Improving Efficiency and Lifetime of Logic-in-Memory by Combining IMPLY and MAGIC Families

Minhui Zou\textsuperscript{a}, Junlong Zhou\textsuperscript{a,}\textsuperscript{c}, Jin Sun\textsuperscript{a}, Chengliang Wang\textsuperscript{b}, Shahar Kvatinsky\textsuperscript{c}

\textsuperscript{a} School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, 210094, China
\textsuperscript{b} School of Computer Science, Chongqing University, Chongqing, 400044, China
\textsuperscript{c} Electrical and Computer Engineering, Technion Israel Institute of Technology, Haifa, 3200003, Israel

A R T I C L E I N F O

Keywords:
Memristor
In-memory computing
Logic in memory
Synthesize and mapping
Efficiency
Lifetime

A B S T R A C T

Memristor-based in-memory computing has attracted much attention recently. By combining the storability and computability of memristor devices together, the memristor-based in-memory computing could break the so-called von Neumann bottleneck. Logic-in-memory (LIM) aims at implementing any computing task in memory. IMPLY family and MAGIC family are two of the most popular stateful logic families of LIM. The two families have their own respective advantages and disadvantages. However, there is few work combining the two family gates together in a same memristor crossbar. In this paper, we would present X-IMPLY family gates that eliminates the required external resistors of conventional IMPLY family gates. We then show by combining X-IMPLY and MAGIC family gates, the advantages of both logic families are exploited. At last, we evaluate the proposed method on state-of-art benchmarks. The results show, averagely, our method saves more than 15% of cell usage and is more than 22% faster and increases lifetime by more than 41% compared with MAGIC SIPLER for different size constrains of crossbar row/column.

1. Introduction

The past decades have seen magnitude orders’ improvement of energy efficiency brought by the scaling of the CMOS technology as predicted by the Moore’s law. However, as the scaling of transistor size approaching to its physical limit, it is increasingly harder for the conventional computer architecture to continue improving energy efficiency through process advancement [1]. Besides, the data-intensive applications requiring huge amount of data movement between the computing units and the memory units become more and more popular [2–4]. The traditional computer architecture that separates computing units and memory units is challenged [5].

Memristor-based in-memory computing (IMC) has been a potential solution to the aforementioned challenges. By combining the storability and computability together, the memristor-based IMC could process the data where it is to avoid the data movement between the computing units and the memory units. The memristor-based IMC could be broadly categorized as neuromorphic computing (NC) and Logic-in-memory (LIM). The former takes advantage of analogous matrix-vector-multiplications with memristor crossbars to boost the processing of neural network computing [6–9]. However, NC is a specific task and cannot be used for other tasks. The latter is for general IMC, which aims at implementing any logic function with memristor devices [10–13]. The first work of LIM is IMPLY [10], which implemented an IMP gate with two memristor cells and one resistor, as shown in Fig. 3(a).

To overcome the problem of input overwriting of IMPLY family gates, [15] proposed to insert additional gates in gate-level netlists to avoid fan-out. The method causes heavy overhead when the fan-out exists in higher layers of the netlists. [12,14] proposed to copy the inputs before they are overwritten. However, the execution of a copy operation in LIM takes two steps, either an initialization step plus an operation step or a readout step plus a write step. Too many copy operations could affect the computing latency greatly. MAGIC [11] put forward a memristor-based NOR gate, which avoids the problem of input overwriting, as is shown in Fig. 3(c). The MAGIC NOR gate is
we improve it to fit for the combination of X-IMPLY family and MAGIC SIMPLER [18] and SAID [19] in terms of throughput and parallelism. Crossbar row/column, which outperforms other mapping tools, such as MAGIC SIMPLER [13] maps a gate-level netlist into a single memristor family gates, a new mapping method is required. The mapping tool better exploit the combination of X-IMPLY family gates and MAGIC family gates is connecting external resistors to the memristor crossbars. However, external resistors bring in additional hardware overhead. Also, the resistors will be in the periphery and not inside the memristor crossbars, and will be connected through the crossbar row/column decoder, which complicates the decode. In this paper, we propose X-IMPLY family gates by replacing the resistors of the IMPLY family gates with in-crossbar memristor cells so that the compatibility of X-IMPLY family gates and MAGIC family gates is enabled without additional hardware overhead. Besides, in order to better exploit the combination of X-IMPLY family gates and MAGIC family gates, a new mapping method is required. The mapping tool MAGIC SIMPLER [13] maps a gate-level netlist into a single memristor crossbar row/column, which outperforms other mapping tools, such as SIMPLE [18] and SAID [19] in terms of throughput and parallelism. We follow MAGIC SIMPLER’s idea of single-row/column mapping, but we improve it to fit for the combination of X-IMPLY family and MAGIC family within a same memristor crossbar.

