Towards Area-Efficient Path-Based In-Memory Computing using Graph Isomorphisms

Sven Thijssen*, Muhammad Rashedul Haq Rashed†, Hao Zheng*, 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, hao.zheng, rickard.ewetz}@ucf.edu, jha@cs.fiu.edu

***Abstract*—In-memory computing has attracted significant attention due to its potential to alleviate the issues caused by the von Neumann bottleneck. Path-based computing is a recently proposed in-memory computing paradigm for evaluating Boolean functions using nanoscale crossbars. Unlike state-of-the-art paradigms that use expensive WRITE operations to execute functions, path-based computing only relies on READ operations, which translates into benefits of low power consumption and low computational delay. Unfortunately, path-based computing comes with the penalty of substantial area overhead. In this paper, we introduce the ISO framework, a hardware-software solution for minimizing the area overhead of path-based computing systems. The framework is based on mapping computation to in-memory kernels using an intermediate k-LUT representation. The k-LUTs facilitate reusing hardware resources that realize the same computational structures. The reuse is performed by detecting identical subfunctions using isomorphic graphs. We also present a set of program instruction and scheduling algorithms to facilitate the hardware reuse. We have evaluated our proposed ISO framework on the 10 ISCAS85 benchmarks. Our experimental evaluation indicates that our proposed architecture improves energy consumption, latency, and area by $1.30\times$, $76.59\times$, and $2.79\times$ on the average compared with previous state-of-the-art methods for path-based computing.**

I. Introduction

In an era where 328.77EB amounts of digital data are created daily, computing systems are pushed to the limit to process this information [1]. While these amounts of digital data have facilitated the success of data-intensive applications, such as deep learning, the applications are constrained by the von Neumann bottleneck [2], which results from the data movement between memory units and processing units. In-memory computing promises to eliminate this bottleneck by merging the memory and computing units.

Over the years, many in-memory computing paradigms have been proposed, both in the analog and digital domain. While the analog domain provides low energy and high density, the digital domain provides robustness [3]. In the digital domain, several computing paradigms have been proposed, such as IMPLY [4], MAGIC [5], FLOW [6], and PATH [7]. The first three paradigms use WRITE operations to perform Boolean logic operations while the latter solely relies on READ operations. As WRITE operations in ReRAM require orders of magnitude higher energy consumption compared with READ operations, PATH is favorable compared with the other aforementioned digital in-memory computing paradigms in terms of energy consumption [8]. Unfortunately, the state-of-the-art synthesis methods for path-based computing suffer from large area overhead [7].

The synthesis method directly maps functions represented using binary decision diagrams (BDDs) into crossbar designs. However, it is well-known that many classes of arithmetic functions do not have compact BDD representations, resulting in substantial area overheads. Therefore, there is immense potential to improve computer-aided design tools for in-memory computing systems by leveraging the success stories from the well-established VLSI industry.

In this paper, we propose the ISO framework to address the aforementioned problems. In the ISO framework, our main objective is to reduce the overall area. To accomplish this, we make multiple contributions. First, we draw concepts from the VLSI community to synthesize large benchmarks into smaller intermediary data structures, called k-LUTs [9]. A k-LUT is a look-up table with at most k input variables. Each k-LUT will be synthesized into a BDD, which is subsequently synthesized into a crossbar design. From our experimental results, we conclude that this already results in significant hardware savings. Second, we observed that for a fixed value of k, the number of possible BDD structures is bounded. This entails that the BDD structures are *isomorphic*. A graph isomorphism is a one-to-one mapping between the nodes of two graphs such that the structure is preserved [10]. To identify these isomorphic graphs, we use the Weisfeiler-Lehman kernel [11] and we exploit this observation to make even more hardware savings by reusing the same hardware resources.

We make the following contributions:

1) We leverage k-LUT representations to map Boolean functions into crossbar designs.
2) We made the observation that for a fixed value of k, the number of different BDD structures is bounded. By exploiting isomorphic properties between these BDDs, we can make significant hardware savings.
3) Third, we have developed an end-to-end synthesis framework, ISO, to compile a Boolean function into a computing system for path-based computing. A tight hardware-software co-design is proposed, including crossbar design and program execution.

