

An MIG-based Compiler for Programmable Logic-in-Memory Architectures

Mathias Soeken¹ Saeideh Shirinzadeh² Pierre-Emmanuel Gaillardon³
Luca Gaetano Amarú¹ Rolf Drechsler^{2,4} Giovanni De Micheli¹

¹ Integrated Systems Laboratory, EPFL, Lausanne, Switzerland

² Group of Computer Architecture, University of Bremen, Bremen, Germany

³ Electrical and Computer Engineering Department, University of Utah, Salt Lake City, UT, USA

⁴ Cyber-Physical Systems, DFKI GmbH, Bremen, Germany

ABSTRACT

Resistive memories have gained high research attention for enabling design of in-memory computing circuits and systems. We propose for the first time an automatic compilation methodology suited to a recently proposed computer architecture solely based on resistive memory arrays. Our approach uses *Majority-Inverter Graphs* (MIGs) to manage the computational operations. In order to obtain a performance and resource efficient program, we employ optimization techniques both to the underlying MIG as well as to the compilation procedure itself. In addition, our proposed approach optimizes the program with respect to memory endurance constraints which is of particular importance for in-memory computing architectures.

1. INTRODUCTION

Resistive Random Access Memories (RRAMs) have gained high attention for their promising applications in non-volatile and ultra compact memories [8, 11, 14]. The resistive switching property of RRAMs have also made them of interest in synthesis of primary logic gates enabling in-memory computing [5, 12]. Logic circuits using the resistive switching proposed so far are mostly based on logic implementation that imposes long sequences of operations. Very recently a majority oriented logic has been proposed for synthesis of RRAM-based logic circuits that shows much lower latency and area overhead compared to the sequential implication based approach [13]. In [7], a computer architecture using only RRAMs called *Programmable Logic-in-Memory* (PLiM) was proposed. For programs executed on this architecture, the number of instructions and the required number of RRAMs are important cost metrics to measure the quality. The instruction set for the PLiM consists of a single instruction called RM₃, which computes the majority-of-three operation in which one input is inverted. This corresponds directly to the physical implementation of the RRAM proposed in [10]. Consequently, *Majority-Inverter Graphs* (MIG, [3]), which are logic representations that use the majority-of-three operation and inverters as only logic primitives, are the natural abstraction to derive PLiM programs, i.e., se-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '16, June 05-09, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-4236-0/16/06...\$15.00

DOI: <http://dx.doi.org/10.1145/2897937.2897985>

quences of RM₃ instructions. This idea has been originally proposed in [7], but no automatic translation scheme has been proposed.

One function can be represented by several structurally different but functionally equivalent MIGs. Each MIG leads to PLiM programs of different quality. Further, even for the same MIG representation, there exists several ways to translate it into PLiM programs that vary in their quality.

In this paper, we propose an efficient and fully automated compilation procedure of Boolean functions to programs for the PLiM architecture based on MIGs. The key contributions of this paper are:

1. An optimization algorithm based on MIG rewriting [1, 13] that optimizes the MIG w.r.t. the expected number of instructions and required RRAMs in the translated PLiM program.
2. A smart automatic translation algorithm that traverses the MIG in a way to minimize the costs of the resulting PLiM program.

Our experimental evaluation shows that our proposed techniques have a significant impact on the resulting programs compared to a naïve translation approach: (i) the number of instructions can be reduced by up to 19.95% and the number of required RRAMs can be reduced by up to 61.4%.

Our techniques unlock the potential of the PLiM architecture [7] to process large scale computer programs using in-memory computing. This makes this promising emerging technology immediately available for realistic applications.

2. BACKGROUND

2.1 Majority-Inverter Graphs

An MIG is a data structure for Boolean function representation and optimization. An MIG is defined as a logic network that consists of 3-input majority nodes and regular/complemented edges [1, 2, 3].

MIGs can efficiently represent Boolean functions thanks to the expressive power of the *majority operator* (MAJ) $\langle abc \rangle = ab \vee ac \vee bc = (a \vee b)(a \vee c)(b \vee c)$. Indeed, a majority operator can be configured to behave as a traditional conjunction (AND) or disjunction (OR) operator. In the case of 3-input majority operator, fixing one input to 0 realizes an AND while fixing one input to 1 realizes an OR. As a consequence of the AND/OR inclusion by MAJ, traditional *And-Or-Inverter Graphs* (AOIGs) are a special case of MIGs and MIGs can be easily derived from AOIGs. An example MIG representation derived from its optimal AOIG is depicted in Fig. 1(a). AND/OR operators are replaced node-wise by MAJ operators with a constant input.

