than on the original machine – no recompilation needed. This is reflected by comparing *TS1/IS3* (*TS1/IS2*) vs. *TS0/TS0*.

Another observation is that among abstract ISAs, higher abstraction usually exhibits better performance, as shown by comparing *TS0/IS3* vs. *TS0/IS2* (13%) and *TS1/IS3* vs. *TS1/IS2* (13%). With higher abstraction it is expected that the execution time will increase compared to native configuration since using basic instructions allows finer granularity. On the other hand, using abstract ISAs reduces the number of instructions, together with the number of initialization cycles. The two opposite trends cause different benchmark behaviors.

The flexibility and code size reduction advantages of abstractPIM come with a cost. The additional execution cycles per benchmark result in proportional additional energy consumption and lower effective lifetime. We believe that higher abstraction is worth the cost of these limitations.

VII. Conclusions

This paper presents a hierarchical compilation concept and method for logic execution within a memristive crossbar array. The proposed method provides flexibility, portability, abstraction and code size reduction. The abstractPIM flow lays a solid foundation for a compiler for a memristor-based architecture, by enabling automatic mapping and execution of any logic function within the memory, using a defined ISA.

Acknowledgment

This research is supported by the ERC under the European Unions Horizon 2020 Research and Innovation Programme (grant agreement no. 757259).

References

[1] S. Hamdioui *et al.*, "Memristor for computing: Myth or reality?," *DATE*, pp. 722–731, Mar. 2017.

[2] D. Ielmini and H.-S. P. Wong, "In-memory computing with resistive switching devices," *Nature Electronics*, vol. 1, pp. 333–343, Jun. 2018.

[3] M. Angel Lastras-Montaño and K.-T. Cheng, "Resistive random-access memory based on ratioed memristors," *Nature Electronics*, vol. 1, pp. 466–472, Aug. 2018.

[4] W. Woods and C. Teuscher, "Approximate vector matrix multiplication implementations for neuromorphic applications using memristive crossbars," *IEEE/ACM NANOARCH*, pp. 103–108, Jul. 2017.

[5] L. Deng *et al.*, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, pp. 1–48, Mar. 2020.

[6] S. Kvatinsky *et al.*, "MAGIC-memristor-aided logic," *IEEE TCAS II*, vol. 61, pp. 895–899, Nov. 2014.

[7] J. Borghetti *et al.*, "'memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, p. 873—876, Apr. 2010.

[8] E. Testa *et al.*, "Inversion optimization in majority-inverter graphs," *NANOARCH*, pp. 15–20, Jul. 2016.

[9] V. Tenace *et al.*, "SAID: A supergate-aided logic synthesis flow for memristive crossbars," *DATE*, pp. 372–377, Mar. 2019.

[10] R. Ben-Hur *et al.*, "SIMPLER MAGIC: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *IEEE TCAD*, Jul. 2019.

[11] J. Bürger *et al.*, "Digital logic synthesis for memristors," *Reed-Muller*, pp. 31–40, Jan. 2013.

[12] E. Linn *et al.*, "Beyond von neumann - logic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, vol. 23, p. 305205, Jul. 2012.

[13] "P6 family of processors hardware developer's manual." http://download.intel.com/design/PentiumII/manuals/24400101.pdf.

[14] J. Reuben *et al.*, "Memristive logic: A framework for evaluation and comparison," *PATMOS*, pp. 1–8, Sep. 2017.

[15] D. N. Yadav, P. L. Thangkhiew, and K. Datta, "Look-ahead mapping of boolean functions in memristive crossbar array," *Integration*, vol. 64, pp. 152 – 162, Jan. 2019.

[16] R. Ben Hur *et al.*, "SIMPLE MAGIC: Synthesis and in-memory mapping of logic execution for memristor-aided logic," *IEEE/ACM ICCAD*, pp. 225–232, Nov. 2017.

[17] P. Kurup *et al.*, *Logic Synthesis Using Synopsys*. Springer Publishing Company, Incorporated, 2nd ed., 2011.

[18] L. Amarù, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," *IWLS*, 2015.