RIVER: Sneak Path Aware READ-based In-Memory Computing for 1T1M Memristive Crossbars

Till Schnittka[ψ], Chandan Kumar Jha[ψ], Sallar Ahmadi-Pour[ψ], Rolf Drechsler[ψ,†]
University of Bremen, Bremen, Germany[ψ]
DFKI GmbH, Bremen, Germany[†]
schnitti@uni-bremen.de, chajha@uni-bremen.de, sallar@uni-bremen.de, drechsler@uni-bremen.de

***Abstract*—In-memory Computing (IMC)** **using emerging devices has shown immense potential. Among these devices, memristors have emerged as one of the most popular for performing digital IMC. While several methods exist for digital IMC using memristors, most require expensive write operations in terms of energy, latency, and endurance. Hence, READ-based IMC techniques have been proposed to reduce the number of writes to the memristor crossbar. However, existing techniques rely on simple gates that can be mapped to the memristive crossbar, making them non-optimal, and they suffer from unwanted sneak paths causing undesired behavior. In this work, we alleviate these limitations and propose an optimized synthesis methodology for 1T1M crossbars called RIVER. RIVER supports more complex gates and is sneak-path aware. When comparing RIVER with the state-of-the-art using ISCAS' 85 and EPFL benchmarks, we achieve 33% less gate utilization on average while reducing the average staircase length by 37%. Moreover, these enhancements result in a 58% reduction in the required crossbar area. After eliminating sneak paths, RIVER still shows 38% less area usage on average as compared to the state-of-the-art.**

I. INTRODUCTION

IMC has gained immense popularity in recent years owing to the large benefits it provides in power and performance [1]. IMC using emerging devices, in particular memristors, have seen a lot of focus as they are capable of both analog and digital computations [2], [3]. Memristors are two terminal devices that can be configured to be in a *High Resistance State (HRS)* or a *Low Resistance State (LRS)*, depending upon the magnitude and the direction of the applied voltage [4]. The LRS and HRS are used as logic '1' and logic '0' in digital computations, respectively. In this work, we focus on digital computation, i.e., implementing IMC using memristors. Various approaches for implementing IMC using memristors have been developed over the years. Methods like IMPLY [5], MAGIC [6], FELIX [7], Majority Logic [8], PATH [9], etc., are effective in performing IMC using memristors. These *Logic-in-Memory (LiM)* techniques require the memristors to be written while evaluating the Boolean function. However, writing operations for the memristors require a lot of energy, as the state of the memristor needs to be changed. The write operation is also a slow process, thus the latency also increases significantly [10]. Repeated writing to the memristors also reduces their lifetime, as they have a limited endurance [11].

Recently, there has been a method proposed for performing IMC using only READ operations [12], called STREAM. However, this methodology requires a staircase of memristors, i.e., the memristor crossbars need to be connected in series. The memristors are written only once at the start, when the mapping of the Boolean function is done to the memristor crossbar. The inputs are then applied as voltage signals to the crossbars to perform the Boolean operation. Since there are no write operations to the memristors during the Boolean function evaluation, it results in significant energy savings as well as improved latency. While this work showed that we can perform digital IMC using only READ operations, the work was limited to using OR and NOR gates, which led to suboptimal designs. Additionally, they do not consider the sneak paths that can cause undesired behavior in the memristor crossbar and lead to incorrect outputs [13], [14].

