Journal of Systems Architecture 119 (2021) 102232

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

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

Minhui Zou [a], Junlong Zhou [a,*], Jin Sun [a], Chengliang Wang [b], Shahar Kvatinsky [c]

[a] School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, 210094, China
[b] School of Computer Science, Chongqing University, Chongqing, 400044, China
[c] Electrical and Computer Engineering, Technion Israel Institute of Technology, Haifa, 3200003, Israel

ARTICLE INFO

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

ABSTRACT

Memristor-based 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 SIMPLER 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). The inputs and output of the IMP gate are purely stored in memristor devices. [14] extended the IMP gate and put forwards three other IMP-like gates also implemented with two memristor cells and one resistor, which are OR gate, AND gate, and NIMP gate. Let us collectively denote the IMP gate and the extended IMP-like gates as **IMPLY family gates**. The IMPLY family gates do not need initialization steps. However, an IMPLY family gate overwrites one of its inputs, which could be a barrier for multi-cascading.

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

* Corresponding author.
E-mail addresses: zouminhui@njust.edu.cn (M. Zou), jlzhou@njust.edu.cn (J. Zhou), sunj@njust.edu.cn (J. Sun), wangcl@cqu.edu.cn (C. Wang), shahar@ee.technion.ac.il (S. Kvatinsky).

https://doi.org/10.1016/j.sysarc.2021.102232
Received 20 February 2021; Received in revised form 26 May 2021; Accepted 28 June 2021
Available online 9 July 2021

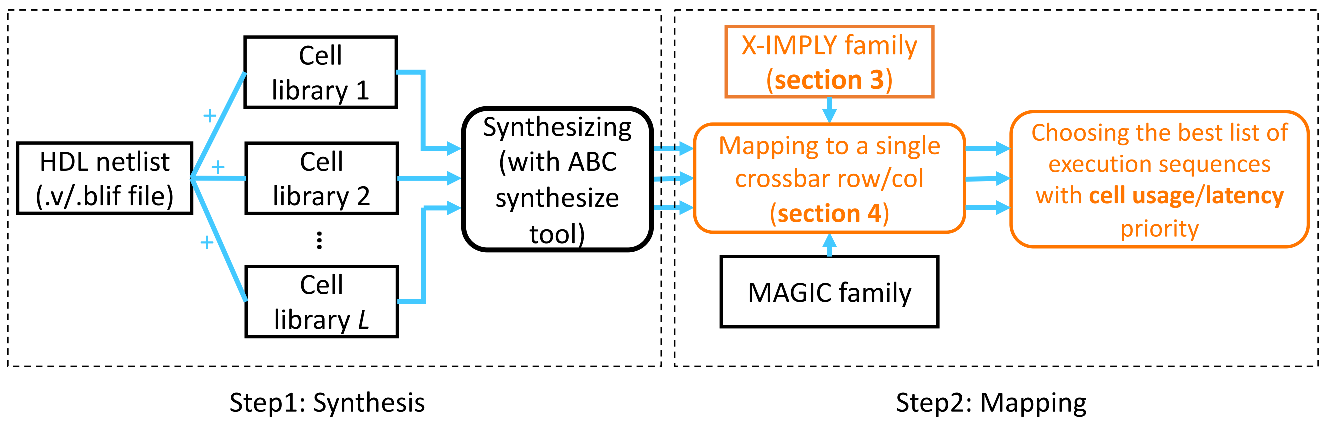

HDL netlist (.v/.blif file)
Cell library 1
Cell library 2
Cell library L
Synthesizing (with ABC synthesize tool)
X-IMPLY family (section 3)
Mapping to a single crossbar row/col (section 4)
MAGIC family
Choosing the best list of execution sequences with cell usage/latency priority
Step1: Synthesis
Step2: Mapping

Fig. 1. The workflow of the proposed method: the HDL netlist is synthesized by using ABC synthesize tool with L different cell libraries; then individually map each generated gate-level netlist to a crossbar row/column with both X-IMPLY and MAGIC family gates to generate a list of execution sequences; at last choose the optimal list of execution sequences with the least cell usage or latency.

implemented with three memristor cells, one of which is dedicated for storing the operation output. Thus the inputs remain unchanged after the execution of the gate. [16,17] extended the MAGIC NOR gate and put forwards some other MAGIC-like gates, such as OR gate and NIMP gate. Let us collectively denote the MAGIC NOR gate and the extended MAGIC-like gates as **MAGIC family gates**. Although a MAGIC family gate does not overwrite its inputs, it needs one additional initialization step and three memristor cells, which cost more time and memristor cells compared with an IMPLY family gate.

