A Compiler for Parallel and Resource-Constrained Programmable in-Memory Computing

Giulia Meuli* Mathias Soeken* Pierre-Emmanuel Gaillardon† Giovanni De Micheli*
*Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
†Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT, USA

***Abstract*—Solid-state resistive switches have recently enabled low-power, scalable, non-volatile memories. Their proven intrinsic logic operation, allowing design of in-memory computing systems, attracted the attention of the research community. A Programmable Logic-in-Memory (PLiM) computer has been proposed operating on an RRAM memory array and an instruction set to access the in-memory capabilities of the RRAM cells. Since the RRAM's intrinsic logic operation is based on the majority-of-three function, *Majority Inverter Graphs* (MIGs) can be used to implement a compiler that translates Boolean functions into PLiM instructions.**

This work presents a fast MIG-based PLiM compiler aiming at parallelizing RRAM instructions under resource constraints. Considering valid the assumption that all RRAM cells in the PLiM computer can be enabled in parallel, helps evaluating the theoretical potential of the PLiM parallelization as part of a larger architecture exploration effort. More complex scenarios can easily be adapted by our approach. The code is optimized to reduce number of accesses to the memory and its structure enables a very low runtime compared to the state-of-the-art approach. Resource constraints allow to fit the PLiM instructions into a given maximum number of RRAM cells.

I. Introduction

Metal-oxide resistive switching technology is currently being investigated for a wide range of applications, allowing high performances and low power systems [1]. It is used with success to develop Resistive Random Access Memories (RRAMs) [2], [3], obtaining promising non-volatile characteristics. The integration of these memories into FPGAs enables the reduction of both the delay and the energy consumption [4]. Also the field of the Internet-of-Things (IoTs) is investigating how to take advantage of RRAMs' peculiarities [5]. In addition to these applications, cross-bar RRAM arrays are largely considered for neuromorphic computing [6].

The possibility of performing in-memory computation by means of the intrinsic logic operation of resistive switches [7] led to the recent proposal of a *Programmable Logic-in-Memory* (PLiM) computer [8]. The PLiM architecture has the ability to compute functions on a standard resistive memory by means of an additional low-overhead controller in the PLiM computer. Instructions are read from the memory banks, decoded, and then computed on the dedicated array section. The PLiM architecture is fully programmable and is capable of computing any logic functions, assuming that a sufficient number of cells is available.

The logic computation has to be performed on arrays of RRAM-based switches. In particular, both *Bipolar Resistive Switches* (BRS) [1] and *Complementary Resistive Switches* (CRS) [9] can be used to implement logic operations. Indeed, a single BRS or CRS cell can implement 14 Boolean functions over the total of 16 [10]. The present work exploits the majority intrinsic logic operation of the resistive switches.

A compiler has been proposed in [11] which derives the PLiM program of a logic function. A PLiM program is a set of instructions that can be run on the memory array. The compiler makes use of the *Majority Inverter Graphs* (MIGs) [12], [13], a logic representation in which all operations are majority-of-three $\langle xyz \rangle = xy \vee xz \vee yz$ or inversions.

The original PLiM computer architecture, as discussed in [8], considers one instruction per time; a choice made for the sake of simplicity. The controller acts on the inputs of a single array cell every time. This mode causes the function to be processed slowly on the array. The compiler in [11] works conforming to this operating mode. It is aiming at optimizing the number of instructions, using quite involved algorithms and data structures. Its performances are achieved at the expense of the runtime of the compiler.

The present work extends the capabilities of the original PLIM compiler proposing an automatic compiler based on the MIG representation which exploits parallelism. The computer architecture could be developed to enable concurrent accesses to several cells in the array. In order to do so, the compiler must create a program which takes into account the parallel control of cells. Parallelizing the computation ensures a better exploitation of the array computing capabilities and a faster computation. It is important to underline that the algorithm has been designed considering that all RRAM cells are accessible at the same time. This assumption gives the opportunity to investigate the benefits of parallel PLiM to trigger more architectural exploration. The algorithm can be easily adapted, when applied to more restricted applications.

The proposed algorithm takes a standard MIG as input and, proceeding level by level, returns the parallelized PLiM program of the function. It aims to get a small number of occupied RRAMs, an highly parallelized code and a very small runtime. It has been implemented in C++, tested on EPFL and ISCAS benchmarks and compared to the previous compiler [11]. As results, it shows a maximum speed-up of 2000$\times$ with a contained increase of the number of RRAMs.