The contributions of this work are summarized below:

- This work first proposes X-IMPLY family gates by replacing the resistors of the IMPLY family gates with in-crossbar memristor cells. To the best of our knowledge, this is the first work on this topic. We also examine the compatibility of X-IMPLY family gates and MAGIC family gates within a same memristor crossbar.
- This work then shows an improved single-row/column mapping method to combine both family gates in a same crossbar to reduce cell usage and latency.
- At last, the proposed technique is tested on benchmark circuits. Comparing to MAGIC SIMPLER, the proposed technique averagely saves more than 15% of cell usage and reduces more than 22% of latency and increases lifetime by more than 41% for different size constraints of crossbar row/column.

The rest of the paper is organized as follows. Section 2 provides the background about the memristor devices and the memristor-based IMPLY and MAGIC family gates. Section 3 presents the X-IMPLY family and its compatibility with MAGIC family. Section 4 proposes the synthesis and mapping method of combining X-IMPLY and MAGIC family gates. Section 5 shows the experimental results and Section 6 concludes this paper.

2. Background and definitions

2.1. Background

2.1.1. RRAM devices and threshold-based switching

To our best knowledge, all the stateful memristor logic families take use of the characteristic of threshold-switching memristor [10,11,14, 17]. The voltage-threshold memristor model of this paper is similar to [14]. As shown in Fig. 2(a), the polar with thicker line of a memristor device is labeled as positive polar and the other polar is labeled as negative polar. The resistance of a memristor device is controlled by the voltage across it, which is denoted as \( v_{pn} \). The resistance of the memristor device is denoted as \( R \). The values of \( R \) are for guaranteed \( v_{on} \) and \( v_{off} \) are reverse-directed. The magnitude of \( v_{on} \) represents logic 0 and \( v_{off} \) represents logic 1. The process of a memristor device’s resistance changing from \( R_{off} \) to \( R_{on} \) is called \( Set \) and the opposite process is called \( Reset \), which is shown in Fig. 2(b).

There are four threshold voltages for the memristor device: \( V_{on} \), \( V_{off} \), \( V_{Set} \), and \( V_{Reset} \). Note that, \( V_{off} \) and \( V_{Reset} \) are forward-directed and \( V_{on} \) and \( V_{Set} \) are reverse-directed. The magnitude of \( V_{Reset} \) (\( V_{Set} \)) is higher than that of \( V_{off} \) (\( V_{on} \)). \( V_{Reset} \) and \( V_{Set} \) are for guaranteed \( Reset \) and \( Set \) initializations, respectively. The behavior of the memristor model is explained as below:

Fig. 2. (a) A memristor device; (b) State switching of memristor devices.
2.1.2. Memristor-based logic gates and logic synthesis

