READ-based In-Memory Computing using Sentential Decision Diagrams

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
‡Computer Science Department, Florida International University, Miami, USA
{sven.thijssen, muhammad.rashed, rickard.ewetz}@ucf.edu, jha@cs.fiu.edu

2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC) | 979-8-3503-9354-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ASP-DAC58780.2024.10473963

***Abstract*—Processing-in-memory (PIM) has the potential to unleash unprecedented computing capabilities. While most in-memory computing paradigms rely on repeatedly programming the non-volatile memory devices, recent computing paradigms are capable of evaluating Boolean functions by simply observing the flow of electrical currents within a crossbar of non-volatile memory. Synthesizing Boolean functions into such crossbar designs is a fundamental problem for next-generation in-memory computing systems. The selection of the data structure used to guide the synthesis process has a first-order impact on the overall system performance. State-of-the-art in-memory computing paradigms leverage representations such as majority inverter graphs (MIGs), and binary decision diagrams (BDDs). In this paper, we propose the Cascading Crossbar Synthesis using SDDs (C^2S^2) framework for automatically synthesizing Boolean logic into crossbar designs. The cornerstone of the C^2S^2 framework is a newly invented data structure called sentential decision diagrams (SDDs). It has been proved that SDDs are more succinct than binary decision diagrams (BDDs). To minimize expensive data transfer on the system bus, C^2S^2 maps computation to multiple crossbars that are connected together in series. The C^2S^2 framework is evaluated using 13 benchmark circuits. Compared with state-of-the-art paradigms such as CONTRA, FLOW, and PATH, C^2S^2 improves energy-efficiency by $6.8\times$while maintaining similar latency.**

I. Introduction

The rise of the Internet of Things (IoT) [1] and 5G technology [2] has powered the emergence of data-intensive applications. However, today's high-performance computing systems are inhibited by the von Neumann bottleneck [3]. Processing in-memory (PIM) using non-volatile memory is a promising solution to overcome this bottleneck, either in the analog or digital realm. While analog in-memory computing can efficiently execute approximate matrix-vector multiplication, digital in-memory computing can deliver deterministic precision for high-assurance applications.

Within digital in-memory computing, we distinguish between WRITE-based [4] and READ-based computing paradigms [5]. The WRITE-based in-memory computing paradigms require the non-volatile memory devices to be reprogrammed for each function evaluation, which results in high latency, power consumption, and poor reliability. In contrast, READ-based digital in-memory computing paradigms do not suffer from these ramifications. In a READ-based in-memory computing paradigm, the resistive devices are only programmed once to their states, and do not need to be reprogrammed for each computation. Examples of such paradigms are OR-plane logic [6] and path-based computing [7]. We focus on path-based computing because OR-plane logic requires selector devices that are difficult to fabricate.

The authors were in part supported by NSF awards # 2319399 and # 2113307, and DOE award DE-SC0024576.

TABLE I
Data structures for digital in-memory computing.

Framework	Citation	Data structure
CONTRA	[8]	NOR netlist
COMPACT	[9]	BDD
PATH	[7]	BDD
C^2S^2	(this work)	SDD

Path-based computing evaluates Boolean functions using one-transistor one-memristor (1T1M) crossbar arrays. The non-volatile memory devices are programmed one time and Boolean variables are assigned to the selectorlines. The Boolean function evaluates to true if there exists a path from an input wordline to an output wordline using only low-resistive devices and closed access transistors.

The success of the semiconductor industry over the past two decades has been driven by automatically synthesizing Boolean functions into logic gates (or standard cells) [10]. The selection of the underlying data structure used to guide the synthesis process has a first-order impact on the overall system performance. In Table I, we show the data structures used to guide the synthesis process of Boolean functions to in-memory computing hardware. Most data structures were invented before year 2000. This paper is motivated by more recently invented sentential decision diagrams (SDDs) [11]. SDDs have been shown to be exponentially more succinct than previous decision diagram representations [12].

In this paper, we propose the Cascading Crossbar Synthesis using SDDs, C^2S^2 pronounced as "C-2-S-2", framework for automatically synthesizing Boolean functions into 1T1M crossbar arrays for path-based in-memory computing. The cornerstone of the C^2S^2 framework is the recently invented SDD representation. Moreover, multiple crossbars are connected into staircase-like cascading structures to minimize the utilization of the system bus. The main innovations of C^2S^2 are summarized, as follows:

1) We propose synthesis algorithms for mapping SDDs to 1T1M crossbars. First, Boolean functions are synthesized into crossbars using SDDs, which are then evaluated using low-latency and energy-efficient READ operations.
2) Using cascading structures to minimize inter-crossbar communication on the system bus, we present new algorithms for synthesizing Boolean functions into path-based computing systems with architectural constraints.