The Section II introduces the MIG data structure, the intrinsic majority operation of resistive switches and the PLiM architecture. Section III explains what is a PLiM program and how it looks when parallelization is introduced. In Section IV the proposed PLiM compiler algorithm is thoroughly explained. Finally Section V shows and discusses the obtained results.

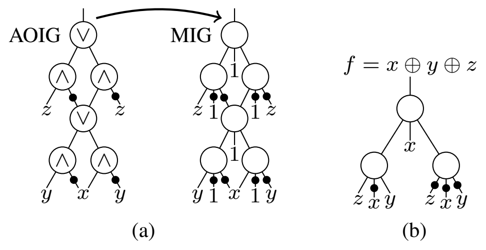


Fig. 1. Example of MIG (a) Equivalence between an AOIG and MIG rapresentations (b) Equivalent optimized MIG

II. Preliminaries

A. Majority Inverter Graph

An MIG is a logic network with 3-input nodes performing a majority operation and with complemented edges performing the inversion. An example of a MIG, directly translated from an *And Or Inverter Graph* (AOIG) is shown in Fig. 1(a). In Fig. 1(b), the same function is optimized by means of the complete algebra associated to this data structure. More details can be found in [12], [13].

The input function of the compiler is given as an already optimized MIG. The algorithm does not introduce any modification to the graph. The applied pre-optimization method makes use of the inverter propagation axiom [14]:

$$\Omega.I\colon \overline{\langle xyz\rangle} = \langle \bar{x}\bar{y}\bar{z}\rangle \tag{1}$$

As a result, all nodes require at most one complemented edge without increasing the size and the depth of the MIG.

B. Intrinsic Majority Operation

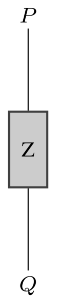
A resistive switch is a two terminal device (schematic shown in Fig. 2(a)) which is able to switch between a *High Resistance State* (HRS) and a *Low Resistance State* (LRS), according to the voltage applied to its terminals. The switch itself is able to implement a three input logic function. The first two inputs are the voltages applied to its two terminals P and Q and the third one is the resistive state of the cell Z. When the cell is in the LRS then $Z = 0$, otherwise $Z = 1$.

The intrinsic logic function is described in Fig. 2(b). The top table shows the logic function implemented by the cell when in LRS, and the bottom one when it is in HRS. By combining the tables, it is possible to obtain the global function that is used as basic instruction for the PLiM program:

$$\begin{aligned} Z_n &= (P\overline{Q})\overline{Z} \vee (P \vee \overline{Q})Z \\ &= PZ \vee \overline{Q}Z \vee P\overline{Q}\,\overline{Z} \\ &= PZ \vee \overline{Q}Z \vee P\overline{Q} \\ &= \langle P, \overline{Q}, Z\rangle = RM_3(P,Q,Z) \end{aligned} \tag{2}$$

This function is referred to as 3-input *Resistive Majority* (RM_3) [8]. The operation that a resistive switch intrinsically performs is based on the majority operation. The peculiarity is in the complemented second input.

The result of the function Z_n is stored as the next resistive state of the memory cell that performed the computation. This means that the same cell is used as destination for the computed result overwriting its previously stored value.



P	Q	Z	Z_n
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	0

$$Z_n = P\overline{Q}$$

P	Q	Z	Z_n
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	1

$$Z_n = P \vee \overline{Q}$$

(a) (b)

Fig. 2. Resistive switch logic operation.

C. PLiM Computer Architecture

Exploiting the intrinsic majority operation of resistive switches, an architecture capable of performing computation directly on a memory array has been proposed in [8]. The PLiM computer architecture can enable both memory and computing operations to be performed on a standard array. It is composed by a traditional multi-bank memory with the addition of a PLiM controller. The controller is a simple finite state machine that reads instructions from the memory and, when required, performs the computing operation within the memory. More detailed information about the PLiM architecture can be found in [8].

III. PLiM Program

A PLiM program is a set of instructions to be run on the memory array in order to compute a Boolean function. This section is explaining the elements that build the program and how they are grouped together.

A. PLiM Instructions

The above-described intrinsic logic operation RM_3 is used to build the PLiM program together with a set of RM_3-derived instructions introduced for the sake of convenience. It is possible to derive them in a very intuitive way, by simply setting one or more inputs to a constant value. Setting the input Z to a constant value means initializing the cell in one of its encoded states.