As shown in Fig. 3(a), a memristor-based IMPLY family gate consists of two memristor devices \((i1\) and \(i2\)) and one resistor \((R_G)\). Memristor \(i1\), \(i2\) and resistor \(R_G\) are connected to voltage sources \(V_{i1}\), \(V_{i2}\) and GND, respectively. There are four combinations for input \((i1, i2)\), which are \((0,0)\), \((0,1)\), \((1,0)\), and \((1,1)\). The principle of IMPLY family gates is creating a conditional switching \((\text{Set or Reset})\) for memristor \(i2\) by choosing the values of \(V_{i1}\), \(V_{i2}\) and \(R_G\) while keeping memristor \(i1\) unchanged. For example, a conditional switching could be created that \(i2\) switches only when the input \((i1, i2)\) is \((0,0)\) and does not switch for any other input combinations. This conditional switching is analogous to logic gate IMP, of which the truth table is shown in Fig. 3(b). The output of the IMPLY family gate is the logic value of memristor \(i2\) after the execution of the gate, thus the output \(o\) overwrites the input \(i2\). Similarly, [14] puts forward IMPLY AND, IMPLY OR, and IMPLY NIMP gates by taking use of the conditional switching for the memristor \(i2\) when the input \((i1, i2)\) are \((0,1)\), \((1,0)\), and \((1,1)\), respectively.

As shown in Fig. 3(c), a memristor-based MAGIC family gate is made up of three memristor devices \((i1, i2\) and \(o\)). Both \(i1\) and \(i2\) are connected to voltage source \(V_{i1}\) and \(V_{i2}\) and connected to GND. The MAGIC family gate requires the output memristor \(o\) to be initialized before the execution of the gate. The principle of MAGIC family gates is creating a conditional switching \((\text{Set or Reset})\) for memristor \(o\) by choosing the values of \(V_{i1}\) while keeping memristor \(i1\) and \(i2\) unchanged. For example, a conditional switching could be created that memristor \(o\) switches when the input \((i1, i2)\) is one of \((0,1)\), \((1,0)\) and \((1,1)\) and does not switch only when the input combination is \((0,0)\). This conditional switching is analogous to logic gate NOR, of which the truth table is shown in Fig. 3(d). Similarly, by taking use of other conditional switching for memristor \(o\), other MAGIC family gates could be implemented. For example, memristor \(o\) is initialized as \(0\) and only when the input combination \((i1, i2)\) is \((0,1)\), the memristor \(o\) switches to \(1\). This conditional switching is analogous to logic gate NIMP.

Table 1 shows the comparison of IMPLY family gates and MAGIC family gates. Compared with an IMPLY family gate, a MAGIC family gate has higher cell usage and longer latency (initialization step + execution step). However, MAGIC NOR gate does not require resistors connected to the memristor crossbars.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>IMPLY family</th>
<th>MAGIC family</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell usage</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Require initialization</td>
<td>NO</td>
<td>Yes</td>
</tr>
<tr>
<td>Require resistor</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

MAGIC SIMPLER is a state-of-the-art synthesis and mapping tool that converts any logic function into a gate-level netlist consists of only NOR and NOT gates and then maps it to a single crossbar row/column. The mapping results are the execution orders of each gate of the netlist and the physical location of each net of the netlist on the memristor crossbars.

2.2. Definitions

For the ease of discussion, let us introduce the definitions for this paper.

Definition 1. function gate, a gate that is not a NOT gate, i.e. IMP gate, NIMP gate or OR gate;

Definition 2. in1 input, the input net of a function gate that is not overwrittern by the output when the gate is implemented with (X-)IMPLY family;

Definition 3. in2 input, the input net of a function gate that is overwrittern by the output when the gate is implemented with (X-)IMPLY family;

Definition 4. ReadIN2 gate, a function gate with its in1 input being the in2 input of other function gates or a NOT gate with its input being the in2 input of other function gates.

3. X-IMPLY family and its compatibility with MAGIC family

To make a memristor crossbar support both families at the same time, one obvious option is to add resistors to the memristor crossbar. For example, for a memristor crossbar with size of \(256 \times 256\), to enable an IMPLY family gate execute both in any crossbar row or column, each crossbar row and column needs a resistor, as shown in Fig. 4. Then the memristor crossbar needs 512 resistors, the hardware overhead of which is not neglected. Additionally, the resistors will be in the periphery and not inside the memristor crossbars, and will be connected through the crossbar row/column decoder, which complicates the decode. In this section, we would propose X-IMPLY family gates by replacing the resistors of the IMPLY family gates with in-crossbar memristor cells, thus the versatility of the crossbars is enabled without additional hardware overhead.

