BDD complexity analysis of multiplier circuits

I. Grudenic and N. Bogunovic
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Faculty of electrical engineering and computing, University of Zagreb
Unska 3, Zagreb, Croatia
Phone: 01-6129-999 int. 548 Fax: 01-6129-653 E-mail: igor.grudenic@fer.hr

Abstract – The equivalence of Boolean functions is an important issue in hardware design. In order to check the equivalence of two combinatorial circuits SAT solvers and BDD based algorithms are used. For complex hardware designs efficient BDD manipulation algorithms must be employed. There is a number of industrial combinatorial circuitry for which BDD representation explodes exponentially in the number of inputs. An example of such a circuitry is multiplier that is known to have no efficient BDD representation regardless of the employed variable ordering. In this paper we analyze some important properties of the BDD representation of multipliers, as well as the complexity of the involved computations.

I. INTRODUCTION

The growth of the integrated circuits components (in the number of elements per chip) makes these designs more error prone. In order to minimize design flaws efficient formal verification methods must be used.

In the verification of combinatorial (stateless) circuits equivalence checking is the most important formal verification algorithm. Many equivalence checking algorithms [1][2] rely on the Binary Decision Diagrams (BDDs) as its central data structure. Binary decisions diagrams are known to be efficient in representing common patterns, like digital adders, in combinatorial circuit design.

Since BDDs represent logical function the representation cannot be efficient for all the functions because Boolean function representation is known to be NP complete. Multiplier is an important example of combinatorial circuitry with no efficient BDD representation. It is proved in [3] that at least one of the multiplier outputs has its BDD representation exponential in the number of the operand bits. This is true regardless of the variable ordering used while creating BDD outputs. Even more, if each of the outputs is presented using different variable ordering (which is not feasible for computation) the size of at least one of the output would be exponential in the number of variables.

In this paper we analyze some important properties of multiplier BDD representation such as: the size of the representation, computation issues and variable ordering. In section II we define BDDs and its basic properties. Multiplier circuit is described in section III. The size of the multiplier BDD representation and complexity of the computation involved are outlined in section IV. Different variable orderings and their effect on the BDD properties of the multipliers are discussed in section V.

II. BDD DEFINITION

BDD is a directed acyclic graph used to represent Boolean functions. The nodes in the graph correspond to Boolean variables, while the arcs define evaluation of the variables. The paths in the graph therefore stand for different interpretations of the Boolean function. The sinks of the graph are equivalents of true and false value of the function. BDDs for the outputs of the full adder are given in Fig. 1.

$x \oplus y \oplus z$ $\qquad$ $x \cdot y + x \cdot z + y \cdot z$

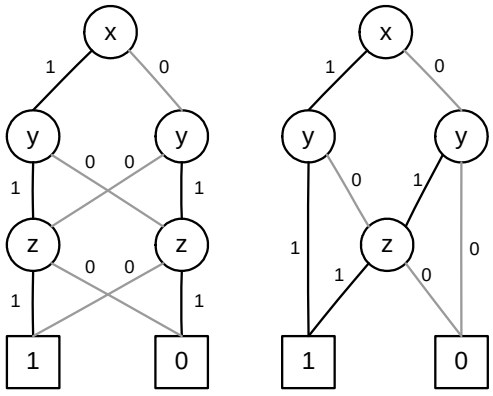

Fig. 1. BDD representation of full adder

The lower bit of the result, presented by the left BDD, evaluates to *true* if the odd number of inputs evaluates to *true.* The right BDD represents the higher bit of the results which is evaluated to *true* if any pair of the inputs evaluates to *true*.

BDDs are canonical, which is very important for equivalence checking, if the following three conditions are satisfied:

1. variable ordering is the same for all the functions,
2. there is no isomorphic subgraphs in the diagram
3. node collapses if both arcs are pointing to the same child

The basis for the efficient BDD manipulation is the following recursive expression:

$$f\langle op\rangle g = x \cdot \left(f_x \langle op\rangle g_x\right) + \bar{x} \cdot \left(f_{\bar{x}} \langle op\rangle g_{\bar{x}}\right)$$

where f and g are BDDs representing functions f and g. The cofactors f_x and $f_{\bar{x}}$ are defined as:

$$f_{\overline{x_i}} = f(x_1,...,x_{i-1},0,x_{i+1},...,x_n)$$
$$f_{x_i} = f(x_1,...,x_{i-1},1,x_{i+1},...,x_n)$$

and can be easily obtained from BDD f if the x_i is the top variable.

