Start Small But Dream Big: On Choosing a Static Variable Order for Multiplier BDDs

Khushboo Qayyum, Alireza Mahzoon, and Rolf Drechsler

1 Introduction

With the size of *Integrated Circuits* (ICs) getting smaller and their functionality getting more complex, the task to assert the correctness of an IC becomes crucial. It is imperative that functionality of chips is thoroughly verified before silicon to prevent bugs from escaping into the final product. These bugs not only cause malfunctions but are a threat to the security of systems and a cause of monetary losses [1]. In this regard, formal verification techniques allow reliable verification of ICs using mathematical proof. Among formal verification methods, *Binary Decision Diagram* (BDD)-based methods are widely used to prove the correctness of ICs. A BDD is a canonical representation of a Boolean function as a *Directed Acyclic Graph* (DAG) [2]. Therefore, if two Boolean functions perform the same task, their BDDs will be the same if the input variables of both functions are in the same order regardless of how the Boolean function is defined. This attribute of canonicity allows two circuits to be easily compared and verified. State-of-the-art tools can perform this verification by a simple root pointer comparison of the BDDs of two Boolean functions [1].

However, one of the main challenges of using BDDs is to find a good order of the input variables. The size of BDDs is very sensitive to the input variable order; therefore, a good order of input variables may produce a size of BDD within polynomial order, but a bad order can cause the size of a BDD to be of exponential

K. Qayyum (✉) · A. Mahzoon
University of Bremen, Bremen, Germany
e-mail: khushboo@uni-bremen.de; mahzoon@uni-bremen.de

R. Drechsler
Institute of Computer Science, University of Bremen and Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
e-mail: drechsle@uni-bremen.de

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_11

order. This attribute of a BDD calls for the choice of input variable order to be perceptive. A number of different heuristics have been developed in the past that try to associate the arrangement of input variables with different aspects of circuits' architecture or by using searching and sorting techniques [3–7]. These heuristics can be divided into two main categories based on when the input order is applied to the BDD construction. In *static variable ordering* heuristics, the input variable order is arranged and decided before the construction of BDDs, and in *dynamic variable ordering* heuristics, the input variable order is applied during the construction of BDD.[1]

The erratic behavior of the BDD size is not just limited to the ordering of the input variables, but the size is also sensitive to the structure of underlying function. This behavior is particularly evident in BDDs of complex arithmetic circuits. Arithmetic circuits make integral part of ICs; therefore, their correct functionality is essential. Bugs like Pentium FDIV can render a chip useless if they are not identified before silicon. Within the category of arithmetic circuits, *multipliers* have piqued the interest of researchers for a long time as the BDDs of multiplier circuits tend to explode in size even at substantially small multipliers [2]. Due to this explosion, constructing the BDDs for multipliers requires tremendous hardware resources [8]. Furthermore, using systems with insufficient resource requirements lead to prolonged runtimes only to result in failure at the end. Keeping the concern for the size of BDDs for multipliers in mind, the choice of an appropriate input variable ordering for a multiplier becomes paramount as a good input variable ordering can help in reducing this size explosion. Additionally, an early estimation of memory requirement can facilitate an appropriate selection of resources and thus save time and effort.

Contribution In this paper, we present a methodology to choose an optimal static variable ordering heuristic for larger multipliers with an early estimation of the *endsize*, *peaksize*, and *memory* required for constructing the BDD nodes. Our proposed methodology allows a fast and resource-efficient optimal static variable ordering heuristic selection. The estimation of the size and memory requirements of the large multiplier allows a more insightful selection of resources for the BDD construction. In our methodology, we first obtain the smaller version of the target multiplier and perform analysis using various static variable ordering heuristics. Our results show that a static variable ordering heuristic that is optimal for a scaled-down circuit is also optimal for the larger circuit while requiring only a fraction of time and memory resources. For the endsize estimation, we reuse chosen heuristics and incrementally increase the size of the circuit to estimate the endsize and the peaksize of the larger circuits. Using the estimated peaksize of the BDD, a conservative estimation of the memory requirements for the BDD nodes can be determined with high accuracy. We perform extensive experiments on multipliers obtained using the GenMul [9] multiplier generator.

[1] In this work, we overlook dynamic variable ordering heuristics as these heuristics can be counterintuitively slow and thus prohibitive for complex circuits like multipliers.

2 Related Work

Multipliers have been a subject of interest for a long time, and multiple strategies have been developed specifically to address the size complexity of the multiplier BDDs. In [10], the authors address the complexity of multipliers BDDs through the introduction of input variables although the complexity is reduced but not solved completely. Recently, in [11] the authors also present a method to decrease the complexity of verification of the multiplier BDDs; however, an optimal variable ordering is not addressed. Multiple different static variable ordering heuristics have been developed in the past. In [5–7] authors exploit circuit architecture to come up with a suitable input variable ordering, whereas in [3, 4], searching algorithms are applied to the circuit to find a good input variable ordering. Most of these consider only the endsize of the BDD and only for a single output.

