PATH: Evaluation of Boolean Logic using Path-based In-Memory Computing Systems

Sven Thijssen*, Muhammad Rashedul Haq Rashed†, Sumit Kumar Jha‡, and Rickard Ewetz†
*Department of Computer Science, University of Central Florida, Orlando, USA
†Department of Electrical and Computer Engineering, University of Central Florida, Orlando, USA
‡Department of Computer Science, Florida International University, Miami, FL, USA
{sven.thijssen, muhammad.rashed, rickard.ewetz}@ucf.edu, jha@cs.fiu.edu

Abstract—**In-memory computing using non-volatile memory is a promising pathway to accelerate data-intensive applications. While substantial research efforts have been dedicated to executing Boolean logic using digital in-memory computing, the limitation of state-of-the-art paradigms is that they heavily rely on repeatedly switching the state of the non-volatile resistive devices using expensive WRITE operations. In this paper, we propose a new in-memory computing paradigm called path-based computing for evaluating Boolean logic. Computation within the paradigm is performed using a one-time expensive compilation phase and a fast and efficient evaluation phase. The key property of the paradigm is that the execution phase only involves cheap READ operations. First, we define an analogy between binary decision diagrams (BDDs) and one-transistor one-memristor (1T1M) crossbars that allows Boolean functions to be mapped into crossbar designs. When such crossbar design becomes too large to be physically realizable, we propose to synthesize the Boolean function into a path-based computing system. A path-based computing system consists of a topology of staircase structures. A staircase structure is a cascade of hardwired crossbars, which minimizes inter-crossbar communication. We evaluate the proposed paradigm using ten circuits from the Revlib benchmark suite, eight control circuits of the EPFL benchmark suite, and eight ISCAS85 benchmarks. Compared with state-of-the-art digital in-memory computing paradigms, path-based computing improves energy and latency with $1006\times$ and $10\times$ on average, respectively.**

I. Introduction

The growth of digital data accelerates at a high pace. In 2025, the total amount of digital data is expected to be 175ZB [1]. This growth is driven by a variety of factors, one being the collection of sensor data using IoT devices [2]. The development of 5G and 6G networks will only accelerate the amassment of this data further [3]. Another contributing factor is the emergence of data-driven technologies, such as deep neural networks [4], and foundational AI models, which require internet-scale amounts of digital data for unsupervised pre-training [5]. Unfortunately, these data-intensive techniques suffer from the Von Neumann bottleneck [6]. The bottleneck denotes the energy-inefficiency of a bus to transfer data between a computer's memory and computing units. Several other factors, such as the End of Moore's law [7] and the End of Dennard Scaling [8], are challenging the performance of these data-intensive applications.

Processing in-memory using non-volatile memory has recently attracted significant attention to mitigate the aforementioned limitations [17]. Non-volatile memory technology includes memristors, resistive random access memory (ReRAM) [18], phase change memory (PCM) [19], and spin-transfer torque magnetic random access memory (STT-MRAM) [20]. Analog in-memory computing is well-known for performing matrix-vector multiplication at high speed and with low energy consumption. These computations are carried out in dense crossbar arrays. Unfortunately, analog in-memory computing is limited to matrix-vector multiplication, and related arithmetic operations [21]. Some efforts have been made to improve accuracy while maintaining these energy and latency advantages [22]. Unfortunately, despite these efforts, analog in-memory computing cannot deliver the deterministic precision required for high-assurance applications. However, digital computing is more robust due to the clear distinct states for a logical zero and one [23]. For comprehensive reviews on in-memory computing, we refer to [24]–[26].

This work was in part supported by DARPA #FA87502320501, DARPA #HR00112420004, DOE #DESC0024576, NSF #2319399, NSF #2404036, and NSF #2408925.

TABLE I
Comparison of in-memory logic styles in terms of underlying operation and evaluated logic complexity.

Digital logic style	Representative Studies	Operations in each phase	
		Compile	Execute
IMPLY	[9], [10]	WRITE	WRITE+READ
MAGIC	[11], [12]	WRITE	WRITE+READ
MAJORITY	[13], [14]	WRITE	WRITE+READ
FLOW	[15], [16]	WRITE	WRITE+READ
Path-based	**(this paper)**	WRITE	READ

Several noteworthy digital in-memory computing paradigms are IMPLY [9], MAGIC [11], MAJORITY [13], and FLOW [15]. These in-memory computing paradigms more or less have the following in common: the paradigms consist of two broad phases. First, there is a one-time compilation phase and, second, an execution phase that is performed for each function input. In Table I, we show the READ and WRITE operations performed in each phase for the different logic styles. It can be observed that all previous paradigms use WRITE operations in the execution phase. WRITE operations are orders of magnitude more expensive than READ operations [27]. Further, WRITE operations are detrimental to the endurance of the memristor's lifetime [28]. In contrast, the proposed path-based computing paradigm evaluates Boolean logic using READ operations in the execution phase, mitigating the high energy consumption for the WRITE operations and thus extending the system's lifetime.

Further, design automation tools are essential to map computation into hardware designs. Hardware-software co-design is a trending approach in a variety of novel computing schemes, including photonic computing [29], [30], quantum computing [31], [32], and in-memory computing [33], [34] to optimize the hardware resources. In this work, we explore a tight hardware-software co-design for 1T1M crossbars and Boolean functions. To achieve this strong relation between hardware and software, we base ourselves on an analogy between BDDs and 1T1M crossbars.

Lastly, in previous works [12], [16], little or no attention is made to the underlying architecture. Many of these rely on a simple computing architecture consisting of multiple crossbars connected to a bus. Unfortunately, such bus-crossbar architecture is not energy- and latency-efficient. In our work, we target staircase architectures where a staircase is a hardwired collection of crossbars. The main idea is that the bus utilization will be reduced, which translates into energy and latency improvements of the overall computing system.

In this paper, we propose a new computing paradigm called path-based in-memory computing. The paradigm is capable of evaluating Boolean functions using 1T1M crossbar arrays. We also propose a framework called PATH to automatically map computation to 1T1M crossbars or path-based computing systems with staircase structures. The main innovations of the paper are summarized, as follows:

- A new computing paradigm, called path-based in-memory computing, is introduced. The paradigm executes Boolean functions fast and efficiently using only READ operations instead of using slow and energy-consuming WRITE operations.
- We introduce a framework, called PATH, to synthesize Boolean functions into a single crossbar design. The PATH framework exploits an analogy between BDDs and 1T1M crossbars to map Boolean functions into crossbar designs. A BDD with $|V|$ nodes and $|E|$ edges can be mapped into a to a crossbar of dimensions $|V| \times |E|$.
- We further introduce an equivalent bipartite graph data structure for the BDD. By means of node merging, this bipartite graph can be further compressed into a smaller, equivalent graph. This compression results in an area reduction of 16%.
- When the crossbar design becomes too large to be physically realizable, the PATH framework provides a partitioning algorithm to map Boolean functions to staircase structures. The objective is to minimize the bus utilization by minimizing the hardware resources in terms of the number of staircases.
- The experimental evaluation is performed on ten circuits from the Revlib benchmark suite, eight control circuits from the EPFL benchmark suite, and eight ISCAS85 benchmarks. Compared with the state-of-the-art in-memory paradigm COMPACT [16], PATH improves energy and latency with $1006\times$ and $10\times$ on average.

The remainder of the paper is organized as follows: preliminaries are provided in Section II. The path-based computing paradigm is introduced in Section III. In Section IV, we review some state-of-the-art digital in-memory computing paradigms. Problem formulation and a high-level overview of the PATH framework are given in Section V. The crossbar-level synthesis framework is detailed in Section VI, and the partitioning algorithm is provided in Section VII. Optimization steps are introduced in Section VIII. The experimental evaluation is performed in Section IX. The paper is concluded in Section X.

II. Preliminaries

A. Binary Decision Diagrams

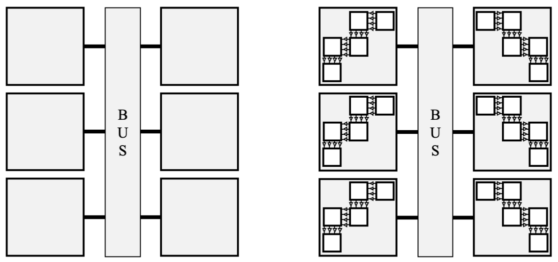
A binary decision diagram (BDD) is a graph representation of a Boolean function. The directed acyclic graph (DAG) consist of internal decision nodes and two leaf (terminal) nodes. The terminal nodes represent the output '0' and '1', respectively. The internal decision nodes are assigned a Boolean variable, and each internal decision node has a positive and negative output edge. The positive edge corresponds to the positive literal, and the negative edge corresponds to the negative literal. A BDD is evaluated by traversing the graph from the root nodes to one of the leaf nodes based on an instance of the Boolean variables. BDDs commonly refer to reduced order binary decision diagrams (ROBDDs) where nodes and edge have been eliminated to reduce the size of the representation [35]. When a BDD is used to represent a multi-output function, the BDD will have a separate root node for each output of the Boolean function [36].

B. Memristor Crossbar Arrays

In this section, we will review one-transistor one-memristor (1T1M) crossbars [37]. A model for a 1T1M crossbar is illustrated in Figure 2(c). A 1T1M crossbar array consists of wordlines, bitlines, and selectorlines. Each wordline is connected to each bitline using a series-connected memristor and access transistor. The vertically aligned access transistors share a single selectorline. Both the memristors and the access transistors act functionally as switches that can be turned ON and OFF. The switch corresponding to a memristor is ON (or OFF) based on if the memristor is programmed to have low (or high) resistance. The switch corresponding to the access transistor is turned ON (or OFF) based on if the selectorline is charged (or discharged, depending on the type of transistor).

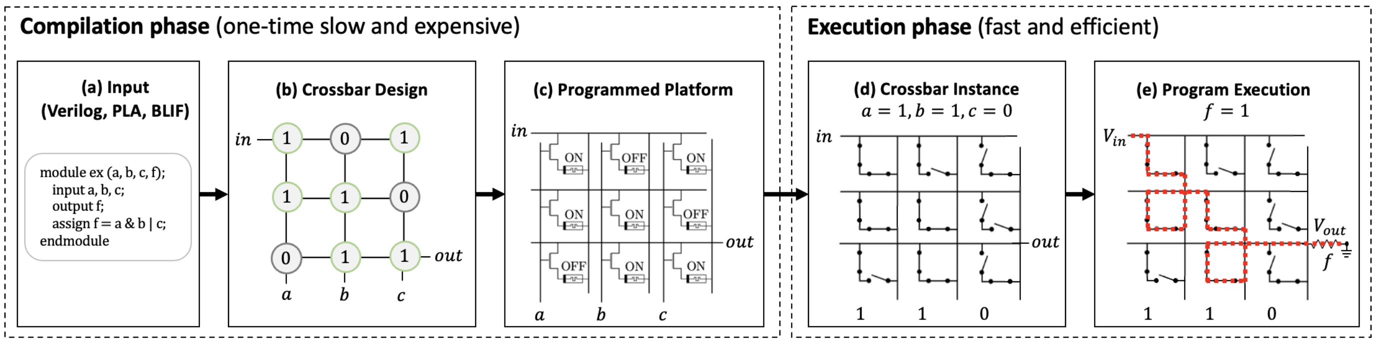
C. In-memory computing architecture

Traditionally, bus architectures have been leveraged for in-memory computing [38]–[40]. In this computing architecture, the crossbars are connected to a bus. An example of a bus architecture with six crossbars is illustrated in Figure 1(a). However, in our work, we target a path-based computing system with staircase structures. A staircase structure is a collection of crossbars that have hardwired inter-connections. In Figure 1(b), we illustrate a staircase architecture of six staircases where each staircase consists of five hardwired crossbars.