In Section II, we provide preliminaries, and we motivate our work in Section III. The ISO framework is introduced in Section IV. Area optimization is discussed in Section V. Experimental evaluation in Section VI, and the paper is concluded in Section VII.

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

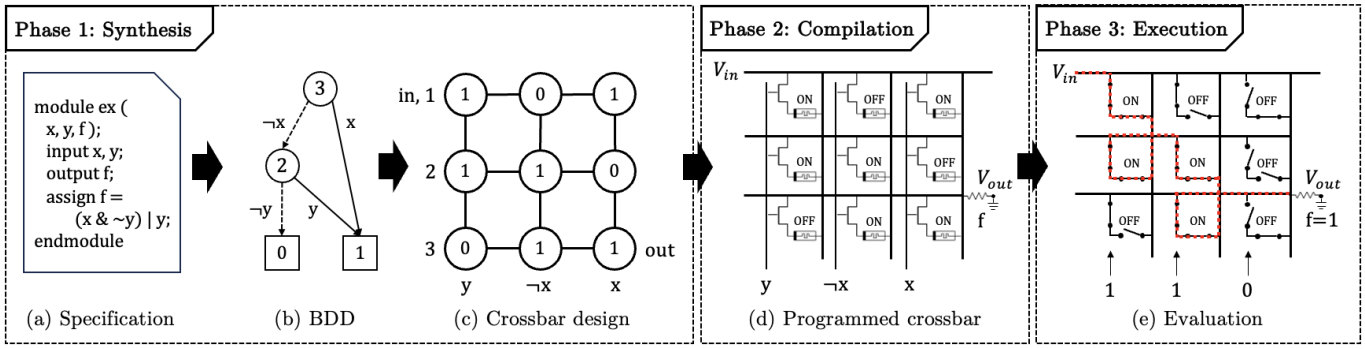


Fig. 2. High-level overview of the synthesis, compilation, and execution phase for path-based computing. The synthesis and compilation phase must be done once for each Boolean function, and the execution phase is repeated for each input vector.

II. Preliminaries

A. Binary decision diagram

A Binary Decision Diagram (BDD) is a directed acyclic graph (DAG) representation for a Boolean function (see Figure 2(b)). A BDD consists of layers of nodes where each node in a layer is assigned the same input variable. Each node, except for two terminal nodes, have two outgoing edges. One edge (solid) is labeled the positive literal, and one edge is labeled with a negative literal (dashed). The terminal nodes are labeled '0' and '1'. For evaluation, we are given an input vector, and the BDD is traversed from the root node to a terminal node. At each node, take the edge such that the truth value of the literal evaluates to '1'. When reaching the '1' ('0') terminal node, the Boolean function evaluates to true (false).

B. Isomorphism

An isomorphism between a graph G and H is a bijection B from the nodes in G to the nodes in H such that $B{:}V(G) \to V(H){:}\forall(u,v) \in E(G) \mapsto (B(u), B(v)) \in E(H)$ [10]. In other words, there is a mapping from the nodes in G to the nodes in H such that for every edge in G there exists an edge in H while adhering to this mapping. In Figure 1, we provide an example of an isomorphism between two graphs, G and H.

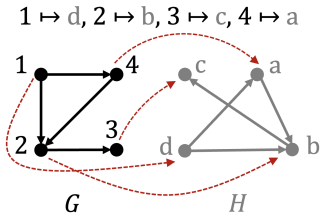


Fig. 1. Example of an isomorphism between two graphs G and H.

C. Path-based computing

Path-based computing is a READ-based digital in-memory computing paradigm. The paradigm consists of three phases: synthesis, compilation, and execution.

Phase I: Synthesis. The input of this phase is a specification, and the output is a crossbar design. In Figure 2(a), the specification is the Boolean function $f = (x \wedge \neg y) \vee y$, which is subsequently synthesized into a binary decision diagram (BDD). The resulting BDD is shown in Figure 2(b). Then, the BDD is mapped to a crossbar design, as shown in Figure 2(c). In a crossbar design, the Boolean literals are assigned to the selectorlines, and the memristors are assigned binary truth values. The synthesis phase is performed only once for a given Boolean function.

