Monitoring the Effects of Static Variable Orders on the Construction of BDDs

Khushboo Qayyum
University of Bremen
Bremen, Germany
khushboo@uni-bremen.de

Alireza Mahzoon
University of Bremen
Bremen, Germany
mahzoon@informatik.uni-bremen.de

Rolf Drechsler
University of Bremen/DFKI
Bremen, Germany
drechsler@uni-bremen.de

***Abstract*—The increasing complexity of modern *Integrated Circuits* (ICs) results in more bug occurrences during their design. In this regard, formal verification techniques such as the methods based on *Binary Decision Diagrams* (BDDs) mathematically ensure the correctness of these ICs before fabrication. However, BDDs are very sensitive to input variable ordering. Most of the time, the construction of BDDs fails due to size explosion caused by unfavorable input variable ordering. A lot of research has been done in the past to find a good variable ordering method; however, the focus has always been on the *endsize* of BDDs which may not present the whole picture.**

In this paper, we present a framework to monitor the BDD size during construction. We argue that in addition to the *endsize*, the highest number of nodes, i.e. the *peaksize*, should also be taken into account when selecting the static variable ordering method. The proposed framework uses a *variable order generator* to compute an input variable order for a circuit and a *BDD generator*, with CUDD at its heart, to monitor important parameters in particular *peaksize*. With the help of the framework, we observe how popular static variable ordering heuristics affect the *peaksize* and *endsize* of BDDs using popular benchmark sets, i.e., ISCAS85, ISCAS89.

I. Introduction

Recent advancements in technology have caused a huge increase in smart devices used per capita. With Industry 4.0 and *Internet of Things* (IoT) in play, the demand for functionally rich ICs has sky-rocketed. This demand for powerful devices along with the ever-increasing number of transistors on a microchip has caused a significant growth in the complexity and size of circuits. On one hand, a tiny IC is now capable of performing extraordinary tasks but on the other hand, the increasing size and functional complexity have made the design and verification of these ICs a laborious task. This difficulty in ensuring correctness leads to an increase in bugs escaping into silicon.

These bugs not only lead to faulty devices but can also cause monetary losses in cases like data loss and product recall [1]. Thus, it is crucial to verify these ICs before fabrication. Several methods based on simulation and formal verification have been developed to ensure the correctness of these circuits. In simulation-based techniques, the correctness of output values is checked for several input values. However, it is usually impossible to cover the whole input space for large designs. On the other hand, the formal verification methods ensure the correctness of ICs via mathematical procedures. As a result, the 100% correctness can be guaranteed. This makes them very desirable for both industry and academia, since simulation-based techniques cannot guarantee complete correctness. After decades of research, the area of formal verification has grown immensely. A vast number of methods and techniques are developed under formal verification which allows us to ensure the correct function of circuits before they come into physical form. Verification methods based on *Binary Decision Diagrams* (BDDs) [2], [3] are among the popular formal methods that prove the correctness of circuit function through equivalence checking. They are also used for *Polynomial Formal Verification* (PFV) of various digital circuits [4]–[9].

A BDD is a *Directed Acyclic Graph* (DAG) that represents a Boolean function. This representation is canonical in nature, i.e., every Boolean function has its unique BDD given a specific order of input variables. This simplifies the comparison of two circuits and thus allows easier functionality checks. A lot of research has been done to maximize the potential of BDDs in the last few decades. Despite their potential, they also have drawbacks like their size and their sensitivity towards certain architectures of circuits. The size of a BDD is heavily influenced by how the input variables of the respective Boolean function are ordered. Choosing a good input variable order can lead to size within polynomial limits, but an unfavorable order can result in an exponentially sized BDD. This exponential increase in size can lead to undesirably long run-times and in the worst-case scenario, the construction of a BDD fails as a result of insufficient memory. Therefore, the choice of a good input variable order becomes extremely crucial. In general, there are two ways to set the input variable orders:

- **Static Variable Ordering**: the order is determined/applied before the construction of a BDD.
- **Dynamic Variable Ordering**: the order is determined/applied during the construction of a BDD.