The memory usage issue of BDDs is addressed by several works [12–15] using different approaches in constructing and manipulation of the BDDs. The estimation of size of BDDs has been attempted using timed automata by [16, 17] for generic circuits.

Our work differs in how the problem is approached with only multipliers as our target circuits. We focus mainly on choosing an optimal static variable ordering heuristic from the already developed heuristics. For calculations, we consider all the outputs for endsize and also consider the peaksize of the BDDs. Additionally, we focus on memory estimation instead of memory management.

3 Preliminaries

3.1 Binary Decision Diagrams

BDDs are a tree-like representation of a Boolean function created using Shannon expansion. Once ordered and reduced, these *Reduced Ordered BDDs* (ROBDDs) form a canonical DAG for the given Boolean function. In our work, we refer to the ROBDDs as BDDs. The canonicity of the BDDs is indifferent to the architecture of the underlying function. That is, given two Boolean functions, their BDDs will be the same if :

- both the functions perform the same tasks regardless of the underlying architecture and
- both the graphs are made with input variables arranged in the same order.

This makes the comparison of two circuits a trivial task. For this reason, the BDDs are favored in the area of formal verification. However, the size of BDDs and their sensitivity to certain circuit architectures sometimes undermines its performance and ease. The ordering of input variables of a Boolean function heavily influences the size of its BDD. Thus, a good choice of the input variable order plays a decisive role in finding an optimally sized BDD. Although finding an optimal input variable order for a BDD is NP-hard [2], various heuristics have been developed to address this problem. Using these heuristics, the input variables can be arranged during the construction of BDD using dynamic variable ordering heuristics or before the construction of BDD using static variable ordering heuristic like the ones given below.

- **Initial Order:** Input variable order as they are defined in the circuit description.
- **Reverse Order:** Reverse of the initial order.
- **Dependency Order:** The variables influencing more outputs of the circuit get precedence [7].
- **Depth-First Search Order:** Depth-First Search (DFS) is used to determine input variable order [3].
- **Fanin Order:** The inputs that are deeper in the circuit get precedence [6].
- **Fanout Order:** The inputs with more fanouts in the circuit get precedence [4].
- **Random Order:** The input variable order is generated randomly.
- **Breadth-First Search Order** Breadth-First Search is used to determine the ordering for the given circuit [3].

Although dynamic variable ordering heuristics are capable of producing better outcomes, their excessive runtimes make them counterproductive. For this reason, in this work, we only focus on static variable ordering heuristics.

3.2 Multipliers

Multipliers are essential components in modern ICs. Many pivotal applications like encryption, digital signal processing, etc. require multipliers. Different types of multiplier architectures have been developed over time to satisfy demands in aspects like power, speed, area, and accuracy. When individually compared, these architectures are apparently different, but based on their internal functions, a multiplier can be broadly represented as a three-stage structure as represented in Fig. 1. Each of these stages performs the following task:

- *Partial Product Generator* (PPG): generates the partial product from the multiplier and multiplicand.
- *Partial Product Accumulator* (PPA): reduces and aggregates the partial products.
- *Final Stage Adder* (FSA): sums up all the result of the PPA to generate the final product.

The PPGs for multipliers can be implemented using simple AND gates and Booth encoding algorithm. Likewise, some of the examples of PPAs are array, Wallace tree, counter-based Wallace, and Dadda tree algorithms. For the FSA stage, architectures like Brent-Kung, ripple carry, carry look-ahead, Lander-Fischer, Kogge-Stone, and Carry-skip can be used.

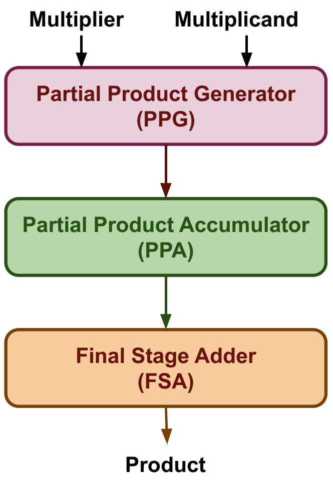

Multiplier
Multiplicand
Partial Product Generator (PPG)
Partial Product Accumulator (PPA)
Final Stage Adder (FSA)
Product

Fig. 1 Multiplier structure

4 Methodology

In this section, we explain our proposed methodology. First, we present an overview and later we explain each step of our methodology.

