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

Sven Thijssen
sven.thijssen@knights.ucf.edu
University of Central Florida
Orlando, Florida, USA

Sumit Kumar Jha
sumit.jha@utsa.edu
University of Texas at San Antonio
San Antonio, Texas, USA

Rickard Ewetz
rickard.ewetz@ucf.edu
University of Central Florida
Orlando, Florida, USA

ABSTRACT

Processing in-memory breaks von Neumann-based constructs to accelerate data-intensive applications. Noteworthy efforts have been devoted 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 compile phase and a fast and efficient evaluation phase. The key property of the paradigm is that the execution phase only involves cheap READ operations. Moreover, a synthesis tool called PATH is proposed to automatically map computation to a single crossbar design. The PATH tool also supports the synthesis of path-based computing systems where the total number of crossbars and the number of inter-crossbar connections are minimized. We evaluate the proposed paradigm using 10 circuits from the RevLib benchmark suite. Compared with state-of-the-art digital in-memory computing paradigms, path-based computing improves energy and latency up to 4.7X and 8.5X, respectively.

1 INTRODUCTION

The rapidly growing number of sensor devices in the Internet of Things has increased the accessibility to digital data. The amount of available digital data is expected to reach 175 *ZB* by 2025 [16]. This has powered the emergence of data-driven applications such as decision-making [5], drug discovery [27], and deep neural networks [13]. Unfortunately, it is challenging to execute data-intensive application on today's high performance computing systems [14]. This is because the separation of memory and computing units within the von Neumann architecture introduces bandwidth-limited and power-hungry data transfer [3].

Processing in-memory using non-volatile memory has recently attracted significant attention. Non-volatile memory technology includes memristor, resistive random access memory (ReRAM) [2], phase change memory (PCM) [7], spin-transfer torque magnetic random access memory (STT-MRAM) [9]. By integrating the non-volatile memory devices into dense crossbar arrays, mathematical operations can be executed energy-efficiently with high speed. Analog in-memory computing is energy-efficient but it is limited to arithmetic operations [8]. In contrast, any Boolean function can be accelerated using digital in-memory computing. Digital in-memory computing can be performed using logic styles such as IMPLY [11], MAGIC [10], and FLOW [23]. The logic style defines the input and output of Boolean operations as analog voltages and states for the non-volatile memory devices. The system performance is largely dictated by the properties of the in-memory logic style.

Table 1: 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	[11, 18]	WRITE	WRITE+READ
MAGIC	[4, 10]	WRITE	WRITE+READ
FLOW	[22, 23]	WRITE	WRITE+READ
Path-based	**(this paper)**	WRITE	READ

Computation within in-memory paradigms can be divided into a one-time compile phase and an execution phase that is performed one-time for each function input. In Table 1, 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 [26]. In contrast, the proposed path-based paradigm evaluates Boolean logic using cheap READ operations in the execution phase.

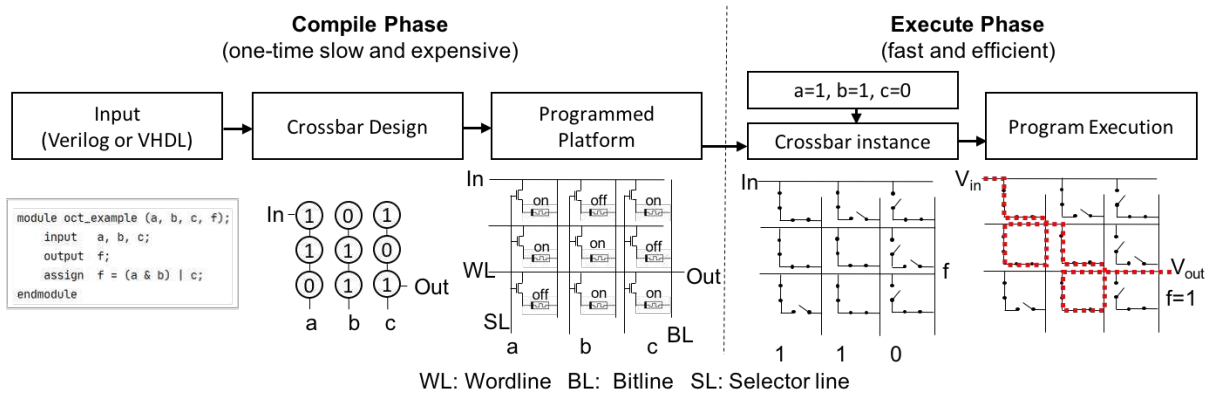
In this paper, we propose a new computing paradigm called path-based in-memory computing. The paradigm is capable of evaluating arbitrary Boolean functions using one-transistor one-memristor (1T1M) crossbar arrays. We also propose a framework called PATH to automatically map computation to 1T1M crossbars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