The RM_3-deduced operations are here described:

$$\begin{aligned} &ZERO(X_1): && RM_3(0,1,X_1) \mid X_1' \leftarrow 0 \\ &ONE(X_1): && RM_3(1,0,X_1) \mid X_1' \leftarrow 1 \\ &NOT(v,X_1): && RM_3(0,1,X_1) \mid X_1' \leftarrow 0 \\ & && RM_3(1,v,X_1) \mid X_1' \leftarrow \overline{v} \\ &BUF(v,X_1): && RM_3(0,1,X_1) \mid X_1' \leftarrow 0 \\ & && RM_3(v,0,X_1) \mid X_1' \leftarrow v \end{aligned}$$

The inputs applied to the terminals of the cell are in lowercase letters (z) and the resistive state of the cell is in uppercase letters

(X_1). It is shown how the next resistive state X_1' corresponds to the result of the RM_3 operation.

The first two instructions are used as initialization: the $\mathrm{ZERO}(X_1)$ instruction sets the resistive state $@X_1$ to 0 while the $\mathrm{ONE}(X_1)$ sets it to 1. The next two operations copy the value of an input (BUF) or its complemented value (NOT) into the cell resistive state. In both cases, a ZERO step on the destination cell is required. Note that these last two operations consist of two RM_3 instructions to be performed on the same cell: initialization and loading of the desired value. This means that the two instructions cannot be performed at the same time.

B. Layers

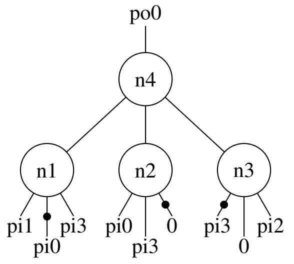
The compiler described in this work will return a code which is built taking into account that some operations can be performed on different cells at the same time, without compromising the final result. All the operations that can be executed in parallel are grouped together in containers that are referred to as "layers". Layers are a way to represent data dependencies between operations. Instructions to compute values on one layer can only access operands that were computed on preceding layers. At the moment, only the data dependencies given by the MIG logic representation are considered, thus being independent from the actual architecture. However, architecture constraints may be more restrictive, which can be captured in additional data dependencies.

The example in Fig. 3 is used to explain the concept of layer. It is shown how the compiler translates a complete graph into a code. The input graph is an MIG on which $\Omega.I$ has been applied. It represents a 4-inputs, single output function. The inputs are named pi0, pi1, pi2 and pi3, while the output is po0. The graph has 4 nodes: n1, performing a MAJ operation; n2, performing an OR; n3 performing an AND and the last n4, performing a MAJ. The corresponding code shows the set of instructions, grouped in layers. For each of them, the generating node is indicated.

Considering the node n1, it performs the operation $\langle pi1, \overline{pi0}, pi3\rangle = \mathrm{RM}_3(pi1, pi0, pi3)$ and the result is saved in the resistive cell named $X_{\text{dest\_1}}$. This location is first initialized with the $pi3$ value, then the RM_3 operation is computed. All the instructions associated with this node need to be placed in different layers, because they are all performed on the same resistive cell $X_{\text{dest\_1}}$. Nevertheless, the analogue operations of node n2 and n3 can be parallelized. As it can be seen, all the BUF operations are performed on the cells $X_{\text{dest\_1}}$, $X_{\text{dest\_2}}$ and $X_{\text{dest\_3}}$ at the same time, being grouped together in the first two layers. At this point is important to notice that BUF and NOT require two layers each, because of the initialization step they both include. The output value of the function, po0 is stored into X_{dest_3}. The shown program needs a total of 4 RRAMs for the computation and has a total of 12 RM_3 instructions, grouped in 6 layers.

IV. PLiM Compiler

The proposed compiler translates a standard MIG graph into a PLiM program. It proceeds level by level, from the inputs to the outputs, associating graph's nodes to instructions. In contrast to [11], it allows the parallel execution of instruction on different memory cells. Instructions that can be performed in parallel are grouped in layers.