Both methods have been studied and researched for a long time. Several heuristics have been developed to find the connection between the arrangement of the input variable order to different aspects of a circuit. Typically, the goal of these heuristics is to find an input variable order that yields the smallest possible *endsize* of BDDs. However, there are aspects other than *endsize* that still remain overlooked. One of these

aspects is the growth pattern of BDD sizes with respect to an input variable order which sets the *peaksize*. The *peaksize* is a crucial parameter that determines the required memory for the BDD construction.

Although the *peaksize* can be monitored for both methods, in this work, we only consider the static variable ordering heuristics for the following reasons:

- to speed up the BDD construction, since dynamic variable ordering methods are slow and impractical for large circuits,
- to keep results deterministic,
- to associate the *peaksize* to a specific input variable order arrangement, and
- to be able to reproduce experiments and results.

In this paper, we propose a framework to provide a deeper insight into the construction of BDDs. We highlight that in addition to the lower final size (i.e., *endsize*) of a BDD, the maximum intermediate size (i.e., *peaksize*) has to be considered for a good input variable ordering. Reducing *peaksize* can help us with reducing the required memory during the BDD construction. Furthermore, it can also expedite the construction process. In order to monitor the size of BDD during the construction, we developed a framework with a *BDD generator* and a *Variable Order Generator* (VOG). The VOG uses static variable ordering heuristics to compute input variable order for a circuit. The BDD Generator, with CUDD [10] at its backbone, uses these input variable orders generated by the VOG to construct the BDD and monitor important parameters such as *peaksize*.

II. Related works

Several research works have proposed to improve the runtime and memory usage during BDD construction. They take advantage of various manipulation techniques to reduce the size of BDDs. The authors of [11] proposed a new streaming BDD manipulation that never causes memory overflow. The proposed method in [12] uses a new technique for computing BDDs, where the operands are themselves BDDs. The authors of [13] took advantage of a combination of top-down (decomposition-based) and bottom-up (composition-based) approaches to build BDDs. The work of [14] presented a new approach that replaces recursive synthesis operations using *If-Then-Else* (ITE) with MORE which is based on exchanges of neighboring variables and existential quantification.

In addition to manipulation techniques, several research works related to static variable ordering have been done for BDDs. The authors of [15] proposed two static ordering techniques based on the *fanin* heuristic and use the term "Max BDD" by which they actually refer to the largest BDD within all *Primary Outputs* (POs). Similarly, [16] proposed a *fanout* heuristic that considers only one output of the circuit for variable ordering. The authors of [17] used *Depth-first search* with interleaving for multiple outputs. The work of [18] considers multiple outputs for the *Breadth-First Search* (BFS) and *Depth-First Search* (DFS) heuristics. However, these works only monitor the *endsize* of BDDs and they do not take the *peaksize* into account.

In our work, we observe the effects of static variable ordering heuristics on the *endsize* and *peaksize* of BDDs.

III. Preliminaries

To make this paper self-contained, in this section we mention some important concepts that form the core of our work.

A. Binary Decision Diagrams

We first briefly summarize some basics of BDD:

- **BDD:** a directed, acyclic graph whose nodes have two edges associated with the values of the variables 0 and 1. A BDD contains two terminal nodes (leaves) that are associated with the values of the function 0 or 1.
- **Ordered BDD (OBDD):** a BDD, where the variables occur in the same order along each path from the root to a leaf.
- **Reduced OBDD (ROBDD):** an OBDD that contains a minimum number of nodes for a given variable order.

We refer to ROBDD as BDD in the rest of the paper. BDDs are canonical; therefore, if two circuits have identical functionality, their BDDs will also be identical under the same input variable ordering, regardless of the underlying architectures. Thus, the comparison of two circuits becomes trivial when BDDs are used since we only need to compare the root pointers of the BDDs [19]. This makes equivalence checking using BDDs a very powerful tool in the area of formal verification.