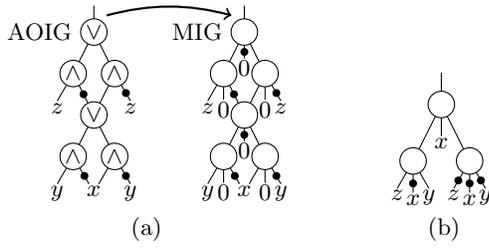


Figure 1: Example MIG representation (a) derived by transposing its optimal AOIG representation, and (b) after optimization

Intuitively, MIGs are at least as compact as AOIGs. However, even smaller MIG representation arise when fully exploiting the majority functionality, i.e., with nonconstant inputs [3]. We are interested in compact MIG representations because they translate in smaller and faster physical implementations. In order to manipulate MIGs and reach advantageous MIG representations, a dedicated Boolean algebra was introduced in [1]. The axiomatic system for the MIG Boolean algebra, referred to as Ω , is defined by the five following primitive axioms.

$$\Omega \left\{ \begin{array}{l} \text{Commutativity} - \Omega.C \\ \langle xyz \rangle = \langle yxz \rangle = \langle zyx \rangle \\ \text{Majority} - \Omega.M \\ \begin{cases} \langle xyz \rangle = x = y & \text{if } x = y \\ \langle xyz \rangle = z & \text{if } x = \bar{y} \end{cases} \\ \text{Associativity} - \Omega.A \\ \langle xu \langle yuz \rangle \rangle = \langle zu \langle yux \rangle \rangle \\ \text{Distributivity} - \Omega.D \\ \langle xy \langle uvz \rangle \rangle = \langle \langle xyu \rangle \langle xyv \rangle z \rangle \\ \text{Inverter Propagation} - \Omega.I \\ \langle x \bar{y} z \rangle = \langle \bar{x} y \bar{z} \rangle \end{array} \right.$$

The axioms are inspired from median algebra [9] and the properties of the median operator in a distributive lattice [4]. A strong property of MIGs and their algebraic framework concerns reachability. It has been proved that by using a sequence of transformations drawn from Ω it is possible to traverse the entire MIG representation space [3]. In other words, given two equivalent MIG representations, it is possible to transform one into the other by just using axioms in Ω . This result is of paramount interest to logic synthesis because it guarantees that the best MIG can always be reached. Unfortunately, deriving a sequence of Ω transformations is an intractable problem. As for traditional logic optimization, heuristic techniques provide here fast solutions with reasonable quality [6].

By using the MIG algebraic framework it is possible to obtain a better MIG for the example in Fig. 1(a). Fig. 1(b) shows the MIG structure, which is optimized in both depth (number of levels) and size (number of nodes). Such MIGs can be reached using a sequence of Ω axioms starting from their unoptimized structures. We refer the reader to paper [3] for an in-depth discussion on MIG optimization recipes.

2.2 PLiM Architecture

Resistive memories have the capability of natively implementing a universal majority-based logic operator [10]. Indeed, by denoting Z the value stored in the memory, after the application of signals on its top and bottom terminals, denoted P and Q , it is possible to express the updated value of Z as $Z \leftarrow PZ \vee QZ \vee PQ = \langle PQZ \rangle$. This basic operation

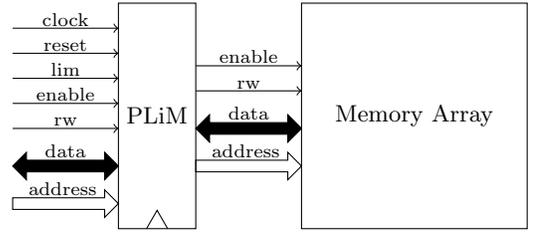


Figure 2: The PLiM architecture: The PLiM controller operates as a wrapper over the RRAM array and schedules the RM_3 operations

natively embedding both a majority-of-three and an inversion can be leveraged as a universal computing operator. We refer to this operation as 3-input *Resistive Majority* (RM_3) with $RM_3(P, Q, Z) = \langle P\bar{Q}Z \rangle$.