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2024 at 13:01:52 UTC from IEEE Xplore. Restrictions apply.

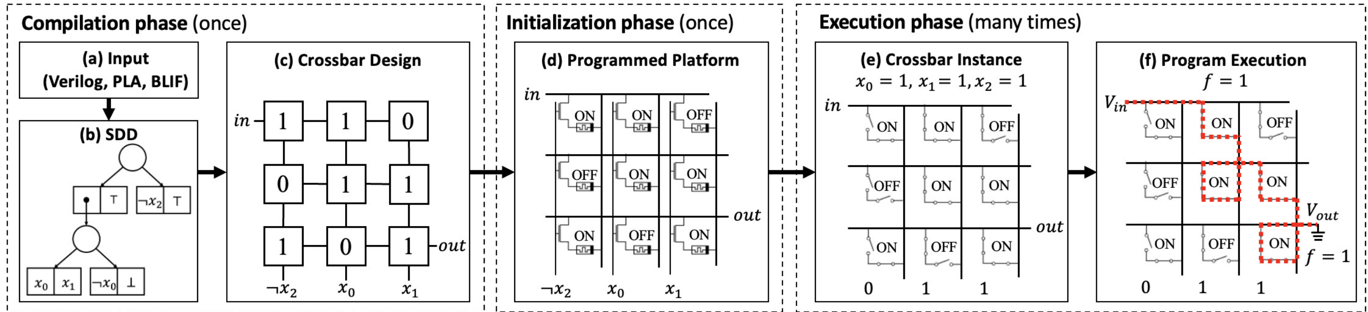
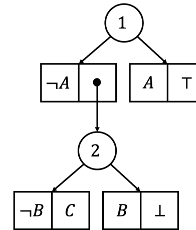


Fig. 2. Overview of the compilation phase and evaluation phase for path-based computing. (a) First, a Boolean function is specified in a hardware description language such as Verilog. (b) Then, the Boolean function is synthesized into an SDD. (c) The SDD is mapped into a crossbar design. (d) Next, the memristors are programmed to their resistive states. (e) Given an input vector for evaluation, the selectorlines are charged accordingly. (f) Finally, the output is computed by applying a high input voltage to the top-most wordline.

3) Compared with state-of-the-art paradigms, the experimental results demonstrate energy improvements of $6.8\times$while maintaining similar latency.

In Section II, we provide preliminaries. In Section III, we explain path-based computing and formally define the problem. Our proposed framework $\mathrm{C^2S^2}$ is introduced in Section IV. Optimization of hardware resources in Section V. Experimental evaluation is in Section VI. Summary and future work in Section VII.



(a) SDD

	BDD	SDD
pdc	6,363	4,142
spla	7,176	4,142
urf4	13,926	9,026
frg2	30,543	11,132

(b) SDD vs BDD size

Fig. 1. (a) An SDD. (b) Comparison of SDD and BDD sizes.

II. Preliminaries

A. Sentential Decision Diagrams (SDDs)

A sentential decision diagram (SDD) is a graph representation of a Boolean function f [11]. The SDD of a Boolean function $f = \neg A \wedge \neg B \wedge C \vee A$ is illustrated in Figure 1(a). In contrast with binary decision diagrams (BDDs), sentential decision diagrams are defined by a *vtree* instead of a variable ordering. This generalization from a strict linear ordering to a local partial ordering imposed by the underlying vtree of an SDD yields greater flexibility and allows SDDs to explore more partitions as compared to BDDs. This ultimately results in the enhanced succinctness of SDDs compared to equivalent BDDs. The BDD and SDD sizes of four representative benchmarks are shown in Figure 1(b).

An SDD is a directed acyclic graph (DAG) with two types of nodes: OR-nodes, and AND-nodes. OR-nodes are represented by circles, and AND-nodes are represented by rectangles, as shown in Figure 1(a). An OR-node is a disjunction over all its outgoing edges; an AND-node is a conjunction of a prime (left element in the rectangle), and a sub (right element in the rectangle). A prime or a sub can either be literals (a Boolean variable or the negation of a Boolean variable) or Boolean functions. SDDs are interpreted as follows: starting from the root node of the DAG, we traverse all nodes to the leaf nodes. All AND-nodes with $\bot$ (false) as sub are tautologically equivalent with $\bot$, and can thus be omitted from the Boolean function. For the OR-nodes, we disjoin the function of its children, and for the AND-nodes we conjoin the prime and the sub. For Figure 1(a), we would have $f = \neg A \wedge (\neg B \wedge C \vee B \wedge \bot) \vee A \wedge \top \equiv \neg A \wedge \neg B \wedge C \vee A$. Efficient algorithms for manipulating SDDs enable their use in our synthesis approach.