Despite being such a powerful tool in formal verification, BDDs have some weaknesses as well. One such weakness is the size of BDDs, which is very sensitive to how the input variables are arranged for constructing a BDD. It is possible that for a certain input variable order, the size of a BDD is polynomial, and for another one it may be exponential in size. Fig. 1 shows the BDDs created for the function $f(x1, x2, x3, x4, x5, x6)$ using two different variable orderings. The input variable ordering B produces a compact BDD whereas the input variable ordering A produces a large BDD.

The choice of input variable orders can be made before the construction using *static variable ordering heuristics* or during the construction of BDDs using the *dynamic variable ordering heuristics*.

B. Static Variable Ordering Heuristics

In static variable ordering heuristics, the input variables of a circuit are usually arranged by trying to find some relations between the architecture of the circuit and the ordering of input variables. A number of different heuristics have been developed in the past by leveraging the architecture of a given circuit or employing modern sorting or searching algorithms. Some heuristics are as follows:

- **Initial Order:** Variables are ordered as they are declared in the circuit description.
- **Reverse Order:** Initial input variable order is reversed.

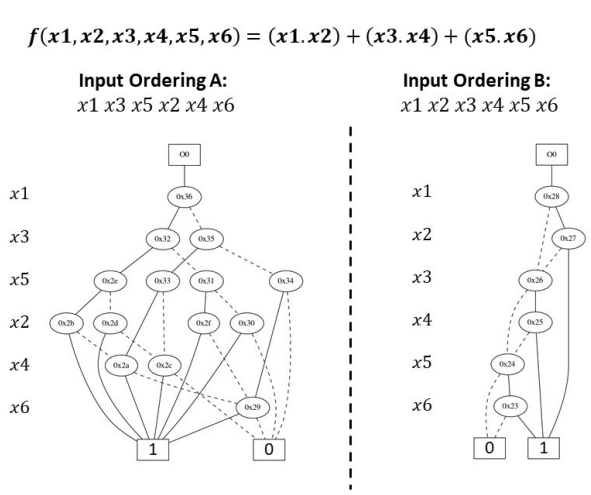


Fig. 1: BDD of a $f(x1, x2, x3, x4, x5, x6)$ using two different input variable orderings

- **Dependency Order:** The variables which influence more outputs are given priority [20].
- **DFS Order:** DFS applied to circuit [18].
- **Fanin Order:** The deeper inputs in the circuit are given priority [15].
- **Fanout Order:** The inputs with more fanouts in the circuit are given priority [17].
- **Random Order:** The input variable orders are generated randomly.
- **BFS Order with Interleaving:** BFS is applied to the circuit and the input variables that are unique to each output are inserted using interleaving [18].
- **BFS Order with Appending:** BFS is applied to the circuit and the input variables that are unique to each output are appended to the end [18] [17].

Some heuristics calculate input variable orders per output and the final input variable order for the complete circuit is created by arranging inputs variables based on priority of outputs they influence (i.e., the inputs of the output with higher priority have precedence).

C. CUDD

Colorado University Decision Diagrams (CUDD) is a package in C/C++ language that allows easy manipulation of BDDs. The package consists of a large set of functions that can be used to work with BDDs in different ways. With the help of the built-in functions, different statistics related to BDDs can be acquired.

IV. Proposed Work

In this section, we describe the proposed framework to monitor the effects of static variable ordering heuristics on the construction of BDDs. First, we provide an overview of the framework. Then, we describe each module of the framework in detail.