The Programmable Logic-in-Memory (PLiM) architecture aims at enabling logic operations on a regular RRAM array. While every memory node can implement basic operations, the difficulty of operating logic on a memory array lies in the distribution of signals and the scheduling of operations. The PLiM controller consists of a wrapper of the RRAM array (Fig. 2) and works as a simple processor core, reading instructions from the memory array and performing computing operations (majority) within the memory array. As a wrapper, the PLiM uses the addressing and read/write peripheral circuitries of the RRAM array. When $LiM = 0$, the controller is *off* and the whole array works as a standard RAM system. When $LiM = 1$, the circuit starts performing computation. The controller consists of a simple finite state machine and few work registers, in order to operate the RM_3 instruction, as detailed in [7]. The instruction format consists of the first operand A , the second operand B , and the destination address Z of the results. Single-bit operands A and B are then read from constants or from the memory array, and logic operation is performed during the write operation to the memory location Z by setting P to A and Q to B . The new value stored in the node Z is then $Z \leftarrow \langle \bar{A}BZ \rangle$. When the write operation is completed, a program counter is incremented, and a new cycle of operation is triggered.

3. MOTIVATION

The main idea of this work is leveraging MIGs in order to derive RM_3 instruction sequences, which can run as programs on the PLiM architecture. In its current form, the PLiM architecture can only handle serial operations [7]. Therefore, only one MIG node might be computed each time and the total number of instructions is equal to the sum of instructions required to compute each MIG node. Accordingly, reducing the size of the MIG is considered to have a significant impact on the PLiM program with respect to the number of steps. However, still further MIG optimization is possible to lower the costs caused by complemented edges. While the presence of a single complemented edge in an MIG node is of interest for benefiting from the intrinsic majority operation inside an RRAM, a second or third complemented edge imposes extra costs in both number of instructions and required RRAMs. Hence, MIG area rewriting, besides reducing number of nodes with multiple complemented edges, can be highly effective for optimizing the number of instructions, while the latter can also lower the number of required RRAMs.

As an example, consider the two equivalent MIGs in Fig. 3(a), before optimization on the left and after optimization on the right. Translating them into RM_3 instructions

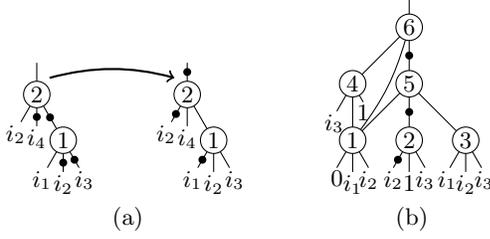


Figure 3: Reducing the number of instructions and RRAMs, after (a) MIG rewriting and (b) node selection and translation

yields:

Before MIG optimization	After MIG optimization
01: 0, 1, @X ₁ X ₁ ← 0	01: 0, 1, @X ₁ X ₁ ← 0
02: 1, i ₃ , @X ₁ X ₁ ← \bar{i}_3	02: i ₃ , 0, @X ₁ X ₁ ← i ₃
03: i ₁ , i ₂ , @X ₁ X ₁ ← N ₁	03: i ₂ , i ₁ , @X ₁ X ₁ ← N ₁
04: 0, 1, @X ₂ X ₂ ← 0	04: i ₄ , i ₂ , @X ₁ X ₁ ← N ₂
05: 1, @X ₁ , @X ₂ X ₂ ← \bar{N}_1	
06: i ₂ , i ₄ , @X ₂ X ₂ ← N ₂	

Here @X_i refers to the address of RRAM X_i and N_j refers to the result of the MIG node j. Program addresses are bold in front of the RM₃ instruction A, B, C, and the second and fourth column comment the action of the instruction.

It can be seen that after optimization both the number of instructions and RRAMs are decreased, from 6 to 4 and from 2 to 1, respectively. The effect of multiple complement edge elimination is much larger when translating a large MIG.

Not only the MIG structure has an effect on the PLiM program, but also the order in which nodes are translated and which of the node's children are selected as operands A, B, and destination Z in the RM₃ instruction. As example, consider the MIG in Fig. 3(b). Translating it in a naïve way, i.e., in order of their node indexes and selecting the RM₃ operands and destination in order of their children (from left to right), will result in the following program:

01: 0, 1, @X ₁ X ₁ ← 0	11: 0, 1, @X ₅ X ₅ ← 0
02: 1, i ₁ , @X ₁ X ₁ ← \bar{i}_1	12: i ₃ , 0, @X ₅ X ₅ ← i ₃
03: 0, 1, @X ₂ X ₂ ← 0	13: i ₁ , @X ₄ , @X ₅ X ₅ ← N ₃
04: i ₂ , 0, @X ₂ X ₂ ← i ₂	14: 0, 1, @X ₆ X ₆ ← 0
05: 0, @X ₁ , @X ₂ X ₂ ← N ₁	15: 1, i ₃ , @X ₆ X ₆ ← \bar{i}_3
06: 0, 1, @X ₃ X ₃ ← 0	16: 1, 0, @X ₇ X ₇ ← 1
07: i ₃ , 0, @X ₃ X ₃ ← i ₃	17: @X ₂ , @X ₆ , @X ₇ X ₇ ← N ₄
08: 1, i ₂ , @X ₃ X ₃ ← N ₂	18: @X ₂ , @X ₃ , @X ₅ X ₅ ← N ₅
09: 0, 1, @X ₄ X ₄ ← 0	19: @X ₇ , @X ₅ , @X ₂ X ₂ ← N ₆
10: 1, i ₂ , @X ₄ X ₄ ← \bar{i}_2	