4.1 Overview

The overview of our proposed methodology is illustrated in Fig. 2. Our proposed methodology is comprised of two steps. In the first step, shown in Fig. 2 by the red solid line, a smaller version of the target multiplier circuit is obtained. The behavior of BDDs for a number of different static variable ordering heuristics is observed for the smaller scaled-down circuit, and the most optimal heuristic is identified. Once the optimal heuristic is selected, the first step of the methodology is concluded and the second step begins. In the second step, shown as a blue dashed line in Fig. 2, we build a set of circuits which are also scaled-down versions of the target multiplier. The circuits in this set have the same structure but incrementally increase in size. The optimal static variable ordering heuristic is applied to this set of circuits, and different parameters related to the size of BDDs of each circuit are collected. Using these parameters, the endsize, peaksize, and memory requirement of the target multiplier are estimated.

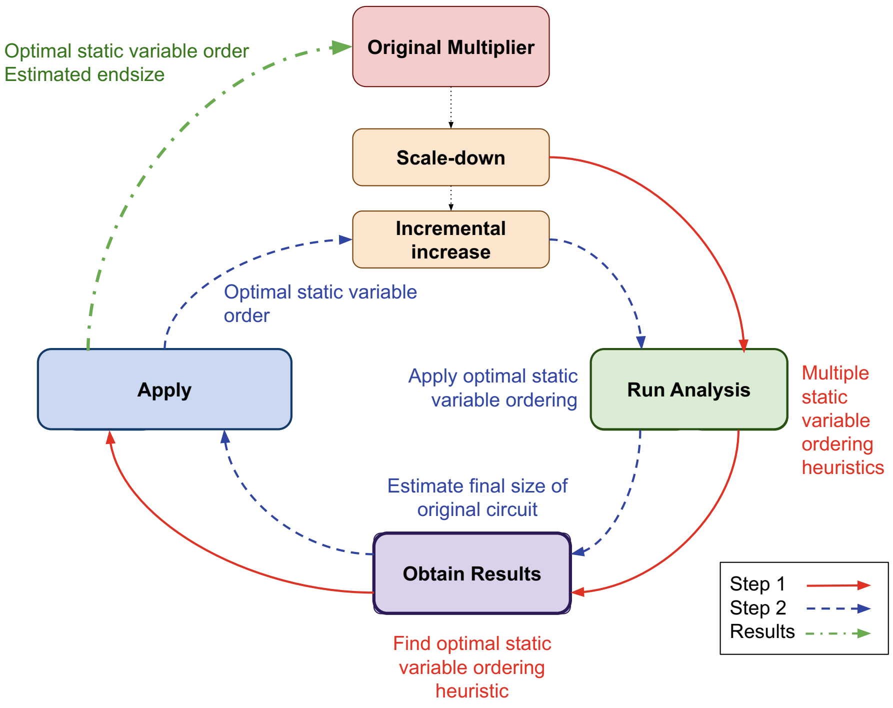

Optimal static variable order
Estimated endsize
Original Multiplier
Scale-down
Incremental increase
Optimal static variable order
Apply
Apply optimal static variable ordering
Run Analysis
Multiple static variable ordering heuristics
Estimate final size of original circuit
Obtain Results
Find optimal static variable ordering heuristic
Step 1
Step 2
Results

Fig. 2 Proposed two-step methodology

4.2 Optimal Static Variable Order Selection

The first step of our proposed methodology aims to find an optimal static variable ordering heuristic for the target multiplier. This process is represented in Fig. 2 by the red solid line. The idea is to work on smaller version of the target circuit to find the optimal static variable ordering heuristic and then use it for target multiplier. This helps to reduce the time and resources required for constructing the BDDs that are otherwise substantial when the larger multiplier is used. Consider a target circuit that is a 64 $\times$ 64 bit signed multiplier with a Array PPA and a Brent-Kung FSA. The scaled-down version is a 8 $\times$ 8 bit signed multiplier with the same Array PPA and a Brent-Kung FSA. We use GenMul [9] to obtain multiplier circuits in various structures and sizes. Our proposed methodology is agnostic to the underlying tools, thus, any BDD construction tool can be adopted that provides information about the endsize and the peaksize of the BDD. In order to find the optimal variable ordering heuristic, we use an in-house framework with CUDD [18] at its heart. This framework finds input variable orders for a given circuit using different heuristics and then uses these input variable orders to construct BDDs for the circuits. In addition to the BDD construction, the framework also monitors different parameters of BDDs like peaksize during construction and endsize. Since the GenMul tool only

Table 1 Results of static variable ordering heuristics for the same multiplier in different sizes

Static variable ordering heuristic	Circuit size		
	8 × 8	10 × 10	12 × 12
Initial order	6480	50,085	391,891
Reverse order	**6386**	**48,876**	**374,537**
Dependent order	12,916	129,321	1,276,275
Fanin order	6969	51610	390512
Fanout order	6480	50085	391891
Random order	16710	230951	2524622
BFS appending	12618	123378	1178732
Initial order interleaved	12916	129321	1276275

provides circuits in Verilog, we convert the circuits into bench format that can be processed by our framework. This can be done using the Yosys tool [19].