From above, we can see that both IMPLY family gates and MAGIC family gates have their own respective advantages and disadvantages and the two logic families complement each other. Based on this observation, we propose to combine the two logic families to exploit the advantages of them both. Both families are compatible with memristor crossbars. However, the IMPLY family gates require external resistors connected to the bitlines or writelines of the memristor crossbars. A straightforward method to adding the compatibility of IMPLY family gates to 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 constrains of crossbar row/column.

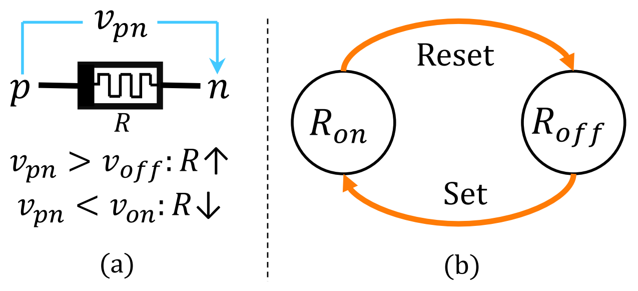


Fig. 2. (a) A memristor device; (b) State switching of memristor devices.

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.

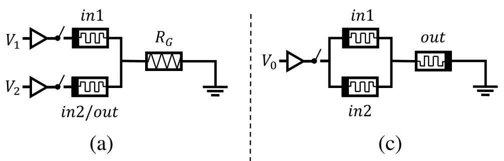
2. Background and definitions

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 denoted as R_{off} and R_{on} when the memristor device is in the highest resistance state and the lowest resistance state, respectively. In this paper, R_{off} represents logic 0 and R_{on} 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 as 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:



in1	in2	out
0	0	1
0	1	1
1	0	0
1	1	1

(b)

in1	in2	out
0	0	1
0	1	0
1	0	0
1	1	0

(d)

Fig. 3. (a) A memristor-based IMPLY family gate; (b) The truth table of IMP gate; (c) A memristor-based MAGIC family gate; (d) The truth table of NOR gate.

- When $v_{pn} > V_{off}$: R increases;
- When $v_{pn} \geq V_{Reset}$: R changes to R_{off} unconditionally, which is for guaranteed $Reset$ initialization;
- When $v_{pn} < V_{on}$: R decreases;
- When $v_{pn} \leq V_{Set}$: R changes to R_{on} unconditionally, which is for guaranteed Set initialization;
- When $V_{on} \leq v_{pn} \leq V_{off}$: R stays unchanged.

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 ($in1$ and $in2$) and one resistor (R_G). Memristor $in1$, $in2$ and resistor R_G are connected to voltage sources V_1, V_2 and GND, respectively. There are four combinations for input ($in1$, $in2$), which are (0,0), (0,1), (1,0), (1,1). The principle of IMPLY family gates is creating a conditional switching (Set or $Reset$) for memristor $in2$ by choosing the values of V_1, V_2, and R_G while keeping memristor $in1$ unchanged. For example, a conditional switching could be created that $in2$ switches only when the input ($in1$, $in2$) 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 $in2$ after the execution of the gate, thus the output out overwrites the input $in2$. Similarly, [14] puts forward IMPLY AND, IMPLY OR, and IMPLY NIMP gates by taking use of the conditional switching for the memristor $in2$ when the input ($in1$, $in2$) 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 ($in1$, $in2$ and out). Both $in1$ and $in2$ are connected to voltage source V_0 and out is connected to GND. The MAGIC family gate requires the output memristor out to be initialized before the execution of the gate. The principle of MAGIC family gates is creating a conditional switching (Set or $Reset$) for memristor out by choosing the values of V_0 while keeping memristor $in1$ and $in2$ unchanged. For example, a conditional switching could be created that memristor out switches when the input ($in1$, $in2$) 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 out, other MAGIC family gates could be implemented. For example, memristor out is initialized as 0 and only when the input combination ($in1$, $in2$) is (0,1), the memristor out 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 1
Comparison of IMPLY family gates and MAGIC family gates.