- A new computing paradigm called path-based in-memory computing. The paradigm executes Boolean functions fast and efficiently using only READ operations.
- The PATH framework exploits an analogy between Boolean decision diagrams (BDDs) and 1T1M crossbars to map Boolean functions into crossbar designs. A BDD with n nodes and m edges can be mapped to a crossbar of dimensions (n)x(m).
- The PATH framework maps larger Boolean functions to multiple connected crossbars (with constraints on the dimensions). The framework minimizes a weighted sum of the number of crossbars and the number of inter-crossbar connections.

The authors acknowledge support from NSF awards #2113307, #1755825, #1908471, and #2008339, the DARPA cooperative agreement #HR00112020002, ONR grant #N000142112332, and the DOE/NNSA.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530596



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

Figure 1: Flow for evaluating Boolean functions using 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 selector lines. (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.

- The experimental evaluation is performed on 10 circuits from the RevLib benchmark suite. Compared with the state-of-the-art in-memory paradigm, PATH improves energy and latency with at least 4.7X and 8.5X, respectively.

The remainder of the paper is organized, as follows: preliminaries are provided in Section 2. The path-based paradigm is introduced in Section 3. The problem formulation is given in Section 4. The crossbar-level synthesis framework is detailed in Section 5. The experimental evaluation is performed in Section 6. The paper is concluded in Section 7.

2 PRELIMINARIES

2.1 Binary Decision Diagrams

A binary decision diagram (BDD) is a graph representation of a Boolean function. The graph consist of internal decision nodes and two leaf nodes. The terminal nodes represent the output '0' and '1', respectively. The internal decision nodes each have an assigned Boolean variable and a positive and negative output edge. 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 [6]. 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 [12].

2.2 Memristor Crossbar Arrays

In this section, we will review one-transistor one-memristor (1T1M) crossbars and one-diode one-memristor (1D1M) crossbars [24]. The paradigm mainly relies on (1T1M) crossbars but (1D1M) crossbars are used to realize routers [15].

A 1T1M crossbar array consists of wordlines, bitlines, and selector lines. Each wordline is connected to each bitline using a series connected memristor and access transistor. The vertically aligned access transistor share a single selector line. 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 selector line is charged.

A 1D1M crossbar only has wordlines and bitlines. A crossbar with dimension (N)x(M) can be used to connect the N inputs to any of the M outputs by appropriately programming the the memristor devices, i.e., a NxM router.

3 PATH-BASED COMPUTING

Path-based computing aims to evaluate Boolean functions using in-memory computing. The flow is illustrated using an example in Figure 1. The flow for path-based computing consists of one-time slow and expensive compile phase and a fast and efficient execution phase. The input to the compile phase is a Boolean function specified in a hardware descriptive language (Verilog, VHDL), which is shown in Figure 1(a). The input is first synthesized into an abstract crossbar design $\mathcal{D}$, which is shown in Figure 1(b). The 1T1M crossbar design specifies the state of each non-volatile memory device (0/1) and the Boolean variable assigned to each selector line. The input and output assignment to the wordlines are also specified. Next, the memory devices within a nanoscale crossbar are programmed (off/on), which is shown in Figure 1(c).