A. Framework

Fig. 2 shows the framework that we developed for monitoring the BDDs during construction. Based on the tasks, the framework can be divided into two parts

- Variable Order Generator
- BDD Generator

1) Variable Order Generator: The VOG, as shown on the top of Fig. 2, takes the circuit file as input and based on the heuristic of choice, generates the input variable order for the respective circuit. The generator starts with parsing the circuit file and sends the parsed data to the *variable extractor*. At the moment, parser is capable of processing *bench* and *isc* files. The parsed data is used by the variable extractor to obtain the necessary information about the circuit, e.g., the number of inputs and outputs and the structure of circuits. This information is utilized by the algorithms that run at the back-end of the VOG. These algorithms, based on their respective heuristics, arrange the input variables for the given circuit. Once the task of the VOG is completed and an input variable order is successfully computed, the order is saved in the *variable order database*; hence, it can be used by the *BDD generator*.

2) BDD Generator: The *BDD generator*, as shown at the bottom of Fig. 2, performs the construction and monitoring of the size of the BDD. When the input variable order is generated, the main BDD generator can proceed with its task of construction. The BDD generator also starts with parsing the circuit file and extracting all the necessary information that is required to construct the BDD. Once the circuit is parsed, a netlist of all the gates and their connections is created. After the netlist generation, the CUDD package is initialized and the input variables are arranged by the *variable sorting* module with the help of the previously generated order by the VOG. After this point, the BDD construction begins, starting from the outputs and traversing each gate in the netlist, until either an input is reached or the generation is complete. During each iteration, the total nodes that are created by the CUDD package for the purpose of constructing the BDD, are monitored by the *peaksize monitor*. The *peaksize* of the circuit along with the gate ID which is traversed during the iteration is saved in a separated file format (CSV) in the *monitoring database*. With every entry into the *monitoring database*, some additional parameters like the number of alive and dead nodes are also saved. The collective BDD nodes of all the outputs and the BDD size of individual outputs are also monitored as the outputs may share some of the nodes. After all the outputs of the circuits are completed, the final statistics like run-time and *endsize* are saved by the *endsize monitor* in the *monitoring database* and the framework is terminated. In this work, we perform the calculations and experiments on the complete circuits and aggregate the results of all the outputs. The *endsize* of the circuit can be defined as

$$endsize = \sum_{i=0}^{n} O_i - r_d, \tag{1}$$

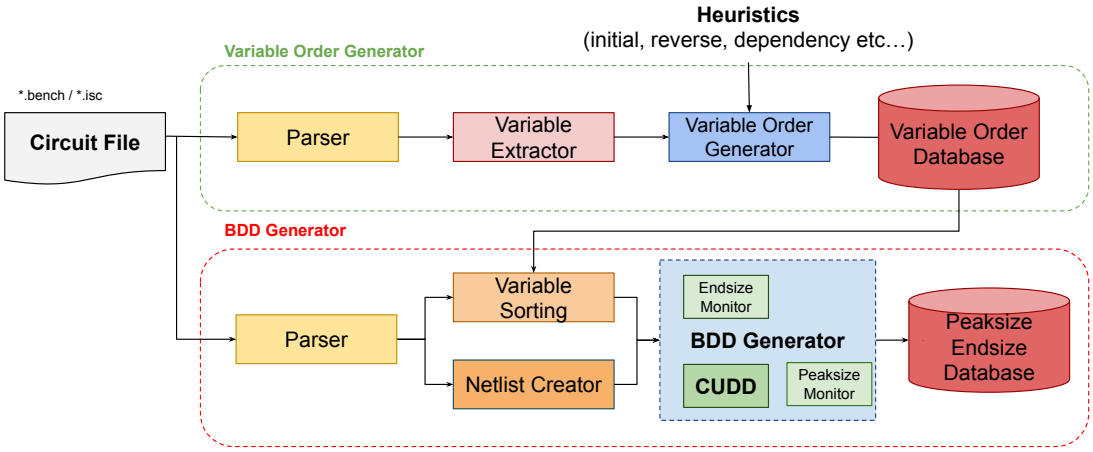
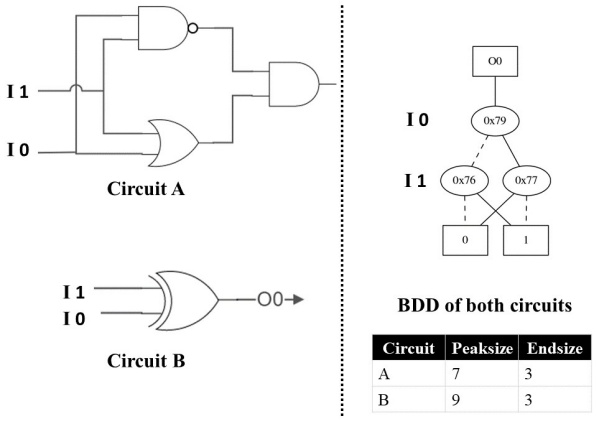