(a) Bus architecture (b) Staircase architecture

Fig. 1. Comparison of a traditional bus architecture and a staircase architecture. Each staircase is a collection of hardwired crossbars.


Compilation phase (one-time slow and expensive)
Execution phase (fast and efficient)
(a) Input (Verilog, PLA, BLIF)
module ex (a, b, c, f);
input a, b, c;
output f;
assign f = a & b | c;
endmodule
(b) Crossbar Design
(c) Programmed Platform
(d) Crossbar Instance a = 1, b = 1, c = 0
(e) Program Execution f = 1
in
out
Vin
Vout
ON
OFF

(a) Verilog code (b) Crossbar design $\mathcal{D}$ (c) 1T1M crossbar reconfiguration (d) Crossbar instance $\mathcal{I}$ (e) Evaluation

Fig. 2. Flow for the synthesis and evaluation of Boolean functions for path-based computing. (a) A program in Verilog code. (b) The abstract crossbar design obtained through synthesis. (c) The physical crossbar with the non-volatile memory devices programmed and Boolean variables assigned to the selectorlines. (d) The state of the switches (open/closed) with respect to the state of the non-volatile memory devices (ON/OFF) and the instance (a,b,c)=(1,1,0) of the Boolean variables. (e) The Boolean function f evaluates to 1 because there is a path from the input to the output.

III. Path-based computing

Path-based computing aims to evaluate Boolean functions using in-memory computing. An example of the flow for the synthesis and evaluation of path-based computing is shown in Figure 2. The flow for path-based computing consists of a one-time slow and expensive compilation phase and a fast and efficient execution phase. The input to the compilation phase is a Boolean function specified in a hardware descriptive language (Verilog, VHDL), which is shown in Figure 2(a). The input is first synthesized into an abstract crossbar design $\mathcal{D}$, which is shown in Figure 2(b). The 1T1M crossbar design specifies the state of each non-volatile memory device (0/1) and the Boolean variable assigned to each selectorline. Here, the Boolean literals a, b, and c are assigned to the first, second, and third selectorline, respectively. The input and output assignment to the wordlines are also specified. Next, the memory devices within a nanoscale crossbar are programmed ON (LRS) or OFF (HRS), which is shown in Figure 2(c). The state of the devices are programmed to LRS or HRS by applying a voltage with appropriate polarity and magnitude [41]. We use a write-and-verify scheme to ensure the correct programming [42].

In the execution phase, an instance of Boolean variables is provided to the selectorlines. The selectorlines control the switches represented by the access transistors. The state of the switches controlled by the memory devices are also shown in Figure 2(d). Next, an input voltage is applied to the top-most wordline and an output voltage is measured across a resistor connected to the bottom-most wordline. If the output voltage is high, the Boolean function evaluates to true. Otherwise, the function evaluates to false. For the input instance (a,b,c)=(1,1,0), the function evaluates to true because there exists a path from the input to the output, as illustrated in Figure 2(e). In contrast, the function evaluates to false for the input instance (1,0,0). Observe that the memristors must not be reprogrammed to evaluate the same Boolean function for different input vectors. In Figure 3, a more detailed example is shown for the Boolean function $f = a \vee \neg b$. More specifically, we show the state of the crossbar for all four input vectors. Again, the crossbar must not be reprogrammed for different input vectors.

The one-time compilation phase is both slow and expensive. Mainly, due to the expensive WRITE operations used to program the platform. On the other hand, the cost is amortized across each execution of the Boolean function. The execution phase is fast and efficient because it only involves charging/decharging the selectorlines and performing READ operations. The advantageous properties compared with other in-memory paradigms comes from the *novel* use of the access transistors. No previous paradigms have used the access transistors to perform logic.

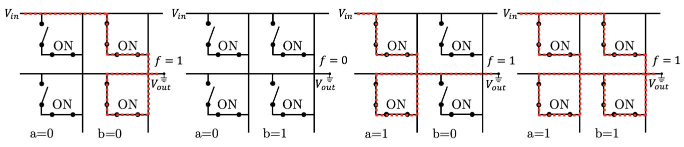


Fig. 3. Execution of all four input vectors on a crossbar for the Boolean function $f = a \vee \neg b$. Observe how the state of the memristors does not change for different input vectors, but only the state of the access transistors changes using the selectorlines.

IV. Comparison of digital in-memory computing paradigms

In this section, we compare and review some of the most prevalent state-of-the-art digital in-memory computing paradigms and our proposed path-based in-memory computing paradigm (PATH). These digital in-memory computing paradigms are IMPLY [45], MAGIC [11], MAJORITY [13], and FLOW [44]. In Figure 4, we illustrate the steps during execution for each of the logic styles to evaluate the Boolean function $f = (a \wedge b) \vee \neg c$ for the input vector $a = 1, b = 1, c = 1$. For consistency, we employ the following basic operations among these logic styles: READ, WRITE, and COPY (READ+WRITE). The definitions are provided in the legend at the top of Figure 4.

A. IMPLY

IMPLY logic is based on the Boolean operation material implication (IMP) [43], [45]. The IMP operation $P \rightarrow Q$ can be realized in hardware using two memristors P and Q. By applying voltages over the memristors P and Q, the result is obtained in the memristor Q. Thus, IMPLY logic is destructive in terms of its inputs [43]. Further, extensive design automation tools for IMPLY-based in-memory computing have not been developed, usually requiring manual labor to design

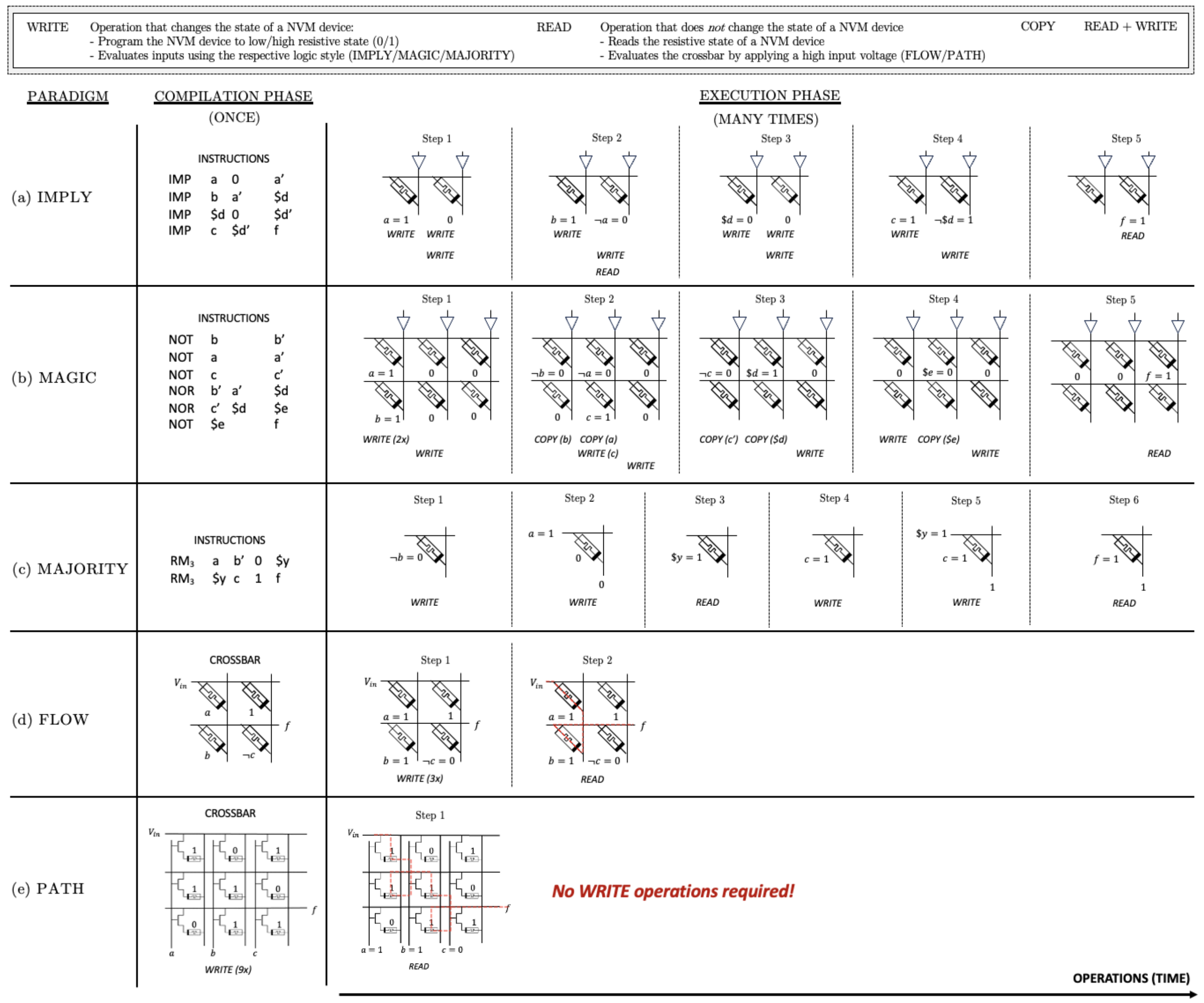

WRITE Operation that changes the state of a NVM device:
- Program the NVM device to low/high resistive state (0/1)
- Evaluates inputs using the respective logic style (IMPLY/MAGIC/MAJORITY)
READ Operation that does not change the state of a NVM device
- Reads the resistive state of a NVM device
- Evaluates the crossbar by applying a high input voltage (FLOW/PATH)
COPY READ + WRITE
PARADIGM
COMPILATION PHASE (ONCE)
EXECUTION PHASE (MANY TIMES)
(a) IMPLY
INSTRUCTIONS
IMP a 0 a'
IMP b a' $d
IMP $d 0 $d'
IMP c $d' f
(b) MAGIC
INSTRUCTIONS
NOT b b'
NOT a a'
NOT c c'
NOR b' a' $d
NOR c' $d $e
NOT $e f
(c) MAJORITY
INSTRUCTIONS
RM3 a b' 0 $y
RM3 $y c 1 f
(d) FLOW
CROSSBAR
WRITE (3x)
(e) PATH
CROSSBAR
WRITE (9x)
No WRITE operations required!
OPERATIONS (TIME)

Fig. 4. Comparison of the last step of the compilation phase and the steps during the execution phase for digital in-memory logic styles IMPLY [43], MAGIC [11], MAJORITY [13], FLOW [44], and the proposed PATH. For a given Boolean function, the compilation phase is only performed once and the execution phase is performed many times. For the compilation phase, either a series of instructions or a crossbar is provided. The Boolean function $f = (a \wedge b) \vee \neg c$ is computed for the input vector $a = 1, b = 1, c = 1$. Intermediate variables are denoted by a dollar sign ($). For each logic style, the operations READ, WRITE, and COPY (= READ + WRITE) are provided, as defined in the legend at the top of the figure. The WRITE operation writes the required resistive state to a memristor or executes the primary Boolean function for IMPLY/MAGIC/MAJORITY by applying the required input voltages. The READ operation reads the resistive state of a memristor or evaluates the Boolean function for FLOW/PATH. The COPY operation performs both a READ and WRITE operation. Observe how path-based computing does not require any WRITE operations during the execution phase.

circuits [46]. One of the few automation tools for IMPLY is described in [47]. In Figure 4(a), we observe that IMPLY requires many intermediate steps of READ and WRITE operations to realize the Boolean function f. The required IMP operations are also provided in Figure 4(a).