Phase II: Compilation. The memristors of a crossbar are programmed to either a high (OFF/'0') or low (ON/'1') resistive state, depending on the provided crossbar design. Orthogonal to other digital in-memory computing paradigms, programming of the memristors is only done once during the system's lifecycle. In Figure 2(d), we illustrate the state of the crossbar, and the assignment of literals to the selectorlines.

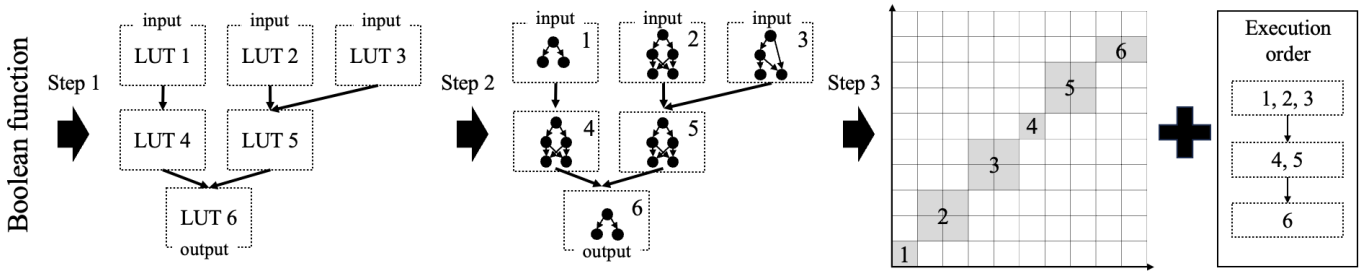
Phase III: Execution. The Boolean function f is evaluated repeatedly for different input vectors. The selectorlines are charged according to the truth value of the respective literal. For example, in Figure 2(e), the Boolean function f is evaluated for the input vector $x = 0$, $y = 1$. When the selectorline is assigned '1', then it is charged and the transistors connected to this selectorline are closed. Otherwise, when the selectorline is assigned '0', it is not charged and open. Then, a high input voltage is applied to the input wordline, and an electrical current flows through the crossbar along memristors which are ON. When the electrical current reaches the output wordline, f evaluates to true. Otherwise, f evaluates to false.

III. Motivation

In this section, we outline the limitations of previous work, and motivate our proposed ISO framework. Path-based computing has the benefit of low energy consumption and low latency, but the computing paradigm currently suffers from large area overhead. The state-of-the-art synthesis methods map a single BDD directly to a crossbar. This brings three problems with it:

1) A single BDD may result in larger area than a collection of small BDDs.
2) BDDs may grow large in the number of input variables (if they can be synthesized after all for real-world benchmarks), and consequently the resulting area is larger than physical crossbar dimensions allow. Consequently, a posteriori partitioning algorithms are required.

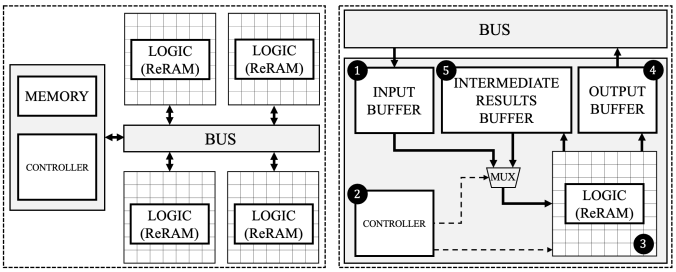
To address these problems, we will leverage k-LUTs as intermediate data structure to synthesize a Boolean function. Instead of creating a single BDD for the whole Boolean function, we will create a BDD for each LUT. This directly solves the first two problems. Further, we made the observation that when k is fixed, the possible number of BDD structures is bounded. This enables the reuse of hardware resources as the crossbar design is structurally the same for different LUTs.



(a) k-LUT topology (b) BDD topology (c) Unbounded crossbar design + execution order

Fig. 5. Synthesis of a Boolean function using k-LUTs. In step 1, the Boolean function is synthesized into a k-LUT topology. In step 2, the k-LUT topology in (a) is converted in a BDD topology in (b). Next, in step 3, the individual BDDs are mapped into a crossbar design using the synthesis method described in Section II-C such that we obtain an unbounded crossbar design with execution order in (c).