By changing the order in which the nodes are translated and also the order in which children are selected as operands and destination for the RM₃ instructions, a shorter program can be found (for the same MIG representation):

01: 0, 1, @X ₁ X ₁ ← 0	09: i ₃ , 0, @X ₄ X ₅ ← i ₃
02: i ₂ , 0, @X ₁ X ₁ ← i ₂	10: i ₁ , @X ₃ , @X ₄ X ₄ ← N ₃
03: i ₁ , 1, @X ₁ X ₂ ← N ₁	11: @X ₁ , @X ₂ , @X ₄ X ₄ ← N ₅
04: 1, 0, @X ₂ X ₂ ← 1	12: 0, 1, @X ₂ X ₂ ← 0
05: i ₃ , i ₂ , @X ₂ X ₂ ← N ₂	13: i ₃ , 0, @X ₂ X ₂ ← i ₃
06: 0, 1, @X ₃ X ₃ ← 0	14: @X ₁ , 0, @X ₂ X ₂ ← N ₄
07: 1, i ₂ , @X ₃ X ₃ ← \bar{i}_2	15: @X ₁ , @X ₄ , @X ₂ X ₂ ← N ₆
08: 0, 1, @X ₄ X ₄ ← 0	

Based on this observations, the next section describes algorithms for automatically finding a good MIG representation and for translating a MIG representation in an effective way to get a small PLiM program.

4. THE PLiM COMPILER

This section describes the compilation process and the optimization approaches we employed. The first subsection describes the customized MIG rewriting algorithm that optimizes MIG structures to be more convenient for compiling into RM₃ instructions. The second subsection describes compilation in detail.

4.1 MIG Rewriting

As discussed in Section 3 both the size of an MIG and the distribution of complemented edges have an effect on the PLiM program in number of instructions and number of RRAMs. Hence, we are interested in an MIG rewriting algorithm that (i) reduces the size of the MIG, and (ii) reduces the number of MIG nodes with multiple complemented edges.

```

1 for (cycles = 0; cycles < effort; cycles++) do
2   | Ω.M; Ω.DR→L;
3   | Ω.A; Ω.C;
4   | Ω.M; Ω.DR→L;
5   | Ω.IR→L(1-3);
6   | Ω.IR→L;
7 end

```

Algorithm 1: MIG rewriting for PLiM architecture

The proposed MIG rewriting approach is given in Algorithm 1 and follows the rewriting idea of [1]. It can be iterated for a certain number of times, controlled by *effort*. The first three lines of Algorithm 1 are based on the conventional MIG area rewriting approach proposed in [1]. It is clear that Ω.M reduces the size of MIG by eliminating the unnecessary nodes. Distributivity from right to left (Ω.D_{R→L}) also reduces the number of nodes by one. These node elimination techniques are repeated after reshaping the MIG by applying Ω.A; Ω.C, which may provide further size reduction opportunities.

To reduce the number of nodes with multiple inverted edges, we first apply an extended inverter propagation axiom from right to left denoted by Ω.I_{R→L(1-3)}. Ω.I_{R→L(1-3)} includes the three transformations (1) $\langle \bar{x}\bar{y}\bar{z} \rangle = \langle x\bar{y}\bar{z} \rangle$, (2) $\langle \bar{x}\bar{y}\bar{z} \rangle = \langle x\bar{y}z \rangle$, and (3) $\langle \bar{x}\bar{y}\bar{z} \rangle = \langle x\bar{y}z \rangle$.

Transformation (1) decreases the number of required negations for computing the node from two to one. Since the node still lacks the single complemented edge after transformation, it might be favorable by possibility of creating the ideal single complement case for the node at its fanout target. Nevertheless, transferring a complemented edge can be also unfavorable if the target node already has a single complemented edge. Transformations (1) and (2) make the node ideal for computation. The same arguments about moving a complemented edge to the node at the fanout target or removing it also exist for (2) and (3). At the end, since the MIG might have been changed after the three aforementioned transformations, Ω.I_{R→L} is applied again to ensure the most costly case is eliminated. In general, applying the last two lines of Algorithm 1 over the entire MIG repetitively can lead to much fewer instructions and RRAM cost.