B. MAGIC

The MAGIC logic style [11] is based on the Boolean operation NOR, and can be considered the successor of IMPLY. The NOR operation can be realized using three memristors. The NOT operation is a NOR operation where one input is always '1'. In contrast with IMPLY, MAGIC is not destructive for its inputs [11] when applying the appropriate voltage. Further, there is an additional memristor for the output to be realized. Over the years, many papers have been proposed using the MAGIC logic style [12], [46], [48]. In Figure 4(b), we show the steps to realize the Boolean function f using READ, WRITE, and COPY operations.

C. MAJORITY

The majority operation is a Boolean function that evaluates to true when half or more of its inputs evaluate true. For in-memory computing, the majority operation with three inputs is primarily interesting due to its one-to-one correspondence with a single memristor [13]. We define the majority operation as $Z' = M(X, \neg Y, Z) = (X \wedge Z) \vee (\neg Y \wedge Z) \vee (X \wedge \neg Y)$. Then let X and Y be the inputs to the two terminals of the memristor, and let Z be the resistive state of the memristor. By applying the appropriate voltages to the inputs and programming the memristor to the appropriate resistive state, the majority function can be executed in-situ. The resulting value Z' is then stored as a resistive value in the memristor [49]. Several synthesis methods have been proposed in recent years [14], [49]–[52], many of which rely on majority inverter graphs (MIGs) as data structure. In Figure 4(c), we illustrate the steps using READ and WRITE operations for the majority logic style.

D. FLOW

FLOW (flow-based computing) is a digital in-memory computing paradigm which relies on the absence/presence of electrical current to perform its computations [16], [44], [53]. Initially, the input variables, their negations, and the Boolean truth values (0/1) are assigned to the memristors. Program execution consists of two steps. In the first step, the memristors are programmed to their resistive states (0 for high, 1 for low), as shown in the first step of Figure 4(d). In the second step, a high input voltage is applied to the input wordline (V_{in}), and the Boolean function is *read* out as follows: if there is a path from the input wordline to the output wordline through memristors in a low resistive state, then the Boolean function evaluates to true. Otherwise, the Boolean function evaluates to false.

E. PATH

In our proposed path-based computing (PATH), the program execution solely relies on READ operations, and the application of an input voltage to perform computations. WRITE operations are only performed once during the previous step, i.e. the compilation phase, for a given Boolean function. In the first step of Figure 4(e), we are show the crossbar design. The memristors are in their resistive states, which were programmed only once prior, during 1T1M crossbar reconfiguration (see previous Section III). During program execution there are no WRITE operations, and thus these resistive states do not change. However, the selectorlines are charged accordingly to open/close access transistors. The memristors are programmed to their resistive states only once for a given Boolean function. Given another input vector $a = 0, b = 1, c = 0$, the crossbar design in Figure 4(e), would remain, and is thus invariant to the input vector. Then, in the second step, the evaluation is *read* out by sensing the presence/absence of electrical current. The crossbar invariancy makes PATH a strong contender for repeated computations.

V. The PATH Framework

First, we outline the problem formulations in Section V-A. Then, we give a high-level overview the PATH framework in Section V-B.

A. Problem formulation

Our overall objective is to synthesize a Boolean function ϕ into a path-based computing system. We approach this larger problem by solving two smaller problems, as follows:

- **Problem I:** We propose a synthesis method to construct a crossbar design $\mathcal{D}$ for a Boolean function ϕ. The algorithm is based on an analogy between a BDD for the function ϕ and a 1T1M crossbar. We further improve the synthesis method by transforming the BDD into an equivalent graph-based data structure such that we can reduce its graph size by merging nodes. This transformation results in smaller crossbar designs, and subsequently power and latency improvements.
- **Problem II:** Based on the analogy of Problem I, we propose a synthesis method to construct a topology $\mathcal{T}$ for a path-based computing system of staircase structures S_j. A staircase structure S_j is an ordered set of crossbars X_i. Between each X_i and X_{i+1}, there are hardwired inter-crossbar connections from the wordlines of crossbar X_i to the selectorlines of crossbar X_{i+1}.

B. Overview of framework

In this section, we give an outline of the PATH framework. An overview of the synthesis flow is shown in Figure 5. First, in Section VI, we discuss synthesis for a single crossbar. Then, in Section VII, we discuss the staircase partitioning. For the crossbar synthesis, we introduce an algorithm to construct a single crossbar design $\mathcal{D}$ based on an analogy between a bipartite graph, derived from a BDD for the Boolean function(s), and a 1T1M crossbar. This algorithm consists of four steps: graph pre-processing, graph transformation, node merging, and crossbar realization. Then, in Section VII, we introduce a partitioning algorithm. The main steps for the partitioning are the graph partitioning, and the realization of staircase intra- and inter-connections. The framework is illustrated with an example in Figure 6.

VI. Crossbar synthesis

The input to the framework is a BDD, and the output is a crossbar design. The BDD is obtained using CUDD [54] which is subsequently pruned into a graph G. The details are provided in Section VI-A. In the next step, the graph transformation step in Section VI-B, the pruned graph G is converted into a bipartite graph B. This graph is then compressed into a new bipartite graph B' in Section VI-C. The last step is the crossbar synthesis step, which constructs a crossbar design $\mathcal{D}$ for the given bipartite graph B'. The details are provided in Section VI-D.

A. Graph pre-processing

The input to the graph pre-processing step is a BDD. In Figure 6(a), a multi-output BDD for a full adder is provided. The Boolean functions are $c_{out} = (a_0 \wedge b_0) \vee (a_0 \wedge c_{in}) \vee (b_0 \wedge c_{in})$ and $s_0 = a_0 \oplus b_0 \oplus c_{in}$, respectively. The graph pre-processing involves removing the zero output node and all the edges connected to the zero terminal node. The zero terminal node can be removed because it corresponds to $\neg c_{out}$ and $\neg s_0$. The one terminal node will be connected to the input, which we label in. The edges in the BDD are labeled with their respective decision variables. The positive (negative) edge connected to node with the decision variable x_i will be labeled x_i ($\neg x_i$). Finally, we reverse the edges and we label the nodes from 1 to $|V|$ where $|V|$ is the number of nodes. The resulting graph of the BDD in Figure 6(a) is shown in Figure 6(b).

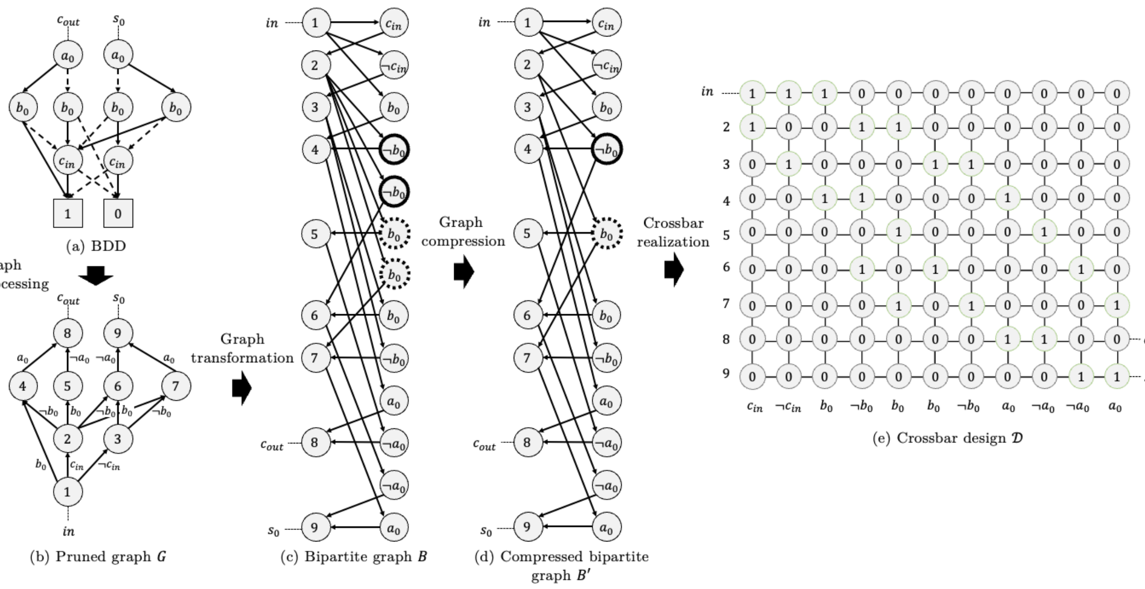


Fig. 6. Example of the synthesis flow. (a) A multi-output BDD for a full adder with Boolean functions c_{out} and s_0. (b) The nodes of the BDD are relabeled and the edges are assigned a Boolean literal based on their shape (positive literal for solid edges and negative literal for dashed edges). Further, the negative terminal node 0 is removed. (c) The bipartite graph $B = (U_1, U_2, F)$ is constructed from the pruned graph G. (d) The bipartite graph B is compressed into an equivalent bipartite graph $B' = (U_1', U_2', F')$ using node merging. (e) A crossbar design $\mathcal{D}$ is constructed with dimensions $|U_1'| \times |U_2'|$ where each node $u_1 \in U_1'$ is assigned to a wordline and each node $u_2 \in U_2'$ is assigned to a bitline-selectorline pair. For each edge $f = (u_1, u_2)$ or $f = (u_2, u_1)$, $u_1 \in U_1', u_2 \in U_2'$, a memristor is programmed to a low resistive state (ON) to realize the connections.

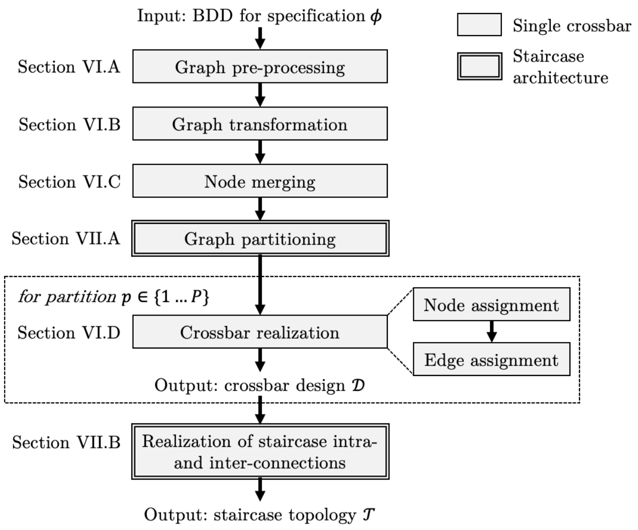


Fig. 5. Overview of the PATH framework, including the crossbar synthesis and staircase partitioning.

B. Graph transformation

In this step, the resulting pruned graph is converted into a directed bipartite graph. This graph transformation is introduced as an intermediary data structure for the node merging step. Let $G = (V, E)$ be the pruned graph where V is a set of nodes and E is a set of edges, and let $B = (U_1, U_2, F)$ be a bipartite graph where U_1 and U_2 are sets of nodes and F is a set of edges. The sets U_1 and U_2 are disjoint and independent [55], and F is a new set of edges between nodes from U_1 and U_2. Let $v \in V$ correspond to a node $u_1 \in U_1$, and let $e \in E$ correspond to a node $u_2 \in U_2$. For each node $v \in V$, we introduce a node $u_1 \in U_1$. For each edge in the BDD, we introduce a node with two edges. More specifically, for an edge $e = (v_1, v_2, l) \in E$ where $v_1 \in V$, $v_2 \in V$ and l is a literal, we create a new node $u_2 = (u_1^1, u_1^2, l) \in U_2$ where u_1^1 is the image of v_1 and u_1^2 is the image of v_2. Then, we realize the connections between nodes and edges by introducing two new edges in F for each node $u_2 \in U_2$ such that $F = \{(u_1^1, u_2), (u_2, u_1^2) \mid u_2 = (u_1^1, u_1^2, l),\ u_2 \in U_2\}$. An example of the transformation of the pruned graph G into a bipartite graph B is illustrated in Figure 6(c). Note that we represent the nodes in U_2 with their literals l instead of the triple (u_1^1, u_2^2, l) for clarity.