B. In-Memory Computing Architecture

The architectural design of an in-memory computing platform is an important design choice. If the desired computation cannot fit within a single crossbar, it must be partitioned across multiple crossbars. However, inter-crossbar communication on the system bus is power-hungry and bandwidth-limited. To overcome this limitation, we propose to use crossbars connected into cascading staircase-like structures. The cascading structure introduces additional constraints on the data movement but greatly reduces the amount of communication on the system bus in our experimental investigations.

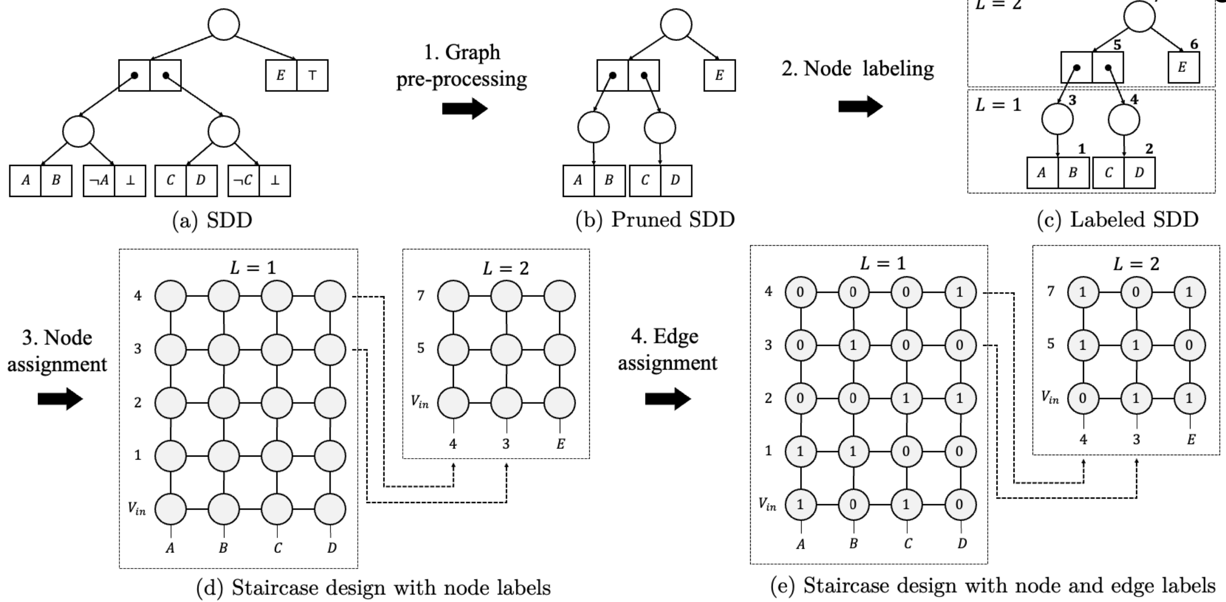
III. Path-based computing using SDDs

Path-based computing is a READ-based digital in-memory computing paradigm [7]. The synthesis process in that work was guided by a relatively direct mapping from a BDD to a crossbar. We show an overview of path-based computing using SDDs in Figure 2. The example is shown using a single crossbar, but we consider cascading staircase-like structures in the $\mathrm{C^2S^2}$ framework. The computing paradigm consists of three phases: compilation, initialization, and execution.

In the compilation phase, a Boolean function f is provided in a hardware description language such as Verilog, PLA, or BLIF. This is illustrated in Figure 2(a). Then, the function is synthesized into an SDD, as shown in Figure 2(b). Next, the SDD is compiled into a crossbar design $\mathcal{D}$ for 1T1M crossbar arrays, as shown in Figure 2(c). In the initialization phase, the memristors are programmed once to their resistive state, as in Figure 2(d). This results in lower energy consumption and latency compared with WRITE-based computing paradigms.

Finally, in the execution phase, an input vector is provided to evaluate the Boolean function f. In Figure 2(e), the input vector to be evaluated is $\{x_0{=}1, x_1{=}1, x_2{=}1\}$ and the selectorlines

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2024 at 13:01:52 UTC from IEEE Xplore. Restrictions apply.