Fig. 2: BDD monitoring framework



Circuit	Peaksize	Endsize
A	7	3
B	9	3

Fig. 3: Comparison of peaksize of two simple circuits with same functionality and their BDD

where O_i is the final size of the ith BDD and n is the total number of outputs in a circuits. r_d is the number of nodes that are shared by BDD of each output.

With the help of the framework, the growth of BDDs can be closely monitored. This can be beneficial in two ways: 1) the growth pattern allows us to select static variable ordering heuristics that reduce the memory requirements of the BDD construction by reducing the *peaksize* of BDD. Choosing a heuristic that reduces the memory requirements can allow the BDDs to be constructed on resource-constrained devices. 2) Since the size of BDD is monitored so meticulously, the point where the BDDs construction fails is recorded. This can help us to identify and mitigate failures due to resource constraints by exploring circuit optimization techniques.

Fig. 3 shows the *peaksize* and *endsize* of two circuits with initial ordering (i.e., the input order is same as given in the circuit file). Both circuits represent an XOR gate.

$$f(A,B) = A \oplus B = (A+B)\cdot(\bar{A}+\bar{B}). \quad (2)$$

Circuit A uses the basic gates to implement this XOR gate (NAND, OR and AND) and Circuit B directly implements the XOR gate. It can be seen in the table in Fig. 3 that both of them have the same *endsize* but the *peaksize* is different. Thus a different choice of gates used can lead to different requirements of memory for the BDD construction. Therefore, by exploring different gate-level optimizations, the memory consumption during the BDD construction of a circuit can be reduced.

V. Experimental Results

The experiments have been carried out on an Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz with 15 GByte of main memory. The timeout (T.O.) for constructing BDDs is set to 3600 seconds.

We evaluate the efficiency of our framework using ISCAS85 [21], ISCAS89 [22] benchmark sets. The circuits with sequential profiles were first converted into combinational circuits, i.e., the flip flops within these circuits were removed and substituted with new input and output variables. The wide variety of circuits within these benchmark sets allows us to observe the trends and the effects of our desired parameters. We ran extensive experiments on all the circuits from each benchmark set but we present results for only some selected static ordering heuristics and circuits from each benchmark set to keep the results concise.

A. ISCAS85 circuits

Table I shows the circuits from the ISCAS85 benchmark set. The first column shows our selected variable ordering heuristic and the first row shows the name of the circuits. Within the ISCAS85 benchmark, we have selected c499 which is a 32-bit *Single Error Correcting* (SEC) circuit, and c880 which is an 8-bit *Arithmetic Logic Unit* (ALU). The c1355 circuit performs the same functionality as that of a c499 circuit but the XOR gates are replaced with an equivalent made from four NAND gates. The c2670 circuit is a 12-bit ALU and lastly, c6288 is a 16x16 multiplier circuit. The details about the selected

static variable ordering heuristics can be seen in the section III-B. The *peaksize* in the table shows the maximum number of nodes that were created during the construction of the BDD using the respective static variable ordering heuristics and the *endsize* shows the final number of nodes of all the outputs of the circuit as defined in Equation (1). The bold text within each column shows the smallest *peaksize* and *endsize* among the given static variable ordering heuristics for each circuit.