4.2 Compilation

In this section we describe how an optimized MIG is compiled into a PLiM program. Algorithm 2 gives an overview of the algorithm, details are explained in the remainder of this section. The algorithm keeps track of a map COMP[v] that stores for each MIG node v whether it has been computed or not. Initially, all leaves, i.e., primary inputs and the constant, are set to be computed. A priority queue Q keeps track of all vertices that can possibly be translated, called candidates.

```

Input : MIG  $M$ 
Output : PLiM program  $P = \{I_1, I_2, \dots, I_k\}$ 
1 foreach leaf in  $M$  do
2 | set  $\text{COMP}[v] \leftarrow \top$ ;
3 end
4 foreach MIG node in  $M$  do
5 | if all children of  $v$  are computed then
6 | |  $Q.\text{enqueue}(v)$ ;
7 | end
8 end
9 while  $Q$  is not empty do
10 | set  $c \leftarrow Q.\text{pop}()$ ;
11 | set  $P \leftarrow P \cup \text{translate}(c)$ ;
12 | set  $\text{COMP}[c] \leftarrow \top$ ;
13 | foreach parent of  $c$  do
14 | | if all children of  $v$  are computed then
15 | | |  $Q.\text{enqueue}(v)$ ;
16 | | end
17 | end
18 end

```

Algorithm 2: Outline of compilation algorithm

A vertex is a *candidate* if all its children are computed. The sorting criteria for Q is described in Section 4.2.1.

The main loop of the algorithm starts by popping the best candidate c from the priority queue and translating it into a sequence of PLiM instructions. Section 4.2.2 describes the translation process in detail. Afterwards, for each parent, it is checked whether it is computable, and if this is the case, it is inserted into Q .

4.2.1 Candidate Selection

Our candidate selection strategy is based on two principles: (i) releasing the RRAMs in-use as early as possible, and (ii) allocating RRAMs at the right time such that they are blocked as short as possible. We show two example MIGs to clarify the principles. Fig. 4(a) shows an MIG with two candidates u and v , for which all of their children nodes are already computed. Candidate u has two *releasing children*, i.e., children who have single fan-out, while v has only one releasing child. In the case that u is selected for computation first, the RRAMs keeping its releasing children can be freed and reused for the next candidate.

Fig. 4(b) shows a small MIG with two candidates u and v to illustrate the second principle. The output of u is only required when v is already computed. In other words, the number of RRAMs in use can increase if u is computed before v since the RRAM keeping u cannot be released before computing the root node of the MIG. This way, v is computed when a RRAM has been already allocated to retain the value of u . The number of additional RRAMs required in such condition can be considerable for large number of nodes.

In order to sort nodes in the priority queue in Algorithm 2, two nodes u and v are compared. Node u is preferred over v if (i) its number of releasing children is greater, or (ii) if u 's parent with the largest level (ordered from PIs to POs) is on a lower level than v 's parent with the smallest level. If no criteria is fulfilled, u and v are compared according to their node index.

4.2.2 Node Translation

This section explains how a node in the MIG is translated into a RM_3 instruction with operands A and B , and destination Z . The operands A and B can be RRAMs or constants and the destination Z is a RRAM. Recall that the instruction

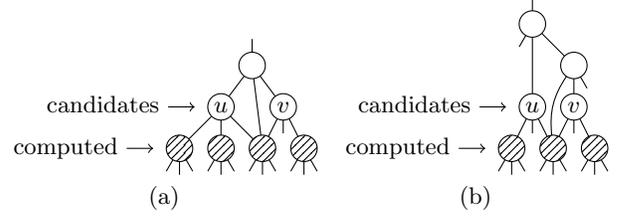


Figure 4: Reducing the number of RRAMs by selecting the candidate with (a) more releasing children and (b) smaller fanout level index

computes $Z \leftarrow \langle A\bar{B}Z \rangle$. In the ideal case each MIG node can be translated into exactly one RM_3 instructions and can reuse one of its children's RRAMs as destination. In other cases additional RM_3 instructions and/or additional RRAMs are required.

Select operand B. We first select which of the node's children should serve as operand B , i.e., the second operand of the RM_3 instruction. In total, four cases with subcases are checked in the given order which are illustrated in Fig. 5. Only the last two subcases require two additional instructions and one additional RRAM.