3.1. X-IMPLY family gates

As shown in Fig. 5(a), the X-IMPLY family gate replaces the resistor \(R_G\) of IMPLY family gates with an in-crossbar memristor cell \(R'_{ij}\), which is connected to voltage source \(V_{ij}\). The memristor cell \(R'_{ij}\) does not store any operand or output. Instead, it functions as a load resistor just like the resistor \(R_G\) of IMPLY family gates does. The operation result \(o\) still overwrites the input \(i2\). For the ease of later discussion, let us denote the voltage potential of the shared net among \(i1\), \(i2\), and \(R'_{ij}\), as \(V_{ij}\). The resistance of the resistor \(R_G\) of an IMPLY family gate is constant and does change throughout the execution of the gate function. The resistance of the memristor \(R'_{ij}\) of an X-IMPLY family gate needs to be constant, too. However, as mentioned in Section 2, only within the voltage range \([V_{\text{on}}, V_{\text{off}}]\), the resistance of memristor devices persists.
Thus, the voltage across $R_{g}^{'1}$ throughout the execution of the gate must be within the range $[V_{in1}, V_{off}]$.

Let us take the X-IMPLY IMP gate as example. Table 2 lists the constrains for all the input combinations of X-IMPLY IMP gate. According to Kirchhoff's current law, the relation of $V_{1}$, $V_{2}$ and $V_{3}$ satisfies $V_{1} - V_{2} + V_{3} = 0$, where $R_{in1}$ and $R_{in2}$ stand for the resistance of $in1$ and $in2$, respectively. For example, when the input ($in1$, $in2$) is (0, 1), $R_{in1}$ is $R_{off}$ and $R_{in2}$ is $R_{on}$. The $in1$ of the X-IMPLY IMP gate is not overwritten for all input combinations and stays unchanged. When $in1$ is 0, the voltage across it satisfies $V_{1} - V_{2} \geq V_{on}$; and when $in1$ is 1, the voltage across it satisfies $V_{1} - V_{2} \leq V_{off}$. For $in2$, when the input ($in1$, $in2$) is (0, 0), its value would be overwritten by output $out$. In this case, to ensure $in2$ switches from 0 to 1, the voltage across it satisfies $V_{2} - V_{3} \leq V_{sat}$. For other input combinations, the value of $in2$ does not change and the voltage across it resembles that of $in1$. The resistance of $R_{g}^{'1}$ is between $R_{on}$ and $R_{off}$ and does not change for all input combinations, thus the voltage across $R_{g}^{'1}$ satisfies $V_{on} \leq V_{3} - V_{2} \leq V_{off}$.

Similarly, we examine all the other IMP family gates to transform them into IMP family gates. The memristor model parameters are shown in Table 3, which are similar to those of [14]. However, only IMP gate, OR gate, and NIMP that are transformed could work under the memristor model parameters. The AND gate works only when $|V_{out}| > 1$.

Note that the location of the memristor cell $R_{g}^{'1}$ is not fixed. If an X-IMPLY family gate is to be implemented in a crossbar row/column, the $R_{g}^{'1}$ could be any memristor cell in that row/column.

### 3.2. Compatibility of X-IMPLY family gates and MAGIC family gates

As is shown in Fig. 5(b), to relax the design constrains for MAGIC family gates, the positive polar of $in1$ and $in2$ are connected to different voltage sources $V_{1}$ and $V_{2}$, respectively [14]. Besides, the positive polar of the memristor cell $out$ is connected to voltage source $V_{4}$ instead of GND and an memristor $R_{g}^{'1}$ functioning as a load resistor is added, which is different from [14]. The memristor $R_{g}^{'1}$ is connected to voltage source $V_{4}$. Similarly, the voltage potential of the shared net among $in1$, $in2$, $R_{g}^{'1}$, and $out$ is also denoted as $V_{g}$.

Let us take the MAGIC IMP gate as example. Table 4 lists the constrains for all the input combinations of MAGIC IMP gate. The $out$ is initialized as 1 for each input combination and it switches to 0 only when ($in1$, $in2$) is (1, 0). The memristor $in1$, $in2$ and $R_{g}^{'1}$ stay unchanged for all input combinations.