In the execution phase, an instance of Boolean variables is provided to the selector lines. The selector lines control the switches represented by the access transistors. The state of the switches controlled by the memory devices are also shown in Figure 1(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 1(e). In contrast, the function evaluates to false for the input instance (1,0,0).

The one-time compile 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 execute phase is fast and efficient because it only involves charging/decharging the selector lines 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.

4 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:** Synthesize a Boolean function ϕ into a crossbar design $\mathcal{D}$. The objective is to minimize the dimensions of the synthesized crossbar design.
- **Problem II:** Synthesize a Boolean function ϕ into a path-based computing system consisting of multiple connected crossbars with fixed dimensions. The objective is to minimize a weighted sum of the number of crossbars and the number of inter-crossbar connections.

We approach the first problem by converting ϕ into a BDD. Next, we develop a one-to-one scheme of mapping a BDD into a crossbar design $\mathcal{D}$. The limitation is that there are dimensional constraints on 1T1M crossbars. To handle function ϕ that cannot fit into a single crossbar, we solve the second problem by partitioning the BDD into multiple parts such that each part can fit into a single crossbar. We perform the partitioning while minimizing the number of crossbars and the number of inter-crossbar connections.

5 THE PATH FRAMEWORK

In this section, we present the PATH framework that is capable of automatically mapping Boolean functions to a path-based computing platform with multiple connected crossbar designs $\mathcal{D}$. The framework consists of three main steps: graph pre-processing, crossbar partitioning, and crossbar synthesis. The flow of the framework is shown in Figure 2. The framework is illustrated with examples in Figure 3 and Figure 4.

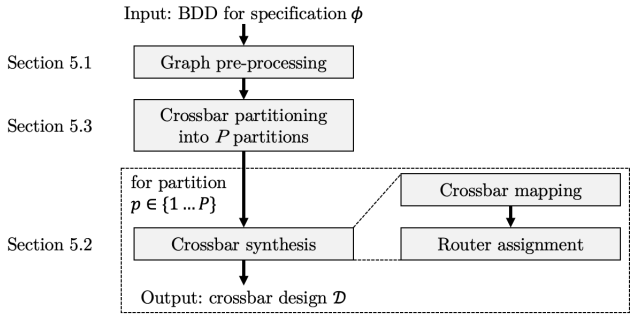


Figure 2: Flow of crossbar synthesis.

The input to the framework is a BDD of the function ϕ obtained using CUDD [19]. In the graph pre-processing step, the BDD is converted into an undirected graph G. The details are provided in Section 5.1. In the crossbar partitioning step, the graph is partitioned into P subgraphs such that each subgraph can fit into a crossbar with specified dimensions. The details of the partitioning are provided in Section 5.3. In the crossbar synthesis step, each subgraph partition is mapped into a crossbar design using a crossbar mapping and routing assignment step. The details are provided in Section 5.2.

We explain the crossbar synthesis step before the partitioning step because it provides us the guidelines for how the graph should be partitioned.

5.1 Graph pre-processing

The input to the graph pre-processing step is a BDD of ϕ. 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 $\bar{\phi}$. The one terminal node will be connected to the input, which we label *in*. The root node is labeled with *out*. 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, the edges are made unidirectional and the nodes are labeled from 1 to M. The resulting graph of the BDD in Figure 3(a) is shown in Figure 3(b).

5.2 Crossbar Synthesis

The crossbar synthesis step consists of a crossbar mapping step and a router assignment step. The crossbar mapping is used to convert the undirected graph G into a crossbar design $\mathcal{D}$ in Section 5.2.1. The router assignment is used to connect the primary inputs to the crossbar design using a 1D1M routing crossbar in Section 5.2.2.

The outlined mapping algorithm is based on an **analogy between graphs of BDDs and 1T1M crossbars**. The nodes and edges correspond to wordlines and bitline-selector line pairs. Each node in the graph G is assigned to a wordline. Each edge in G is realized using a bitline-selector line pair. 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, which are realized using the bitline-selector line pairs.