- (a) *There is exactly one complemented child:* B is the RRAM storing this complemented child.
- (b) *There is more than one complemented child, but also a constant child:* The nonconstant complemented child is selected for B , since constants allow for more flexibility when selecting the remaining operands.
- (c) *There is no complemented child, but there is a constant child:* B is assigned the inverse of the constant. Since we consider MIGs that only have the constant 0 child, B is assigned 1.
- (d) *There is more than one complemented child, but at least one with multiple fan-out:* We select the RRAM of the child with multiple fan-out, as this excludes its use as destination.
- (e) *There is more than one complemented child, none with multiple fan-out:* The RRAM of the first child is selected.
- (f) *There is no complemented child, but for one child there exists a RRAM with its complemented value:* Each node is associated with an RRAM which holds or has held its computed result. In addition, if its inverted value is computed once and stored in an additional RRAM X_i , it is remembered for future use. In this case B can be assigned X_i .
- (g) *There is no complemented child, but one child has multiple fan-out:* The child with multiple fan-out is selected with the same argumentation as above. Since it is not inverted, an additional RRAM X_i needs to be allocated, loaded with 0, and then set to the complement of X_n . As described above, X_i is associated to the child for future use.
- (h) *There is no inverted child and none has multiple fan-out:* The first child is selected and an additional RRAM X_i is allocated to store the complement of X_n .

Select destination Z. After the inverter selection, the destination RRAM, i.e., the third argument of the RM_3 instruction is selected. The aim is to reuse one of the children's RRAMs as work RRAM instead of creating a new one. In total, four cases (with subcases) are checked which are illustrated in Fig. 6. Only the first case allows to reuse an RRAM of the children for the destination, all the other cases require one or two additional instructions and one additional RRAM. Note that one of the children has already been se-

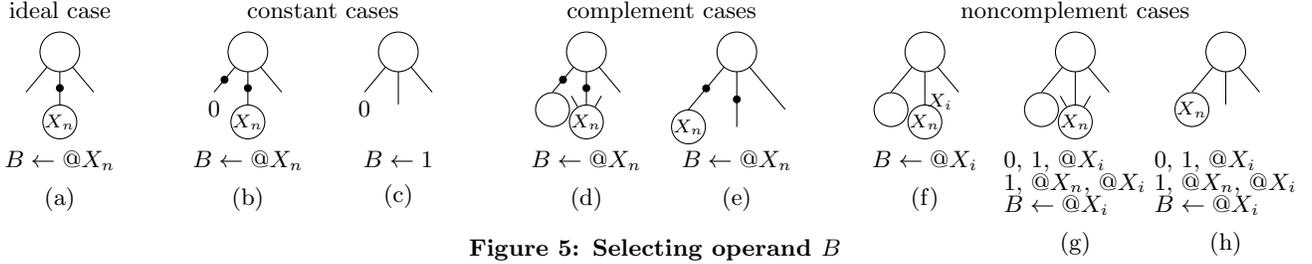


Figure 5: Selecting operand B

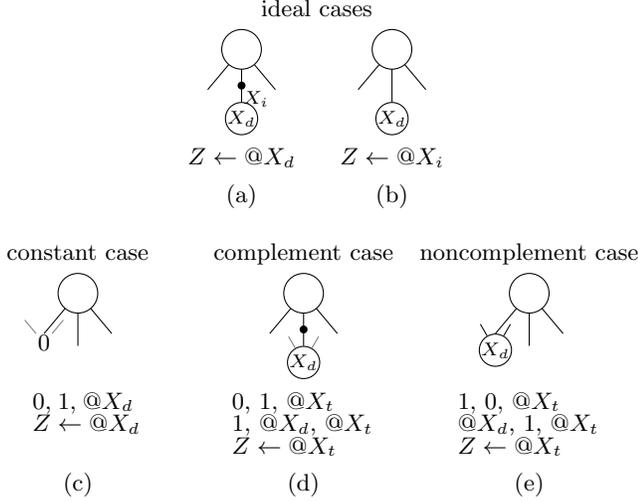


Figure 6: Selecting destination Z

lected as operand B and that this is implicitly considered in the following descriptions.

(a) *There is a complemented child with one fan-out, and there exists an RRAM with its complemented value:* The existing RRAM X_i for the complemented value can be used and it is safe to override it, since the child does not fan out to other parents.

(b) *There is a noncomplemented child with one fan-out:* The RRAM of this child can be used as destination and it is safe to override it. Note that case (a) is preferable compared to this one to avoid complemented children for operand A .

(c) *There is a constant child:* If there is a constant child (with or without multiple fan-out) a new RRAM is allocated and initialized to the constant value (considering complemented edges into account).