	IMPLY family	MAGIC family
Cell usage	2	3
Require initialization	NO	Yes
Require resistor	Yes	No

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 overwritten 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 overwritten 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 $256 * 256$, 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'_G, which is connected to voltage source V_g. The memristor cell R'_G 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 out still overwrites the input $in2$. For the ease of later discussion, let us denote the voltage potential of the shared net among $in1$, $in2$, and R'_G as V_3. 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'_G of an X-IMPLY family gate needs to be constant, too. However, as mentioned in Section 2, only within the voltage range $[V_{on}, V_{off}]$, the resistance of memristor devices persists.

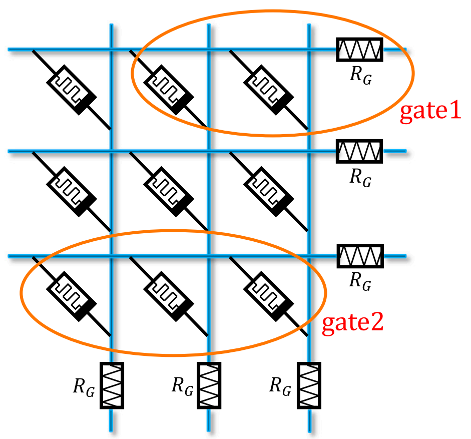


Fig. 4. An IMPLY family gate (gate1) and a MAGIC family gate (gate2) in a same memristor crossbar.

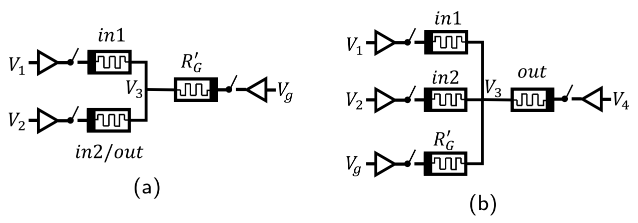


Fig. 5. (a) An memristor-based X-IMPLY family gate; (b) Relaxing the design constrains of memristor-based MAGIC family gates.

Table 2
Constraints for X-IMPLY IMP gate.

Inputs and output	Constraints
$in1$=0, $in2$=0, out=1	$\frac{V_1-V_3}{R_{off}}+\frac{V_2-V_3}{R_{off}}+\frac{V_g-V_3}{R'_G}=0$
	$V_1-V_3 \geq V_{on}$
	$V_2-V_3 \leq V_{Set}$
	$V_{on} \leq V_g-V_3 \leq V_{off}$
$in1$=0, $in2$=1, out=1	$\frac{V_1-V_3}{R_{off}}+\frac{V_2-V_3}{R_{on}}+\frac{V_g-V_3}{R'_G}=0$
	$V_1-V_3 \geq V_{on}$
	$V_2-V_3 \leq V_{off}$
	$V_{on} \leq V_g-V_3 \leq V_{off}$
$in1$=1, $in2$=0, out=0	$\frac{V_1-V_3}{R_{on}}+\frac{V_2-V_3}{R_{off}}+\frac{V_g-V_3}{R'_G}=0$
	$V_1-V_3 \leq V_{off}$
	$V_2-V_3 \geq V_{on}$
	$V_{on} \leq V_g-V_3 \leq V_{off}$
$in1$=1, $in2$=1, out=1	$\frac{V_1-V_3}{R_{on}}+\frac{V_2-V_3}{R_{on}}+\frac{V_g-V_3}{R'_G}=0$
	$V_1-V_3 \leq V_{off}$
	$V_2-V_3 \leq V_{off}$
	$V_{on} \leq V_g-V_3 \leq V_{off}$

Table 3
The memristor model parameters.

R_{on}	1 kΩ
R_{off}	300 kΩ
V_{on}	−2.5 V
V_{off}	1.9 V
V_{Set}	−3 V
V_{Reset}	2.4 V

Thus, the voltage across R'_G throughout the execution of the gate must be within the range $[V_{on}, V_{off}]$.