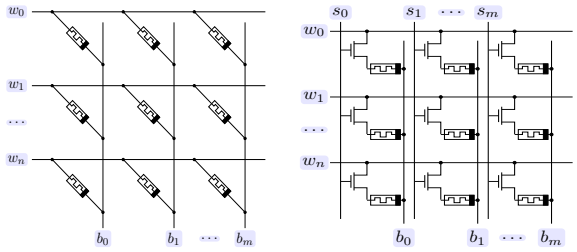


Fig. 1: Layout of a a) Passive Crossbar, b) 1T1M Crossbar

In this work, we alleviate the limitations of the prior work and propose an efficient READ-based IMC methodology called RIVER. Since STREAM is based on passive crossbars, it was only capable of implementing OR-NOR operations. In RIVER, we exploit the architecture of the 1T1M crossbar to support AND-OR gates in addition to one supported by STREAM. We show for the first time that AND-OR operations like $F = i_0 \wedge (i_1 \vee i_2 \vee \ldots)$ can also be effectively mapped to the 1T1M crossbar staircase. Other methods that require writes to the memristors have previously shown AND-OR operations. However, we show this for the first time for the READ-based IMC using flow-based computing. Other works have shown AND implementation but they are not based on flow-based computing [15]. Implementing AND-OR gates is crucial as it leads to a more optimized synthesis of the Boolean function. We also show that STREAM does not consider sneak paths while mapping even though they exist and can cause incorrect outputs. In the RIVER methodology, we also show a technique of mitigating the sneak paths. Following are the contributions of our work. First, we propose a novel methodology called RIVER for performing READ-based IMC using 1T1M crossbar staircases. Second, we show for the first time that AND-OR operations can be effectively mapped to the 1T1M crossbar in the RIVER methodology. Third, we propose an effective synthesis and mapping algorithm for the RIVER methodology that eliminates the sneak paths in READ-based IMC. Last, we evaluated the RIVER methodology on the ISCAS' 85 and EPFL

benchmarks and achieved 33% less gate utilization and 37% reduction in the memristor crossbar staircase length.

The rest of the paper is organized as follows: In Section II, we discuss the necessary background. In Section III, we discuss the related works and their limitations. In Section IV, we discuss the proposed RIVER methodology. In Section V, we discuss the results, and conclude the paper in Section VI.

II. PRELIMINARIES

A. Memristor Crossbar Architectures

The memristor crossbars can be either passive or active, as shown in Fig. 1 [16]. The passive crossbar is a crossbar of wires, where each word line $W = \{w_0, \ldots, w_n\}$ and bit line $B = \{b_0, \ldots, b_m\}$ is connected via a memristor $M = \{m_{n,m}\}$ (n = row and m = column) [17]. For the current to flow between a word line w_0 and a bit line b_0, the respective memristor $m_{0,0}$ must be in the LRS. However, in some cases, undesired sneak paths can occur that can connect the bit line to the word line. For our example, the w_0 and a bit line b_0 can be connected if the following memristors $m_{0,1}, m_{1,1}, m_{1,0}$ are in LRS. Active crossbars, specifically the 1T1M crossbars, contain additional selector lines $S = \{s_0, \ldots, s_n\}$ parallel to the bit lines, and a transistor in series with the memristor [18] [19]. In contrast to a passive crossbar, in a 1T1M crossbar, a word line w_i and bit line b_i are connected via the function $s_j \wedge m_{i,j}$, i.e, the memristor needs to be in LRS and the selector corresponding to the memristor needs to be ON.

B. Staircases

In the context of memristor crossbars, a staircase describes a series of memristor crossbars in levels $L = \{l_0, l_1, \ldots l_k\}$, where some wires of each staircase level $l_{i \geq 1}$ are connected to the wires of the previous staircase level l_{i-1} [12]. This is done to enable the synthesis of larger Boolean functions across multiple crossbars. Depending upon the Boolean functions, the connections are decided for the crossbars to generate the staircases. Fig. 2 shows an abstract representation of a crossbar staircase. Each staircase level is represented by a grid containing white or black dots, where the columns and rows of the grid represent the word lines and bit lines of the crossbar, respectively. White dots denote memristors in HRS, and black dots denote the memristors in LRS. Interconnects between staircase levels are represented by arrows originating from the bit line of the previous staircase. A black dot at the beginning of an arrow indicates a negation, while an arrow going into the top of a crossbar (like in the second level in Fig. 5) shows a connection to the selector line.

C. READ-based Computing

READ-based computing is an emerging direction in the area of IMC that uses flow-based computing. Most design styles require write operations to be performed on the memristors while implementing the Boolean function. This can happen as some values in the memristor crossbars are rewritten during the function evaluation or the inputs of the computation are encoded as memristor states. However, write operations to the memristor crossbar are expensive in terms of energy, latency, and endurance. READ-based computing mitigates this by keeping

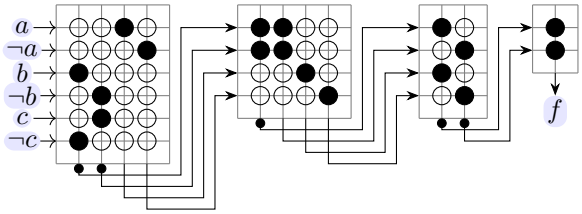


Fig. 2: Mapping of $a \oplus b \oplus c$ using STREAM-O

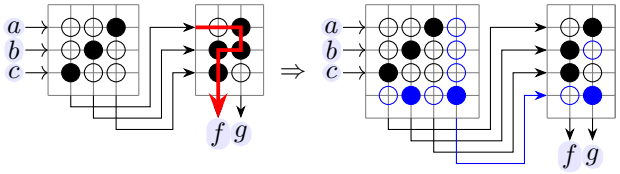


Fig. 3: Example of Sneak Path Elimination

the state of the memristors in the crossbar the same throughout the computations, and inputs are fed to the crossbar using only bit lines, word lines, and selector lines. Writing to memristors consumes a lot of energy and takes time. On the other hand, reading from a memristor consumes almost no energy at all. Hence, READ-based IMC drastically improves the energy consumption and latency of the operations [12].