5.2.1 Crossbar mapping. The crossbar mapping step maps the graph G in Figure 3(b) into a crossbar design $\mathcal{D}$ shown in Figure 3(c). The crossbar mapping consists of a node assignment step and an edge assignment step. The node assignment involves assigning the nodes (in order) to the first M wordlines of the crossbar. Next, edge assignment is performed by appropriately assigning a state to the memristors and Boolean variables to the selector lines. Consider an edge $e_l \in E$ connecting nodes i and j with label x_k (or $\neg x_k$). The edge e_l will be realized using bitline l and selector line l. The label x_k (or $\neg x_k$) is assigned as the input to selector line l. All memristors along bitline l are programmed to be off except the memristors intersection with wordline i and j. The resulting crossbar of the graph in Figure 3(b) is shown in Figure 3(c).

5.2.2 Router assignment. The input to the router configuration step is the crossbar design $\mathcal{D}$. In this step, a 1D1M crossbar configured for routing is connected to the crossbar design $\mathcal{D}$. The key observation is that many input variables (or complemented input variables) are required to be provided to multiple selector lines. The routing crossbar takes one instance of the primary inputs and

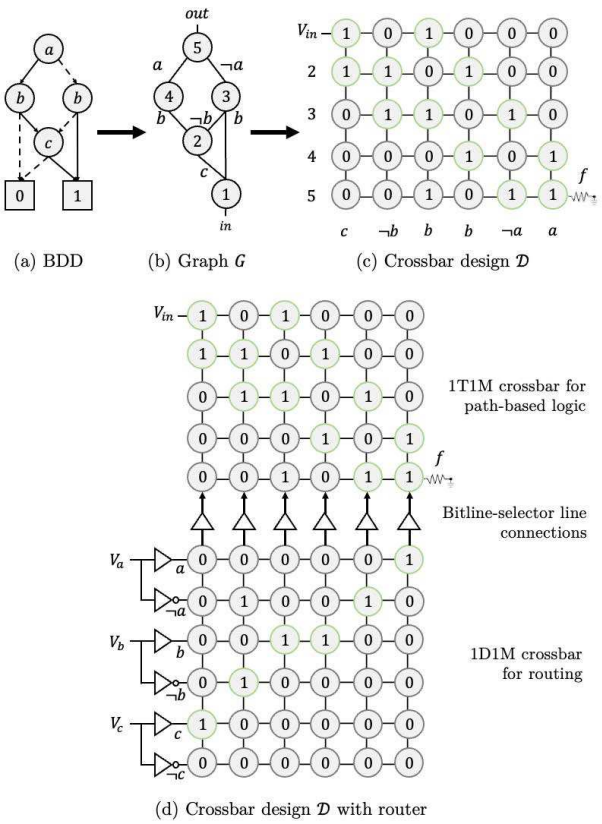


Figure 3: (a) BDD of ϕ. (c) Graph obtained from graph preprocessing. (d) Crossbar design from crossbar synthesis. (d) 1T1M crossbar design with 1D1M crossbar for routing.

routes the signals to the appropriate selector lines of the crossbar design $\mathcal{D}$. In particular, each primary input x_i is provided to a buffer and an inverter to generate the inputs x_i and $\neg x_i$. The literals are connected to two separate wordlines. The bitlines of the 1D1M crossbar are connected to the selector lines of the 1T1M crossbar. The memristors in the 1D1M crossbar are programmed appropriately to route the primary inputs to the appropriate selector lines.

5.3 Crossbar partitioning

In this section, we propose a crossbar partitioning method to handle constraints on the maximum crossbar dimension. We illustrate the partitioning using an example below. In Section 5.3.1, we formulate the partitioning problem. We provide a MIP based solution to the problem in Section 5.3.2.

We illustrate the crossbar partitioning with an example in Figure 4. The crossbar dimensions are set to 4x4. The graph in figure 4(a) has 7 nodes and 8 edges, which would require a crossbar with dimension of 7x8. In Figure 4(b), we label all the nodes with a partition number (letter for clarity). Some nodes are labeled with two partitioning numbers because they are required to be placed in two crossbars. This stems from the fact that edges correspond to connections that can only be realized within a graph. The three crossbars and the inter-crossbar connections are shown in Figure 4(c). We show the nodes for each crossbar in the middle and the replicated nodes on the edges between the crossbars.