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 $\frac{V_1-V_3}{R_{in1}}+\frac{V_2-V_3}{R_{in2}}+\frac{V_g-V_3}{R'_G}=0$, 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 $V_1-V_3 \geq V_{on}$; and when $in1$ is 1, the voltage across it satisfies $V_1-V_3 \leq V_{off}$. 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 $V_2-V_3 \leq V_{Set}$. 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 is between R_{on} and R_{off} and does not change for all input combinations, thus the voltage across R'_G satisfies $V_{on} \leq V_g-V_3 \leq V_{off}$.

Similarly, we examine all the other IMPLY family gates to transform them into X-IMPLY 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 $\frac{|V_{on}|}{|V_{off}|}<1$.

Note that the location of the memristor cell R'_G is not fixed. If an X-IMPLY family gate is to be implemented in a crossbar row/column, the R'_G 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 polars 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 functioning as a load resistor is added, which is different from [14]. The memristor R'_G is connected to voltage source V_g. Similarly, the voltage potential of the shared net among $in1$, $in2$, R'_G, and out is also denoted as V_3.

Let us take the MAGIC IMP gate as example. Table 4 lists the constrains for all the input combinations of MAGIC IMP gate. The memristor 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 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 to be tuned into related resistance ranges.

NOT gate is required by many synthesis and mapping tools [11,13,19,20], 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. Synthesizing 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 4
Constraints for MAGIC IMP gate.

Inputs and output	Constraints
$in1=0$, $in2=0$, $out=1$	$\frac{V_1-V_3}{R_{off}}+\frac{V_2-V_3}{R_{off}}+\frac{V_g-V_3}{R'_G}+\frac{V_4-V_3}{R_{on}}=0$
	$V_1-V_3\geq V_{on}$
	$V_2-V_3\geq V_{on}$
	$V_{on}\leq V_g-V_3\leq V_{off}$
	$V_4-V_3\leq V_{off}$
$in1=0$, $in2=1$, $out=1$	$\frac{V_1-V_3}{R_{off}}+\frac{V_2-V_3}{R_{on}}+\frac{V_g-V_3}{R'_G}+\frac{V_4-V_3}{R_{on}}=0$
	$V_1-V_3\geq V_{on}$
	$V_2-V_3\leq V_{off}$
	$V_{on}\leq V_g-V_3\leq V_{off}$
	$V_4-V_3\leq V_{off}$
$in1=1$, $in2=0$, $out=0$	$\frac{V_1-V_3}{R_{on}}+\frac{V_2-V_3}{R_{off}}+\frac{V_g-V_3}{R'_G}+\frac{V_4-V_3}{R_{on}}=0$
	$V_1-V_3\leq V_{off}$
	$V_2-V_3\geq V_{on}$
	$V_{on}\leq V_g-V_3\leq V_{off}$
	$V_4-V_3\geq V_{Reset}$
$in1=1$, $in2=1$, $out=1$	$\frac{V_1-V_3}{R_{on}}+\frac{V_2-V_3}{R_{on}}+\frac{V_g-V_3}{R'_G}+\frac{V_4-V_3}{R_{on}}=0$
	$V_1-V_3\leq V_{off}$
	$V_2-V_3\leq V_{off}$
	$V_{on}\leq V_g-V_3\leq V_{off}$
	$V_4-V_3\leq V_{off}$

Table 5
Compatible X-IMPLY family gates and MAGIC family gates.

	V_1	V_2	V_4	V_g	R'_G
X-IMPLY IMP	−1.0 V	−2.5 V	–	1.0 V	$R_{on}\sim 1.11$ kΩ
X-IMPLY OR	−1.0 V	−2.5 V	–	floating	–
X-IMPLY NIMP	−2.5 V	1.7 V	–	−2.0 V	$R_{on}\sim 1.18$ kΩ
MAGIC IMP	−2.0 V	1.7 V	1.7 V	−2.0 V	$R_{on}\sim 1.04$ kΩ
MAGIC OR	2.5 V	2.5 V	−1.0 V	floating	–
MAGIC NIMP	0 V	2.5 V	−1.0 V	floating	–