(a) SDD (b) Pruned SDD (c) Labeled SDD

(d) Staircase design with node labels (e) Staircase design with node and edge labels

Fig. 4. Example of the synthesis flow for the proposed framework C^2S^2. The input of the framework is an SDD and the output is a staircase design. The graph of the SDD in (a) is pre-processed such that we obtain a pruned SDD in (b). Next, the nodes of the graph are labeled and separated in layers defined by the OR-nodes as shown in (c). Then, in (d), a cascade of crossbars is created by assigning the nodes to wordlines and selectorlines. More precisely, the variables are assigned to the selectorlines and inter-crossbar connections are created. Finally, memristors are assigned a low resistive state ('1') to realize the edges, resulting in a staircase design in (e).

are charged accordingly. To evaluate the Boolean function f, we apply a high voltage to the top-most nanowire (the input) and ground the bottom-most nanowire (the output). An electrical current will flow through the memristor crossbar array. We say that a Boolean function f evaluates to true if and only if there exists a path from the input to the output along closed gates. In Figure 2(f), we observe that f evaluates to true as there is such a path from input to output, which is indicated using a red dotted line.

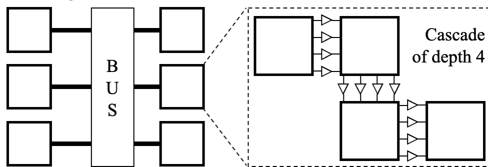


Fig. 3. Example of cascade architecture with four crossbars connected in series.

A. Problem formulation

In this paper, we address the problem of mapping a Boolean function into cascading staircase-like structures of 1T1M crossbars using sentential decision diagrams (SDDs), as illustrated in Figure 3. We decompose the problem into two sub-problems:

1) The algorithmic synthesis of a Boolean function into cascading 1T1M crossbar structures based on SDDs without any explicit constraints on the number or dimensions of crossbars.
2) The partitioning of the SDD-based design into multiple cascading 1T1M crossbars while obeying hardware constraints, such as the number of series connected crossbars and the dimensions of the crossbars.

We resolve these twin problems by presenting a graph-based mapping of SDDs onto crossbars and creating a partitioning algorithm based on an ILP formulation to meet hardware constraints and merge underutilized cascading structures.

IV. The C^2S^2 Framework

In this section, we describe our proposed Cascading Crossbar Synthesis using SDDs, C^2S^2, framework. The input of the framework is an SDD G and the output is a cascading design $\mathcal{S}$. Figure 4 presents an overview of the flow of the framework. The synthesis consists of four main steps: graph pre-processing, node labeling, node assignment, and edge assignment.

A. Graph Pre-Processing

The input of the first step is a graph G representing the SDD of a Boolean function f. The SDD is generated using the SDD tool [13]. In Figure 4(a), the graph G for the SDD is illustrated. Following the rules in Section II-A, AND-nodes with $\bot$ as prime/sub are removed. AND-nodes with $\top$ as prime/sub are changed into BUF-nodes (buffer nodes). This resulting graph illustrated in Figure 4(b).

B. Level and Node Labeling

In the second step, the nodes of the graph are labeled. The input is the pruned SDD from Figure 4(b) and the output is a graph with node labels, as illustrated in Figure 4(c). Further, the nodes in different levels are separated. Each level is defined by the distance between the root node and the OR-nodes. In Figure 4(c), the OR-nodes 2 and 4 define a layer with its AND-nodes 1 and 3 as children.

C. Node Assignment

In this section, we solve the problem of assigning nodes of the labeled graph G' to construct a cascading design. The nodes of G' will be assigned to the wordlines and selectorlines of crossbars, depending on their layer. Whether a node will be assigned to a wordline or selectorline depends on the type of node:

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2024 at 13:01:52 UTC from IEEE Xplore. Restrictions apply.

- Each AND-node will be assigned to a wordline, and its prime and sub will each be assigned to a selectorline.
- Each BUF-node will be assigned to a selectorline.
- Each OR-node will be assigned to a wordline.

In Figure 4(d), a cascading design is illustrated where the nodes of the graph in Figure 4(c) have been assigned to the wordlines and selectorlines. For example, the AND-node 1 has been assigned to the second wordline from the bottom in the first crossbar, and its prime A and sub B have been assigned to the first and second selectorline in the first crossbar.

D. Edge Assignment

In this step, the memristors are assigned a resistive state to realize the Boolean operations. An AND-node is realized by programming three memristors to a low resistive state ('1'): the memristor connecting the prime and the input voltage, and the memristors connecting the prime and the sub. An OR-node is realized by assigning '1' to all memristors between the wordline of the OR-node and the selectorlines of its children.