Layers 1 - 2	n1	BUF(pi1, $X_{\text{dest\_1}}$)
	n3	BUF(pi2, $X_{\text{dest\_2}}$)
	n2	BUF(pi3, $X_{\text{dest\_3}}$)
Layer 3	n1	RM3(pi3, pi0, $X_{\text{dest\_1}}$)
	n3	RM3(0, pi3, $X_{\text{dest\_2}}$)
	n2	RM3(pi0, 0, $X_{\text{dest\_3}}$)
Layers 4 - 5	n4	NOT($X_{\text{dest\_2}}$, $X_{\text{neg\_1}}$)
Layer 6	n4	RM3($X_{\text{dest\_1}}$, $X_{\text{neg\_1}}$, $X_{\text{dest\_3}}$)

Fig. 3. An MIG graph and its PLiM program

Section IV-A explains how a single node is translated and how the algorithm proceeds through the graph. The way the algorithm reshapes itself when a constraint on the number of cells is introduced is described in Section IV-B. Section IV-C clarifies the processing of the outputs when the algorithm is constrained. Section IV-D shows an optimization to reduce the number of layers is shown.

A. Algorithm Structure

1) Single MIG-node Translation: The core of the algorithm is the processing of a single MIG node. The function $\mathrm{process\_n}$ takes a node as input, checks its children-nodes' characteristics and extracts the corresponding instructions. Depending on its configuration, each node requires one to three layers. The pseudo code of the $\mathrm{process\_n}$ function is shown in Alg. 1. The main purpose is to assign to each child of the node the corresponding operands for the RM_3 operation: a, b, and X_{dest}. Those inputs must satisfy specific characteristics:

1) the edge corresponding to input b must be complemented.
2) the cell used as destination must be overwritable.

To justify the first requirement, consider that a MIG node performs the majority operation. This is slightly different from RM_3, because of the complemented second input. The reason for the second requirement is that the result of the operation is encoded in the resistive state of cell X_{dest}. Consequently the previous state is overwritten by each operation and the data is lost. If a child of the processed node has a fanout > 1, it cannot be overwritten without compromising the next operation's result. Referring to Alg. 1, the function $\mathrm{free\_ch}$ stores, in a container, all the MIG node's children that can be overwritten. The cell corresponding to a child node can be used as destination of the computation X_{dest} only if:

```
    **Data:** MIG-node n
    **Result:** code-node
 1  consider one node $n$ of the MIG graph;
 2  set *free* $\leftarrow$ free_ch$(n)$;
 3  set *order* $\leftarrow$ get_op_order$(n)$;
 4  set $a \leftarrow order[0]$;
 5  given free_stack;
 6  **if** *order[1] is complemented* **then**
 7      set $b \leftarrow order[1]$;
 8  **else**
 9      **if** *free_stack is empty* **then**
10          set $X_{not} \leftarrow X_{new}$;
11      **else**
12          set $X_{not} \leftarrow X_{stack}$;
13      **end**
14      $\mathrm{NOT}(order[1], X_{not})$;
15      set $b \leftarrow X_{not}$;
16  **end**
17  **if** *order[2]* $\in$ *free* **then**
18      set $X_{dest} \leftarrow order[2]$;
19  **else**
20      **if** *free_stack is empty* **then**
21          set $X_{dest} \leftarrow X_{new}$;
22      **else**
23          set $X_{dest} \leftarrow X_{stack}$;
24      **end**
25      $\mathrm{BUF}(order[2], X_{dest})$
26  **end**
27  $\mathrm{RM}_3(a, b, X_{dest})$
```

Algorithm 1: Function process_n to process a single MIG node

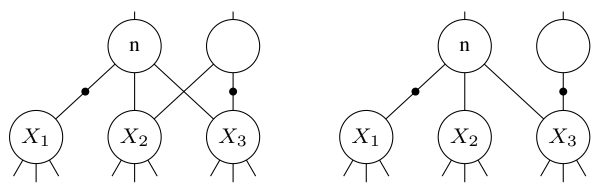
1) is not an input of the function;
2) has a fanout equal to one.

The first case obviously is because the inputs are signals coming from the outside. The second case ensures that, as explained above, the data cannot be destroyed if it has to be used by other nodes.

The next function applied is get_op_order(n) that checks again the children and put them in order. If there is a complemented child it is placed in the second position. The third position is then occupied by a child that can be rewritten, if there is one. If among the three children there is a node that is complemented and also overwritable, priority is given to the first property and it is located in the second position. Indeed there is maximum one complemented child per node, while many children might be overwritable.

The container free_stack is a reservoir of cells to be used when needed. It is used to keep record of all the locations whose value has been already used and can be overwritten without damaging the result of the computation. How this stack is filled is explained in the following section.