III. RELATED WORK AND LIMITATIONS

Flow-based computing has received significant attention in recent years [12], [20]–[22]. Some of the state-of-the-art techniques that are based on flow-based computing are called COMPACT [22] and Stream [12]. COMPACT [22] encodes the input values and logic within the states of memristors in a passive memristor crossbar. A low resistance path between predefined points on the crossbar then indicates the result of the function. However, since the input of the function is encoded in the state of the memristor, COMPACT requires writing to a set of memristors for each operation, which consumes a large amount of energy. Additionally, COMPACT only uses a single large crossbar, which is impractical to implement [23].

STREAM [12] uses staircases of passive Memristor crossbars to implement flow-based computing based on OR-plane logic. There are two techniques proposed in STREAM, namely STREAM-O and STREAM-M. More specifically, STREAM-O implements n-input OR and NOR gates by applying the inputs of these gates to the word lines of a memristor crossbar. Memristors are then used to connect the word lines of the gate's input to the bit line of the gate's output, and the interconnections between the crossbars are used to implement the negation. An example of such a mapping can be seen in Fig. 2, which maps the function $a \oplus b \oplus c$ into four staircase levels. It can be seen from Fig. 2, that the first crossbar maps six inputs to the rows $(a, \neg a, b, \neg b, c, \neg c)$ to four outputs to the column, two of which are negated, as indicated by the black dot on the bottom of the crossbar. The four outputs of the first staircase map the functions $\neg(b \vee \neg c)$, $\neg(\neg b \vee c)$, a and $\neg a$. STREAM-M [12] also uses passive memristor crossbars, although instead of a tree-covering algorithm, the logic tool SIS is used to map logic functions into a hierarchy of *Sum-Of-Products (SOP)* terms [24]. These are then

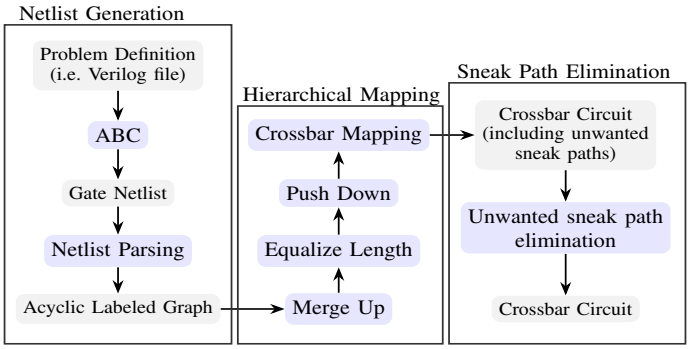


Fig. 4: Overview of the RIVER Methodology

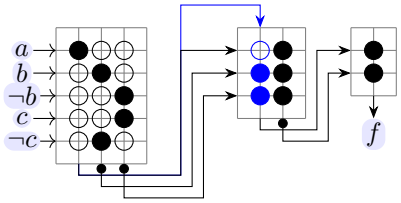


Fig. 5: Mapping of $a \oplus b \oplus c$ using RIVER

converted into OR-gates using De Morgan's theorem. However, STREAM has the following severe limitations.

A. Limitation 1: Sneak Path

The mapping algorithm in STREAM does not take into account the undesired sneak paths [13], [25]. Hence, sneak paths are an issue within STREAM when mapping to passive memristor crossbars. Fig. 3 demonstrates this issue: The crossbar mapping is intended for $f = b \vee c, g = a \vee b$, but the highlighted sneak path allows current to flow through f even if only a is true. The same sneak paths also exist for g, distorting the mapped functions to $f = g = a \vee b \vee c$. We will discuss how our proposed methodology handles sneak paths in detail in Section IV-C using additional rows. An example to mitigate the sneak path is shown in Fig. 3.