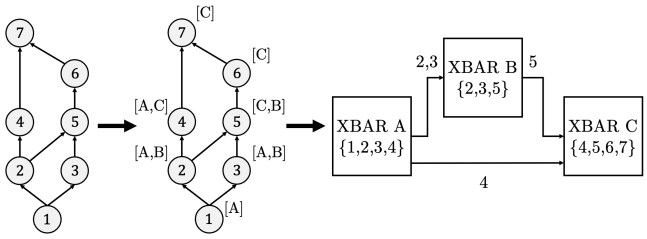


Figure 4: Example of a graph partitioning. (a) Graph G. (b) Assignment of partition letters A, B, and C to each node. (c) Three crossbar designs A, B, and C for each partition.

5.3.1 The Crossbar Partitioning Problem. The crossbar partitioning problem is a problem of assigning the nodes and edges in the graph G to one or more crossbars (or partitions). The problem is formulated as follows:

- Each node in G is required to be assigned to one or multiple crossbars.
- Each edge in G is required to be assigned to one crossbar. The adjacent nodes are required to be assigned to the same crossbar.
- A crossbar can fit at most D nodes and D edges, where DxD are the crossbar dimensions.
- The objective of the partitioning is to minimize the total number of crossbars and the number or nodes that are assigned to multiple partitions (inter-crossbar connections).

5.3.2 Partitioning using MIP formulation. In this section, we formulate and solve a MIP to perform the crossbar partitioning. Let the graph G have vertices V and edges E. Let K be an overestimate on the number of required crossbars with dimension (D)x(D). Let v_i^k be a binary variable indicating if the corresponding node is assigned to crossbar k. Let x_k be a binary variable indicating if crossbar k is utilized. Using the introduced notation, the crossbar partitioning problem is formulated as follows:

$$\min \quad \alpha N + (1-\alpha) I \tag{1}$$

$$\text{s.t.} \quad \sum_{k \in K} x_k = N \tag{2}$$

$$v_i^k \leq x_k, \qquad \forall k \in K, \forall i \in V \tag{3}$$

$$\sum_{k \in K,\, v \in V} v_i^k - |V| = I \tag{4}$$

$$\sum_{k \in K} e_i^k = 1, \qquad \forall i \in V \tag{5}$$

$$2 \cdot e_i^k \leq v_p^k + v_q^k, \qquad e_i = (p,q) \in E \tag{6}$$

$$\sum_{i \in V} v_i^k \leq D, \qquad \forall k \in K \tag{7}$$

$$\sum_{i \in E} e_i^k \leq D, \qquad \forall k \in K \tag{8}$$

$$x_k \geq x_{k+1}, \qquad \forall k \in \{1, \ldots, K-1\} \tag{9}$$

where α is a user-specified parameter for balancing the total number of crossbars N and the number of inter-crossbar connections I. The first constraint (on line 2) sets N to the number of crossbars being utilized. The constraint on line 3 requires each crossbar k to be used if a node is assigned to it. Line 4 defines the number of inter-crossbar connections. Line 5 and line 6 ensure that each edge is

placed in one partition and that the nodes adjacent to the edge are placed in the same partition. The next two constraints on line 7 and line 8 ensure that the crossbar dimension constraints are respected. The last constraint ensures that crossbar $(k+1)$ is not used before crossbar k. This eliminates a substantial amount of degenerate solutions, which results in speed-ups.

6 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]. CPLEX [1] is used as ILP solver for the graph partitioning and a timeout of 1h is set. After that we simply query the best feasible solution (it is easy to find a feasible solution due to the structure of the problem). Within the ILP formulation, we have set $\alpha = 0.5$ and $K = 1.5 \times \frac{\max(|V|,|E|)}{D}$ as the overestimate for the number of crossbars. In Table 2, an overview is provided of 10 benchmarks of the RevLib benchmark suite [25].