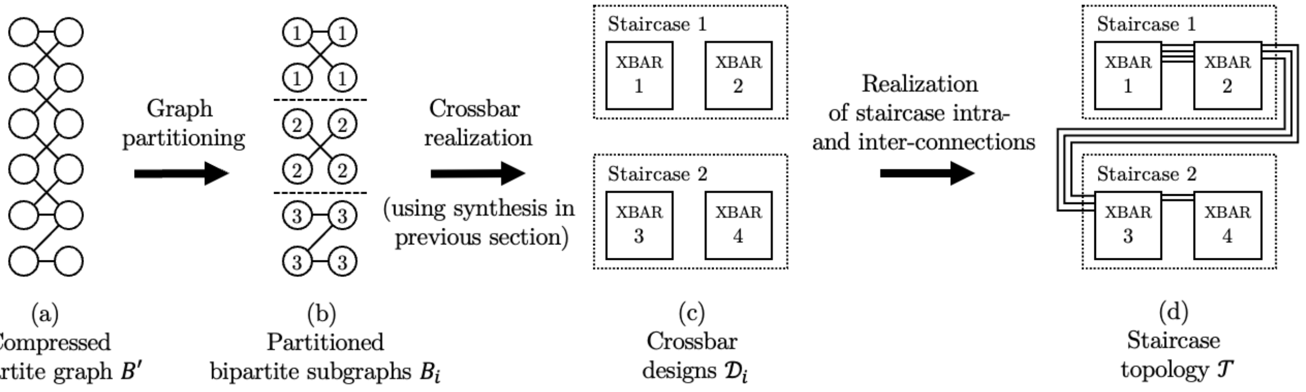
C. Node merging

In the bipartite graph, we observe that a node $u_1 \in U_1$ may have outgoing edges to more than one node $u_2 \in U_2$ with the same literal l. For example, in Figure 6(c), we observe that node 2 in the bipartite graph B has two outgoing edges to two distinct nodes with both label $\neg b$. In this section, we propose to merge such nodes with the same label into a single node.

More formally, let $B = (U_1, U_2, F)$ be the bipartite graph and let $u_1 \in U_1$ be a node with outgoing edges to nodes $u_2^i = (u_1, u_i, l)$ and $u_2^j = (u_1, u_j, l)$ where $i \neq j$, and $u_1, u_i, u_j \in U_1$. Then we define a mapping $B = (U_1, U_2, F) \rightarrow B' = (U_1', U_2', F')$ as follows:

$$
\begin{aligned}
&U_1 \rightarrow U_1' : u_1 \mapsto u_1 \\
&U_2 \rightarrow U_2' : u_2 = (u_1, u_i, l) \mapsto u_2' = (u_1, l) \\
&F \rightarrow F' : f = (u_1, (u_1, u_i, l)) \mapsto f' = (u_1, (u_1, l)) \text{ and} \\
&\qquad\quad f = ((u_1, u_i, l), u_1) \mapsto f' = ((u_1, l), u_1)
\end{aligned}
$$

Based on the aforementioned mapping function, we can merge the two nodes with label $\neg b$ into one node such that we obtain a compressed bipartite graph B' as illustrated in Figure 6(d). This operation is valid due to that the nodes $u_2 \in U_2$ represent literals l, and the edges between $u_1 \in U_1$ and $u_2 \in U_2$ represent conjunctions between u_1 and u_2. Thus, for two such edges (u_1, u_i) and (u_1, u_j), we have the following: $u_1 \wedge u_i = u_1 \wedge l = u_1 \wedge u_j$.



Graph partitioning
Crossbar realization
(using synthesis in previous section)
Realization of staircase intra- and inter-connections
Staircase 1
XBAR 1
XBAR 2
Staircase 2
XBAR 3
XBAR 4
(a) Compressed bipartite graph B'
(b) Partitioned bipartite subgraphs B_i
(c) Crossbar designs $\mathcal{D}_i$
(d) Staircase topology $\mathcal{T}$

Fig. 7. A high-level overview of the partitioning scheme. (a) The input of the partitioning scheme is the compressed bipartite graph B' and the user-defined parameter $T = 2$. (b) Using the graph partitioning algorithm explained in Section VII-A, the bipartite graph is decomposed into smaller bipartite subgraphs B_i. (c) The individual subgraphs B_i are each synthesized into crossbar designs $\mathcal{D}_i$ using the crossbar realization in Section VI-D. (d) Finally, a staircase topology $\mathcal{T}$ is constructed by realizing the intra-staircase and inter-staircase connections, as explained in Section VII-B.

D. Crossbar realization

The outlined crossbar realization is based on an analogy between the bipartite graph $B' = (U_1', U_2', F')$ and 1T1M crossbars. The nodes $u_1 \in U_1'$ correspond to wordlines and the nodes $u_2 \in U_2'$ correspond to bitline-selectorline pairs. The path-based paradigm is based on creating paths by turning on and off connections in the crossbar design. The connections correspond with the edges $f \in F'$, which are realized using the memristors. The crossbar mapping consists of a node assignment step and an edge assignment step.

1) Node assignment: The node assignment involves assigning the nodes $u_1 \in U_1'$ to the wordlines of the crossbar design $\mathcal{D}$ and the nodes $u_2 \in U_2'$ to the bitline-selectorline pairs of crossbar design $\mathcal{D}$.

2) Edge assignment: Next, for each edge $f = (u_1, u_2)$ or $f = (u_2, u_1)$, $u_1 \in U_1, u_2 \in U_2, f \in F$, we program the corresponding memristor at the intersection of wordline u_1 and selectorline u_2 to a low resistive state (ON). Further, the input and output are assigned to the respective wordlines. The resulting crossbar design $\mathcal{D}$ for the Boolean functions f_1 and f_2 is shown in Figure 6(e).

VII. Partitioning for staircase structures

In this section, we propose a partitioning algorithm to synthesize the Boolean function ϕ into a topology $\mathcal{T}$ of staircase structures. A topology is a directed acyclic graph (DAG) of staircase structures with potentially multiple edges between different staircase structures where each staircase structure is an ordered set of crossbars with inter-crossbar connections between two consecutive crossbars. An overview of the partitioning scheme is illustrated in Figure 7.

The input of the partitioning algorithm is a bipartite graph $B = (U_1, U_2, F)$ and the output is a topology $\mathcal{T}$ of staircase structures. The bipartite graph B is obtained by means of the pre-processing steps described in Section VI-A and VI-B. The idea of the partitioning scheme is that the given bipartite graph B is partitioned into smaller bipartite graphs $B_i = (U_{1,i}, U_{2,i}, F_i), |U_{1,i} + U_{2,i}| \leq |U_1 + U_2|$. For each B_i, a crossbar design $\mathcal{D}_i$ is constructed, which is part of a staircase structure. Unfortunately, it is not straightforward to partition the graph B into B_i such that the size of B_i is maximized while meeting the dimensions of crossbar X_i. The partitioning makes that intermediate evaluations must be propagated to other crossbars and/or staircases. Further, only the first crossbar X_1 in a staircase structure is connected to the bus, which brings that the intermediate results and literals can only be fed to this first crossbar. To address these constraints, we propose the following: a user-defined parameter defines the maximum dimensions which may be used to synthesize a bipartite graph B_i. Here, we assume the number of wordlines and the number of bitline-selectorline pairs is equal for a crossbar. An algorithm to construct such topology $\mathcal{T}$ is provided in Section VII-A. Next, staircase intra- and inter-connections must be realized for the aforementioned constraints. These are discussed in Section VII-B.

Algorithm 1 Partitioning algorithm for staircase structures

```
Input: $B = (U_1, U_2, F)$, $T = \{T_0, ..., T_L\}$
Output: $\mathcal{T}$ // Set of staircase designs
 1: function TopologicalStaircasePartitioning($B$, $T$)
 2:     $i = 1, V_{i,1} = \emptyset, V_{i,2} = \emptyset, S \leftarrow \emptyset$
 3:     $\mathcal{T} \leftarrow \emptyset$
 4:     for $u_2 \in$ TopologicalSort($U_2$) do
 5:         $V_{i,1}' \leftarrow V_{i,1} \cup \{u_1 | f = (u_1, u_2) \vee f = (u_2, u_1), \forall f \in F\}$
 6:         $V_{i,2}' \leftarrow V_{i,2} \cup \{u_2\}$
 7:         if $|V_{i,1}'| \leq T_i \wedge |V_{i,2}'| \leq T_i$ then
 8:             $V_{i,1} \leftarrow V_{i,1}'$
 9:             $V_{i,2} \leftarrow V_{i,2}'$
10:         else
11:             $F_i \leftarrow \{f | \exists u_1 \in V_{i,1}, u_2 \in V_{i,2} :$
                        $f = (u_1, u_2) \vee f = (u_2, u_1), f \in F\}$
12:             $B_i \leftarrow (V_{i,1}, V_{i,2}, F_i)$ // Create bipartite subgraph
13:             $S \leftarrow S \cup \{B_i\}$
14:             $i \leftarrow i + 1$
15:             $V_{i,1} = \{u_1 | f = (u_1, u_2) \vee f = (u_2, u_1), \forall f \in F\}$
16:             $V_{i,2} = \{u_2\}$
17:             if $|S| = L$ then
18:                 $\mathcal{T} \leftarrow \mathcal{T} \cup \{S\}$
19:                 $i \leftarrow 1, S \leftarrow \emptyset$
20:             end if
21:         end if
22:     end for
23:     return $\mathcal{T}$
24: end function
```

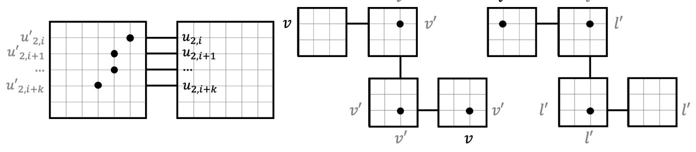
A. Graph partitioning

In Algorithm 1, we provide the first part of the partitioning scheme. We are given a bipartite graph $B = (U_1, U_2, F)$ as input, and a user-defined threshold T_i for the amount of logic that will placed within each crossbar X_i. The output of the algorithm is a topology $\mathcal{T}$ of staircase structures S where each S is an ordered set of crossbars X_i such that X_i precedes X_{i+1}. The partitioning algorithm has two auxiliary variables $V_{i,1}$ and $V_{i,2}$ which will contain the nodes assigned to the wordlines and selectorlines, respectively. The nodes that are assigned to $V_{i,1}$ are in U_1, and the nodes that are assigned to $V_{i,2}$ are in U_2.

The algorithm iterates in a topological sort over the nodes $u_2 \in U_2$. In each iteration, node u_2 is assigned to a crossbar, together with its neighboring nodes. Recall that the nodes u_2 are the edges $e \in E$ in our original graph $G = (V, E)$. When assigning a node $u_2 \in U_2$ to a crossbar X_i, we want each neighboring node u_1 to be assigned to X_i as well. This is due to that u_2 represents an edge $e = (v_1, v_2) \in E$ between two nodes $v_1, v_2 \in V$. Thus, we want both its endpoints to be present in the crossbar X_i.