Table 5 lists the compatible X-IMPLY family gates and MAGIC family gates. The second, third, and fourth rows are X-IMPLY family gates. The rest rows below are MAGIC family gates. The X-IMPLY IMP gate, X-IMPLY NIMP gate, and MAGIC IMP gate require $R_{g}^{'1}$ to be tuned into related resistance ranges.

NOT gate is required by many synthesis and mapping tools [11,13,18,20], which is required as well in this work. The NOT gate (NOT($in1$)) could be implemented as X-IMPLY IMP($in1,0$), in which the $in2$ is first initialized as 0 and then overwritten by the output.

## 4. Combining X-IMPLY family gates and MAGIC family gates

Section 3 has shown a memristor crossbar could support both X-IMPLY family gates and MAGIC family gates at the same time. Based on that, we could implement any logic function in memristor crossbars. As shown in Fig. 1, for a given logic function (in the form of HDL netlist), the first step is to synthesize it into a gate-level netlist consists of supported memristor-based logic gates by the memristor crossbars. The second step is to map the gate-level netlist to the memristor crossbars, i.e. generating the execution sequences.

### 4.1. Synthesising the given HDL netlist

[12,13,18,20] synthesize the given HDL netlist with cell library (NOR, NOT) by using ABC synthesize tool [21] to generate a gate-level netlist consists of only NOR gates and NOT gates. The cell library

---

**Table 2** Constraints for X-IMPLY IMP gate.

<table>
<thead>
<tr>
<th>Inputs and output</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$in1=0$, $in2=0$, $out=1$</td>
<td>$\frac{V_{in1} - R_{on}}{R_{in1}} + \frac{R_{in1}}{R_{in2}} + \frac{V_{in2}}{R_{in2}} = 0$</td>
</tr>
<tr>
<td>$in1=0$, $in2=1$, $out=1$</td>
<td>$\frac{V_{in1} - R_{on}}{R_{in1}} + \frac{R_{in1}}{R_{in2}} + \frac{V_{in2}}{R_{in2}} = 0$</td>
</tr>
<tr>
<td>$in1=1$, $in2=0$, $out=0$</td>
<td>$\frac{V_{in1} - R_{on}}{R_{in1}} + \frac{R_{in1}}{R_{in2}} + \frac{V_{in2}}{R_{in2}} = 0$</td>
</tr>
<tr>
<td>$in1=1$, $in2=1$, $out=1$</td>
<td>$\frac{V_{in1} - R_{on}}{R_{in1}} + \frac{R_{in1}}{R_{in2}} + \frac{V_{in2}}{R_{in2}} = 0$</td>
</tr>
</tbody>
</table>

**Table 3** The memristor model parameters.

| $R_{on}$ | 1 kΩ |
| $R_{off}$ | 300 kΩ |
| $V_{sat}$ | −2.5 V |
| $V_{off}$ | 1.9 V |
| $V_{out}$ | −3 V |
| $V_{GND}$ | 2.4 V |
by adding one or more supported gates. All the cell libraries we extend the three least-size cell libraries {IMP, NOT}, {NIMP, NOT}, and {OR, NOT}. As shown in Table 6, {IMP, NOT}, {NIMP, NOT}, {OR, NOT} are also functionally complete. As mentioned in Section 2, IMPLY family gates save cell usage and latency at the best effort. We propose to combine X-IMPLY family gates and MAGIC family gates to reduce cell usage and latency and at the same time avoid input overwriting at the best effort.