is chosen because the combination of NOR gate and NOT gate is a 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 $5xp1$ 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 least memristor cells and latency. From the results, we could see the cell library {NOR, NOT} is not optimal for many logic functions. For example, for benchmark $clip$ the optimal cell library is {IMP, NOT} and for benchmark $cm163a$ the optimal cell library is {NIMP, NOT}. In this work, we also use ABC synthesize tool to synthesize the given HDL netlist. Differently, we synthesize L times and each time with a different cell library, as shown in Fig. 1. To increase the synthesis flexibility, we extend the three least-size cell libraries {IMP, NOT}, {NIMP, NOT}, {OR, NOT} by adding one or more supported gates. All the cell libraries are {IMP, NOT}, {NIMP, NOT}, {OR, NOT}, {IMP, NIMP, NOT}, {IMP, OR, NOT}, {NIMP, OR, NOT}, {IMP, NIMP, OR, NOT}. Thus, L is 7 in this paper.

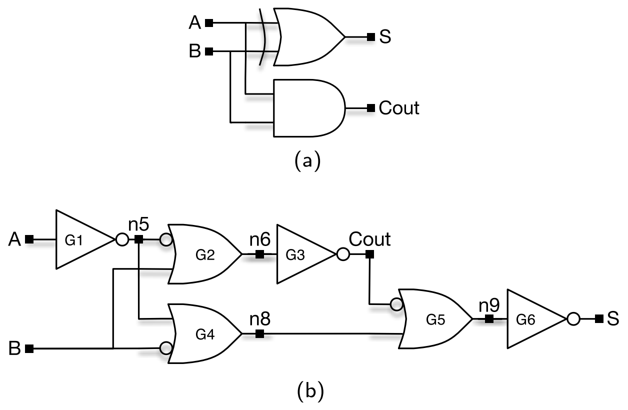


Fig. 6. (a) HDL netlist of a half adder; (b) The gate-level netlist of the half adder consists of only IMP gates and NOT gates.

4.2. Mapping the generated gate-level netlist

We follow the idea of MAGIC SIMPLER that maps the generated gate-level netlist to a single crossbar row/column because it outperforms the other mapping methods in terms of throughput and parallelism [13]. As mentioned in Section 2, IMPLY family gates save cell usage and time while MAGIC family gates could avoid the problem of input overwriting. 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. For IMP gate $G2$, the in2 input $n5$ is shared by another gate $G4$. If gate $G4$ is executed before $G2$, $G2$ 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 6
Result comparison of synthesizing LGsynth91 benchmarks with different cell libraries.

	NOR + NOT		IMP + NOT		OR + NOT		NIMP + NOT	
	# of NOR	# of NOT	# of IMP	# of NOT	# of OR	# of NOT	# of NIMP	# of NOT
5xp1	80	32	80	30	84	72	84	27
clip	95	49	95	27	95	81	95	33
cm150a	46	16	46	8	46	56	46	17
cm162a	33	27	33	19	33	23	33	16
cm163a	32	29	32	18	32	22	32	15
misex1	48	30	48	12	42	36	42	18
parity	45	31	45	21	45	61	45	22
x2	37	31	37	14	37	28	37	19

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 net 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.