When assigning a node u_2 to the wordlines of a crossbar $V_{i,2}$, we must not exceed the logic threshold T_i we have set. Similar for its neighboring nodes u_1 when assigning to the selectorlines $V_{i,1}$ (condition of if statement on line 7). If the condition fails, we create a bipartite subgraph $B_i = (V_{i,1}, V_{i,2}, F_i)$ (line $11-12$), and we add B_i to the current staircase S (line 13). When the current staircase S has reached its maximum depth L (line 17), then we will add the current staircase S to the topology $\mathcal{T}$ (line 18), and we create a new staircase S (line 19). The algorithm stops when all nodes $u_2 \in U_2$ have been processed.

In Figure 7(a), we take the compressed bipartite graph B' and the user-defined parameter $T = 2$ as input for the partitioning algorithm. In Figure 7(b), we illustrate the partitioning of the bipartite graph into multiple subgraphs B_i. Each subgraph B_i is delineated by a dashed line, and all its nodes have the same number i. These subgraphs are subsequently synthesized into crossbar designs $\mathcal{D}_i$, as explained in Section VI-D and grouped into staircase structures.



(a) Edge prep. (b) Node prop. (c) Literal prop.

Fig. 8. Example of the intra-connections.

B. Realization of staircase intra- and inter-connections

While the algorithm partitions the bipartite graph B into bipartite subgraphs B_i, which are mapped to crossbars, the hardware architecture imposes additional constraints on the design. We have identified three staircase intra- and inter-connections that must be made to realize the crossbar mapping to a partitioning over staircase structures: edge preparation, node propagation, and literal propagation. In Figure 7(c), we take the crossbar designs $\mathcal{D}_i$ as input. The output is a topology $\mathcal{T}$ of staircases by realizing the staircase intra- and inter-connections, as illustrated in Figure 7(d).

1) Edge preparation: For each crossbar X_i, $i > 1$, the selectorlines are connected to the wordlines of previous crossbar X_{i-1}. In the mapping algorithm in VI-D, the nodes $u_2 \in U_2$ are assigned to the selectorlines. This entails that the nodes must be *prepared* in crossbar X_{i-1}. In Figure 8(a), is illustrated how the nodes u_2, j for crossbar X_1 (in black) are prepared in crossbar X_0 (in gray).

2) Node propagation: A node $u_1 \in U_1$ may appear in multiple crossbars X_i among multiple staircases S_j. From the structure of a pruned graph G, we know that each node $v \in V$ has at most two outgoing edges. At some point, the node will be *realized*, i.e., its two outgoing edges have been assigned. Let that point be denoted as X_r. From this point X_r forward, any other occurrence of v is to realize incoming edges of v. When v occurs at some later point in the same staircase $X_i, i > r$, we must propagate v to that crossbar X_i. This is illustrated in Figure 8(b) where node v is realized in crossbar X_1 and propagated to crossbar X_4. The intermediary nodes v' are in gray, and the start and end point are in black. Similarly, when v occurs in multiple staircases, then node v must be propagated from its point of realization X_r to all other staircases.

3) Literal propagation: A literal l may appear in a crossbar $X_i, i \geq 2$. For each such literal l, we must propagate the literal up to layer X_{i-2}. For example, in Figure 8(c), the literal l appears in crossbar X_4, and is thus propagated from the first crossbar X_1 to the last crossbar X_4.

Algorithm 2 Binary search over the threshold

```
Input: $B = (U_1, U_2, F)$, $D$
Output: $\mathcal{T}$ // Set of staircase designs
 1: function BinarySearch($T$)
 2:     $low \leftarrow 0, high \leftarrow D$
 3:     $\mathcal{T} \leftarrow \emptyset$
 4:     $T \leftarrow \lfloor (low + high)/2 \rfloor$
 5:     while $low \neq high$ do
 6:         $\mathcal{T}' \leftarrow$TopologicalStaircasePartitioning($B, T$)
 7:         if $\mathcal{T}' \neq \emptyset$ then // Solution found
 8:             $\mathcal{T} \leftarrow \mathcal{T}'$
 9:             $low \leftarrow T$ // Increase lower bound
10:         else // No solution found
11:             $high \leftarrow T$ // Decrease upper bound
12:         end if
13:         $T \leftarrow \lfloor (low + high)/2 \rfloor$
14:     end while
15:     return $\mathcal{T}$
16: end function
```

VIII. Additional optimization

In this section, we introduce two additional optimization steps. In Section VIII-A, we propose a method to improve the search over the user-defined threshold T, and in Section VIII-B, we propose an optimization to have a more fine-grained exploration of the search space.

A. Partitioning search

The partitioning algorithm presented in Section VII requires a user-defined parameter T, which is a threshold for the amount of logic that will be placed in a crossbar X. As this variable is unknown in advance, we propose a binary search over T. Let all crossbars X_i in a staircase structure have the same dimensions $D \times D$. In Algorithm 2, we provide the binary search algorithm for the topological staircase partitioning. The input is the bipartite graph $B = (U_1, U_2, F)$, and the dimensions D of the crossbars. The output is a topology $\mathcal{T}$. The idea is that when for a given threshold T, no solution can be found, we decrease the threshold T. Potentially, no solution is found due to the intra- and inter-connections explained in Section VII-B. The node propagations, literal propagations, and edge preparations may result in that a crossbar exceeds its dimensions while constructing, and consequently the partitioning algorithm fails to find a solution for the given constraints. In the other case, when for a given $\mathcal{T}$, a solution can be found, we retain this solution and attempt to find a better solution by increasing the threshold $\mathcal{T}$.

B. Node splitting

In this section, we provide an optimization step to improve the overall synthesis. Due to the node merging optimization laid out in previous Section VI-C, the node degree for all nodes $u_2 \in U_2$ may increase. The partitioning algorithm in Algorithm 1 assigns such nodes u_2 and its neighboring nodes $u_1 \in U_1$ to a single crossbar in a staircase. When the node degree of u_2, $\delta(u_2)$, is greater than the logic threshold T, such node cannot be assigned to a crossbar. A solution would be to increase the threshold T, but this brings with it that there is less room for node propagations, literal propagations, and edge preparations. Hence, there is a fine balance which must be sought between the threshold T and the node degree $\delta(u_2)$. Therefore, we propose to split nodes $u_2 \in U_2$ for which $\delta(u_2) > T$ into two nodes u_2^1 and u_2^2.

In Algorithm 3, we present an algorithm to cope with such nodes. The algorithm can be used in combination with Algorithm 2. More specifically, line 6 in Algorithm 2 can be replaced with $\mathcal{T}' \leftarrow$SPLITWRAPPER(B, T).

Algorithm 3 consists of two parts: SPLITWRAPPER(B, T), and an auxiliary function SPLITNODE(B, T). The former continues to split nodes $u_2^* \in U_2$ with maximum degree $\delta(u_2^*)$ as long as a node in B is changed (line 16). We use the auxiliary function SPLITNODE(B, T) to perform this operation. On line 2, we seek such a node $u_2^* \in U_2$ with maximum node degree $\delta(u_2^*)$. When this node degree is smaller than the threshold, we do not need to split. Hence we return our current bipartite graph B (lines $3-5$). Otherwise, we create a new bipartite graph B' where u_2^* is replaced by two new nodes u_2^1, and u_2^2 such that its number of edges is equal, or differs at most by one edge (lines $6-12$).

Algorithm 3 Node splitting algorithm

```
Input: $B = (U_1, U_2, F)$, $T$
Output: $\mathcal{T}$ // Set of staircase designs
 1: function SPLITNODE($B$, $T$)
 2:     $u_2^* \leftarrow \arg\max \delta(u_2), \forall u_2 \in U_2$
 3:     if $\delta(u_2^*) \leq T$ then
 4:         return $B$
 5:     end if
 6:     $X \leftarrow \{u_1 | (u_1, u_2^*) \in F\}$
 7:     $Y \leftarrow \{u_1 | (u_2^*, u_1) \in F\}$
 8:     $U_1' \leftarrow U_1$
 9:     $U_2' \leftarrow U_2 \setminus \{u_2^*\} \cup \{u_2^1, u_2^2\}$
10:     $F' \leftarrow \{(u_1, u_2^1) | u_1 \in X\} \cup \{(u_1, u_2^2) | u_1 \in X\} \cup$
                $\{(u_1^1, u_2^1), (u_1^2, u_2^2) | u_1^1 \in Y_1, u_1^2 \in Y_2,$
                $Y_1 \subseteq Y, Y_2 \subseteq Y, ||Y_1| - |Y_2| \leq 1|\}$
11:     $B' \leftarrow (U_1', U_2', F')$
12:     return $B'$
13: end function

14: function SPLITWRAPPER($B$, $T$)
15:     $B' \leftarrow \emptyset, \mathcal{T} \leftarrow \emptyset$
16:     while $B' \neq B$ do
17:         $\mathcal{T} \leftarrow$TOPOLOGICALSTAIRCASEPARTITIONING($B, T$)
18:         if $\mathcal{T} \neq \emptyset$ then
19:             return $\mathcal{T}$
20:         else
21:             $B' \leftarrow$SPLITNODE($B$, $T$)
22:         end if
23:     end while
24:     return $\mathcal{T}$
25: end function
```

IX. EXPERIMENTAL EVALUATION

The experiments are conducted on a machine with 20 Intel Core i9-9900X and 128GB RAM. The framework is implemented in Python 3.8 and the source code is publicly available on GitHub[1]. We use the ABC [56] binding for CUDD [54] to construct the BDDs with dynamic variable reordering based on symmetric sifting [57]. In Table II, an overview is provided of ten benchmarks from the Revlib benchmark suite [58], eight control benchmarks from the EPFL benchmark suite [59], and eight ISCAS85 benchmarks [60]. We report the number of inputs, outputs for each benchmark, as well as the number of nodes and edges for the respective BDD.

We evaluate the path-based computing systems by building an architectural model. In Figure 9, we illustrate the high-level architecture. The architecture consists of several tiles T on a bank, as illustrated in Figure 9(a). Each tile T has a H-Tree of staircases S as topology [61], [62], as shown in Figure 9(b). Four staircases with an I/O of 128 bits are connected to a Wide-I/O bus. Each staircase S is a series of crossbars X, as illustrated in Figure 9(c).

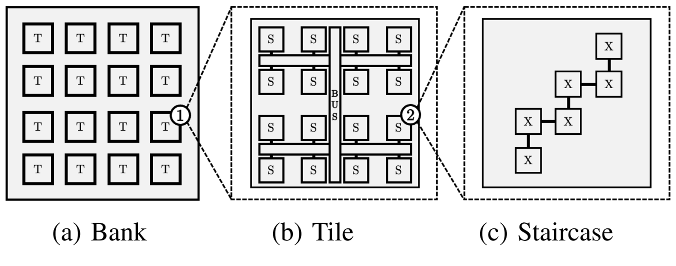


Fig. 9. High-level overview of the architecture. (a) A bank consists of multiple tiles T. (b) Each tile T contains multiple staircases S. The topology of the staircases is according to a H-Tree. (c) Each staircase S contains a series of crossbars X.