In Figure 3(a) and (b), we show an overview of the architecture targeted in previous work [7], and our work, respectively. In previous work, the logic units are connected to the bus, which is power-hungry and slow. In our work, we aim for locality. First, the inputs are provided from the bus to the input buffer ❶. The controller ❷ fetches the data from the buffer, and executes the operations in the logic unit ❸. The output is either written to the output buffer ❹, which is connected to the bus, or to the intermediate results buffer ❺ such that the values can be applied in a next iteration.



(a) Previous work (b) Proposed architecture

Fig. 3. High-level overview of the targeted architecture.

IV. Proposed ISO framework

In this section, we introduce the ISO framework, which synthesizes Boolean functions into k-LUTs, and subsequently maps these to crossbar designs for path-based computing.

A. Overview

The ISO framework consists of three main steps. In Figure 4, we give a high-level overview of the ISO framework. First, we illustrate in Section IV-B how a Boolean function can be synthesized using a topology of k-LUTs into a crossbar design with unbounded dimensions. Then, we illustrate in Section V-A how graph isomorphisms can be used to reduce the overall area. Finally, in Section V-B, we target the proposed architecture from previous section where the crossbar dimensions are constrained.

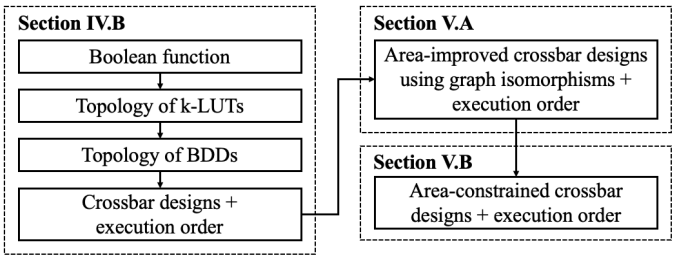


Fig. 4. High-level overview of the ISO framework.

B. Synthesis of a Boolean function using k-LUTs

In this section, we describe the synthesis of a Boolean function using a topology of k-LUTs. The input is a Boolean function (specification), and the output is a set of crossbar designs and an execution order. First, the Boolean function is converted into a k-LUT topology, as shown in Figure 5(a). The k-LUT topology is a data-flow graph, represented as a DAG where the leaf node is the output, and the root nodes are the inputs. Next, each k-LUT is converted into a topology of BDDs, as shown in Figure 5(b). Each BDD is synthesized into a crossbar design according to the methodology of the synthesis method described in Section II-C. The output is a set of crossbar designs with unbounded dimensions and an execution order, as shown in Figure 5(c). The execution order is a series of evaluation instructions (EVAL). An EVAL instruction fetches the input data from the buffer, applies the steps required for evaluation as described in Section II-C, and writes the output data back to the buffer. The execution order consists of multiple cycles, where each cycle corresponds to a generation in the topological sort of the BDD topology.

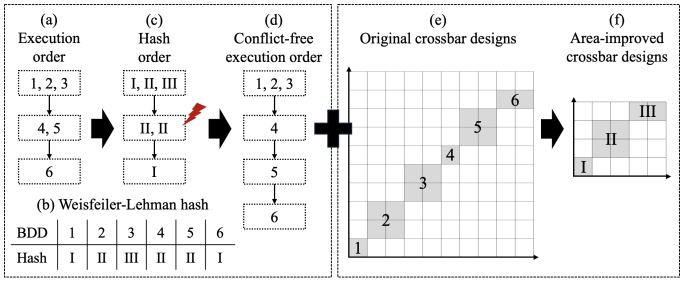


Fig. 6. Area improvement using graph isomorphisms.

V. Area optimizations

A. Area-improved synthesis using graph isomorphisms

In this section, we explain how the overall area requirements can be improved by leveraging isomorphic properties of graphs. When the number of inputs k for a k-LUT is small, we observe that the number of possible BDD structures is also small. In other words, while the inputs may vary, the structure of some BDDs may be invariant. We will leverage this observation to reduce the overall area of the crossbar design. More specifically, when two BDDs G and H are isomorphic, we will reuse the same hardware resources for both G and H. The only difference is that the inputs, which are aligned with the selectorlines, may differ.