Table 2: Overview of 10 input circuits from RevLib [25].

Benchmark	Inputs	Outputs
in0	15	11
apex2	39	3
spla	16	46
pdc	16	40
misex3	14	14
tial	14	8
apex4	9	19
cps	24	109
apex5	117	88
seq	41	35

We evaluate the path-based computing systems by building a complete architectural model with crossbars, peripheral circuitry, and buses for inter-crossbar connections. Most properties of each circuit component in terms of power, area, and latency are obtained from [17, 22]. The power consumption for the bus, crossbar, buffers, and inverters are 13mW, 0.3mW, 0.109μW, and 0.218μW. The area for the respective components are 0.2μm^2, 15.7mm^2, 6mm^2, and 12mm^2. The latency for the respective components are 15ns, 100ns, 3.29ps, and 1.6425ps. The power and area are obtained from the architecture level model. The latency for PATH is proportional to the length of the critical path among the crossbar inter-connections.

We compare PATH with the state-of-the-art paradigms for FLOW and MAGIC paradigms, i.e., COMPACT [22] and CONTRA [4]. The results for the paradigms are obtained using the same circuit level parameters using the models in [4, 22]. We have obtained both tools from GitHub. For COMPACT and CONTRA, we use 0.39nJ and 50.88ns for the write energy and latency, respectively [20].

In Section 6.1, we evaluate the proposed path-based in-memory computing paradigm and the effectiveness of the PATH framework. In Section 6.2, we compare the PATH framework with state-of-the-art in-memory computing paradigms.

6.1 Evaluation of the PATH framework

In this section, we evaluate the path-based paradigm and the effectiveness of the PATH framework.

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

Table 3: BDD, graph, and crossbar design properties without dimensional constraints.

Benchmark	BDD		Graph		Crossbar design	
	Nodes (num)	Edges (num)	Nodes (num)	Edges (num)	Rows (num)	Columns (num)
in0	385	766	384	680	384	680
apex2	567	1130	566	1042	566	1042
spla	594	1184	593	864	593	864
pdc	621	1238	620	887	620	887
misex3	674	1344	673	1094	673	1094
tial	897	1790	896	1717	896	1717
apex4	990	1976	989	1874	990	1874
cps	1080	2156	1079	1633	1080	1633
apex5	1259	2514	1258	2387	1259	2387
seq	1302	2600	1301	2041	1301	2041
Norm.	1.00	1.00	1.00	0.85	1.00	0.85

First, we evaluate mapping Boolean functions ϕ to crossbar designs without constraints on the dimensions. In Table 3, the columns are labeled with the number of nodes and edges in the BDD, the number of nodes and edges in the pre-processed graph G, and the number of rows and columns in the resulting crossbars. The synthesis time is less than 3 seconds for all the circuits. It can be observed that the graph has one less node and fewer edges than the BDD. This is the result of removing the zero terminal node and the adjacent edges. There is a one-to-one correlation between the number of nodes and edges in the graph and the dimensions of the crossbar design. A graph with $|V|$ edges and $|E|$ nodes results in a crossbar with dimensions of $|V|$x$|E|$. This allows the crossbar partitioning to easily handle the crossbar dimension constraints. The crossbar partitioning is necessary as it is not expected that we will be able to fabricate crossbars of dimension 1258x2387.

Table 4: Evaluation of PATH using crossbars with dimension of DxD = 128x128.

Benchmarks	Crossbars (num)	Inter-connections (num)	Critical path (num)	Power (mW)	Latency (μs)	Area (mm^2)
in0	7	426	5	35.0	1.03	34.52
apex2	9	763	9	37.6	1.82	39.90
spla	9	732	9	37.6	1.82	39.90
pdc	9	728	9	37.6	1.82	39.90
misex3	10	785	10	38.9	2.02	42.58
tial	17	1335	15	47.9	3.02	61.40
apex4	18	1628	18	49.2	3.62	64.09
cps	16	1508	16	46.7	3.22	58.71
apex5	23	1805	22	55.7	4.42	77.53
seq	20	1884	20	51.8	4.02	69.47