In our experimental evaluation, we will compare the performance of the proposed PATH framework with COMPACT [16], ArC [50], and CONTRA [12]. The performance is compared in terms of energy, latency, and area. The parameters for the comparisons are given below. To evaluate our architecture, we set the power consumption for the bus and 128x128 crossbar to 13mW, 0.3mW [38]. For our design, we have a 4-

[1]https://github.com/sventhijssen/path

TABLE II
OVERVIEW OF TEN REVLIB BENCHMARKS, EIGHT EPFL CONTROL BENCHMARKS, AND EIGHT ISCAS85 BENCHMARKS. FOR EACH BENCHMARK, THE NUMBER OF INPUTS AND OUTPUTS IS GIVEN. FOR THEIR RESPECTIVE BDDS, THE NUMBER OF NODES AND EDGES IS GIVEN.

Benchmark	Benchmark		BDD		Pruned	
	Inputs (num)	Outputs (num)	Nodes (num)	Edges (num)	Nodes (num)	Edges (num)
Revlib						
in0	15	11	385	766	384	680
apex2	39	3	567	1130	566	1042
spla	16	46	594	1184	593	864
pdc	16	40	621	1238	620	887
misex3	14	14	674	1344	673	1094
tial	14	8	897	1790	896	1717
apex4	9	19	990	1976	989	1874
cps	24	109	1080	2156	1079	1633
apex5	117	88	1259	2514	1258	2387
seq	41	35	1302	2600	1301	2041
EPFL						
arbiter	256	129	25109	50214	25108	49758
cavlc	10	11	436	868	435	776
ctrl	7	26	89	174	88	128
dec	8	256	512	1020	511	510
i2c	147	142	1204	2404	1203	1936
int2float	11	7	159	314	158	301
priority	128	8	772	1540	771	1539
router	60	30	219	434	218	379
ISCAS85						
c432	36	7	1291	2578	1290	2463
c499	41	32	111146	222164	111114	212466
c880	60	26	5776	11448	5750	11151
c1355	41	32	111146	222164	111114	212466
c1908	33	25	30605	61110	30580	57308
c2670	233	140	8249	15940	8109	14621
c5315	178	123	15454	30416	15331	27477
c7552	207	108	33983	67534	33875	65400
Normalized			1.00	1.00	1.00	**0.85**

channel 128-bit Wide-IO bus with a rate of 400MHz [63]. The area for the respective components are $0.2\mu\text{m}^2$, 15.7mm^2 [38]. For COMPACT, we extrapolate the area. The latency for the bus and crossbar components are 15ns and 100ns, respectively.

In Section IX-A, we first evaluate the crossbar synthesis. Then, in Section IX-B, we evaluate the proposed PATH framework, including the proposed node merging and partitioning algorithm. Finally, in Section IX-C, we make a comparison of the PATH framework with other digital in-memory computing paradigms.

A. Evaluation of crossbar synthesis

In this section, we will evaluate the crossbar synthesis. For the evaluation, we do not impose any restrictions on the crossbar dimensions, such that the number of wordlines (rows), and the number of bitline-selectorline pairs (columns) can be infinitely large. We evaluate the crossbar synthesis first without and then with the proposed node merging. In Table III, we provide the number of nodes and edges for the pruned graph G, as well as the hardware resources for both approaches. For the synthesis without node merging, we observe that the number of rows and the number of columns correspond to the number of nodes and edges of the pruned graph, respectively. This is due to the analogy between BDDs and 1T1M crossbars. Next, we report the number of rows and columns for the approach with node merging. We observe that the number of columns (selectorline-bitlines pairs) reduces by 16% on average, resulting in an area reduction of 16% on average. From this, we conclude that it is advantageous to work with the compressed bipartite graph B', and we will use this graph in subsequent sections. Thus, a BDD with $|V|$ nodes and $|E|$ edges can be synthesized into a crossbar of dimensions $|V| \times |E|$, which is an upper bound. Empirically, we conclude that on average a BDD with $|V|$ nodes and $|E|$ edges can be synthesized into a crossbar of dimensions $|V| \times 0.84|E|$.

B. Evaluation of the PATH framework

In this section, we evaluate the PATH framework. In our first experiment, we evaluate the hardware resources for varying staircase depth L, i.e., the number of crossbars in a staircase structure. These hardware resources are the number of staircases, the number of staircase inter-connections, and the critical path length. In Table IV, we give an overview of these hardware resources as well as the synthesis time for varying staircase depths $L \in \{1, 2, 4, 6\}$.

We observe that the number of required staircases decreases when the staircase depth L increases, with a reduction of 24% on average for a staircase structure of six layers compared with a single crossbar. For example, for the benchmark arbiter of the EPFL benchmark suite, the number of staircases reduces from 889 for $L = 1$ to 691 for $L = 6$. The number of inter-connections may increase or decrease, depending on the benchmark. For example, for arbiter, the number of inter-connections increases from $49,973$ to for $L = 1$ to $51,035$ for $L = 6$. This is due to that the logic threshold tends to be lower for larger staircase structures, requiring more node splits, and consequently more node propagations. However, for the majority of the benchmarks (17 out of 26), the number of inter-crossbar connections decreases, with a reduction of 8% on average for six layers compared with a single crossbar. For example, for benchmark cavlc of the Revlib benchmark suite, the number of staircase inter-connections decreases from 610 for $L = 1$ to 566 for $L = 6$. Finally, we observe that the critical path length reduces by 17% on average for $L = 6$ compared with $L = 1$. The reduction of the number of staircases brings with it that the critical path length decreases. This is because the critical path length is at most the number of staircases, and the number of staircases for $L = 6$ is lower than the number of staircases for $L = 1$. From these results, we conclude it is best to utilize a path-based computing system with larger staircase structures.

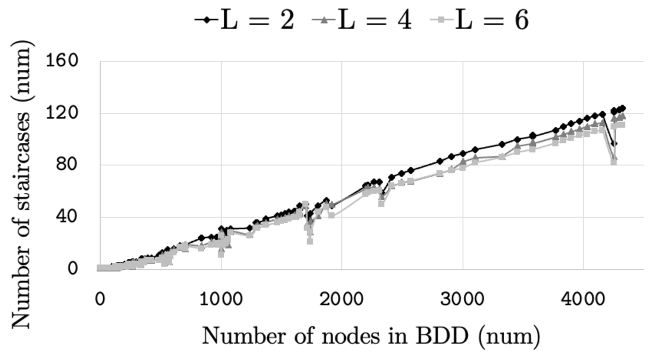


Fig. 10. Number of staircases in terms of BDD size for eight ISCAS85 benchmarks.

Next, we make an analysis of the hardware utilization in terms of the intermediate data structure. More specifically,

TABLE IV
COMPARISON OF THE HARDWARE RESOURCES AND SYNTHESIS TIME (T) FOR VARYING PATH-BASED COMPUTING ARCHITECTURES. THE HARDWARE RESOURCES ARE EXPRESSED IN TERMS OF NUMBER OF STAIRCASES (S), NUMBER OF INTER-CONNECTIONS (I), AND CRITICAL PATH LENGTH (C).

Benchmark	L = 1				L = 2				L = 4				L = 6			
	S (num)	I (num)	C (num)	T (min)	S (num)	I (num)	C (num)	T (min)	S (num)	I (num)	C (num)	T (min)	S (num)	I (num)	C (num)	T (min)
Revlib																
in0	11	547	11	0.1	10	527	10	0.2	9	526	9	0.4	9	529	9	0.4
apex2	16	767	16	0.1	13	706	13	0.2	11	630	11	0.2	10	579	10	0.4
spla	14	694	14	0.2	12	666	12	0.3	9	513	9	0.2	7	430	7	0.2
pdc	14	645	14	0.2	12	622	12	0.2	9	526	9	0.2	8	457	8	0.2
misex3	16	835	15	0.2	15	839	15	0.2	13	819	13	0.3	12	814	12	0.4
tial	27	1422	23	0.2	23	1404	21	0.4	21	1429	19	0.3	21	1414	19	0.8
apex4	31	1683	31	0.2	27	1693	26	0.5	25	1686	24	1.0	25	1698	24	1.0
cps	27	1393	25	0.4	24	1388	23	0.5	20	1368	20	0.4	20	1364	20	0.7
apex5	47	2077	27	0.4	36	2060	26	0.5	34	2102	28	0.9	33	2110	27	0.9
seq	37	1826	25	0.4	31	1821	27	0.6	28	1811	22	1.1	28	1831	27	0.9
EPFL																
arbiter	889	49973	302	8.6	762	50348	314	12.7	717	50546	311	17.8	691	51035	307	15.5
cavlc	11	610	11	0.0	11	627	11	0.0	9	593	9	0.0	9	593	9	0.1
ctrl	1	0	1	0.0	1	0	1	0.0	1	0	1	0.0	1	0	1	0.0
dec	6	192	4	0.9	6	202	4	0.0	5	206	4	0.1	5	196	4	0.1
i2c	32	1616	25	0.4	30	1664	27	1.6	28	1639	25	0.3	29	1647	25	0.5
int2float	4	146	4	0.0	3	115	3	0.0	3	106	3	0.0	2	67	2	0.0
priority	19	495	18	0.1	15	425	14	0.3	14	426	14	0.0	16	481	16	4.3
router	4	87	4	0.0	4	95	4	0.1	4	99	4	0.0	4	97	4	0.2
ISCAS85																
c432	40	2121	40	0.3	36	2086	36	0.5	33	2071	33	1.6	32	2049	32	1.0
c499	3592	160724	108	26.4	3212	163939	101	32.7	3020	165236	91	106.9	2884	161093	88	82.0
c880	189	8004	43	1.5	167	7931	43	4.5	155	7750	42	4.9	150	7666	43	4.8
c1355	3592	160724	108	26.1	3212	163939	101	32.0	3020	165236	91	112.1	2884	161093	88	80.4
c1908	939	39903	50	8.7	835	39959	42	9.5	765	40000	45	31.3	731	38959	43	24.9
c2670	352	8576	45	2.7	314	8235	41	5.1	295	7914	40	8.1	272	7199	41	11.3
c5315	393	10699	25	3.9	353	10114	22	7.0	293	8694	20	7.8	277	7834	19	10.6
c7552	1032	35470	108	7.7	894	34416	87	14.6	779	32417	82	21.2	694	30593	78	365.2
Normalized	1.00	1.00	1.00	**1.00**	0.89	0.98	0.92	1.72	0.80	0.95	0.85	2.27	**0.76**	**0.92**	**0.83**	5.66

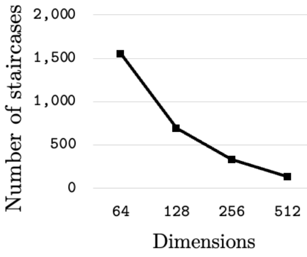
TABLE III
THE NUMBER OF NODES AND EDGES FOR THE PRUNED GRAPH G, AND THE CORRESPONDING HARDWARE RESOURCES FOR A CROSSBAR DESIGN USING THE UNCOMPRESSED AND COMPRESSED BIPARTITE GRAPH.