B. Limitation 2: Non Optimality

STREAM uses a passive crossbar and is limited to OR-Plane logic. With 1T1M crossbars, we show that the AND-OR gate can be effectively implemented. We show that the selector line can be used to implement the logic function $f = a \wedge (b_0 \vee b_1 \vee b_2 \vee \ldots)$. This gate allows for a significantly smaller representation of some logic functions. An example of this is the function $a \oplus b \oplus c$. Fig. 2 shows a mapping of this function with OR and NOR gates, which STREAM-O uses. Fig. 5 shows a mapping made possible through the use of one AND-OR gate in the second crossbar. The AND-OR operation is represented by the arrow going into the top of the second crossbar, which uses the layout of the 1T1M crossbar to model the function $i_0 \wedge (i_1 \vee i_2)$, which, in this case, is $a \wedge ((\neg(b \vee \neg c)) \vee (\neg(\neg b \vee c)))$ using only a single crossbar column. This decreases both the length of the staircase and the size of the crossbar at each staircase level.

Hence, we see that the RIVER methodology effectively handles both limitations, the undesired sneak paths, and gives a more optimum mapping as compared to STREAM.

IV. RIVER METHODOLOGY

In this work, we propose RIVER, an efficient methodology based on READ-based computing on 1T1M that has the capability to a) eliminate undesired sneak paths and b) exploit the 1T1M crossbar to enable support for complex gates. These complex gates, allow for better synthesis of the Boolean function and enable a more optimal mapping as compared to STREAM. The overall RIVER methodology is shown in Fig. 4. The RIVER methodology consists of three steps. 1) We use the ABC tool to map the input to a set of logic gates. These logic gates are chosen such that they can be efficiently mapped to a 1T1M crossbar (Section IV-A). We then generate an acyclic-labeled graph encoding the behavior of these gates. 2) We apply a series of graph transformations that gives an effective mapping to a crossbar staircase. 3) We detect the sneak paths and eliminate them to obtain the final mapping (Section IV-C).

A. Netlist Generation with ABC

First, the input functions need to be optimized and mapped to a set of logic gates that can be represented using a 1T1M crossbar. We achieve this using the ABC tool.

Unlike STREAM-O which can only support OR- and NOR-gates, we allow the mapping using additional gates. This is enabled by the use of 1T1M crossbars. In RIVER methodology, the gates that we have used for the mapping are as follows:

- OR ($i_0 \vee i_1 \vee \ldots$),
- NOR ($\neg(i_0 \vee i_1 \vee \ldots)$),
- AND-OR ($i_0 \wedge (i_1 \vee i_2 \vee \ldots)$), and
- NOT-AND-OR ($\neg(i_0 \wedge (i_1 \vee i_2 \vee \ldots))$),

We limited the number of inputs of the OR-gate to 5 in both AND-OR-gates and NOT-AND-OR-gates. This was done as we observed that the ABC tool does not utilize larger gates when mapping. However, in the next section, we will show how to merge smaller gates to improve the mapping. Additionally, we also define two gates: A buffer gate, which has a single input and forwards it to the output, and an inverter gate, which inverts the input. To allow for easier notation later on, we will name the buffer and inverter gate OR_1 and NOR_1, respectively.

To convert the input function, we repeatedly apply `resyn`, `resyn2`, and `resyn2rs` followed by `balance`. The output gates are written into a Verilog file. We then parse the output gates to generate an acyclic-labeled graph. We label all the edges from the AND-OR and NOT-AND-OR gates with a, and all other edges with o, to represent whether they belong to an AND or an OR operation. Additionally, nodes are labeled with $O, \overline{O}, A, \overline{A}$ for the gates OR, NOR, AND-OR, and NOT-AND-OR, respectively. Fig. 6 shows an example of such a graph for the input function $f = (a \vee b) \wedge (c \oplus d)$. We will use this as an example to explain the steps of the RIVER methodology. Each node is described by a number n_i, which is shown at the top or bottom right side of a node in the tree. Each egde $e_{i,j}$ is defined by the nodes that it connects. We also introduce the expression f_i to describe the function of the node i in the tree. For example, node 7 in Fig. 6 describes the following function: $f_7 = \neg(f_1 \vee f_4)$

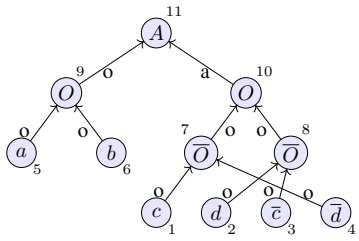


Fig. 6: The acyclic labelled graph during the mapping of $f = (a \vee b) \wedge (c \oplus d)$, derived from an ABC netlist

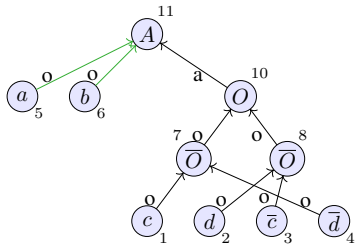


Fig. 7: The acyclic labelled graph during the mapping of $f = (a \vee b) \wedge (c \oplus d)$ after the *Merge-Up* operation