Algorithm 1 Partitioning of the SDD G

```
Input: SDD G = (V, E), constraints C
Output: T // A topology of cascading staircase-like structures
 1: function PARTITION(G)
 2:     while V ≠ ∅ do
 3:         R = ∅ // Initially, no root nodes for our cone
 4:         S = □ // Initially, no staircase structure
 5:         for r in GETCANDIDATES(G) do
 6:             R_temp ← R ∪ {r}
 7:             H ← INDUCEDSUBGRAPH(G, R)
 8:             if CONSTRAINTSSATISFIED(C, H) then
 9:                 S = STAIRCASE(H)
10:                 R ← R_temp
11:             end if
12:         end for
13:         T ← T ∪ {S}
14:     end while
15: end function
```

V. Constrained cascading synthesis

The synthesis method explained in previous Section IV results in cascading designs of arbitrary dimensions and depths, and may not be optimal in realistic environments. Therefore, we provide a partitioning algorithm to synthesize the given graph G into subgraphs G' such that G' can be realized in physical hardware.

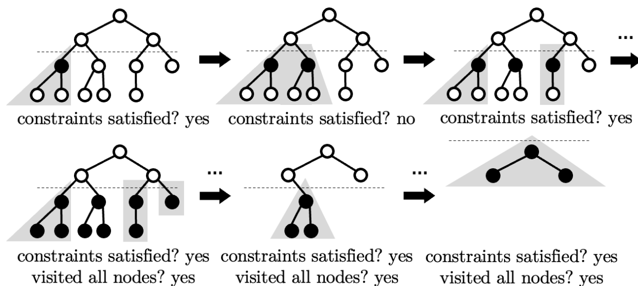


Fig. 5. Example of the partitioning algorithm. In each block, candidate root nodes are visited in a reverse topological sort.

A. Partitioning

In Algorithm 1, we provide the partitioning algorithm. The input of the partitioning algorithm is the graph $G = (V, E)$ of an SDD, and the constraints C. The constraints C include the crossbars dimensions, the number of inter-crossbar connections, and the cascade depth. The output is a topology of cascading staircase-like designs $\mathcal{T}$. The algorithm iterates over the nodes in V and stops when all nodes in V have been processed, i.e. when all nodes in V have been assigned to a cascading structure.

The algorithm works as follows: given the cascade depth L, we consider candidate root nodes R that will define a set of cones. A cone is defined by a root node and all its descendants. The candidate root nodes are a sorted collection of nodes, based on their level in the graph. All nodes are assigned a level l, as explained in Section IV-B. Then we define a block of nodes as nodes between layer l and $l+L$ where $l \bmod L = 0$. Candidate root nodes are being considered for each block in a reverse topological sort (top to bottom). For each candidate root node, an induced subgraph H is constructed based on the candidate root nodes R_{temp}. If the constraints are satisfied when assigning the induced subgraph to the cascading design (line 8), then we extend R with the candidate root node r. This is illustrated in Figure 5. Otherwise, we do not extend R with the candidate root node r and we proceed to the next candidate root node.

$$\min N \tag{1}$$
$$\text{s.t.} \sum_{j \in \mathcal{T}} t_j = N \tag{2}$$
$$\sum_{j \in \mathcal{T}} s_i^j = 1, \quad \forall i \in |\mathcal{T}| \tag{3}$$
$$s_i^j \leq t_j, \quad \forall i \in |\mathcal{T}|, \forall j \in |\mathcal{T}| \tag{4}$$
$$s_{i_0}^j + s_{i_1}^j \leq 1, \quad \forall (i_0, i_1) \in CS, \forall j \in |\mathcal{T}| \tag{5}$$
$$\sum_{i \in \mathcal{T}} s_i^j \times W_i \leq D, \quad \forall j \in |\mathcal{T}| \tag{6}$$
$$\sum_{i \in \mathcal{T}} s_i^j \times S_i \leq D, \quad \forall j \in |\mathcal{T}| \tag{7}$$

B. Aggregation

The resulting cascading topology $\mathcal{T}$ from previous section results in cascading designs that are underutilized. To mitigate this problem, we propose an ILP formulation to merge these underutilized cascades. The objective is to minimize the number of cascades N (line 1). The idea is to assign each cascade in the original topology $\mathcal{T}$ to a cascade in a new topology $\mathcal{T}'$. Thus, for each staircase $\mathcal{S}_i \in \mathcal{T}$, we assign it to a staircase $\mathcal{S}_j \in \mathcal{T}'$. The ILP formulation contains both variables and constants. These variables are s_i^j and t_j (besides N). The constants are W_i, S_i, and D.