Benchmark	Pruned graph		Without node merging		With node merging	
	Nodes (num)	Edges (num)	Rows (num)	Cols (num)	Rows (num)	Cols (num)
Revlib						
in0	384	680	384	680	384	565
apex2	566	1042	566	1042	566	879
spla	593	864	593	864	593	767
pdc	620	887	620	887	620	750
misex3	673	1094	673	1094	673	849
tial	896	1717	896	1717	896	1143
apex4	989	1874	989	1874	989	1157
cps	1079	1633	1079	1633	1079	1248
apex5	1258	2387	1258	2387	1258	2132
seq	1301	2041	1301	2041	1301	1560
EPFL						
arbiter	25108	49758	25108	49758	25108	41441
cavlc	435	776	435	776	435	530
ctrl	88	128	88	128	88	100
dec	511	510	511	510	511	510
i2c	1203	1936	1203	1936	1203	1837
int2float	158	301	158	301	158	265
priority	771	1539	771	1539	771	1539
router	218	379	218	379	218	351
ISCAS85						
c432	1290	2463	1290	2463	1290	1929
c499	111114	212466	111114	212466	111114	198936
c880	5750	11151	5750	11151	5750	8540
c1355	111114	212466	111114	212466	111114	198936
c1908	30580	57308	30580	57308	30580	53100
c2670	8109	14621	8111	14622	8111	13337
c5315	15331	27477	15331	27477	15331	23850
c7552	33875	65400	33875	65400	33875	53708
Normalized avg.	1.00	1.00	1.00	1.00	**1.00**	**0.84**

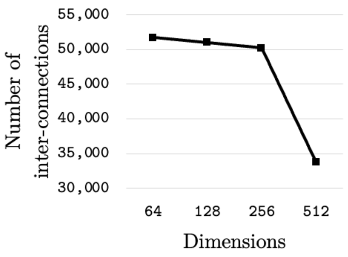
in Figure 10, we show the number of required staircases in terms of the number of BDD nodes for different staircase depths. The crossbar dimensions are 128x128, and the BDDs are collected from the eight ISCAS85 benchmarks. From this figure, we clearly observe there is a linear trend between these two dimensions. Further, we observe at first glance that the number of required staircases decreases for increasing staircase depth (the line for $L = 2$ lies higher than for $L = 4$, and $L = 4$ lies higher than for $L = 6$). This corresponds with the results in Table IV. For $L = 2$, $L = 4$, and $L = 6$, the trendline is described by the following equations, respectively: $0.0285x - 0.0126$, $0.0268x - 0.02782$, and $0.0255x - 0.3139$ where x is the number of BDD nodes.

Now, we will evaluate the PATH framework in terms of the crossbar dimensions. We evaluate on the benchmark arbiter using a staircase depth of six crossbars. In Figure 11(a), we show the trendline for the number of staircases in function of the crossbar dimensions. We observe that for increasing crossbar dimensions, the number of staircases decreases. This is expected as there is more room in a crossbar for both logic and node propagations. In Figure 11(b), we observe that the number of inter-connections decreases as the crossbar dimensions increase. This is also expected as more logic can be realized within a single staircase, and thus less inter-staircase communication is required.

In Section VII-B, we have highlighted that the partitioning method requires some intra- and inter-connections in order to be a functional computing paradigm. We make an analysis of the components that constitute to the overall synthesis using partitioning. These components are logic, edge preparation, node propagation, and literal propagation. This analysis may give further insight in the synthesis method with the objective to improve any future work on our proposed framework.



(a) Staircases



(b) Inter-connections

Fig. 11. Number of staircases and inter-connections for the EPFL benchmark arbiter for varying dimensions D.

In Figure 12, we show the percentages for each of the components for varying number of layers L where $L \in \{1, 2, 4, 6\}$. We observe that the percentage of logic, which is defined by the threshold T, decreases for increasing number of layers L. This is due to that the number of node propagations increases when the number of layers L increases, which can also be seen in the figure. As mentioned earlier in Section VIII-B, there is a fine balance between the threshold T and the node degree $\delta(u_2)$. When the node degree decreases, the number of node propagations increase, and vice versa.

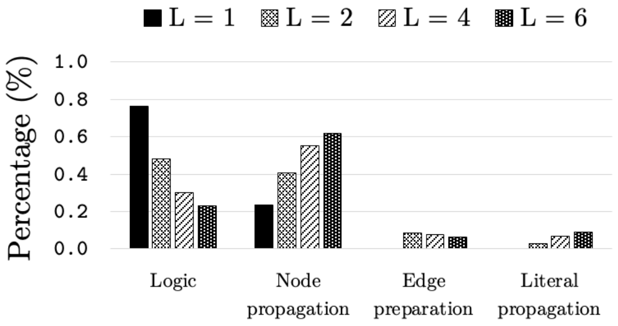


Fig. 12. Percentage for each of the components in a staircase topology using the PATH framework. These components include logic, node propagation, edge preparation, and literal propagation. The percentages are the averages over the ten Revlib, eight EPFL benchmarks, and eight ISCAS85 benchmarks.

C. Comparison with other digital in-memory computing paradigms

In this section, we make a comparison of the path-based computing paradigm with other digital in-memory computing paradigms. More specifically, we compare with COMPACT [16], ArC [50], and CONTRA [12]. COMPACT is the state-of-the-art synthesis method for flow-based computing, ArC for MAJORITY, and CONTRA is the state-of-the-art MAGIC-based general-purpose synthesis method. No comparison is provided with IMPLY based logic [9], [43] as recent papers have shown that IMPLY-based logic is inferior to MAGIC-based logic [46], [64].

In Figure 13, a detailed comparison is given for the normalized energy consumption, latency, and area for all benchmarks (except spla and pdc as CONTRA failed to generate a result) using PATH, COMPACT, and CONTRA. For ArC, only the ISCAS85 benchmarks are reported based on the results in [12]. Note that the latency only reflects the execution time, i.e., the runtime to evaluate Boolean functions, and does not include the synthesis time as reported in Table IV nor the crossbar programming. Compared with PATH, COMPACT requires approximately $1006\times$ more energy, and has approximately $10\times$ longer latency on average. The advantageous performance mainly stems from that COMPACT is a flow-based computing framework where the devices are continuously switched for each evaluation, resulting in many expensive (in terms of energy and latency) WRITE operations. No partitioning scheme exists for COMPACT, so we extrapolated the crossbar size of a 128x128 crossbar to the required dimensions. For COMPACT, some benchmarks require more area than PATH, so we have truncated the plot at unity for clarity (e.g. arbiter has $4\times$ area). On average, COMPACT requires $5.8\times$ of the area of PATH. Further, we observe that CONTRA consumes approximately $2166\times$ more energy and is approximately $15\times$ slower than PATH on average. Similarly to previous argument, CONTRA is much less energy-efficient and slower than PATH due to the large number of write operations. The path-based paradigm only utilizes WRITE operations in the compilation phase, which is amortized across many function evaluations. On average, CONTRA requires only 2% of the area of PATH. Lastly, ArC requires on average $175.96\times$ more energy than PATH and is $8.30\times$ slower than PATH due to the many WRITE operations.

X. Summary and future work

In this paper, we have introduced a new READ-based in-memory computing paradigm, called path-based computing, by leveraging access transistors to perform logic. We have introduced a framework, called PATH, to automatically synthesize Boolean circuits to path-based computing systems. The PATH framework relies on an analogy between bipartite graphs and 1T1M crossbars. The bipartite graphs are derived from BDDs, and serve as an intermediate data representation. Further, we have introduced an optimization technique wherein these bipartite graphs are compressed, resulting in an area reduction of 16%. Finally, we have introduced a partitioning algorithm to map Boolean functions to a topology of staircase structures, where a staircase structure is an ordered set of crossbars, which have hardwired connections between them. By introducing staircases, the bus utilization diminishes, which results in high energy and latency improvements. Our experimental results demonstrate that the paradigm is orders of magnitude faster than state-of-the-art in-memory computing paradigms with energy improvements of $1006\times$ on average. The latency improvements are $10\times$ on average. For future work, we envision that leveraging alternative intermediate data structures may improve the overall synthesis method. Further, alternative or orthogonal approaches to our proposed partitioning algorithm are an interesting trajectory for further research.

References

[1] D. R.-J. G.-J. Rydning, "The digitization of the world from edge to core," *Framingham: International Data Corporation*, p. 16, 2018.
[2] S. Li, L. D. Xu, and S. Zhao, "The internet of things: a survey," *Information systems frontiers*, vol. 17, pp. 243–259, 2015.
[3] I. Tomkos, D. Klonidis, E. Pikasis, and S. Theodoridis, "Toward the 6g network era: Opportunities and challenges," *IT Professional*, vol. 22, no. 1, pp. 34–38, 2020.
[4] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

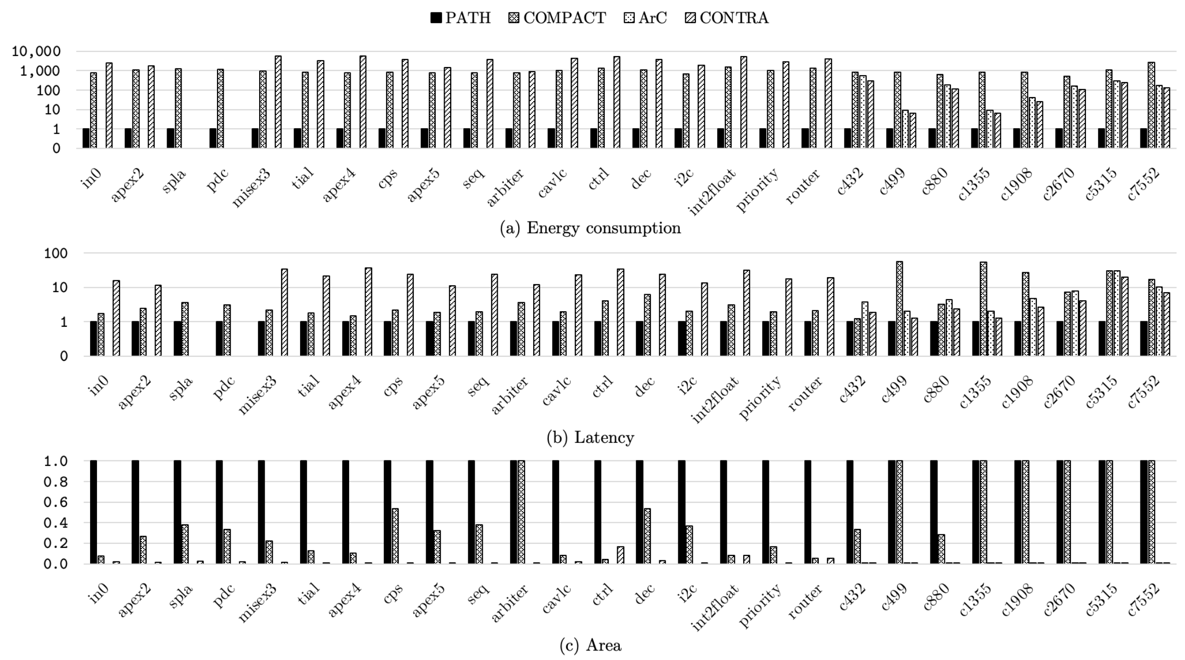


Fig. 13. Normalized energy, latency,and area for PATH, COMPACT [16], ArC [50], and CONTRA [12] for ten Revlib benchmarks, eight EPFL benchmarks, and eight ISCAS85 benchmarks.

[5] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[6] J. Backus, "Can programming be liberated from the von neumann style?," *CACM*, vol. 21, no. 8, pp. 613–641, 1978.

[7] T. N. Theis and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology," *CiSE*, vol. 19, no. 2, pp. 41–50, 2017.

[8] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *ISCA'11*, pp. 365–376, IEEE, 2011.

[9] E. Lehtonen, J. Poikonen, and M. Laiho, "Implication logic synthesis methods for memristors," in *ISCAS'12*, pp. 2441–2444, IEEE, 2012.

[10] S. Shirinzadeh, M. Soeken, and R. Drechsler, "Multi-objective bdd optimization for rram circuit design," in *IEEE DDECS 2016*, pp. 1–6, 2016.