We can see from the results that generally, the *peaksize* of the circuits is larger in all cases, but even within the same circuit, some static variable ordering heuristics produce a *peaksize* that is substantially larger than the others. For example, in the circuit c880, the *peaksize* for the initial order is $\approx 22x$ larger than the BFS heuristic. This difference is also notable in the run-times of both scripts, in which the *initial order* is $\approx 10x$ slower with a run-time of $1.2sec$ and *BFS order* has a runtime of $0.12sec$. Another notable thing is the comparison of *peaksizes* and *endsizes* of c499 and c1355. Despite having the same functionality, the *peaksize* of c1355 is $\approx 3x$ larger than the *peaksize* of c499 and the construction of BDD for the initial order is two times slower for c1355. Using the BFS ordering heuristics, the ordering of c499 and c1355 generated by the algorithm is different which explains the different *endsizes*, which is in line with our claim. The c2670 also shows similar behavior where the heuristic with smaller *peaksize* is faster despite having a larger *endsize*. It should be noted that the last recorded *peaksize* for an ordering that is not completed is not always a large value. A smaller value of the last recorded *peaksize* for c2670 using *fanin order* points out a possible size explosion at the next step. The circuit c6288 in the ISCAS85 benchmark set fails to produce final BDDs for almost all the ordering heuristics. The BDDs of multipliers tend to explode and this is why almost every heuristic produces a large *peaksize* [2], [23]. The heuristics that manage to produce a final BDD, do it at an expense of a huge *peaksize* and longer run-times. In some cases, the construction time of BDDs skews a little despite having similar *peaksize* and *endsize*, this is due the code instrumentation done by the framework to monitor the *peaksize* and other attributes of the underlying circuits.

B. ISCAS89 circuits

Table II shows the circuits that were selected from the ISCAS89 benchmark set. Within the ISCAS89 benchmark set, we selected fractional multipliers cs420, cs838a, real chip based circuits cs9234 and cs13207, and a cs1423 circuit. The first column shows the selected static variable ordering heuristic and the first row shows the circuit. The *peaksize* values with the asterisk show the last known *peaksize* value that was recorded by the framework before the code was terminated. The bold values show the smallest *peaksize* and *endsize* of the BDDs of the respective circuits among different static variable ordering heuristics. It can be seen that the static variable ordering heuristics that produce higher *peaksize* take longer time to run. Most of the orders that produce higher *peaksize* also produce higher *endsize*. However in circuit cs9234, it can be seen that *fanin order*, despite producing a higher *endsize*, takes less time to execute than the *fanout* or *initial order* and the difference lies in the *peaksize* of both the circuits. The slight skew in the run times of circuits despite having similar *peaksizes* and *endsizes* is due to the code instrumentation. Sometimes two heuristics produce the same input variable order which yields identical *peaksizes* and *endsize*

VI. Discussion and Future Work

In this section, we discuss some observations and propose some future works with respect to the framework and *peaksize*.

In most of the cases, within the benchmark set, the trend of *peaksize* followed that of the *endsize*. Within the selected benchmark set, there was a wide variety of circuits available, but it was difficult to group them based on similarities in the circuit architecture. This hampered our ability to closely observe any architecture related trends. A more focused study of a similar category of circuits can be beneficial as it can allow clearer insight into the *peaksize* and how different types of circuits influence *peaksizes* with static variable ordering heuristics. For instance, it will be useful to see how static variable orderings affect the *peaksizes* of BDDs for complex arithmetic circuits.

This work was aimed to highlight the importance of *peaksize*. The aspect of optimizations on circuits to reduce *peaksize* is still unexplored. Exploration of optimization techniques to reduce the *peaksize* can help with the memory limitations as well as the long runtimes when constructing BDDs of complex circuits.