The framework generates input variable orderings for the scaled-down multiplier using different heuristics. Using the generated input variable orders, it constructs the BDDs for the circuit and records their endsizes and peaksizes in a database. The details of these static variable ordering heuristics are given in Sect. 3.1. Once the results of all the static variable ordering heuristics are available, the optimal heuristic is chosen. The choice of the optimal heuristic can be based on different factors, e.g., smallest endsize and resource usage. With the decision of the optimal static variable ordering heuristic, the first step of the methodology concludes, and the second step for the endsize and peaksize estimation for the target multiplier begins.

Revisiting the earlier example, Table 1 shows the endsizes of the Array PPA and Brent-Kung FSA multiplier for three different sizes. The bold values show the smallest values. It can be seen that heuristic that performs well for 8 × 8 also performs well for the larger 10 × 10 and 12 × 12 multiplier thus in line with our claim that the heuristic that performs well for smaller circuits also performs well for larger circuits.

4.3 BDD Endsize and Peaksize Estimation

In the second step, we estimate the endsize and the peaksize of the target multiplier. Since the BDDs of multipliers usually explode in size, an estimation of the peaksize and endsize can help in projecting the required resources for the construction of their BDDs. This process is shown by the blue dashed line in Fig. 2. In this step, we use the optimal static variable ordering heuristic that is previously selected in the first step of the methodology. We obtain two or more circuits from the GenMul tool with the same structure to perform the experiments. These circuits are slightly larger than the scaled-down version of the target multiplier. In our set of circuits, we obtain multipliers with the same architecture with 8 × 8, 9 × 9, and 10 × 10 bit-size. Our framework constructs the BDDs of these circuits and extracts vital information such as the endsize and peaksize of the BDDs of each circuit using the selected static variable order heuristic. Once this information is available, trends are observed with respect to the growth of BDD. Using these trends, the growth factors are calculated, and these growth factors are used to estimate the endsize and peaksize of the target multiplier using the following equations:

$$\hat{e_y} = {d_e}^{y-x} \times e_x \tag{1}$$

$$\hat{p_y} = {d_p}^{y-x} \times p_x \tag{2}$$

where $\hat{e_y}$ and $\hat{p_y}$ are the estimated endsizes and peaksizes of the target multiplier and e_x and p_x are the endsize and peaksize of the scaled-down multiplier. x is the bit-size of the inputs of the scaled-down multiplier and y is the bit-size of the input of the target multiplier. d_e and d_p are the growth factors per bit of the endsize and peaksize, respectively, for the given multiplier structure. The peaksize of a BDD shows the maximum number of nodes that were created throughout the construction of the BDD. Therefore, the peaksize dictates the memory consumption during the construction of a BDD. Using the peaksize and the memory required by a single node, the estimation of the memory required by the target BDD is calculated as follows:

$$memory\_required = \hat{p_y} \times size\_per\_node \tag{3}$$

However, the memory estimation is conservative as they do not include the auxiliary memory that maybe required for processing. Regardless, they can allow for a more insightful resource allocation and thus produce practical runtimes for BDD construction.

5 Experiments

In this section, we present the experimental results of the proposed methodology to select the optimal static variable ordering heuristic and estimation of peaksize and endsize. In our work, we obtained different multiplier structure combinations using the GenMul tool. We applied our methodology to a wide range of multiplier structures, but for brevity, we present results for only a few static variable ordering heuristics and multiplier structures.