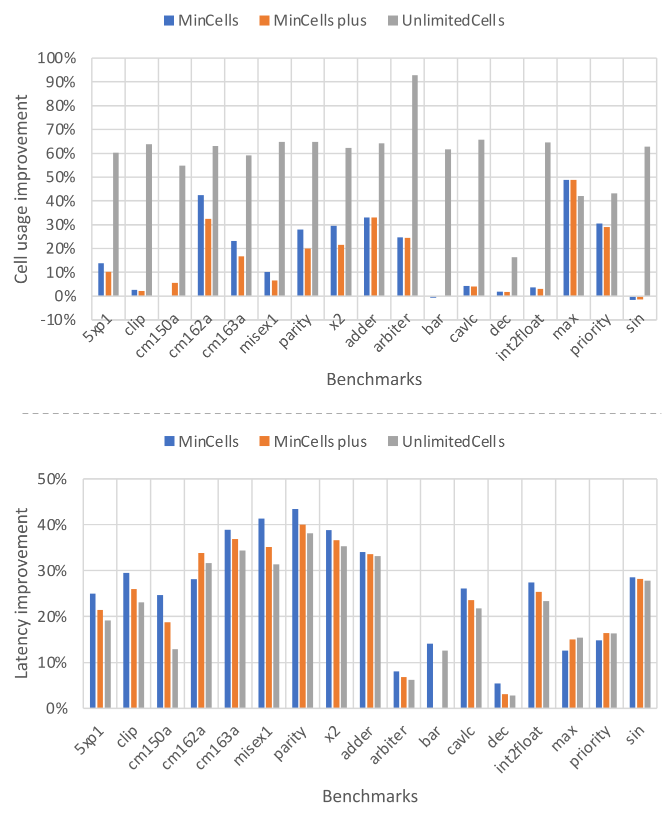


Fig. 7. Improvement of our method compared with MAGIC SIMPLER under different size constrains of crossbar row/column in terms of ***Up:*** cell usage and ***Down:*** latency for mapping the LGsynth91 and EPFL benchmark suite.

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 row size constrains 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 constrains 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 two writing operations because it requires one initialization operation except for the gate operation. Every X-IMPLY 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 increase 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.

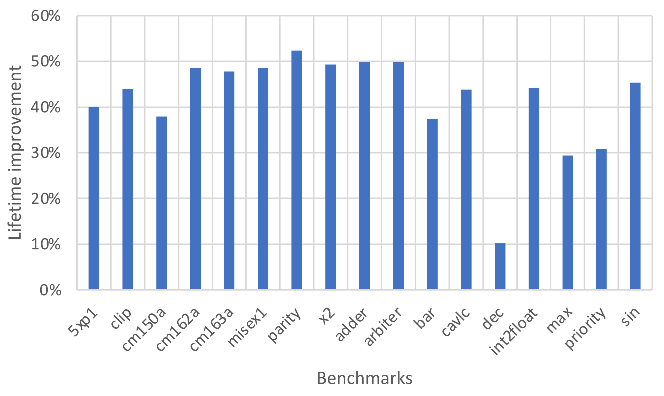


Fig. 8. Improvement of our method compared with MAGIC SIMPLER in terms of lifetime of memristor-based computing system for LGsynth91 and EPFL benchmarks.

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

[1] M. Horowitz, Computing's energy problem (and what we can do about it), in: 2014 IEEE Int. Solid-State Circuits Conf. Dig. Tech. Pap., IEEE, 2014, pp. 10–14, http://dx.doi.org/10.1109/ISSCC.2014.6757323, http://ieeexplore.ieee.org/document/6757323/.

[2] L.-C. Hsu, C.-T. Chiu, K.-T. Lin, H.-H. Chou, Y.-Y. Pu, ESSA: An energy-Aware bit-Serial streaming deep convolutional neural network accelerator, J. Syst. Architect. 111 (2020) 101831, http://dx.doi.org/10.1016/j.sysarc.2020.101831.

[3] R.C.C. Zhe Xu, Binary convolutional neural network acceleration framework for rapid system prototyping, J. Syst. Architect. 109 (2020) 101762, http://dx.doi.org/10.1016/j.sysarc.2020.101762.

[4] S. Mittal, A survey on modeling and improving reliability of dnn algorithms and accelerators, J. Syst. Archit. 104 (2020) 101689, http://dx.doi.org/10.1016/j.sysarc.2019.101689.

[5] O. Mutlu, S. Ghose, J. Gómez-Luna, R. Ausavarungnirun, Processing data where it makes sense: Enabling in-memory computation, Microprocess. Microsyst. 67 (2019) 28–41, http://dx.doi.org/10.1016/j.micpro.2019.01.009, https://linkinghub.elsevier.com/retrieve/pii/S0141933118302291.

[6] Z. Zhu, H. Sun, Y. Lin, G. Dai, L. Xia, S. Han, Y. Wang, H. Yang, A configurable multi-precision CNN computing framework based on single bit RRAM, in: Proc. 56th Annu. Des. Autom. Conf. 2019 - DAC '19, ACM Press, New York, New York, USA, 2019, pp. 1–6, http://dx.doi.org/10.1145/3316781.3317739, http://dl.acm.org/citation.cfm?doid=3316781.3317739.