In the remaining part of Alg. 1 the container order is verified. The member in position one can be assigned to a, i.e., the first member of the RM_3 operation. In the next step the child in position two is analyzed. If get_op_order found a complemented child, than it is directly assigned to b. Otherwise



01: $\mathrm{ZERO}(X_{dest})$
02: $\mathrm{BUFF}(X_3, X_{dest})$
03: $\mathrm{RM}_3(X_2, X_1, X_{dest})$

(a)

01: $\mathrm{RM}_3(X_3, X_1, X_2)$

(b)

Fig. 4. (a) Example of MIG node *n* translated into three layers (b) Example of MIG node *n* translated into one single layer

a new cell must be occupied and it must be initialized to the value of this child. In order to proceed with the initialization the new location X_{not} is first initialized to 0 and then the negated value of order[1] is copied to it by the NOT operation. Finally X_{not} is assigned to b. Successively the child given in third position is analyzed: this is the one supposed to be used as destination. If it is contained in free than X_{dest} is directly assigned. Otherwise a new location needs to be occupied and the value of the third child is copied by the BUF after the initialization to 0. Every time there is the need of a new cell location, the free_stack container is checked. If there is a location saved into this container this is used instead than a new one in order to reduce the number of RRAM cells. After that a, b, and X_{dest} are assigned, the RM_3 operation can be performed.

Two examples are given in the Fig. 4. The node to be processed is shown together with its children. Each children's value is stored into a cell location. So the names X_1, X_2, X_3 indicated both graph node and the corresponding resistive cell in which the value is stored. For the case of Fig. 4(a) the row 2 and 3 of Alg. 1 returns the following vectors: free = X_1 and order = X_2, X_1, X_3. While in the case of Fig. 4(b) the vectors are: free = X_1, X_2 and order = X_3, X_1, X_2. It is also shown how the case (a) is translated into three layers, since the second verification of the order vector is not satisfied (raw 12 of Alg. 1). On the contrary the second example verifies both the verification and fits into a single layer.

2) Level by Level MIG Translation: This section describes how the algorithm traverses the graph. As in Alg. 2, one level per time is considered. All the nodes in the level are processed by the process_n function. Then the function decr_fanout is applied to all the children of the nodes in the level. To understand this last function it is important to know that an MIG node is characterized by the number of fanouts it has. The value of a node with fanout = 2 is used twice. The value of a node with fanout = 0 is no more used for any computation. In the latter case the corresponding cell can be placed in the free_stack. For this reason, at the end of the processing of a level, all the nodes whose values have been used are decremented. Once reached this point of the Alg. 2, all the children with null fanout are placed in the free_stack. They might be used by the function process_n during the processing of the next level. The function process_out is responsible for

```
   Data: MIG-graph
   Result: code
1  foreach level $l$ do
2      foreach node $n$ do
3          process_n($n$);
4      end
5      foreach node $n$ do
6          foreach children $ch$ do
7              decr_fanout($ch$)
8          end
9      end
10     foreach node $n$ do
11         foreach children $ch$ | fanout $= 0$ and $ch \notin$ inputs
            do
12             put $ch \rightarrow free\_stack$;
13         end
14     end
15 end
16 process_out
```

Algorithm 2: Algorithm to process the all the MIG levels

```
1  while free_stack is not empty and freezer is not empty do
2      take a node $f \in freezer$;
3      process_n($f$);
4  end
```

Algorithm 3: Algorithm to process frozen node by mean of freed cells

computing all the outputs pointing a node with complemented edge (X_{out}), adding a simple $\text{NOT}(X_{\text{out}})$ function.

B. Resource Constraints

Some modifications allow the algorithm to work with a resource constraint on the number of available RRAM cells. This working mode of the algorithm can be both used to fit the computation into an array of a given size and to look for the minimal number of cells that can implement a function. It can be also used to minimize the number of cells dedicated to a computation in function of the computation itself, leaving more space to store data in a PLiM computer.

The mode is implemented by changing the order in which the nodes are processed. Both the functions process_n and process_out need to be modified.
In order to implement this operating mode, a cell counter is added to Alg. 2, together with the following features.