5.1 Selection of Optimal Heuristic

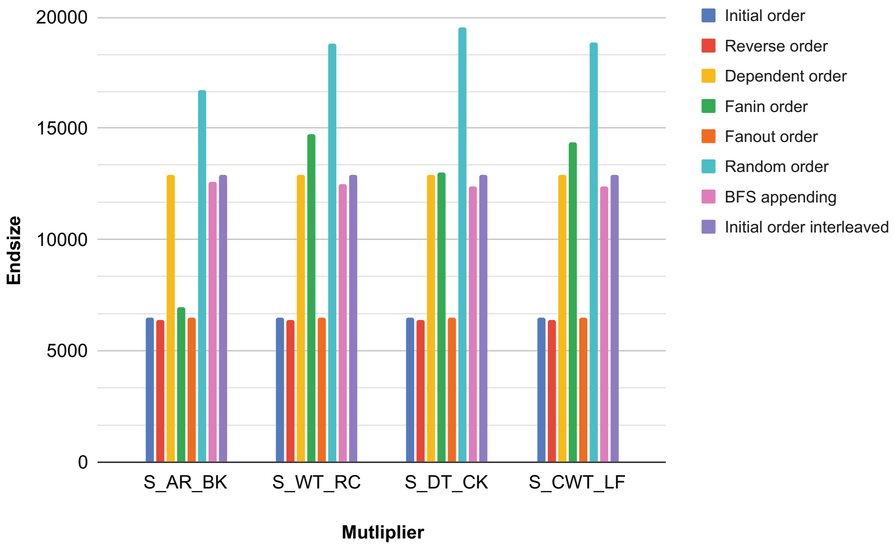
Figure 4 shows the endsize and peaksize of different static variable ordering heuristics for four different 8×8 multiplier structures. The naming of the multipliers in the figures is in the X_A_B format where X is the PPG (S = signed simple, U = unsigned simple), A is the PPA type (AR = Array, WT = Wallace tree, DT = Dadda tree, CWT = counter-based Wallace tree), and B represents its FSA type (BK = Brent-Kung, RC = ripple carry, CK = carry skip, LF = Ladner-Fischer). The x-axis represents the multiplier structure and the y-axis shows the number of nodes. When choosing the optimal static variable ordering heuristic based on the endsize, the lowest value would be considered. From Fig. 3a which shows the endsize for signed multipliers, it can be seen that the *reverse* order performs well for all the multiplier structures, but in Fig. 4a which shows the endsize of unsigned multipliers, *initial* and *fanout* ordering heuristics outperforms the other heuristics. The initial and reverse ordering heuristics are less intuitive, but it is interesting to see that there are other heuristics like the fanout ordering that produce similar results as that of initial order. And consequently its performance would match the reverse order. While some heuristics look oblivious to the structures, heuristics like *fanin order* seem to be affected by the structure of the multiplier. The random order performs the worst for all the selected structures which reinforces that the selection of input variable order should be rational.

Figures 3b and 4b show the peaksizes using different static variable ordering heuristics for signed and unsigned multipliers, respectively. When selecting the static variable ordering heuristic based on the peaksize, the difference seems to be less obvious. It seems so because on average the peaksize is $10x$ larger than the endsize; therefore, the difference between orders is less evident. Within our selected signed multiplier structures (Fig. 3b), the reverse order performs well but this is not universal. For the unsigned multiplier of Array and Brent-Kung combination (U_AR_BK) as evident in Fig. 4b, the BFS produces a much smaller peaksize and thus would consume fewer resources and therefore would be the choice for optimal heuristic when the peaksize is considered.

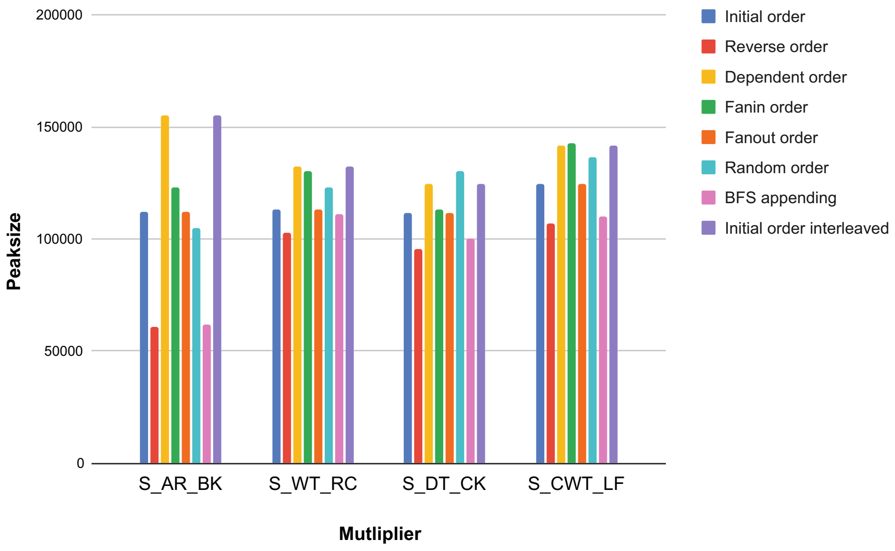
5.2 *Estimation of Endsize and Peaksize of the BDDs*

From the results of step 1, as evident from Fig. 3a, the reverse order was selected as the optimal static variable ordering heuristic for the multiplier structure S_AR_BK. Since our scaled-down version of multiplier was 8×8, the circuits in this set for estimation are of size 9×9, 10×10, and 11×11. Although a set of three increments would suffice, more incremental circuits would result in a better estimation. We obtained a set of circuits of all the multiplier structures with incremental increase in size. However, due to space constraints, we only show the result for one of the multiplier structure, i.e., signed multiplier with Array PPA and Brent-Kung FSA (S_AR_BK). As expected in our results, the growth factor of the peaksize was slightly greater than the endsize (2.8 and 2.6, respectively).