VII. Conclusion

In this paper, we presented a framework to monitor the BDD size during construction. The proposed framework used *variable order generator* to compute an input variable order for a circuit and a *BDD generator*, with CUDD at its heart, to monitor important parameters in particular *peaksize*. With the help of the framework, we observed how popular static variable ordering heuristics affect the *peaksize* and *endsize* of the BDDs using popular benchmark sets (ISCAS85, ISCAS89). With the help of experiments, we were able to show the importance of *peaksize* when considering the static variable ordering heuristics. A static variable ordering heuristic that lowers the *peaksize* of the circuit can help save memory and accelerate the construction.

Acknowledgement

This work was supported in part by DFG within the Reinhart Koselleck Project *PolyVer* (DR 287/36-1).

References

[1] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 1–33, 2016.

[2] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986.

[3] J. Kumar, Y. Miyasaka, A. Srivastava, and M. Fujita, "Formal verification of integer multiplier circuits using binary decision diagrams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2022.

TABLE I: Peaksize and endsizes of circuits from ISCAS85 benchmark set

Order/ Circuit	c499			c880			c1355		
	Peaksize	**Endsize**	**RT(s)**	**Peaksize**	**Endsize**	**RT(s)**	**Peaksize**	**Endsize**	**RT(s)**
Initial Order	**59811**	**45922**	0.13	1217071	346660	1.27	**184085**	**45922**	**0.22**
Reverse Order	128641	115655	0.17	659935	470046	1.11	456535	115655	0.46
Dependent Order	**59811**	**45922**	**0.12**	81712	**23012**	**0.10**	**184085**	**45922**	0.24
Fanin Order	**59811**	**45922**	0.13	1344228	640157	2.26	**184085**	**45922**	0.23
Fanout Order	83793	64684	0.15	460781	422467	1.03	257363	64684	0.32
Random Order	430001	360090	0.95	405047	241534	0.50	1349721	360090	1.91
BFS Order	122084	105086	0.22	**52466**	47344	0.13	454541	114905	0.48

Order/ Circuit	c2670			c3540			c6288		
	Peaksize	**Endsize**	**RT(s)**	**Peaksize**	**Endsize**	**RT(s)**	**Peaksize**	**Endsize**	**RT(s)**
Initial Order	1189426	T.O.	-	2586398	604559	5.48	**233838883**	41421674	826.90
Reverse Order	28175685	18975065	57.61	240807447	T.O.	-	352793222	**40563945**	**730.03**
Dependent Order	13589023	**7001000**	51.86	4466526	835489	10.30	338778640*	T.O.	-
Fanin Order	79018	T.O.	-	108707274	23034806	738.67	350502430*	T.O.	-
Fanout Order	**11688844**	8187717	**17.04**	6973165	1496121	27.67	**233838883**	41421674	943.48
Random Order	38759831	26359885	2715.19	43343705	7215573	270.54	294914597*	T.O.	-
BFS Order	47304115	34585106	1478.42	**1594785**	**366367**	**2.13**	286755982*	T.O.	-

RT = Runtime in seconds - T.O. = Timeout

Values with * show the last recorded peaksize - Bold values show the smallest value within each category

TABLE II: Peaksize and endsizes of circuits from ISCAS89 benchmark set

Order/ Circuit	cs420			cs838a			cs1423		
	Peaksize	**Endsize**	**RT(s)**	**Peaksize**	**Endsize**	**RT(s)**	**Peaksize**	**Endsize**	**RT(s)**
Initial Order	271026	262172	0.24	2099241*	T.O	-	268344	98454	0.51
Reverse Order	**736**	217	**0.03**	**1849**	679	**0.07**	139202	52730	**0.15**
Dependent Order	817	**186**	0.05	3698	**627**	0.09	84669	26505	0.24
Fanin Order	1673	1073	0.05	3084	723	0.10	1141117	601012	2.14
Fanout Order	271026	262172	0.29	2099241*	T.O	**-**	268344	98454	0.41
Random Order	1461	610	0.05	8652	6203	0.08	175435	94502	0.29
BFS	9502	9109	0.05	4427	3663	**0.08**	**43897**	**18481**	0.17