Let us demonstrate the idea with a simple example. Fig. 6(a) shows the HDL netlist of a half adder circuit, which is made up of an XOR gate and an AND gate. Fig. 6(b) shows the corresponding synthesized gate-level netlist with cell library (IMP, NOT). The generated gate-level netlist consists of 3 IMP gates and 3 NOT gates.

is chosen because the combination of NOR gate and NOT gate is functionally complete set, which has the capability to construct any logic function. However, we notice that the sets (IMP, NOT), {NIMP, NOT}, {OR, NOT} are also functionally complete. As shown in Table 6, we have synthesized the LGsynth91 benchmark suite [22] with different cell libraries: (NOR, NOT), (IMP, NOT), (NIMP, NOT), (OR, NOT). For example, the gate-level netlist generated by synthesizing benchmark 5x5p1 with cell library {NOR, NOT} has 80 NOR gates and 32 NOT gates. The number of gates of the generated netlist affects the cell usage and latency overhead after it is mapped to a memristor crossbar. The gate-level netlist with least number of gates normally cost less memristor cells and latency. From the results, we could see the optimal cell library is {IMP, NOT}. In Fig. 6, the in2 input of gate 5 is shared by another gate G4. If gate G4 is executed before gate G2, G2 would be implemented with an X-IMPLY IMP gate since net n5 is no longer accessed by other gates. However, if G4 is executed after G2, n5 must be remains as accessible after the execution of gate G2, and thus G2 could not be implemented with an X-IMPLY IMP gate. For IMP gate G5, its in2 input Cout is not shared by any other gates, and thus gate G5 could be implemented with X-IMPLY IMP gate to save one cell and one initialization step.

Motivated by above example, we put forward a general standard of deciding which logic family to choose to implement for a function gate. The default logic family to choose for a function gate is MAGIC family. In order to reduce cell usage and latency at the best effort, we hope to implement as much function gates with X-IMPLY family gates as possible. A function gate implemented with an X-IMPLY family gate must satisfy either one of the two conditions below.

- Condition 1: The in2 input of the function gate is not shared by any other gates.
- Condition 2: The in2 input of the function gate is shared by other gates, but the function gate is executed after the gates that share the in2 input of the function gate.

Mapping the gate-level netlist into memristor crossbars is deciding the execution order and cell location of each gate of the gate-level netlist. The netlist is regarded as a directed acyclic graph. We follow the synthesis workflow of MAGIC SIMPLER and apply Depth-First Search to process from the root net to the leaf nets. Regarding the branches of a net, MAGIC SIMPLER calculates the cell usage (CU) of each net for the priority of the larger branch. However, to fit for our case, we modify the procedure of deciding which branch first. For the aim of maximizing the advantage of X-IMPLY family gates, we prioritize the branch with more ReadIN2 gates. We use ReadIN2 weight (RW) to estimate the number

<table>
<thead>
<tr>
<th>Inputs and output</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \frac{V_i-V_j}{R_{ij}} + \frac{V_i-V_k}{R_{ik}} = 0 )</td>
</tr>
<tr>
<td>( i=0 ), ( i=0 ), ( o=1 )</td>
<td>( V_i = V_j \geq V_o )</td>
</tr>
<tr>
<td></td>
<td>( V_i = V_k \geq V_o )</td>
</tr>
<tr>
<td></td>
<td>( V_a \leq V_i - V_j \leq V_{off} )</td>
</tr>
<tr>
<td></td>
<td>( V_i - V_j \leq V_{eff} )</td>
</tr>
</tbody>
</table>

| \( i=0 \), \( i=1 \), \( o=1 \) | \( V_i = V_j \geq V_o \) |
|                  | \( V_i = V_k \geq V_o \) |
|                  | \( V_a \leq V_i - V_j \leq V_{off} \) |
|                  | \( V_i - V_j \leq V_{eff} \) |

| \( i=1 \), \( i=2 \), \( o=0 \) | \( V_i = V_j \leq V_{off} \) |
|                  | \( V_i = V_k \geq V_o \) |
|                  | \( V_a \leq V_i - V_j \leq V_{off} \) |
|                  | \( V_i - V_j \leq V_{eff} \) |

| \( i=1 \), \( i=2 \), \( o=1 \) | \( V_i = V_j \leq V_{off} \) |
|                  | \( V_i = V_k \geq V_o \) |
|                  | \( V_a \leq V_i - V_j \leq V_{off} \) |
|                  | \( V_i - V_j \leq V_{eff} \) |