Figure 5a, b shows the estimated endsize and peaksize using Equations 1 and 2 for our selected multiplier structure. The x-axis represents the size of the multiplier in bits, and the primary y-axis shows the number of nodes, and the secondary y-axis

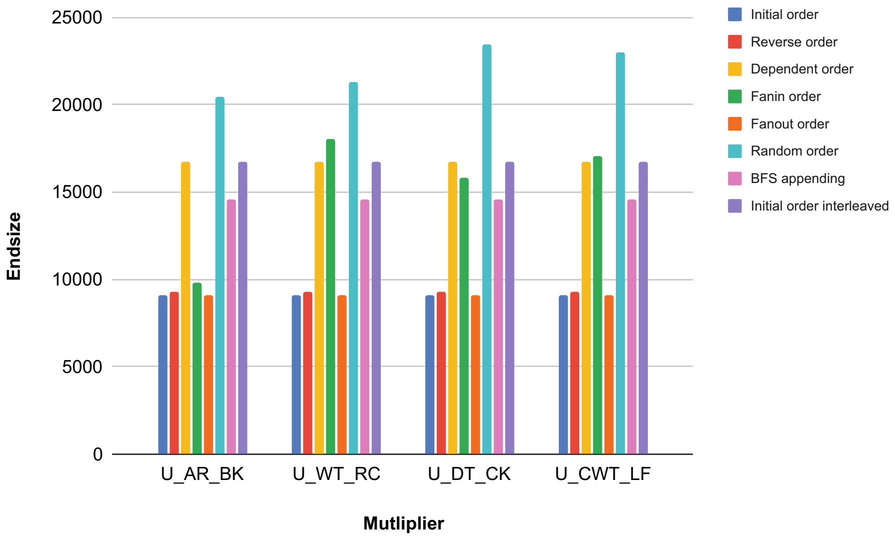


(a)

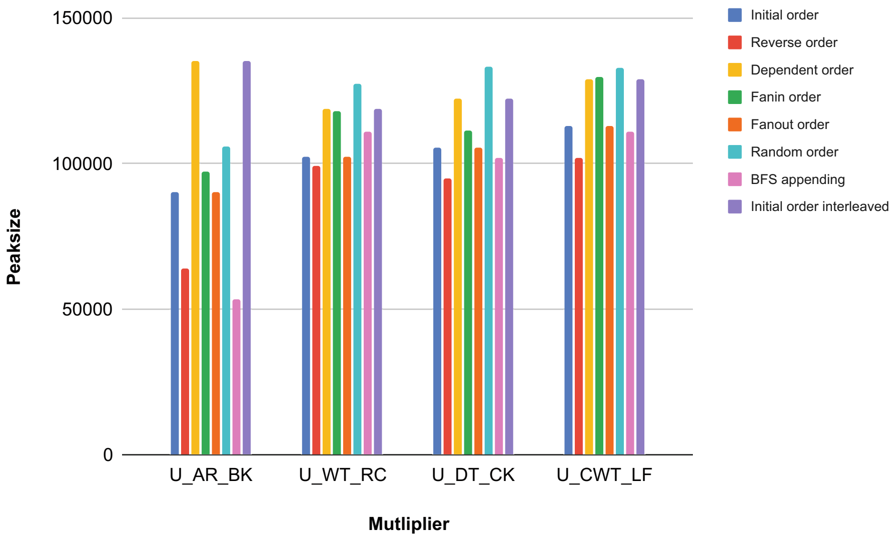


(b)

Fig. 3 Endsize and peaksize of four 8×8 signed multiplier structures for different static variable ordering heuristics. (**a**) Endsize for signed multipliers. (**b**) Peaksize for signed multipliers

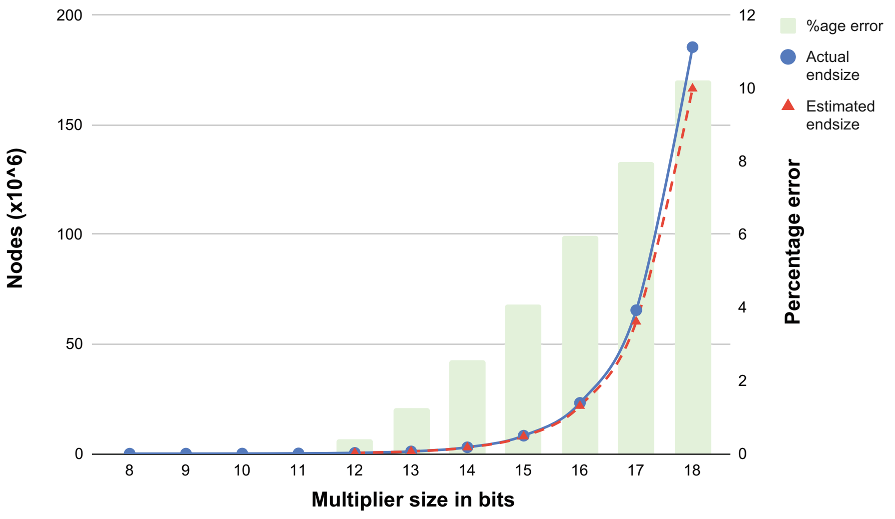


(a)