In Figure 6, we provide a high-level overview of the area-improved synthesis. Given the original execution order and crossbar designs, as shown in Figure 6(a), we now compute the Weisfeiler-Lehman kernel (hash) for each BDD [11], as shown in Figure 6(b). For clarity, we identify the unique hashes using a Roman number. Based on this hash table, we construct a hash order, which maps each BDD in the execution order into its corresponding hash, as shown in Figure 6(c). However, since there will be only a single resource available for each hash, the execution order may have conflicts according to the hash order. A conflict arises when at least two identical hashed are present in the same generation, as highlighted in Figure 6(c). To resolve this, we propose Algorithm 1 where we push down identical hashes with their corresponding BDD into a new generation. For example, in Figure 6(d), BDD 5 is pushed down into a new generation such that we obtain a conflict-free execution order.

Algorithm 1 Conflict-free execution order

Input: $\mathcal{G}$ // The original execution order (list of lists)
Output: $\mathcal{G}^*$ // The new conflict-free execution order (list of lists)
1: $\mathcal{G}^* \leftarrow \emptyset$
2: **while** $|\mathcal{G}| > 0$ **do**
3: $G \leftarrow \mathcal{G}[0]$ // Get first generation
4: $\mathcal{G} \leftarrow \mathcal{G} \setminus \{G\}$ // Remove generation
5: $K \leftarrow \emptyset, H_K \leftarrow \emptyset, P \leftarrow \emptyset$ // Auxiliary data structures
6: **for** $g \in G$ **do**
7: $h \leftarrow$HASH(g) // Convert BDD into its hash
8: **if** $h \notin H_K$ **then** // If hash not already in kept BDDs
9: $K \leftarrow K \cup \{g\}$
10: $H_K \leftarrow H_K \cup \{h\}$
11: **else**
12: $P \leftarrow P \cup \{g\}$
13: **end if**
14: **end for**
15: $\mathcal{G}^* \leftarrow \mathcal{G}^* \cup K$ // Append the kept BDDs
16: **if** $|P| > 0$ **then**
17: $\mathcal{G} \leftarrow P \cup \mathcal{G}$ // Prepend the pushed down BDDs
18: **end if**
19: **end while**
20: **return** $\mathcal{G}^*$

We make the following claim in Theorem 1:

Theorem 1: The conflict-free execution order is optimal in terms of latency.

To illustrate this, we first define Lemma 1.

Lemma 1: Let $\chi = \{X_1, ..., X_N\}$ be an execution order where each X_i is a generation of BDDs to be executed. Then χ is a total order, such that $\forall i, j \in N, i \neq j : X_i < X_j$. Further, let each BDD x belong to a generation X_i such that it is scheduled as early as possible. In other words, $x \in X_i : \nexists i, j : j < i \in N : x \in X_j$.

Proof 1: Whenever two isomorphic BDDs G and H are in the same generation X_i, we retain one BDD, say G, in generation X_i, and move the other, say H, to a new generation X^*. This new generation X^* must be placed between generation X_i and X_{i+1}. H cannot be placed in generation X_{i+1}, as this would contradict Lemma 1. Hence, the order is optimal in terms of latency. ■

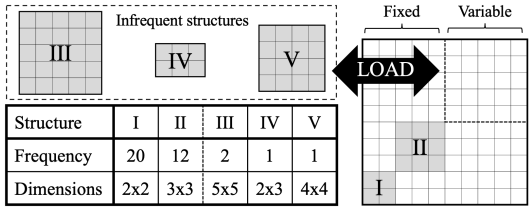
Finally, we realize one crossbar design for each hash. In Figure 6(e), we observe how the original crossbar designs are reduced to the area-improved crossbar designs in Figure 6(f).