[7] Z. Zhu, H. Sun, K. Qiu, L. Xia, G. Krishnan, G. Dai, D. Niu, X. Chen, X.S. Hu, Y. Cao, Y. Xie, Y. Wang, H. Yang, MNSIM 2.0: A behavior-level modeling tool for memristor-based neuromorphic computing systems, in: Proc. 2020 Gt. Lakes Symp. VLSI, ACM, New York, NY, USA, 2020, http://dx.doi.org/10.1145/3386263.3407647.

[8] Y. Cai, Y. Lin, L. Xia, X. Chen, S. Han, Y. Wang, H. Yang, Long live TIME: Improving lifetime and security for NVM-based training-in-memory systems, IEEE Trans. Comput. Des. Integr. Circuits Syst. (2020) http://dx.doi.org/10.1109/TCAD.2020.2977079.

[9] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J.J. Yang, H. Qian, Fully hardware-implemented memristor convolutional neural network, Nature 577 (7792) (2020) 641–646, http://dx.doi.org/10.1038/s41586-020-1942-4, http://www.nature.com/articles/s41586-020-1942-4.

[10] J. Borghetti, G.S. Snider, P.J. Kuekes, J.J. Yang, D.R. Stewart, R.S. Williams, Memristive switches enable stateful logic operations via material implication, Nature 464 (7290) (2010) 873–876, http://dx.doi.org/10.1038/nature08940.

[11] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E.G. Friedman, A. Kolodny, U.C. Weiser, MAGIC-memristor-aided logic, IEEE Trans. Circuits Syst. II Express Briefs 61 (11) (2014) http://dx.doi.org/10.1109/TCSII.2014.2357292.

[12] Z. Zhu, M. Ma, J. Liu, L. Xu, X. Chen, Y. Yang, Y. Wang, H. Yang, A general logic synthesis framework for memristor-based logic design, in: 2019 IEEE/ACM Int. Conf. Comput. Des., IEEE, 2019, http://dx.doi.org/10.1109/ICCAD45719.2019.8942111.

[13] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, S. Kvatinsky, SIMPLER MAGIC: Synthesis and mapping of in-memory logic executed in a single row to improve throughput, IEEE Trans. Comput. Des. Integr. Circuits Syst. 39 (10) (2020) http://dx.doi.org/10.1109/TCAD.2019.2931188.

[14] K.M. Kim, R.S. Williams, A family of stateful memristor gates for complete cascading logic, IEEE Trans. Circuits Syst. I Regul. Pap. 66 (11) (2019) 4348–4355, http://dx.doi.org/10.1109/TCSI.2019.2926811, https://ieeexplore.ieee.org/document/8776649/.

[15] J. Bùrger, C. Teuscher, M. Perkowski, Digital logic synthesis for memristors, Reed-Muller 2013 (2013) 31–40.

[16] S. Gupta, M. Imani, T. Rosing, FELIX: Fast and energy-efficient logic in memory, in: Proc. Int. Conf. Comput. Des., ACM, New York, NY, USA, 2018, pp. 1–7, http://dx.doi.org/10.1145/3240765.3240811, https://dl.acm.org/doi/10.1145/3240765.3240811.

[17] B. Hoffer, V. Rana, S. Menzel, R. Waser, S. Kvatinsky, Experimental demonstration of memristor-aided logic (MAGIC) using valence change memory (VCM), IEEE Trans. Electron Devices 67 (8) (2020) 3115–3122, http://dx.doi.org/10.1109/TED.2020.3001247, https://ieeexplore.ieee.org/document/9125983/.

[18] R. Ben Hur, N. Wald, N. Talati, S. Kvatinsky, Simple magic: Synthesis and in-memory mapping of logic execution for memristor-aided logic, in: 2017 IEEE/ACM Int. Conf. Comput. Des., IEEE, 2017, pp. 225–232, http://dx.doi.org/10.1109/ICCAD.2017.8203782, http://ieeexplore.ieee.org/document/8203782/.

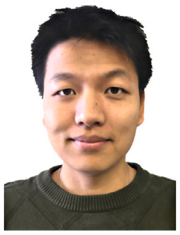
[19] V. Tenace, R.G. Rizzo, D. Bhattacharjee, A. Chattopadhyay, A. Calimera, SAID: A supergate-aided logic synthesis flow for memristive crossbars, in: Proc. 2019 Des. Autom. Test Eur. Conf. Exhib. DATE 2019, 2019, pp. 372–377, http://dx.doi.org/10.23919/DATE.2019.8714939.