(d) *There is a complemented child:* If there is an inverted child X_d (with or without multiple fan-out), a new RRAM X_t is allocated and initialized to the complement of X_d using two RM₃ instructions.

(e) *There is a noncomplemented child with multiple fan-out:* The first child X_d is selected and its value is copied into a new allocated RRAM X_t using two RM₃ instructions.

Select operand A . The child that is selected as operand A is uniquely determined at this point since operand B and destination Z have been selected. Consequently, there is no case distinction w.r.t. to preference. However, there are still different actions to be taken depending on the child node.

(a) *The child node is constant:* A is set to the constant taking the complement edge into account.

(b) *The child node is noncomplemented:* A is set to the

RRAM of the child node.

(c) *The child node is complemented, and there exists an RRAM with its complemented value:* A is set to the computed RRAM of the complemented value.

(d) *The child node is complemented, but there does not exist an RRAM with its complemented value:* A new RRAM X_i is allocated and assigned to the inverted value of the node. A is set to X_i .

At least one instruction is required and no additional RRAM needs to be allocated in order to translate one node. In the worst case, six additional instructions and three additional RRAMs are required, e.g., cases (h), (e), and (d) for selecting operand B , destination Z , and operand A , respectively.

4.2.3 RRAM Allocation

We are interested in finding programs with a small number of instructions and a small number of RRAMs, i.e., optimized w.r.t. time and space. The rewriting algorithm that has been described in Section 4.1 addresses and affects both optimization criteria whereas the node translation described in Section 4.2.2 mainly targets the number of steps. In order to reduce the number of RRAMs, we have added a RRAM allocation. It implements an interface with two operations: (i) *request*, which returns an RRAM that is ready to use, and (ii) *release*, which releases an RRAM that is not required anymore. We implement this interface by using a free list that is populated with released RRAMs. Whenever an RRAM is requested, first it is checked whether a free released RRAM exists that can be re-used, or a new fresh RRAM is allocated.

RRAMs are requested whenever more than one instruction is required to translate a node (e.g., cases (g) and (h) for selecting operand B). RRAMs are released whenever all parents of a child have been computed. In order to address endurance constraints of the in-memory computing architecture, we implemented the interface based on a FIFO strategy, i.e., the *oldest* released RRAM is returned on request. Recently released RRAMs are released later this way.

5. EXPERIMENTAL RESULTS

The results of evaluating our approach for the EPFL benchmarks¹ is given in Table 1. The second column includes results for a naïve translation, where only the candidate selection scheme is disabled, based on the initial nonoptimized MIGs. The third and the fourth columns represent results after MIG rewriting and both rewriting and compilation, respectively. The number of iterations of the MIG rewriting algorithm is set to 4 during all experiments. The instructions and required RRAMs are shown by I and R , and the number of MIG nodes N is provided to give a better understanding of the MIG before and after rewriting. It is clear that N also shows the number of MIG nodes for the compiled PLiM, since the same MIG after rewriting has been used.

¹<http://lsi.epfl.ch/benchmarks>

Table 1: Experimental Evaluation

Benchmark	PI/PO	naïve			MIG rewriting				Rewriting and compilation				
		#N	#I	#R	#N	#I	impr.	#R	impr.	#I	impr.	#R	impr.
adder	256/129	1020	2844	512	1020	2037	28.38%	386	24.61%	1911	32.81%	259	49.41%
bar	135/128	3336	8136	523	3240	5895	27.54%	371	29.06%	6011	26.12%	332	36.52%
div	128/128	57247	146617	687	50841	147026	-0.03%	771	-12.22%	147608	-0.68%	590	14.12%
log2	32/32	32060	78885	1597	31419	60402	23.43%	1487	6.89%	60184	23.71%	1256	21.35%
max	512/130	2865	6731	1021	2845	5092	24.35%	867	15.08%	4996	25.78%	579	43.29%
multiplier	128/128	27062	76156	2798	26951	56428	25.91%	1672	40.24%	56009	26.45%	419	85.03%
sin	24/25	5416	12479	438	5344	10300	17.09%	426	2.73%	10223	18.08%	402	8.22%
sqrt	128/64	24618	60691	375	22351	47454	21.81%	433	-15.46%	49782	17.97%	323	13.87%
square	64/128	18484	54704	3272	18085	33625	38.53%	3247	0.76%	33369	39.00%	452	86.19%
cavlc	10/11	693	1919	262	691	1146	40.28%	236	9.92%	1124	41.43%	102	61.07%
ctrl	7/26	174	499	66	156	258	48.29%	55	16.66%	263	47.29%	39	40.91%
dec	8/256	304	822	257	304	783	4.74%	257	0.00%	777	5.47%	258	-0.39%
i2c	147/142	1342	3314	545	1311	2119	36.05%	487	10.64%	2028	38.81%	234	57.06%
int2float	11/7	260	648	99	257	432	33.33%	83	16.16%	428	33.95%	41	58.59%
mem_ctrl	1204/1231	46836	113244	8127	46519	85785	24.25%	6708	17.46%	84963	24.97%	2223	72.65%
priority	128/8	978	2461	315	977	2126	13.61%	241	23.49%	2147	12.76%	149	52.70%
router	60/30	257	503	117	257	407	19.09%	112	4.27%	401	20.28%	64	45.30%
voter	1001/1	13758	38002	1749	12992	25009	34.19%	1544	11.72%	24990	34.24%	1063	39.22%
Σ		236710	608655	22760	225560	486324	20.09%	19383	14.83%	487214	19.95%	8785	61.40%