Algorithm 2 Fixed and variable crossbar designs

Input: $\mathcal{U}, D$ // The structure hashes sorted in descending order according to their frequency, and the crossbar dimension
Output: $\mathcal{F}, \mathcal{V}$ // Set of fixed and variable structure hashes
1: $L \leftarrow 1, H \leftarrow |\mathcal{U}|, M \leftarrow \lfloor \frac{L+H}{2} \rfloor$
2: **while** $L \leq H$ **do**
3: $R \leftarrow \sum_{g \in \mathcal{U}[1:M]}$ROWS($g$)
4: $C \leftarrow \sum_{g \in \mathcal{U}[1:M]}$COLS($g$)
5: $R_{\max} \leftarrow \max(\{\text{ROWS}(g) \mid \forall g \in \mathcal{U}[M+1{:}|\mathcal{U}|]\})$
6: $C_{\max} \leftarrow \max(\{\text{COLS}(g) \mid \forall g \in \mathcal{U}[M+1{:}|\mathcal{U}|]\})$
7: **if** $R + R_{\max} > D$ or $C + C_{\max} > D$ **then**
8: $H \leftarrow M - 1$
9: **else**
10: $L \leftarrow M + 1$
11: **end if**
12: $M \leftarrow \lfloor \frac{L+H}{2} \rfloor$
13: **end while**
14: $\mathcal{F} \leftarrow \mathcal{U}[1{:}M], \mathcal{V} \leftarrow \mathcal{U}[M+1{:}|\mathcal{U}|]$
15: **return** $\mathcal{F}$, $\mathcal{V}$

B. Area-constrained synthesis

In previous section, we have proposed a synthesis method for k-LUTs to crossbars with arbitrary dimensions. In this section, we address the issue of real-world scenarios where the crossbar dimensions are constrained. We have observed that a few BDD structures occur frequently, and that many other BDD structures occur only a few times. Our proposed execution algorithm is as follows: first, we sort the BDD structures by frequency in descending order. Then, using the binary search algorithm in Algorithm 2, we try to find the maximum number of crossbar designs that can be placed in the crossbar permanently while leaving enough space for replacing irregular BDD structures. The permanent crossbar designs, we will call *fixed* set, and the others *variable* set. The inputs are the set of unique structure hashes with their frequency $\mathcal{U}$, and the crossbar dimensions D. The outputs are the fixed set $\mathcal{F}$ and variable set $\mathcal{V}$.



Structure	I	II	III	IV	V
Frequency	20	12	2	1	1
Dimensions	2x2	3x3	5x5	2x3	4x4

Fig. 7. Frequency and dimensions for different structures, ordered in descending order according to their frequency. Structures I and II are permanently in the crossbar, and structures III, IV, and V are loaded into the crossbar when needed.

In Figure 7, the frequency and the dimensions for different structures are provided in descending order of frequency. We observe that only structures I and II can be placed permanently in the crossbar, and that the infrequent structures III, IV, and V are stored in the buffer. Further, there is space left available in the crossbar for these infrequent structures. Using the LOAD operation, these structures can be loaded into the crossbar when needed.

VI. Experimental evaluation

In this section, we evaluate our proposed ISO framework. First, we evaluate the proposed synthesis method based on k-LUTs. Then, we evaluate the graph isomorphisms. Finally, we compare our proposed framework with previous state-of-the-art digital in-memory computing paradigms. The framework is implemented in Python 3.10 and is available on GitHub[1]. We evaluate our proposed ISO framework on ten ISCAS85 benchmarks.

In Table I, we provide the parameters and specifications for the components in our architecture model. The numbers are obtained from [8], [12], [13], and the CACTI tool [14].

A. Single BDDs vs k-LUTs

First, we illustrate that synthesis using k-LUTs reduces the overall area compared with synthesis using a single BDD. In Table II, we show both the rows, columns, and semiperimeter (rows + columns) for synthesis using a single BDD, for our proposed k-LUTs without isomorphism, and for our proposed k-LUTs with isomorphism. For our proposed synthesis, we have set $k = 4$. For the synthesis using a single BDD, we have used [7]. From Table II, we conclude that our proposed synthesis method using k-LUTs without isomorphisms results in much smaller semiperimeter than [7] with an average normalized reduction of 90%. Our proposed synthesis method using k-LUTs with isomorphisms results in even smaller semiperimeter with a reduction of 99%. The number of cycles increases by $109\times$.