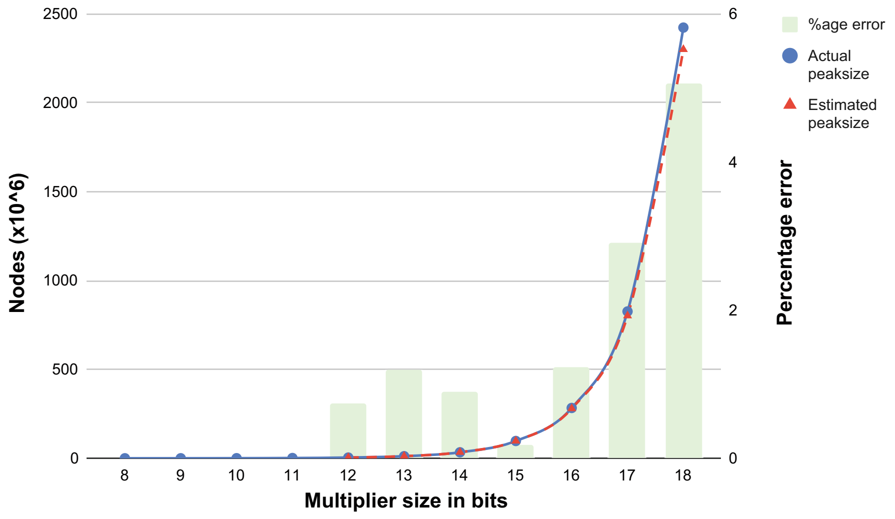


(b)

Fig. 4 Endsize and peaksize of four 8×8 unsigned multiplier structures for different static variable ordering heuristics. (**a**) Endsize for unsigned multipliers. (**b**) Peaksize for unsigned multipliers



(a)



(b)

Fig. 5 Estimated endsize and peaksize for the signed multiplier with Array PPA and Brent-Kung FSA (S_AR_BK). (**a**) Endsize. (**b**) Peaksize

gives the percentage error between the actual and the estimated values. The solid blue lines show the actual endsize and peaksize, and the red dashed lines show the estimated endsize and peaksize for these circuits. The percentage error is shown by the bar graph (green) in the background of the respective graphs.

As visible from Fig. 5a, b, the estimated values follow the same trends as the actual values. Consider the 16×16 bit multiplier in Fig. 5, the estimated endsize is $21, 967, 242$ and the estimated peaksize is $279, 660, 485$. To calculate the percentage error, we constructed the BDD for the circuit sizes that were achievable. For the 16×16 bit multiplier, the percentage error in endsize is $\approx 5.9\%$ and for peaksize it is only $\approx 1.2\%$. Although the error in the estimated endsize shows an increasing trend in this case, increasing the number of circuits to calculate the growth factors will help in decreasing the error percentage. The error in the estimated peaksize does not show a constant increase as the endsize and is very small in contrast to the endsize. Thus, the values that are calculated for the memory required by the BDD nodes using the estimated peaksize can be reliable.

5.3 Memory Usage Estimation

A single node on CUDD package requires 32-bytes when compiled using 64-bit pointer system (16-bytes for 32-bit pointers) [18]. Table 2 shows the estimated values for endsize, peaksize, and memory requirement of BDD nodes constructed using reverse ordering for a signed multiplier with Array PPA and Brent-Kung FSA (S_AR_BK) for larger sizes. Using these values, for the given structures the minimum memory requirement for a 16×16 multiplier, excluding the auxiliary memory required by the CUDD package, is ≈ 9 GB memory for 64-bit systems. Based on the estimated values, for the 18×18 multiplier, we ran it on system with memory resource less than estimated values (Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz, main memory = 64 GB). As expected, the BDD construction failed after running for an extended period (runtime $>$ 24 h). Later, we constructed the BDD on a system having resources greater than our memory estimate (Intel(R) Core(TM) i9-11900KF @ 3.50 GHz, main memory = 125 GB), and the construction was successfully completed in a reasonable time ($\approx$ 1 h). Thus, it reinforces the confidence in the estimated values and in the idea that an early memory estimation allows for a more efficient selection of resource and utilization of time.