1) While processing a node, the RRAM-counter is checked every time a new cell is needed. If there is no cell left to compute the node, this is "frozen" and the process_n function is interrupted.
2) Every node that points to a frozen node needs to be frozen as well.
3) The only way to compute a frozen node is to use a cell from the free_stack.
4) After that a layer has been processed, all the cells that have been freed are used to defrost nodes of the previous levels.

```
   Data: outputs
   Result: code
1  set out_to_process ← outputs;
2  do
3      foreach output ∈ out_to_process do
4          if points to a frozen node then
5              put $freezer \leftarrow output$;
6              continue;
7          end
8          if must be complemented then
9              if free_stack is not empty then
10                 set $X_{not} \leftarrow X_{stack}$;
11             else
12                 if cell-counter < constraint then
13                     set $X_{not} \leftarrow X_{new}$;
14                 else
15                     put $freezer \leftarrow output$;
16                     continue;
17                 end
18             end
19             $\text{NOT}(out\_node, X_{not})$;
20         end
21     end
22     foreach complemented output computed do
23         decrement the fanout of each pointed node
24     end
25     foreach fanout(node)=0 do
26         put free_stack ← $node$
27     end
28     set out_to_process ← $freezer$
29 while free_stack is not empty and out_to_process is not
    empty;
```

Algorithm 4: Function process_out with resource constraint

5) When all the node in a level are frozen the algorithm stops: the number of cells is not sufficient to compile the given Boolean function.

Points 1 and 2 are implemented by modifying the order requirements checking (lines 8, 20 of Alg. 1). The placement of freed cells in the free_stack (point 3) is done at line 12 of Alg. 1. Point 4 is implemented by the Alg. 3 that is added at line 14 of Alg. 2. The check of point 5 is performed after every node in a level has been processed.

C. Outputs Processing

An MIG graph may have multiple outputs that point to their corresponding node *out_node*. Some of this pointing edges might be complemented. Of particular interest is the way the outputs are processed by the function process_out when the resource constraint is required.

When there is a complemented output the algorithm should add a $\text{NOT}(\text{out\_node}, X_{\text{not}})$ operation. Consequently the destination requires a new cell to store the result. For this reason the function is strongly affected by the resource constraint.

The function, described in Alg. 4, computes recursively every output into the container out_to_process which is initialized to contain all the MIG's outputs.

If the output is pointing to a frozen node, it is frozen, because its value has not yet being computed. Then if the output complements a node, the availability of an extra cell is checked. If there is a cell X_{stack} in the free_stack, than this is used to store the complemented value. Otherwise a new one X_{new} is taken from the array. If there is no space for another cell than the output is frozen.

After this first loop through every output some nodes may have been freed. This is the case for all the nodes pointed by only one computed complemented output, which are placed in the free_stack. Before verifying the exit condition, all the outputs in the freezer are placed in the out_to_process container for the next iteration. The loop is stopped when the free_stack is empty or when all the outputs have been computed.

D. Layer Minimization

Because of the parallelization introduced by this compiler, the runtime of the function on the resistive array is strongly dependent on the number of layers of the code. For this reason a function for the layer minimization has been implemented. The function modifies the code after it has been created by the compiler.

As already explained (Fig. 4), a single node may generate from one to three layers, depending on whether it requires pre-computation. Consequently, every level might correspond to one or three layers. The minimization function takes all the nodes that required pre-computing and moves the corresponding instructions in order to reduce the number of layers of that level. In fact, the ZERO initialization instruction can be performed as soon as the destination cell of the operation is ready. For cells taken from the free_stack, this is the level where they have been released, for new cells this is the very first layer.

For every precomputed node p_node the initialization function is moved from the current level location to the location at which it was ready. At the end of the code all the empty layers are eliminated by a specific function.

V. Results

The compiler has been implemented in C++ and evaluated on the EPFL and the ISCAS benchmarks and the results are shown in Table I. The main characteristics of the input MIG graph are presented, together with the key parameters of the PLiM compiler's code. Those are: the number of RRAMs used to compute the function, the total number of instructions, the number of layers and the runtime in seconds. Some parameters are then compared with the ones resulting from the compilation of the state-of-the-art compiler [11].

The two algorithms are different in the choice of the parameter to be optimized and the results underline this difference. The previous algorithm chooses very carefully the order in which the nodes are computed, every time checking the state of all the not yet computed nodes, in order to select the best one. The choice is made in order to minimize the number of cells used and the number of instructions. The main drawback of this approach is the slow runtime of the program. Sometimes, as for *hyp*, the computation requires a very long runtime (more than 4h).