Next, we evaluate the effectiveness of the PATH framework to map the input Boolean functions ϕ to crossbars with dimensions of 128x128. In Table 4, we show the number of crossbars, the number of inter-crossbar connections, the number of crossbars connected in series. We also show the performance in terms of power, latency and area. We observe that PATH is able to successfully meet the hardware constraints for all of the designs. This demonstrates the high fidelity of the PATH framework. The synthesis time is one hour for all the circuits. The runtime is dominated by solving the ILP formulation. Remember we query the best feasible solution after one hour. The structure of the ILP formulation ensures that a feasible solution of reasonable good quality always exists (less than 10% duality gap).

Now we evaluate the sensitivity of the crossbar dimensions on the circuit *seq* in Figure 5. We observe that the total number of crossbars and the number of inter-crossbar connections are reduced when the crossbar dimensions are scaled up in (a) and (b) of Figure 5. We evaluate the performance in terms of power consumption and latency. As expected, the power and latency have improved due to the fewer crossbars and fewer interconnections. For $D = 1024$, the power consumption slightly increases due to underutilization.

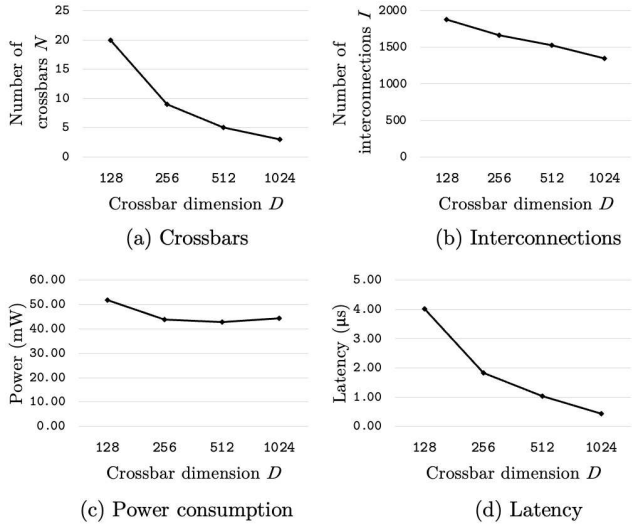


Figure 5: Number of crossbars, interconnections, power consumption, and latency for increasing dimension D.

6.2 Comparison with state-of-the-art in-memory computing paradigms

In this section, we compare PATH with CONTRA [4], the state-of-the-art framework for MAGIC-based in-memory computing, and with COMPACT [22], the state-of-the-art framework for FLOW-based in-memory computing. No comparison has been made with IMPLY [11], considering MAGIC outperforms IMPLY [21].

In Figure 6, the normalized energy consumption and latency are given for PATH, COMPACT, and CONTRA. Compared with PATH, COMPACT results in 4.7X higher energy and 8.5X longer latency. 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. Compared with PATH, we observe that CONTRA consumes 18.12X higher energy and is 85.69X slower. 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 compile phase. The cost of these operations is amortized across multiple evaluations.

7 SUMMARY AND FUTURE WORK

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 the PATH framework to automatically synthesize Boolean circuits into multiple crossbar partitions for path-based computing. Finally, we have demonstrated that the paradigm is orders of magnitude faster and more energy-efficient than state-of-the-art in-memory computing paradigms.

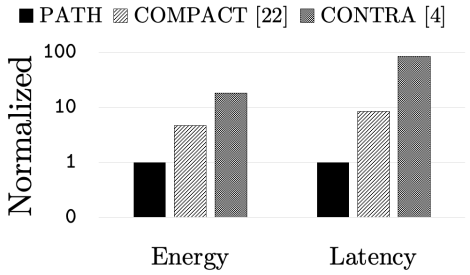


Figure 6: Comparison of the energy consumption and latency for PATH, COMPACT, and CONTRA.