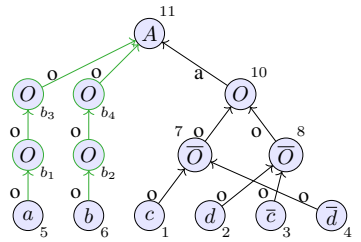


Fig. 8: The acyclic labelled graph during the mapping of $f = (a \vee b) \wedge (c \oplus d)$ after the *Equalize Length* operation

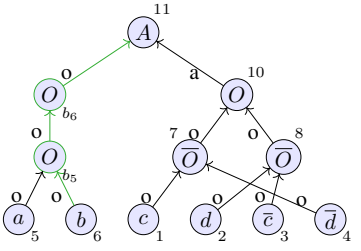


Fig. 9: The acyclic labelled graph during the mapping of $f = (a \vee b) \wedge (c \oplus d)$ after the *Push Down* operation

B. Hierarchical Mapping

The staircases are hardwired, and only the first crossbar in a staircase can receive the primary inputs. We want all outputs to be on the last crossbar of the staircase, therefore the path length for each node needs to be the same. Also, since ABC does not map gates with more than 5 inputs, we want to merge gates wherever possible for better mapping. Hence, we need to modify the graph from Section IV-A to allow it to be mapped to a crossbar staircase efficiently. For this, we apply a set of four graph transformation steps: *Merge Up*, *Equalize Length*, *Push Down*, and *Crossbar Mapping*. Here, it is important to respect the input mapping of each node, since not all the mapped operations are associative.

1) Merge Up: We want to reduce the length of the graph as much as possible, since longer staircases require more buffer nodes to carry results into later staircase levels. We want to remove gates without modifying the value of the function. This is done by evaluating operations as late as possible (by *pushing up* arguments). If there are two OR-operations in sequence, where the first is not negated, we can remove the first OR-node and attach its inputs to the second OR-operation. For such an operation, all paths that are affected must only be labeled with o, since any a-labelled graph would be part of an AND-OR-gate, which cannot receive more than one a-labelled input. An example of this transformation can be seen in Fig. 6. Here, we can see that there is one OR-operation in sequence, which is described by the edges $e_{5,9}$, $e_{6,9}$ and $e_{9,11}$. Note that all of these are marked with o. The *merge up* operation now removes n_9 and $e_{9,11}$, and attaches the edges from nodes 5 and 6 to node 11 directly. This changes f_{11} from $(f_9 \wedge f_{10})$ to $((f_5 \vee f_6) \wedge f_{10})$, which does not change the result, since $f_9 = (f_5 \vee f_6)$. The result of this operation can be seen in Fig. 7.

2) Equalize Length: Next, we insert additional nodes to make sure the path of each node to each of its roots is the same. This can be done by going through every node from the bottom up, counting the length of each of its incoming edges, and adding OR_1-nodes (buffer) to shorter edges. Here, the incoming edge label to the original node must be preserved. If we look at n_{11} in Fig. 7, we can see that the paths $n_5 \overset{e_{5,11}}{\rightarrow} n_{11}$ and $n_6 \overset{e_{6,11}}{\rightarrow} n_{11}$, which have a length of 2, are shorter than the path $n_1 \overset{e_{1,7}}{\rightarrow} n_7 \overset{e_{7,10}}{\rightarrow} n_10 \overset{e_{10,11}}{\rightarrow} n_{11}$, which has a length of 4. We apply the *equalize length* operation by inserting two buffer nodes (b_1 and b_3 as well as b_2 and b_4) to the end of both paths. The result of this is demonstrated in Fig. 8.

3) Push Down: Even though we minimized the number of gates in the *merge up* step, after we have equalized the path length, grouping these larger gates is not always optimal. If, for example, a gate has 3 OR-inputs, where two of these nodes have had buffer nodes inserted to equalize the path length, we can minimize the number of forwarded nodes by evaluating those two inputs with an OR_2 operation lower in the tree, and then forwarding only the result. We can observe such a situation in Fig. 8, where n_{11} has b_3 and b_4 as OR-inputs. We can apply *push down* by merging b_3 and b_4 into a new node b_6. We can then observe the same situation for the newly created b_6, b_1, and b_2, which can be merged into b_5. The result of this operation is shown in Fig. 9.

Additionally, although not the case in the example, the *Equalize Length* operation may introduce unnecessary parallel paths when forwarding an input to multiple different operations. We can merge such paths originating from the same node together, as long as none of the path nodes depend on more than one operation.