B. Sensitivity analysis of k

Now we will determine the sensitivity of k on the area and the delay. The delay is estimated based on the number of cycles in the execution. This value is determined by the critical path of the k-LUT topology. For this experiment, we assume unbounded crossbar dimensions. We observe in Figure 8(a) that the semiperimeter increases for increasing value of k. Further, in Figure 8(b), we observe that the number of cycles decreases for increasing value of k. This is what we would expect; when k increases, the BDD sizes for the k-LUTs increase. This entails that the probability decreases that two BDDs are isomorphic, resulting in larger semiperimeter. On the other hand, the larger k-LUTs capture more logic at once, resulting in a lower number of cycles.

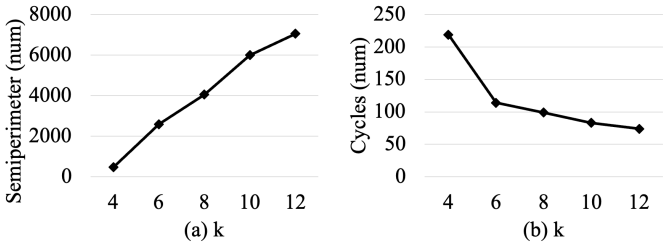


Fig. 8. Sensitivity analysis of the benchmark c7552 for the parameter k.

To illustrate our explanation for the increase of the semiperimeter for increasing value of k, we have analyzed the structure frequency for both $k = 4$ and $k = 12$. In this section, we make two conclusions. In Figure 9(a), we show the frequency of each structure. We observe that for $k = 4$, the frequency for the structures is higher than for $k = 12$. Further, we observe that the graph has a sharp decline and a long tail. This means that a few structures occur many times, while others occur infrequently (Conclusion I). In Figure 9(b), we show the cumulative sum of the structure frequencies. We observe that the cumulative sum of the number of structures for $k = 4$ is much larger than the cumulative sum of the number of structures for $k = 12$ (Conclusion II).

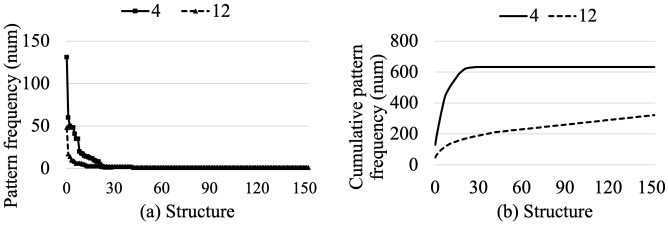


Fig. 9. Structure frequency analysis of the benchmark c7552.

C. Area-constrained synthesis

In this section, we evaluate the area-constrained synthesis, and we finally compare our proposed synthesis method with other digital in-memory computing paradigms. In Figure 10(a), we show the normalized number of LOAD and EVAL instructions for varying values of k (the crossbar dimension is $D = 128$). First, we observe that the LOAD instructions increase for increasing value of k. This corresponds with the Conclusion I of previous section that there are many more structures for higher value of k. Second, the number of EVAL instructions decreases for increasing k. This is due to Conclusion II of previous section. Third, the total number of cycles increases for increasing value of k. This is in contrast with our observation from Figure 8(b). Due to the bounded dimensions, the structures must be regularly swapped whereas for the unbounded dimensions, there are no cycles required for swapping any structures. In the next experiment, we set $k = 4$ as the number of LOAD instructions and the overall number of cycles is low.

TABLE I
Comparison of the number of the required hardware resources for single BDDs and the proposed k-LUTs.

Component	Parameter	Spec
ReRAM crossbar	READ/WRITE energy (pJ)	1.08/3910
	READ/WRITE latency (ns)	29.31/50.88
	Area (mm2)	0.000025
SRAM buffer	READ/WRITE energy (pJ)	58.97/57.57
	READ/WRITE latency (ns)	3.379/3.379
	Area (mm2)	0.083
Bus	Power (mW)	13
	Latency (ns)	53.8379

In Figure 10(b), we show the normalized number of LOAD and EVAL instructions for varying values of D. We observe that the number of LOAD instructions decreases for increasing value of D. The number of EVAL instructions remains constant, as the number k is set to the same value for all dimensions. When the dimensions are bounded, less structures can be placed in the crossbar, resulting in a higher

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