[11] S. Kvatinsky *et al.*, "Magic—memristor-aided logic," *IEEE TCAS-II*, vol. 61, no. 11, pp. 895–899, 2014.

[12] D. Bhattacharjee *et al.*, "Contra: area-constrained technology mapping framework for memristive memory processing unit," in *ICCAD'20*, pp. 1–9, 2020.

[13] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (plim) computer," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 427–432, Ieee, 2016.

[14] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 948–953, IEEE, 2016.

[15] A. Velasquez and S. Jha, "Automated synthesis of crossbars for nanoscale computing using formal methods," in *NANOARCH'15*, pp. 130–136, IEEE, 2015.

[16] S. Thijssen *et al.*, "Compact: Flow-based computing on nanoscale crossbars with minimal semiperimeter," in *DATE*, pp. 232–237, IEEE, 2021.

[17] R. Ronen, A. Eliahu, O. Leitersdorf, N. Peled, K. Korgaonkar, A. Chattopadhyay, B. Perach, and S. Kvatinsky, "The bitlet model: A parameterized analytical model to compare pim and cpu systems," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 2, pp. 1–29, 2022.

[18] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[19] G. W. Burr *et al.*, "Phase change memory technology," *JVST B*, vol. 28, no. 2, pp. 223–262, 2010.

[20] Y. Huai *et al.*, "Spin-transfer torque mram (stt-mram): Challenges and prospects," *AAPPS bulletin*, vol. 18, no. 6, pp. 33–40, 2008.

[21] M. Hu *et al.*, "Memristor-based analog computation and neural network classification with a dot product engine," *Advanced Materials*, vol. 30, no. 9, p. 1705914, 2018.

[22] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 802–815, 2019.

[23] M. R. H. Rashed, S. K. Jha, and R. Ewetz, "Hybrid analog-digital in-memory computing," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2021.

[24] D. Ielmini and H.-S. P. Wong, "In-memory computing with resistive switching devices," *Nature Electronics*, vol. 1, no. 6, pp. 333–343, 2018.

[25] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, pp. 171–243, Springer, 2022.

[26] W. Haensch, A. Raghunathan, K. Roy, B. Chakrabarti, C. M. Phatak, C. Wang, and S. Guha, "Compute in-memory with non-volatile elements for neural networks: A review from a co-design perspective," *Advanced Materials*, p. 2204944, 2022.

[27] C. Xu *et al.*, "Overcoming the challenges of crossbar resistive memory architectures," in *HPCA'15*, pp. 476–488, IEEE, 2015.

[28] Z. Liao, J. Fu, and J. Wang, "Ameliorate performance of memristor-based anns in edge computing," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1299–1310, 2021.

[29] Z. Zhao, D. Liu, M. Li, Z. Ying, L. Zhang, B. Xu, B. Yu, R. T. Chen, and D. Z. Pan, "Hardware-software co-design of slimmed optical neural networks," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 705–710, 2019.

[30] C. Feng, J. Gu, H. Zhu, Z. Ying, Z. Zhao, D. Z. Pan, and R. T. Chen, "A compact butterfly-style silicon photonic–electronic neural chip for hardware-efficient deep learning," *ACS Photonics*, vol. 9, no. 12, pp. 3906–3916, 2022.

[31] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell, *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.

[32] G. Li, A. Wu, Y. Shi, A. Javadi-Abhari, Y. Ding, and Y. Xie, "On the co-design of quantum software and hardware," in *Proceedings of the Eight Annual ACM International Conference on Nanoscale Computing and Communication*, pp. 1–7, 2021.

[33] H. Kim, Y. Kim, S. Ryu, and J.-J. Kim, "Algorithm/hardware co-design for in-memory neural network computing with minimal peripheral circuit overhead," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[34] M. R. H. Rashed, S. Thijssen, S. K. Jha, F. Yao, and R. Ewetz, "Stream: Towards read-based in-memory computing for streaming based data processing," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 690–695, IEEE, 2022.

[35] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a bdd package," in *DAC'90*, pp. 40–45, IEEE, 1990.

[36] S.-i. Minato *et al.*, "Shared binary decision diagram with attributed edges for efficient boolean function manipulation," in *DAC'90*, pp. 52–57, IEEE, 1990.

[37] M. Wang *et al.*, "A selector device based on graphene–oxide heterostructures for memristor crossbar applications," *Appl. Phys. A*, vol. 120, no. 2, pp. 403–407, 2015.

[38] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH*, vol. 44, no. 3, pp. 14–26, 2016.

[39] A. Ranjan, S. Jain, J. R. Stevens, D. Das, B. Kaul, and A. Raghunathan, "X-mann: A crossbar based architecture for memory augmented neural networks," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[40] Z. Zhu, H. Sun, K. Qiu, L. Xia, G. Krishnan, G. Dai, D. Niu, X. Chen, X. S. Hu, Y. Cao, *et al.*, "Mnsim 2.0: A behavior-level modeling tool for memristor-based neuromorphic computing systems," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, pp. 83–88, 2020.

[41] J. Solanki, K. Beckmann, J. Pelton, N. Cady, and M. Liehr, "Effect of resistance variability in vector matrix multiplication operations of 1t1r reram crossbar arrays using an embedded test platform," in *2023 IEEE 32nd Microelectronics Design & Test Symposium (MDTS)*, pp. 1–5, IEEE, 2023.

[42] M. Liehr, J. Hazra, K. Beckmann, S. Rafiq, and N. Cady, "Impact of switching variability of 65nm cmos integrated hafnium dioxide-based reram devices on distinct level operations," in *2020 IEEE International Integrated Reliability Workshop (IIRW)*, pp. 1–4, IEEE, 2020.

[43] S. Kvatinsky *et al.*, "Memristor-based material implication (imply) logic: Design principles and methodologies," *IEEE TVLSI*, vol. 22, no. 10, pp. 2054–2066, 2013.

[44] S. Jha *et al.*, "Computation of boolean formulas using sneak paths in crossbar computing," Apr. 19 2016. US Patent 9,319,047.

[45] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'memristive'switches enable 'stateful'logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[46] R. B. Hur *et al.*, "Simple magic: Synthesis and in-memory mapping of logic execution for memristor-aided logic," in *ICCAD'17*, pp. 225–232, IEEE, 2017.

[47] S. Chakraborti, P. V. Chowdhary, K. Datta, and I. Sengupta, "Bdd based synthesis of boolean functions using memristors," in *2014 9th International Design and Test Symposium (IDT)*, pp. 136–141, IEEE, 2014.

[48] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2434–2447, 2019.

[49] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Logic synthesis for rram-based in-memory computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 7, pp. 1422–1435, 2017.

[50] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "Revamp: Reram based vliw architecture for in-memory computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 782–787, IEEE, 2017.

[51] D. Bhattacharjee, L. Amaŕu, and A. Chattopadhyay, "Technology-aware logic synthesis for reram based in-memory computing," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1435–1440, IEEE, 2018.

[52] D. Bhattacharjee, Y. Tavva, A. Easwaran, and A. Chattopadhyay, "Crossbar-constrained technology mapping for reram based in-memory computing," *IEEE Transactions on Computers*, vol. 69, no. 5, pp. 734–748, 2020.

[53] D. Chakraborty and S. Jha, "Automated synthesis of compact crossbars for sneak-path based in-memory computing," in *DATE*, pp. 770–775, IEEE, 2017.

[54] F. Somenzi, "Cudd: Cu decision diagram package-release 2.4. 0," *University of Colorado at Boulder*, 2012.

[55] D. B. West *et al.*, *Introduction to graph theory*, vol. 2. Prentice hall Upper Saddle River, 2001.

[56] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM CSUR*, vol. 24, no. 3, pp. 293–318, 1992.

[57] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams," in *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pp. 628–631, 1994.

[58] R. Wille *et al.*, "Revlib: An online resource for reversible functions and reversible circuits," in *ISMVL'08*, pp. 220–225, IEEE, 2008.

[59] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The epfl combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, no. CONF, 2015.

[60] F. Brglez, P. Pownall, and R. Hum, "Accelerated atpg and fault grading via testability analysis," in *Proceedings of IEEE Int. Symposium on Circuits and Systems*, pp. 695–698, 1985.

[61] S. Gudaparthi, S. Narayanan, R. Balasubramonian, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon, "Wire-aware architecture and dataflow for cnn accelerators," in *Proceedings of the 52nd annual ieee/acm international symposium on microarchitecture*, pp. 1–13, 2019.

[62] G. Krishnan, S. K. Mandal, C. Chakrabarti, J.-s. Seo, U. Y. Ogras, and Y. Cao, "Interconnect-aware area and energy optimization for in-memory acceleration of dnns," *IEEE Design & Test*, vol. 37, no. 6, pp. 79–87, 2020.

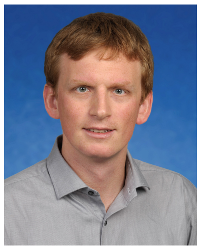
[63] T. Zhang, C. Xu, K. Chen, G. Sun, and Y. Xie, "3d-swift: A high-performance 3d-stacked wide io dram," in *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, pp. 51–56, 2014.

[64] P. L. Thangkhiew, R. Gharpinde, and K. Datta, "Efficient mapping of boolean functions to memristor crossbar using magic nor gates," *TCAS-I*, vol. 65, no. 8, pp. 2466–2476, 2018.

Sven Thijssen is a Ph.D. student in Computer Science at the University of Central Florida (UCF). Sven received his bachelor's degree in Informatics from KU Leuven, Belgium, in 2018, and his master's degree in Computer Science from UCF in 2021. His research interests are in-memory computing and beyond von Neumann computing. In 2020 he has received the ORCGS Doctoral Fellowship from UCF and in 2021 he has received a best paper nomination at DATE.

Muhammad Rashedul Haq Rashed is a Ph.D. candidate in Computer Engineering at the University of Central Florida (UCF). Rashed received his bachelor's degree in Electrical and Electronics Engineering from Bangladesh University of Engineering and Technology (BUET) in 2015. His research interests include EDA for emerging computing paradigms, computer-aided design, and artificial intelligence. He has received a best paper nomination at ICCAD 2022.

Sumit K. Jha is Professor of Computer Science at the University of Texas San Antonio (UTSA). Dr. Jha received his Ph.D. in Computer Science from Carnegie Mellon University. Before joining Carnegie Mellon, he graduated with B.Tech (Honors) in Computer Science and Engineering from the Indian Institute of Technology Kharagpur. Dr. Jha has worked on R&D problems at Microsoft Research India, General Motors, INRIA France and the Air Force Research Lab Information Directorate. His research has been supported by the National Science Foundation (NSF), DARPA, the Office of Naval Research (ONR), the Air Force Office of Scientific Research (AFOSR), the Oak Ridge National Laboratory (ORNL), the Royal Bank of Canada, the Florida Center for Cybersecurity, the Air Force Research Laboratory (AFRL), and National Nuclear Security Administration (NNSA). He is a full member of the Sigma Xi and is a recipient of the IEEE Orlando Engineering Educator Excellence Award. Dr. Jha was awarded the prestigious Air Force Young Investigator Award and his research has led to four Best Paper awards.



Rickard Ewetz received the M.S. degree, in Applied Physics and Electrical Engineering, from Linkopings Universitet in 2011. He received the Ph.D. degree in Electrical and Computer Engineering from Purdue University in 2016. Currently, he is an assistant professor in the Electrical and Computer Engineering Department at the University of Central Florida. His research interests include physical design and computer-aided design for in-memory computing using emerging technologies. He has best paper nominations from ASP-DAC 2019 and DATE 2021.