4) Crossbar Mapping: To map the resulting graph to the crossbar, we first associate each node of the graph with a staircase level. For this, we count the length to each node from any of the graph's roots. We can choose any path since they

all have equal length. This length is then associated with the staircase level (l_i) where, e.g., a node with path length 1 is associated with the staircase level l_1. For example, in Fig. 10 (a), we can see a reduced graph where nodes 1, 2, and 3 belong to a level l_{i-1} and nodes 4 and 5 belong to l_i.

To map level l_i, we first connect each word line w_i to the corresponding hardwired bit line b_i of the previous staircase l_{i-1}. An example of this can be seen in Fig. 10 (b). If a node is mapped to a bit line b_i in l_{i-1} and has an edge marked with a to a node in l_i, the corresponding selector line s_i is also connected in addition to the word line. The bit line corresponding to that selector line b_i is assigned to the node with the incoming edge. This is shown in Fig. 10 (c), where the a-edge from node 1 to 5 is mapped via an additional arrow, and node 5 is associated with the corresponding bit line b_0 of l_i. All other nodes on that level are assigned to any of the remaining bit lines. Each memristor is then programmed to the LRS, if an 'o'-labeled edge exists between the node assigned to the word line and the node assigned to the bit line, otherwise they are programmed to HRS. An example of this can be seen in Fig. 10 (d).

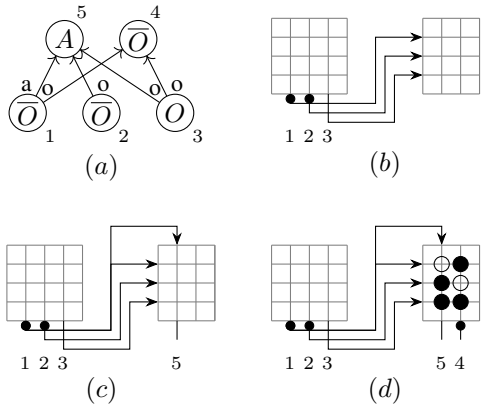


Fig. 10: Example of Staircase Mapping for $l_{i\geq 1}$. (a): Subgraph with the nodes relevant to mapping. (b): Crossbars after word-line assignment. (c): Crossbars after selector line assignment. (d): Crossbars after mapping is complete.

C. Unwanted Sneak Path Elimination

In the proposed hierarchical mapping in Section IV-B, we introduce a lot of sneak paths. As can be seen in Fig. 3, these sneak paths occur every time two outputs, that are not functionally equivalent, of the same crossbar share the same input variables. To eliminate sneak paths, additional inputs are added such that two outputs never share the same input variable. This requires creating a new output in the previous crossbar (if the crossbar is not the topmost crossbar of a staircase). An example of this can be seen in Fig. 3. Here, the sneak path marked in red is eliminated by adding a new input (marked with the blue arrow), which removes the shared input between f and g. To allow for this input, another output is added to the previous crossbar, which copies its information from the original input.

V. Experimental Evaluation

To evaluate the River methodology, we compare our approach with Stream-O [12] on two sets of benchmarks. As the source code for Stream-O is not yet publicly available, we have implemented the algorithms as described in [12]. Our implementation differs only slightly from the original in terms of the gate count. However, we believe that it closely replicates the original Stream-O implementation.

TABLE I: Comparison of River and Stream-O for number of gates and run time. (N)AO is the number of (Not-)And-Or gates, Δgates is the relative difference from River to Stream-O.

Benchmark			Stream-O [12]			River				
Name	Inputs	Outputs	Gates	Buffer	Time [s]	Gates	Buffer	(N)AO	Time [s]	ΔGates
c432	64	7	594	440	0.7	461	294	51	3.2	-22%
c499	82	32	1646	1204	2.1	1062	648	40	4.2	-35%
c880	117	26	999	646	1.3	631	261	112	3.9	-37%
c1355	82	32	1651	1209	2.1	1063	648	40	4.4	-36%
c1908	64	25	2059	1699	6.6	1046	665	81	4.5	-49%
c2670	303	139	1675	910	2.9	1252	487	201	5.2	-25%
c3540	99	22	2371	1590	12.5	1433	608	308	8.7	-40%
c5315	290	123	4359	2983	22.8	2576	1237	499	14.3	-41%
bar	270	128	3078	300	167.9	2273	24	1598	71.3	-26%
cavlc	20	11	1213	671	5.6	813	310	251	6.4	-33%
ctrl	13	25	162	73	0.4	127	27	40	3.4	-22%
i2c	252	141	2331	1309	11.1	1957	961	361	9.5	-16%
square	128	127	71363	55443	7046.4	39207	24386	3841	3198.2	-45%
int2float	22	7	445	265	0.9	312	138	72	4.0	-30%
adder	512	129	49417	48012	2890.5	32979	31454	253	1079.6	-33%
Average										-33%

TABLE II: Comparison of the staircase length between River and Stream-O. Δ is the relative difference from River to Stream-O.