The proposed compiler, proceeding level by level, does not select every time the most convenient node. This leads to a longer number of operations and of RRAMs used. On the other hand it is significantly faster than the previous one. It is possible to see that the compilation speed-up reaches $1000\times$ and $2000\times$ when dealing with large MIGs in terms of their sizes. The number of operations increases by a factor that goes from $1.08\times$ to $1.48\times$ with respect to the previous compiler. An upper bond for the number of layers is three times the number of levels in the graph. The number of layers ranges mainly in between twice and three times the number of levels. The level of parallelism can be approximated comparing the number of levels with the number of instructions.

To overcome this expected result the possibility of performing some instructions in parallel has been introduced. Even with a larger number of instructions, the number of accesses to the array is reduced. With the assumption that all the cells can be simultaneously enabled and that each instruction needs a fixed amount of time, the number of layers of the new code can be compared with the number of operations of the previous one. Resulting in a relevant speed-up of the computation.

Table II is showing the result of running the compiler with a constraint on the number of available cells. The *bar* circuit from the EPFL benchmark is reported as example. Without any constraints, it is compiled to occupy 709 RRAMs. A strict constraint is gradually applied. At the end, for 640 available RRAMs, the capabilities of the compiler reach their limits. Looking at the results it is possible to notice how the number of layers increases while the maximum number of cells decreases. This is due to the fact that every time a node is computed from the freezer, three layers are added to the code. On the other hand, the runtime is kept very low and almost constant.

VI. Conclusion

In this paper a compiler for the PLiM computer has been presented, allowing to translate an MIG graph into a set of RM_3 instructions. Compared with the original compiler [11], it is significantly faster, so that heavy circuits in terms of size, can be compiled, while the previous implementation fails. The output code of the compiler contains instructions that can be run in parallel. This allows a better exploitation of the hardware resources and a faster computation. A key factor to speed up the computation, by means of a lower number of memory accesses, is to reduce the number of levels in the input graph. It would be of interest for successive works to add a low-depth optimization before the compilation. Rewriting the MIG with low-depth optimization would allow a consistent reduction of the number of layers and, therefore decrease the running time of the PLiM program. An MIG rewriting could also take into account the possibility of a parallel computation in addition to the reduction of the number of levels.

Acknowledgments: This research was supported by H2020-ERC-2014-ADG 669354 CyberCare and the Swiss National Science Foundation (200021-169084 MAJesty and and 200021-146600).

TABLE I
EPFL AND ISCAS BENCHMARK RESULTS

Benchmark					Previous PLiM Compiler [11]			PLiM Compiler				Comparisons	
Circuit	#Levels	#I	#O	Size	#RRAMs	#Op	^{1}Time	#RRAMs	#Op	#Layers	Time	Speed-up	#Op
adder	129	256	129	386	258	899	0.27	259	1158	385	0.01	27.00	1.29
bar	14	135	128	3110	341	5730	12.49	709	7304	39	0.03	416.33	1.27
div	4401	128	128	57272	665	99208	5304.13	308	129868	13104	5.02	1056.60	1.31
hyp	9320	256	128	153311	^{3}-	-	-	34643	405001	26763	33.47	-	-
log2	230	32	32	25040	1380	49991	989.32	950	57772	688	0.49	2019.02	1.16
max	290	512	130	2491	560	4006	9.36	571	5289	531	0.03	312.00	1.32
sin	167	24	25	4496	408	8698	26.66	153	10400	478	0.04	666.50	1.20
sqrt	5989	128	64	21066	261	40100	694.56	394	47008	14300	5.96	116.54	1.17
square	156	64	128	13671	487	29177	275.39	4326	36175	420	0.18	1529.94	1.24
arbiter	63	256	129	8957	885	10815	117.39	798	11699	79	0.12	978.25	1.08
cavlc	19	10	11	757	71	1266	0.72	245	1545	55	0.01	72.00	1.22
ctrl	10	7	26	139	36	258	0.02	64	323	27	0.00	^{2}NR	1.25
dec	4	8	256	328	260	839	0.08	290	1010	14	0.00	NR	1.20
i2c	23	147	142	1329	228	2309	2.37	396	2891	60	0.01	237.00	1.25
int2float	18	11	7	263	34	478	0.09	99	569	42	0.00	NR	1.19
mem_ctrl	144	1204	1231	45034	2155	79172	3294.10	5319	100436	432	1.23	2678.13	1.27
priority	245	128	8	993	133	1463	1.42	132	2163	545	0.01	142.00	1.48
router	54	60	30	220	86	411	0.08	100	576	149	0.00	NR	1.40
voter	67	1001	1	7767	1147	16036	103.39	1773	21067	191	0.08	1292.38	1.31
c17	3	5	2	7	7	12	0.00	5	17	6	0.00	NR	1.42
c432	32	36	7	176	66	328	0.04	89	492	61	0.00	NR	1.50
c499	18	41	32	316	53	598	0.14	107	790	37	0.00	NR	1.32
c880	26	60	26	301	72	557	0.28	111	821	76	0.00	NR	1.47
c1355	18	41	32	316	56	598	0.13	107	854	39	0.00	NR	1.43
c1908	28	33	25	291	75	583	0.12	63	777	82	0.01	12.00	1.33
c2670	23	157	64	503	157	833	0.40	173	1269	57	0.00	NR	1.52
c3540	42	50	22	898	131	1576	1.07	186	2112	124	0.01	107.00	1.34
c5315	34	178	123	1283	247	2383	2.30	308	3243	96	0.01	230.00	1.36
c6288	62	32	32	1019	113	2430	1.32	342	2653	176	0.01	132.00	1.09
c7552	39	207	108	1293	295	2470	2.42	401	3357	109	0.01	242.00	1.36