**Table 5**

<table>
<thead>
<tr>
<th>Compatible X-IMPLY family gates and MAGIC family gates.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_i )</td>
</tr>
<tr>
<td>X-IMPLY IMP</td>
</tr>
<tr>
<td>X-IMPLY OR</td>
</tr>
<tr>
<td>X-IMPLY NIMP</td>
</tr>
<tr>
<td>MAGIC IMP</td>
</tr>
<tr>
<td>MAGIC OR</td>
</tr>
<tr>
<td>MAGIC NIMP</td>
</tr>
</tbody>
</table>
of ReadIN2 gates of each branch. The algorithm of calculating \( RW \) is shown in algorithm 1. Given the net \( i \), the algorithm outputs the \( RW \) of net \( i \). The algorithm first checks the gate type of net \( i \). If \( i \) is a primary input, then its \( RW \) is 0. The algorithm then checks whether the driving gate \( gate_i \) of net \( i \) is a ReadIN2 gate. If yes, the \( RW \) of \( i \) is the sum of the \( RW \) of \( gate_i \)'s input nets plus 1; if no, the \( RW \) of \( i \) is just the sum of the \( RW \) of \( gate_i \)'s input nets.

Algorithm 1 ComputeRW

Compute the \( RW \) of net \( i \)
1: Inputs: net \( i \); 
2: Output: \( RW \) of net \( i \); 
3: for net \( i \) do 
4: if net \( i \) is a primary input then 
5: return 0; 
6: else 
7: \( gate_i \) = the driving gate of net \( i \)
8: if \( gate_i \) is a NOT gate then 
9: \( i_{in} \) = the input net of \( gate_i \)
10: if \( gate_i \) is a ReadIN2 gate then 
11: return ComputeRW(\( i_{in} \])+1; 
12: else 
13: return ComputeRW(\( i_{in} \)) 
14: end if 
15: else //\( gate_i \) is a function gate 
16: \( i_{in1} \) = the net as in1 input of \( gate_i \) 
17: \( i_{in2} \) = the net as in2 input of \( gate_i \) 
18: if \( gate_i \) is a ReadIN2 gate then 
19: return ComputeRW(\( i_{in1} \)) + ComputeRW(\( i_{in2} \)) +1; 
20: else 
21: return ComputeRW(\( i_{in1} \)) + ComputeRW(\( i_{in2} \)) 
22: end if 
23: end if 
24: end if 
25: end for

The \( RW \) is the guide for deciding which branch goes first. If the branches of a net have the same value of \( RW \), then we choose the branch with greater \( CU \) as the first.

To combine X-IMPLY and MAGIC family gates, we also modify the AllocateCell algorithm of MAGIC SIMPLER, which is shown as in algorithm 2. The algorithm modifies the part where the fan-out (FO) of the net \( i \) is 1. That is when all the gates sharing the in2 input of \( gate_i \) are already assigned with memristor cells, the in2 input of \( gate_i \) is no longer needed after the execution of it and \( gate_i \) could be implemented with an X-IMPLY gate. For the other parts, the algorithm is the same with the original AllocateCell algorithm.

The time complexity of algorithm 1 and 2 is the same with algorithm ComputeCU and AllocateCell of MAGIC SIMPLER, respectively. In our method, the given netlist is synthesized and mapped for \( L \) times. Thus, the time complexity of our method is \( L \) times of that of MAGIC SIMPLER.

Algorithm 2 NewAllocateCell

Allocate the cell for net \( i \)
1: Input: net \( i \) (not a primary input); 
2: Output: the memory location \( map_i \); 
3: for net \( i \) do 
4: \( gate_i \) = the driving gate of net \( i \)
5: if \( gate_i \) is a NOT gate then 
6: do the same as AllocateCell algorithm does; 
7: else //\( gate_i \) is a function gate 
8: \( i_{in1} \) = the net as in1 input of \( gate_i \) 
9: \( i_{in2} \) = the net as in2 input of \( gate_i \) 
10: if FO(\( i_{in2} \))==1 then //implement \( gate_i \) with an X-IMPLY family gate
11: FO(\( i_{in2} \))=FO(\( i_{in2} \))-1; 
12: map_i =GetMap(\( i_{in2} \)); 
13: \( t=t+1 \) //the gate operation takes one clock 
14: return map_i 
15: else //implement \( gate_i \) with MAGIC family gate 
16: do the same as AllocateCell algorithm does; 
17: end if 
18: end if 
19: end for