[20] P.L. Thangkhiew, K. Datta, Scalable in-memory mapping of boolean functions in memristive crossbar array using simulated annealing, J. Syst. Architect. 89 (2018) 49–59, http://dx.doi.org/10.1016/j.sysarc.2018.07.002, https://linkinghub.elsevier.com/retrieve/pii/S1383762118301012.

[21] A. Mishchenko, (Berkeley Logic Synthesis and VeriïňĄcation Group), ABC: A system for sequential synthesis and VeriïňĄcation, 2012, https://people.eecs.berkeley.edu/{~}alanmi/abc/.

[22] S. Yang, Logic synthesis and optimization benchmarks user guide version 3.0, in: MCNC, 1991.

[23] L. Amarù, P.-E. Gaillardon, G. De Micheli, The EPFL Combinational Benchmark Suite, in: Proc. 24th Int. Work. Log. Synth., 2015.



Minhui Zou received the B.S. degree and Ph.D. degree in computer science and technology from Chongqing University, China, in 2013 and 2018, respectively. He is currently a lecturer with the School of Computer Science and Engineering, Nanjing University of Science and Technology, China. His current research interests include hardware security, edge computing, and memory computing.

Junlong Zhou received the Ph.D. degree in Computer Science and Technology from East China Normal University, Shanghai, China, in 2017. He was a Visiting Scholar with the University of Notre Dame, Notre Dame, IN, USA, during 2014–2015. He is currently an Associate Professor with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China. His research interests include embedded systems, cloud-edge-IoT, and cyber–physical systems, where he has published 2 book chapters and more than 80 refereed papers, including 25+ IEEE/ACM Transactions. He has been an Associate Editor for the Journal of Circuits, Systems, and Computers and the IET Cyber–Physical Systems: Theory & Applications, a Subject Area Editor for the Journal of Systems Architecture, and a Guest Editor for 6 ACM/IET/Elsevier/Wiley Journals such as ACM Transactions on Cyber–Physical Systems.

Jin Sun received the B.S. and M.S. degrees in computer science and technology from Nanjing University of Science and Technology, Nanjing, China, in 2004 and 2006, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Arizona, Tucson, AZ, USA, in 2011. From January 2012 to September 2014, he was with Orora Design Technologies, Inc., as a Member of Technical Staff. He is currently an Associate Professor with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China. His research interests include high-performance computing and electronic design automation.

Chengliang Wang received his B.S. degree in mechatronics in 1996, the M.S. degree in precision instruments and machinery in 1999, and the Ph.D. degree in control theory and engineering in 2004, all from Chongqing University, China. He is now a professor of Chongqing University. His research interests include smart control for complex system, the theory and application of artificial intelligence, wireless network and RFID research. He has published more than 80 writings, including more than 20 articles in SCI, EI, and holds 12 national patents.

Shahar Kvatinsky is an Associate Professor at the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion — Israel Institute of Technology. Shahar received a B.Sc. degree in Computer Engineering and Applied Physics and an MBA degree in 2009 and 2010, respectively, both from the Hebrew University of Jerusalem and the Ph.D. degree in Electrical Engineering from the Technion — Israel Institute of Technology in 2014. From 2006 to 2009, he worked as a circuit designer at Intel. From 2014 and 2015, he was a post-doctoral research fellow at Stanford University. Kvatinsky is an editor of Microelectronics Journal and has been the recipient of numerous awards: 2020 MDPI Electronics Young Investigator Award, 2019 Wolf Foundation's Krill Prize for Excellence in Scientific Research, 2015 IEEE Guillemin-Cauer Best Paper Award, 2015 Best Paper of Computer Architecture Letters, Viterbi Fellowship, Jacobs Fellowship, ERC starting grant, the 2017 Pazy Memorial Award, the 2014 and 2017 Hershel Rich Technion Innovation Awards, 2013 Sanford Kaplan Prize for Creative Management in High Tech, 2010 Benin prize, and seven Technion excellence teaching awards. His current research is focused on circuits and architectures with emerging memory technologies and design of energy-efficient architectures.