For each cascade j in $\mathcal{T}$, we introduce a variable t_j. This variable denotes whether cascade j in our new topology in $\mathcal{T}'$ will be used ($t_j = 1$) or not ($t_j = 0$). The number of used cascades N is then defined by t_j (2). The variable s_i^j indicates whether cascade i from the original topology $\mathcal{T}$ is assigned to cascade j in the new topology $\mathcal{T}'$ or not. If $s_i^j = 1$,

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2024 at 13:01:52 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
NUMBER OF CASCADES, NUMBER OF INTER-CONNECTIONS, AND CRITICAL PATH LENGTH FOR DIFFERENT STAIRCASE DEPTHS.

Benchmark	No cascade				Cascade			
	Bus connections (num)	Inter-connections (num)	Critical path (num)	Synthesis time (min)	Bus connections (num)	Inter-connections (num)	Critical path (num)	Synthesis time (min)
bw	98	179	5	0.32	28	0	1	0.98
apex4	126	2869	9	1.34	56	1248	5	7.45
alu2	25	410	8	0.14	10	111	4	0.63
urf4	159	10086	9	6.86	93	8228	10	27.62
alu4	72	2082	10	1.01	31	1131	8	5.73
tial	73	2278	10	0.95	32	1330	9	6.20
misex3c	106	731	13	0.57	33	181	5	2.31
table3	132	2996	14	2.24	49	1388	6	9.60
misex3	155	1450	13	0.62	43	442	5	4.00
in0	103	1086	14	0.79	30	365	5	2.38
pdc	377	1854	15	1.51	113	585	5	6.53
spla	399	1745	16	1.63	118	566	5	4.22
frg2	980	5172	16	4.55	306	1556	5	14.74
Normalized	1.00	1.00	1.00	**1.00**	**0.36**	**0.36**	**0.49**	4.46

then cascade i is assigned to cascade j. Otherwise, if $s_i^j = 0$, then cascade i is not assigned to cascade j (line 3 and 4). Constraint on line 5 denotes that cascades $i0$ and $i1$ cannot be assigned to the same cascade j. CS is a set of tuples of cascades that are inter-dependent and cannot be assigned to the same cascade. Details are omitted due to the page limit. The constraint allows only s_{i_0} or s_{i_1} to be one. The constraints on line 6 and 7 ensure that the number of wordlines W_i and selectorlines S_i do not exceed the crossbar dimensions D.

VI. EXPERIMENTAL EVALUATION

In this section, we evaluate C^{2}S^2. Experimental evaluation was performed on a Linux machine with Ubuntu version 20.04.4 LTS, 20 i9-900X cores, and 128GB RAM. The source code was programmed in Python 3.8 and can be found on GitHub[1]. For the ILP formulation, we used CPLEX version 20.1.0.0 [14]. To compile the SDDs, we use [13]. Further, to construct the BDDs, we use a BDD tool based CUDD [15]. In Table II, an overview is given of 13 benchmarks selected from the Revlib benchmark suite [16].

TABLE II
OVERVIEW OF 13 BENCHMARKS FROM REVLIB [16].

Benchmark	Circuit		BDD		SDD	
	Inputs	Outputs	AND	OR	AND	OR
bw	5	28	474	237	401	195
apex4	9	19	3432	1716	2751	1069
alu2	10	6	720	360	436	187
urf4	11	11	9284	4642	7143	1883
alu4	14	8	2506	1253	1974	794
tial	14	8	2852	1426	1930	751
misex3c	14	14	1492	746	1063	506
table3	14	14	4820	2410	3747	1632
misex3	14	14	5220	2610	1902	888
in0	15	11	1552	776	1257	578
pdc	16	40	4242	2121	2561	1219
spla	16	46	4784	2392	2815	1327
frg2	143	139	20362	10181	7566	3566
Normalized			1.00	1.00	**0.67**	**0.59**

In all experiments, we set the crossbar dimensions to 128×128 [8]. The power consumption for the data bus connecting the staircases, is set to $13mW$ [17] and the latency $15ns$. For a crossbar $0.3mW$ [17] and $100ns$ with a write energy of $0.391nJ$ and write latency of $50.9ns$ [18]. In Section VI-A, we evaluate the proposed framework C^{2}S^2 in terms of hardware utilization. In Section VI-B, we make a comparison with previous state-of-the-art frameworks for digital in-memory computing.