5. Experiments

In this section we would show the comparison results between the proposed method and MAGIC SIMPLER. The hardware environment of the experiments is i7-9700 CPU and 16GB DRAM memory. Both of the proposed method and MAGIC SIMPLER are implemented on LGsynth91 benchmark suite and EPFL benchmark suite [23]. For the proposed method, each of the \( L \) generated netlists is mapped to generate execution sequences individually. Note that the synthesis and mapping methods are very efficient and each benchmark could be done within a few minutes. The optimal list of execution sequences is chosen with the priority of least cell usage or latency. In this paper we choose the list of execution sequences with least latency as the optimal. The optimal list of execution sequences generated by the proposed method is compared to the list of execution sequences generated by MAGIC SIMPLER in terms of cell usage, latency and lifetime.
5.1. Cell usage and latency

In the first set of experiments, we show the improvement of the proposed method compared with MAGIC SIMPLER in terms of cell usage and latency. In order to have an apple-to-apple comparison with MAGIC SIMPLER, every benchmark is tested on single crossbar row/column with (1) minimal number of memristor cells required (MinCells), (2) MinCells plus max(5% of MinSize, 10) memristor cells (MinCells plus), and (3) unlimited number of memristor cells (UnlimitedCells) as the same with MAGIC SIMPLER does. As is shown in Fig. 7, the proposed method has made notable improvement overall. However, for benchmarks bar and sin, our method lags slightly behind MAGIC SIMPLER in terms of cell usage. For benchmark bar, the generated gate-level netlist of MAGIC SIMPLER requires less gates than that of our method. For benchmark sin, part of ReadIN2 gates are in small branches so that less cells are reused. The proposed method averagely saves more than 15% of cell usage compared with MAGIC SIMPLER for different crossbar size constraints for both benchmark suites. When with unlimited cells, the cell-saving benefit of the proposed method is maximized, which is more than 59% averagely. As to latency, the proposed method is averagely more than 28% and 16% faster for different size constraints of crossbar row/column for LGsynth91 benchmark suite and EPFL benchmark suite, respectively.

5.2. Lifetime

In this section, we would show that our method could increase the lifetime of memristor-based computing system by reducing write operations to the memristor devices. We count one writing operation for each primary input of benchmarks for both X-IMPLY and MAGIC family gates. Every MAGIC family gate is counted as only one writing operation. However, each X-IMPLY NOT gate is counted as two writing operations since it needs to initialize the output memristor cell just like the MAGIC family gates do. The results are shown in Fig. 8. Averagely, the proposed method increases the lifetime of memristor-based computing system by more than 46% for LGsynth91 benchmark suite and 37% for EPFL benchmark suite as compared to MAGIC SIMPLER.

6. Conclusion

The IMPLY family gates and MAGIC family gates see individual development and applications. The two kinds of family gates have their own respective advantages and disadvantages. In this paper, we propose to support both families in a same memristor crossbar. To mitigate the hardware overhead brought by the compatibility of the two families, we propose X-IMPLY family gates by replacing the external resistors of the conventional IMPLY gates with in-crossbar memristor cells. To exploit the advantages of both family gates, we propose an optimized method to combine both family gates. The experiment results show that our method achieves better efficiency and lifetime improvement than MAGIC SIMPLER. In the future work, we would explore the impact of the PVT variation of the memristor devices on the proposed methods and put forward the countermeasures.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by National Natural Science Foundation of China (61672115, 61802185, 61872185), Natural Science Foundation of Jiangsu Province, China (BK20180470, BK20190447), China Postdoctoral Science Foundation (2021T140327, 2020M680068), Fundamental Research Funds for the Central Universities, China (30919011233, 30919011402), National Trusted Embedded Software Engineering Technology Research Center (East China Normal University), and Israel Science Foundation (1514/17).

References