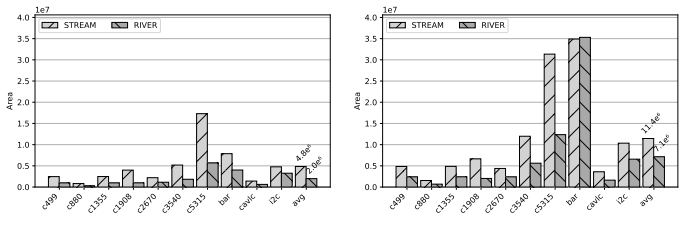
Benchmark	Stream-O [12]	River	Δlength
c432	15	13	-13%
c499	17	12	-29%
c880	17	11	-35%
c1355	17	12	-29%
c1908	26	15	-42%
c2670	17	11	-35%
c3540	35	20	-43%
c5315	27	16	-41%
bar	11	7	-36%
cavlc	17	11	-35%
ctrl	8	5	-38%
i2c	15	9	-40%
square	245	125	-49%
int2float	17	9	-47%
adder	256	130	-49%
Average			-37%

For a comparison between River and Stream-O, we used the ISCAS' 85 [26] and EPFL [27] benchmark suites. In this evaluation, we compare the results regarding the sizes and layouts of the resulting crossbar staircases, the mapping time and lastly, discuss the impact of our proposed sneak path mitigation technique. All experiments were carried out on an AMD FX-8350 with 8 cores @ 4 GHz with 32 GB of RAM running Ubuntu 22.04 LTS.

A. Results

At first, Tab. I shows the number of gates and run time between Stream-O and River. The table is separated into four parts. The left part shows each benchmark from the ISCAS' 85 suite (top half of the table) and the EPFL suite (bottom half of the table). For brevity, we choose to show subsets of these benchmark suits. Each benchmark is shown with its number of inputs and outputs. The two middle columns show the resulting gates, buffer nodes, and run time in seconds for Stream-O and River, respectively. There is an additional column in River for the number of utilized (Not-)And-Or-gates. The last column shows the relative difference between the number of gates obtained through Stream-O with respect to the number of gates from River. In all cases, we observe a reduction in the number of gates, with the smallest difference of 22 % for the c432 benchmark, and the biggest difference of 49 % for the

c1908 benchmark. On average, we observe a reduction in gates of 33 %.



(a) (b)

Fig. 11: Total crossbar area in STREAM-O and RIVER a) before sneak path elimination and b) after sneak path elimination

Next, Tab. II shows the staircase length for each benchmark. The first column shows the benchmark, the middle columns show the staircase length for STREAM-O and RIVER, and the last column shows the relative difference in staircase length between RIVER and STREAM-O, respectively. For every benchmark, we observe an improvement in staircase length, with the smallest difference being 13 % for the c432 benchmark and the biggest difference being 49 % for the adder benchmark. On average, the staircase lengths were improved by 37 %.

At last, we evaluated the impact of the sneak path elimination applied to the crossbar mappings for STREAM-O and RIVER. For this, we utilize the previously obtained results for the benchmark circuits and apply the same sneak path elimination technique to both, STREAM-O and RIVER, respectively. Since sneak path elimination changes both the width and height of crossbars, we used the crossbar area as the sum of all the widths multiplied by the sum of all the heights for all the staircase levels. Fig. 11a shows the total crossbar area before sneak path elimination. The x-axis shows each benchmark for STREAM-O and RIVER with light gray hatched bars and dark gray hatched bars, respectively. The last set of bars shows the average area for both methodologies. We observe that for each benchmark, RIVER has less area than STREAM-O. On average, RIVER performs 58 % better than STREAM-O prior to sneak path elimination. Fig. 11b shows the total crossbar area after sneak path elimination. When comparing the crossbar area before and after the elimination of sneak paths, we observe an average increase in the crossbar area for both STREAM-O and RIVER, respectively. However, even after the sneak path elimination, RIVER still performs 38 % better than STREAM-O.