#N: number of MIG nodes, #I: number of RM₃ instructions, #R: number of RRAMs, improvement is calculated compared to naïve

As expected, the number of MIG nodes have been reduced or remained unchanged for a few cases after MIG rewriting. Although, the number of nodes after MIG rewriting does not show a significant reduction, the sum of the number of instructions is reduced up to 20.09% compared to the naïve translation. This besides the 14.83% reduction achieved in the total number of RRAMs imply the effectiveness of the employed techniques for removing multiple inverted edges.

Performing both MIG rewriting and our optimized compilation approach, the number of required instructions and RRAMs reduces notably. The sum of the number of instructions and RRAM for the compiled PLiM are reduced by up to 19.95% and 61.4%, respectively in comparison with the corresponding values obtained for the naïve PLiM. This represents a significant reduction in both the latency and especially storage space metrics.

6. CONCLUSION

We have presented algorithms to automatically translate large Boolean functions into programs for the in-memory computing PLiM architecture. We observed that both the MIG representation and the way in which an MIG is compiled has a large impact on the resulting PLiM programs—in terms of required instructions as well as number of RRAMs. Experiments show that compared to a naïve translation approach the number of instructions can be reduced by up to 19.95% and the number of required RRAMs can be reduced by up to 61.40%. Our algorithm unlocks for the first time the potential of the PLiM architecture to process large scale computer programs using in-memory computing. This makes this promising emerging technology immediately available for nontrivial applications. Endurance constraints are addressed by RRAM allocation algorithms in our proposed compilation approach. As part of our future research we want to investigate the problem’s complexity and consider constraints in the optimization, e.g., a limited number of RRAMs.

Acknowledgments. This research was partly supported by H2020-ERC-2014-ADG 669354 CyberCare, by the University

of Bremen’s graduate school SyDe, funded by the German Excellence Initiative, and by the Swiss National Science Foundation project number 200021 146600.

7. REFERENCES

- [1] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli. Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization. In *DAC*, pages 194:1–194:6, 2014.
- [2] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli. Boolean logic optimization in majority-inverter graphs. In *DAC*, 2015.
- [3] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli. Majority-inverter graph: A new paradigm for logic optimization. *IEEE T-CAD*, 2015.
- [4] G. Birkhoff and S. A. Kiss. A ternary operation in distributive lattices. *Bull. of the Amer. Math. Soc.*, 53(8):749–752, 1947.
- [5] 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*, 464:873–876, 2010.
- [6] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.
- [7] P.-E. Gaillardon, L. G. Amarù, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli. The PLiM computer: Computing within a resistive memory array. In *DATE*, 2016.
- [8] Y. Ho, Huang, G.M., and P. Li. Dynamical properties and design analysis for nonvolatile memristor memories. *IEEE T-CS*, 58(4):724–736, 2011.
- [9] J. R. Isbell. Median algebra. *Trans. Amer. Math. Soc.*, 260(2):319–362, 1980.
- [10] E. Linn, R. Rosezin, S. Tappertzshofen, U. Böttger, and R. Waser. Beyond von Neumann-logic operations in passive crossbar arrays alongside memory operations. *Nanotechnology*, 23(305205), 2012.
- [11] 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*, 50(5):670–673, 2010.
- [12] S. Shin, K. Kim, and S.-M. Kang. Resistive computing: Memristors-enabled signal multiplication. *IEEE T-CS*, 60(5):1241–1249, 2013.
- [13] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler. Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs. In *DATE*, 2016.
- [14] H. P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai. Metal-oxide RRAM. *Proc. of the IEEE*, 100(6):1951–1970, 2012.