6 Discussion

In this section, we discuss some observations and possible extensions of our work. Although we applied our methodology to only traditional BDD construction methods, we believe that it can easily be adapted to other methods used for constructing BDDs like the one proposed in [10]. In addition to that, the estimated

Table 2 Estimated endsize, peaksize, and memory requirement for S_AR_BK multiplier using reverse ordering

Multiplier sizes	Estimated endsize	Estimated peaksize	Estimated memory required by CUDD[a]
16 × 16	2.18 × 10^7	2.80 × 10^8	9 GB
18 × 18	1.67 × 10^9	2.30 × 10^9	74 GB
32 × 32	2.54 × 10^14	5.87 × 10^15	1.88 × 10^5 TB
64 × 64	3.23 × 10^28	2.58 × 10^30	8.26 × 19^19 TB

[a] conservative estimate for nodes only

memory requirement is not just useful for resource selection for BDD construction; it can also help in exploring other options for construction of multipliers in case the available resources appear to be insufficient. The effects of approaches which strive to reduce memory usage can also be explored and how these methods effect the growth factors. Another interesting aspect would be the assessment of methodology for arithmetic circuits other than multipliers, and it would be insightful to see how the methodology and estimation extends to these circuits.

7 Conclusion

In this paper, we presented a methodology to choose an optimal static variable ordering heuristic for larger multipliers with early estimation of the endsize, peaksize, and memory requirements for constructing the BDD. Using the smaller version of the target multiplier structure, we were able to find an optimal static variable ordering heuristic that also works equally well for the target multiplier. For the endsize estimation, we reused the chosen heuristics and collected a set of multiplier circuits of the same structure with incremental increase in size to find a growth factor per bit for the endsize and peaksize. This growth factor was used to estimate the endsize and the peaksize of the target multiplier. Using the estimated peaksize, we were also able to project the memory required in constructing the BDD. We demonstrated the applicability of our methodology on various multiplier circuits.

References

1. Hu, A.J.: Formal hardware verification with BDDs: An introduction. In: PACRIM, vol. 2, pp. 677–682 (1997)
2. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **100**(8), 677–691 (1986)
3. Butler, K.M., Ross, D.E., Kapur, R., Mercer, M.R.: Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In: Design Automation Conference, 417–420 (1991)

4. Fujii, H., Ootomo, G., Hori, C.: Interleaving based variable ordering methods for ordered binary decision diagrams. In: International Conference on Computer-Aided Design, pp. 38–41 (1993)
5. Fujita, M., Fujisawa, H., Kawato, N.: Evaluation and improvement of Boolean comparison method based on binary decision diagrams. In: International Conference on Computer-Aided Design, vol. 88, pp. 2–5 (1988)
6. Malik, S., Wang, A., Brayton, R., Sangiovanni-Vincentelli, A.: Logic verification using binary decision diagrams in a logic synthesis environment. In: International Conference on Computer-Aided Design, pp. 6–9 (1988)
7. Drechsler, R.: Evaluation of static variable ordering heuristics for MDD construction. In: International Symposium on Multi-Valued Logic, pp. 254–260 (2002)
8. Drechsler, R., Mahzoon, A.: Polynomial formal verification: Ensuring correctness under resource constraints. In: International Conference on Computer-Aided Design (2022)
9. Mahzoon, A., Große, D., Drechsler, R.: GenMul: Generating architecturally complex multipliers to challenge formal verification tools. In: Recent Findings in Boolean Techniques, pp. 177–191. Springer, Berlin (2021)
10. Burch, J.R.: Using BDDs to verify multipliers. In: DAC, pp. 408–412 (1991)
11. Kumar, J., Srivastava, A., Fujita, M.: Formal analysis of integer multipliers by building binary decision diagram of adder trees. In: ISQED, pp. 58–63. IEEE, New York (2022)
12. ichi Minato, S.: Streaming BDD manipulation. IEEE Trans. Comput. **51**(5), 474–485 (2002)
13. Shiple, T.R., Brayton, R.K., Sangiovanni-vincentelli, A.L.: Computing Boolean expressions with OBDDs (1993)
14. Jain, J., Narayan, A., Sangiovanni-Vincentelli, A., Coelho, C., Brayton, R.K., Khatri, S.P., Fujita, M.: Decomposition techniques for efficient ROBDD construction. In: Formal Methods in Computer-Aided Design, pp. 419–434 (1996)
15. Hett, A., Drechsler, R., Becker, B.: MORE: an alternative implementation of BDD packages by multi-operand synthesis. In: Proceedings EURO-DAC'96. European Design Automation Conference with EURO-VHDL'96 and Exhibition, pp. 164–169 (1996)
16. Beyer, D.: Improvements in bdd-based reachability analysis of timed automata. In: FME, pp. 318–343. Springer, Berlin (2001)
17. Beyer, D., Lewerentz, C., Noack, A.: Rabbit: A tool for bdd-based verification of real-time systems. In: International Conference on Computer Aided Verification, pp. 122–125. Springer, Berlin (2003)
18. Somenzi, F.: CUDD: CU decision diagram package release 2.7.0 (2018). https://github.com/ivmai/cudd
19. Wolf, C., Glaser, J., Kepler, J.: Yosys-a free verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip) (2013)