^{1} Runtime in seconds. ^{2} Not Relevant improvement in low-size graphs. ^{3} Graph too large to compile.

REFERENCES

[1] H. S. P. Wong, H. Y. Lee, S. Yu, Y. S. Chen, Y. Wu, P. S. Chen, B. Lee, F. T. Chen, and M. J. Tsai, "Metal-oxide RRAM," *Proceedings of the IEEE*, vol. 100, no. 6, 2012.

[2] Y. Ho, G. M. Huang, S. Member, P. Li, and S. Member, "Dynamical Properties and Design Analysis for Nonvolatile memristor memories," *IEEE T-CS*, vol. 58, no. 4, 2011.

[3] K. C. Liu, W. H. Tzeng, K. M. Chang, Y. C. Chan, C. C. Kuo, and C. W. Cheng, "The resistive switching characteristics of a Ti/Gd2O3/Pt RRAM device," *Microelectronics Reliability*, vol. 50, no. 5, 2010.

[4] X. Tang, P.-E. Gaillardon, and G. De Micheli, "A high-performance low-power near-vt rram-based fpga," in *2014 International Conference on Field-Programmable Technology (FPT)*, Dec 2014.

[5] F. Clermidy, N. Jovanovic, S. Onkaraiah, H. Oucheikh, O. Thomas, O. Turkyilmaz, E. Vianello, J.-M. Portal, and M. Bocquet, "Resistive memories: Which applications?" in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14, 2014.

[6] D. B. Strukov, "Nanotechnology: Smart connections," *Nature*, vol. 476, no. 7361, 2011.

[7] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'Memristive' switches enable 'stateful' logic operations via material implication." *Nature*, vol. 464, no. 7290, 2010.

[8] P.-E. Gaillardon, L. Amaru, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The Programmable Logic-in-Memory (PLiM) computer," *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[9] E. Linn, R. Rosezin, C. Kügeler, and R. Waser, "Complementary resistive switches for passive nanocrossbar memories." *Nature Materials*, vol. 9, no. 5, 2010.

[10] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger, and R. Waser, "Beyond von Neumann—logic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, vol. 23, no. 30, 2012.

[11] M. Soeken, S. Shirinzadeh, P.-E. Gaillardon, L. G. Amarù, R. Drechsler, and G. De Micheli, "An MIG-based compiler for programmable logic-in-memory architectures," in *Design Automation Conference (DAC)*, 2016.

[12] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Design Automation Conference (DAC)*, 2014.

[13] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Boolean logic optimization in majority-inverter graphs," in *Design Automation Conference (DAC)*, June 2015.

[14] E. Testa, M. Soeken, O. Zografos, L. Amaru, P. Raghavan, R. Lauwereins, P.-E. Gaillardon, and G. De Micheli, "Inversion optimization in majority-inverter graphs," in *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, July 2016.

TABLE II
CONSTANT GRAPH'S SIZE - SWEEP ON THE NUMBER OF RRAMS

Circuit	max #RRAMs	#RRAMs	#Op	#Layers	Time
bar	709	709	7304	40	0.03
bar	700	700	7302	65	0.02
bar	690	690	7302	95	0.02
bar	680	680	7302	125	0.02
bar	670	670	7304	158	0.02
bar	660	660	7304	188	0.03
bar	650	650	7304	218	0.02
bar	640	640	7108	468	0.03