[1] https://github.com/sventhijssen/c2s2

TABLE III
NUMBER OF CASCADES, NUMBER OF INTER-CONNECTIONS, AND CRITICAL PATH LENGTH USING BDDS AND SDDS.

Benchmark	BDD			SDD		
	Cascades (num)	Inter-connections (num)	Critical path (num)	Cascades (num)	Inter-connections (num)	Critical path (num)
bw	82	142	3	56	93	2
apex4	99	1763	6	69	1869	4
alu2	18	229	5	15	251	5
urf4	121	5587	10	110	9015	8
alu4	53	1339	9	44	1387	10
tial	58	1538	10	46	1706	9
misex3c	72	699	8	56	416	6
table3	103	2239	10	79	1838	7
misex3	110	2463	9	80	847	6
in0	77	946	8	54	689	7
pdc	262	1822	8	199	1073	7
spla	317	2087	8	211	991	8
frg2	908	10234	13	531	2945	8
Normalized	1.00	1.00	1.00	**0.75**	**0.80**	**0.82**

A. Evaluation of C^{2}S^2

First, we compare our chosen data structure, SDDs, with BDDs. In Table II, the number of AND-nodes and OR-nodes for each benchmark as BDD and SDD is provided.

From Table II, we observe that SDDs reduce the number of AND-nodes and the number of OR-nodes by 33% and 41%, respectively, compared with BDDs. This reduction is due to that SDDs are partitioned based on sub-functions (vtree) whereas BDDs are partitioning based on a single input variable (variable ordering). The use of a vtree instead of variable ordering results in that SDDs have a larger solution space with the potential of being more succinct to represent the Boolean function [19]. Next, we compare the hardware improvements due to the chosen data structure. We evaluate the proposed framework using both BDDs and SDDs. In Table III, we report the hardware utilization in terms of number of cascades, number of inter-connections, and critical path length for both decision diagrams. We set the cascade depth to $L = 4$. Compared with BDDs, we observe that the number of cascades, number of inter-connections and critical path length using SDDs decrease with 25%, 20%, and 18%, respectively. Therefore, we conclude that it is most advantageous to use SDDs compared to BDDs.

Finally, we will perform an analysis of the hardware architecture. In Table IV, the hardware utilization is reported for staircase depths $L = 4$, which is the first saturation point

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2024 at 13:01:52 UTC from IEEE Xplore. Restrictions apply.

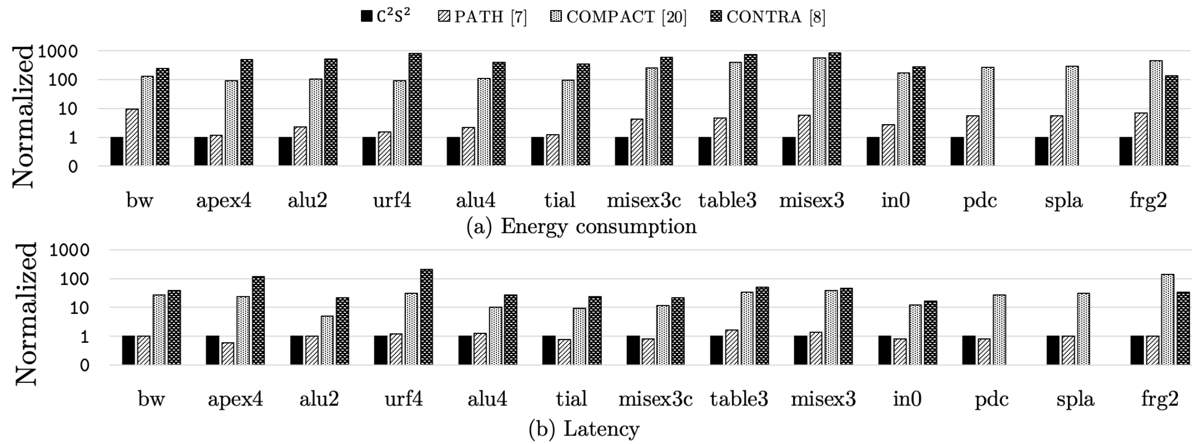


Fig. 6. Normalized energy consumption and latency for the proposed framework C^2S^2, PATH, COMPACT, and CONTRA.

among the benchmarks. In Table IV, we observe that the bus connections have reduced by 64%, the number of interconnections by 64%, and the critical path length by 51%.