Order/ Circuit	cs9234			cs13207			cs35932		
	Peaksize	**Endsize**	**RT(s)**	**Peaksize**	**Endsize**	**RT(s)**	**Peaksize**	**Endsize**	**RT(s)**
Initial Order	7868760	4548997	22.58	26117942	676681	103.41	20810	5708	9.26
Reverse Order	16681646	10481747	28.40	59987033	582638	4404.25	56243	15049	7.73
Dependent Order	1730839	456446	5.18	42895	22730	2.79	**21050**	**5628**	9.58
Fanin Order	5513757	5174415	18.28	33947536*	T.O	**-**	48291	21857	10.10
Fanout Order	7868654	4548993	22.19	26117956	676684	91.06	**21050**	**5628**	7.80
Random Order	4010697	3078084	5.93	20848899*	T.O	**-**	40346	10303	6.52
BFS	**106061**	**78695**	**0.80**	**35877**	**17602**	**1.92**	22229	6299	**6.30**

RT = Runtime in seconds T.O. = Timeout

Values with * show the last recorded peaksize - Bold values show the smallest value within each category

[4] R. Drechsler, A. Mahzoon, and L. Weingarten, "Polynomial formal verification of arithmetic circuits," in *ICCIDE*, 2021, pp. 457–470.

[5] A. Mahzoon and R. Drechsler, "Polynomial formal verification of prefix adders," in *ATS*, 2021, pp. 85–90.

[6] ——, "Late breaking results: Polynomial formal verification of fast adders," in *DAC*, 2021, pp. 1376–1377.

[7] R. Drechsler, A. Mahzoon, and M. Goli, "Towards polynomial formal verification of complex arithmetic circuits," in *DDECS*, 2022, pp. 1–6.

[8] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *DDECS*, 2021, pp. 99–104.

[9] R. Drechsler and A. Mahzoon, "Polynomial formal verification: Ensuring correctness under resource constraints," in *ICCAD*, 2022.

[10] F. Somenzi, "CUDD: CU decision diagram package release 2.7.0," available at https://github.com/ivmai/cudd, 2018.

[11] S. ichi Minato, "Streaming BDD manipulation," *TC*, vol. 51, no. 5, pp. 474–485, 2002.

[12] T. R. Shiple, R. K. Brayton, and A. L. Sangiovanni-vincentelli, "Computing Boolean expressions with OBDDs," 1993.

[13] J. Jain, A. Narayan, A. Sangiovanni-Vincentelli, C. Coelho, R. K. Brayton, S. P. Khatri, and M. Fujita, "Decomposition techniques for efficient ROBDD construction," in *FMCAD*, 1996, pp. 419–434.

[14] A. Hett, R. Drechsler, and B. Becker, "MORE: an alternative implementation of BDD packages by multi-operand synthesis," in *European Design Automation Conf.*, 1996, pp. 164–169.

[15] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *ICCAD*, 1988, pp. 6–9.

[16] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and improvement of Boolean comparison method based on binary decision diagrams," in *ICCAD*, vol. 88, 1988, pp. 2–5.

[17] H. Fujii, G. Ootomo, and C. Hori, "Interleaving based variable ordering methods for ordered binary decision diagrams," in *ICCAD*, 1993, pp. 38–41.

[18] K. M. Butler, D. E. Ross, R. Kapur, and M. R. Mercer, "Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams," in *DAC*, 1991, pp. 417–420.

[19] A. J. Hu, "Formal hardware verification with BDDs: An introduction," in *PACRIM*, vol. 2, 1997, pp. 677–682.

[20] R. Drechsler, "Evaluation of static variable ordering heuristics for MDD construction," in *ISMVL*, 2002, pp. 254–260.

[21] F. Brglez and H. Fujiwara, "A neural netlist of 10 combinational benchmark designs and a special translator in fortran," in *ISCAS*, 1985, p. 669.

[22] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *ISCAS*, 1989, pp. 1929–1934.

[23] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication," *TC*, vol. 40, no. 2, pp. 205–213, 1991.