REFERENCES

[1] [n. d.]. CPLEX optimizer. https://www.ibm.com/analytics/cplex-optimizer
[2] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.
[3] John Backus. 1978. Can programming be liberated from the von Neumann style? *CACM* 21, 8 (1978), 613–641.
[4] Debjyoti Bhattacharjee et al. 2020. CONTRA: area-constrained technology mapping framework for memristive memory processing unit. In *ICCAD'20*. 1–9.
[5] Alexandros Bousdekis et al. 2021. A Review of Data-Driven Decision-Making Methods for Industry 4.0 Maintenance Applications. *Electronics* 10, 7 (2021), 828.
[6] Karl S Brace, Richard L Rudell, and Randal E Bryant. 1990. Efficient implementation of a BDD package. In *DAC'90*. IEEE, 40–45.
[7] Geoffrey W Burr et al. 2010. Phase change memory technology. *JVST B* 28, 2 (2010), 223–262.
[8] Miao Hu et al. 2018. Memristor-based analog computation and neural network classification with a dot product engine. *Advanced Materials* 30, 9 (2018), 1705914.
[9] Yiming Huai et al. 2008. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS bulletin* 18, 6 (2008), 33–40.
[10] Shahar Kvatinsky et al. 2014. MAGIC—Memristor-aided logic. *IEEE TCAS-II* 61, 11 (2014), 895–899.
[11] Eero Lehtonen, Jussi Poikonen, and Mika Laiho. 2012. Implication logic synthesis methods for memristors. In *ISCAS'12*. IEEE, 2441–2444.
[12] Shin-ichi Minato et al. 1990. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *DAC'90*. IEEE, 52–57.
[13] Mehdi Mohammadi et al. 2018. Deep learning for IoT big data and streaming analytics: A survey. *COMST* 20, 4 (2018), 2923–2960.
[14] Giacomo Pedretti et al. 2020. A spiking recurrent neural network with phase-change memory neurons and synapses for the accelerated solution of constraint satisfaction problems. *JXCDC* 6, 1 (2020), 89–97.
[15] Alexander Pisarev et al. 2021. Fabrication technology and electrophysical properties of a composite memristor-diode crossbar used as a basis for hardware implementation of a biomorphic neuroprocessor. *Microelectronic Engineering* 236 (2021), 111471.
[16] David Reinsel-John Gantz-John Rydning. 2018. The digitization of the world from edge to core. *Framingham: International Data Corporation* (2018), 16.
[17] Ali Shafiee et al. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH* 44, 3 (2016), 14–26.
[18] Saeideh Shirinzadeh, Mathias Soeken, and Rolf Drechsler. 2016. Multi-objective BDD optimization for RRAM circuit design. In *IEEE DDECS 2016*. 1–6.
[19] Fabio Somenzi. 2012. CUDD: CU decision diagram package-release 2.4. 0. *University of Colorado at Boulder* (2012).
[20] Linghao Song et al. 2017. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA*. IEEE, 541–552.
[21] Phrangboklang Lyngton Thangkhiew, Rahul Gharpinde, and Kamalika Datta. 2018. Efficient mapping of Boolean functions to memristor crossbar using MAGIC NOR gates. *TCAS-I* 65, 8 (2018), 2466–2476.
[22] Sven Thijssen et al. 2021. COMPACT: Flow-Based Computing on Nanoscale Crossbars with Minimal Semiperimeter. In *DATE*. IEEE, 232–237.
[23] Alvaro Velasquez and Sumit Jha. 2015. Automated synthesis of crossbars for nanoscale computing using formal methods. In *NANOARCH'15*. IEEE, 130–136.
[24] Miao Wang et al. 2015. A selector device based on graphene–oxide heterostructures for memristor crossbar applications. *Appl. Phys. A* 120, 2 (2015), 403–407.
[25] Robert Wille et al. 2008. RevLib: An online resource for reversible functions and reversible circuits. In *ISMVL'08*. IEEE, 220–225.
[26] Cong Xu et al. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *HPCA'15*. IEEE, 476–488.
[27] Linlin Zhao et al. 2020. Advancing computer-aided drug discovery (CADD) by big data and data-driven machine learning modeling. *Drug discovery today* (2020).