B. Comparison with other paradigms

We compare our proposed C^2S^2 framework with other frameworks for digital in-memory computing paradigms. These are PATH [7] for path-based computing, COMPACT [20] for flow-based computing, and CONTRA [8] for MAGIC-based computing. All results, except for the benchmarks pdc and spla for CONTRA due to a runtime error, have been included. Here, we employ the optimization algorithm of Section V-B to aggregate underutilized cascades.

In Figure 6(a), we plot the normalized energy consumption for each benchmark for the aforementioned frameworks. We observe that our framework C^2S^2 reduces the energy consumption with 62.9%, 99.4%, and 99.7% compared with PATH, COMPACT, and CONTRA on average. The energy reduction with the READ-based paradigm PATH stems from the smaller designs as a result of our chosen data structure (SDDs). This translates in less bus connections and thus lower bus utilization. Compared with the WRITE-based paradigms COMPACT and CONTRA, the energy reduction is a result of the high energy consumption for write operations, in combination with high bus utilization.

In Figure 6(b), the normalized latency is given for all benchmarks. Compared with PATH, the latency improves $1.05\times$ using C^2S^2 due to the critical path being of similar length. Compared with COMPACT, and CONTRA, our framework C^2S^2, improves latency by $138\times$ and $33\times$, respectively. This is due to that COMPACT and CONTRA rely on write operations for each evaluation. Further, their bus utilization is higher which leads to increased latency.

VII. Summary and Future Work

In this paper, we have introduced the Cascading Crossbar Synthesis using SDDs, C^2S^2, framework that leverages sentential decision diagrams (SDDs) to compile Boolean functions into circuits for path-based computing. Further, we have proposed an automated synthesis method to compile such SDDs into cascading staircase-like structures, a hardware component consisting of a series of hardwired memristor crossbars. Due to SDDs being smaller than BDDs, hardware utilization can be reduced by approximately 64%. This, combined with the read-based characteristics of path-based computing, the C^2S^2 framework improves energy consumption by $6.8\times$compared with the state-of-the-art synthesis method.

References

[1] S. Li, L. D. Xu, and S. Zhao, "The internet of things: a survey," *Information systems frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[2] A. Gohil, H. Modi, and S. K. Patel, "5g technology of mobile communication: A survey," in *ISSP*, pp. 288–292, IEEE, 2013.

[3] J. Backus, "Can programming be liberated from the von neumann style? a functional style and its algebra of programs," *CACM*, vol. 21, no. 8, pp. 613–641, 1978.

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

[5] M. R. H. Rashed *et al.*, "Stream: Towards read-based in-memory computing for streaming based data processing," in *ASP-DAC*, pp. 690–695, IEEE, 2022.

[6] A. Dehon, "Nanowire-based programmable architectures," *JETC*, vol. 1, no. 2, pp. 109–162, 2005.

[7] S. Thijssen, S. K. Jha, and R. Ewetz, "Path: Evaluation of boolean logic using path-based in-memory computing," in *DAC*, pp. 1129–1134, 2022.

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

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

[10] A. Nardi and A. L. Sangiovanni-Vincentelli, "Logic synthesis for manufacturability," *IEEE Design & Test of Computers*, vol. 21, no. 3, pp. 192–199, 2004.

[11] A. Darwiche, "Sdd: A new canonical representation of propositional knowledge bases," in *IJCAI*, 2011.

[12] S. Bova, "Sdds are exponentially more succinct than obdds," in *AAAI*, vol. 30, 2016.

[13] A. Choi and A. Darwiche, "Sdd advanced-user manual version 2.0." http://reasoning.cs.ucla.edu/sdd/doc/sdd-advanced-manual.pdf, 2018.

[14] IBM, "Cplex optimizer." https://www.ibm.com/analytics/cplex-optimizer, 2020.

[15] M. Vazquez-Chanlatte, "py-aiger-bdd." https://github.com/mvcisback/py-aiger-bdd, Nov 2018.

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

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

[18] T. Yang *et al.*, "Pimgcn: a reram-based pim design for graph convolutional network acceleration," in *DAC*, pp. 583–588, IEEE, 2021.

[19] A. Choi and A. Darwiche, "Dynamic minimization of sentential decision diagrams," in *AAAI*, vol. 27, pp. 187–194, 2013.

[20] S. Thijssen, S. K. Jha, and R. Ewetz, "Compact: Flow-based computing on nanoscale crossbars with minimal semiperimeter and maximum dimension," *TCAD*, 2021.

Authorized licensed use limited to: University of Central Florida. Downloaded on August 02,2024 at 13:01:52 UTC from IEEE Xplore. Restrictions apply.