VI. CONCLUSION AND FUTURE WORK

We have proposed a methodology for doing READ-based computation on the 1T1M crossbars called RIVER. RIVER exploits the 1T1M crossbars to show that complex gates can be effectively mapped, which leads to a more optimized design. In addition to this, in the RIVER methodology, we also show that method of eliminating the sneak paths. We were able to show that our mapping algorithm, which exploits the way complex gates map to 1T1M crossbars, is an improvement over the state-of-the-art mapping algorithm STREAM-O, with an improvement of 33% in area and 37% in delay on average. In the future, we will explore methods to better our proposed approach further by exploring the mapping of other complex gates and tailoring optimal synthesis for staircase representations.

ACKNOWLEDGEMENTS

This work was supported by the German Research Foundation (DFG) within the Project PLiM (DR 287/35-2).

REFERENCES

[1] N. Verma *et al.*, "In-memory computing: Advances and prospects," *IEEE Solid-State Circuits Magazine*, vol. 11, no. 3, pp. 43–55, 2019.
[2] A. Sebastian *et al.*, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.
[3] M. Le Gallo *et al.*, "Mixed-precision in-memory computing," *Nature Electronics*, vol. 1, no. 4, pp. 246–253, 2018.
[4] A. Bende *et al.*, "Experimental validation of memristor-aided logic using 1t1r tao x rram crossbar array," in *2024 37th International Conference on VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID)*. IEEE, 2024, pp. 565–570.
[5] S. Kvatinsky *et al.*, "Memristor-based material implication (imply) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, 2013.
[6] S. Kvatinsky *et al.*, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.
[7] S. Gupta *et al.*, "Felix: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–7.
[8] S. Shirinzadeh *et al.*, "Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 948–953.
[9] S. Thijssen *et al.*, "Path: Evaluation of boolean logic using path-based in-memory computing systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.
[10] S. Singh *et al.*, "Should we even optimize for execution energy? rethinking mapping for magic design style," *IEEE Embedded Systems Letters*, 2023.
[11] K. M. Kim *et al.*, "Voltage divider effect for the improvement of variability and endurance of taox memristor," *Scientific reports*, vol. 6, no. 1, p. 20085, 2016.
[12] M. R. H. Rashed *et al.*, "Stream: Towards read-based in-memory computing for streaming based data processing," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2022, pp. 690–695.
[13] M. A. Zidan *et al.*, "Memristor-based memory: The sneak paths problem and solutions," *Microelectronics journal*, vol. 44, no. 2, pp. 176–183, 2013.
[14] Y. Cassuto *et al.*, "Information-theoretic sneak-path mitigation in memristor crossbar arrays," *IEEE Transactions on Information Theory*, vol. 62, no. 9, pp. 4801–4813, 2016.
[15] L. Xie *et al.*, "Scouting logic: A novel memristor-based logic design for resistive computing," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2017, pp. 176–181.
[16] H. Kim *et al.*, "4k-memristor analog-grade passive crossbar circuit," *Nature communications*, vol. 12, no. 1, p. 5198, 2021.
[17] F. M. Bayat *et al.*, "Implementation of multilayer perceptron network with highly uniform passive memristive crossbar circuits," *Nature communications*, vol. 9, no. 1, p. 2331, 2018.
[18] M. Hu *et al.*, "Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication," in *Proceedings of the 53rd annual design automation conference*, 2016, pp. 1–6.
[19] E. J. Merced-Grafals *et al.*, "Repeatable, accurate, and high speed multi-level programming of memristor 1t1r arrays for power efficient analog computing applications," *Nanotechnology*, vol. 27, no. 36, p. 365202, 2016.
[20] D. Chakraborty *et al.*, "Input-aware flow-based computing on memristor crossbars with applications to edge detection," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 580–591, 2019.
[21] Z. Alamgir *et al.*, "Flow-based computing on nanoscale crossbars: Design and implementation of full adders," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2016, pp. 1870–1873.
[22] S. Thijssen *et al.*, "Compact: Flow-based computing on nanoscale crossbars with minimal semiperimeter and maximum dimension," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4600–4611, 2021.
[23] M. S. Truong *et al.*, "Racer: Bit-pipelined processing using resistive memory," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 100–116.
[24] E. M. S. K. J. Singh *et al.*, "Sis: A system for sequential circuit synthesis," *University of California, Berkeley*, vol. 94720, p. 4, 1992.
[25] L. Shi *et al.*, "Research progress on solutions to the sneak path issue in memristor crossbar arrays," *Nanoscale Advances*, vol. 2, no. 5, pp. 1811–1827, 2020.
[26] M. C. Hansen *et al.*, "Unveiling the iscas-85 benchmarks: A case study in reverse engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.
[27] L. Amarú *et al.*, "The epfl combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.