III. MULTIPLIER DEFINITION

Multiplier is a combinatorial circuit that is consisted of *2n* inputs and *2n* outputs where *n* denotes the number of the operand bits. The inputs are bit vectors defined as:

$$x = [x_1 ... x_n],\ y = [y_1 ... y_n].$$

Bits x_n and y_n are the highest level bits in the x and the y bit vectors, respectively. The result bit vector is

$$z = [z_1 .. z_{2n}]$$

In order to define the outputs in term of the inputs, the following functions that represent outputs of the full adder are used:

$$sum(x, y, z) = x \oplus y \oplus z$$

$$carry(x, y, z) = x \cdot y + x \cdot z + y \cdot z$$

in which the operators $\cdot, \oplus$ and + denote Boolean operators *and, xor* and *or* respectively. Two dimensional bit vectors *s* and *c* are needed to express the output vector *z*. Dimension of the vector *s* is *[n,n+1]* and dimension of the vector *c* is *[n-1,n+1]*. The values of vectors *s* and *c* are given by the following equations:

$$s_{k,i} = \begin{Bmatrix} sum(y_k x_i, 0) & k = 1, i \in [1..n] \\ sum(y_k x_i, sum_{k-1,i+1}, c_{k,i-1}) & k \in [2..n], i \in [1..n] \\ c_{k,i} & k \in [2..n], i = n+1 \end{Bmatrix}$$

$$c_{k,i} = \begin{Bmatrix} 0 & k \in [2..n], i = 0 \\ carry(y_k x_i, sum_{k-1,i+1}, c_{k,i-1}) & k \in [2..n], i \in [1..n] \end{Bmatrix}$$

The order of calculation is determined as an iteration over *k* in the interval *[1,n],* and for the each *k*, iteration for *i* is made over the values *[0,n]*. The values for $c_{k,i-1}$ are calculated before the values $s_{k,i}$.

The output vector *z* is expressed directly from the vector *s* in the following expression:

$$z_i = \begin{Bmatrix} s_{i,i} & i \in [1, n-1] \\ s_{n,i-n+1} & i \in [n, 2 \cdot n] \end{Bmatrix}$$

It is observable that the cover of the upper half of the output vector bits is the entire input set.

IV. MULTIPLIER SIZE AND COMPUTATION PROPERTIES

It is shown in [3] that multipliers have no efficient BDD representation. Using the variable ordering:

$$x_n < x_{n-1} < ... < x_1 < x_0 < y_n < y_{n-1} < .. < y_1 < y_0 \qquad (1)$$

we measured time and space complexity of the circuit for the $n \in [1,14]$. The results are given in the Fig. 2. It can be noticed that the size and the number of operations grows exponentially in *n*. The number of operations is larger than the number of nodes in the result because of the reduction of the isomorphic subgraphs and recalculation of already computed results due to the limited cache size. The ratio between the number of operations and the size of the multipliers BDDs grows linearly with the number of multiplier bits and reaches the factor of four for the 14 bit multiplier. This ratio is obtained with the start cache size

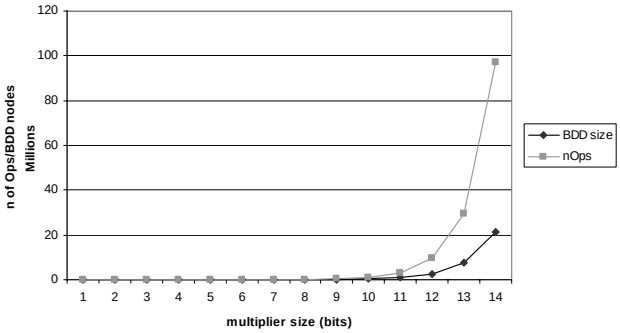

Fig. 2. BDD multiplier time-space complexity

of 4 million entries and the maximum cache size of 16 million entries. The cache size is doubled whenever a threshold is reached. The threshold is set to the point in which 80% of the cache is full.

In order to measure exponential growth of multiplier BDD representation we define factors of growth for both the size *sf* and the number of operations *of* as:

$$sf_i = \frac{size_i}{size_{i-1}} \qquad of_i = \frac{nOpsi_i}{nOps_{i-1}} \qquad (2)$$

where $size_i$ denotes the BDD representation size of the multiplier with *i* bit wide operands. The number of BDD operations processed in order to build the multiplier is expressed as $nOps_i$. Our experimental results show that the size growth factor sf_i converges to the value of 2.88 (*i=14*), while the factor of_i settles at slightly higher value of 3.29 (*i=14*).

Beside the analysis of the time and space complexity of multipliers, we analyzed the same complexities inside the multiplier. Since *n*-bit multiplier is built using *2n* Boolean variables, activity was measured at each of the variable indexes. As shown in Fig. 3. three values were measured: the number of BDD nodes in the unique table and the number of operations with as well as without the cache system enabled. The values are obtained for 10 bit multiplier.

When the number of BDD nodes is observed, it is evident that the majority of data is spread across a few neighbor variable indexes (interval *[13, 15]* in Fig. 3) . It is apparent that this is followed by the number of operations issued at this variable level, when considering the computation made with the cache turned on. It can be also observed, as in [7] that the peak of the computation is one variable index subsequent to the number of BDD nodes peak.

In the scenario where the cache system is not employed,

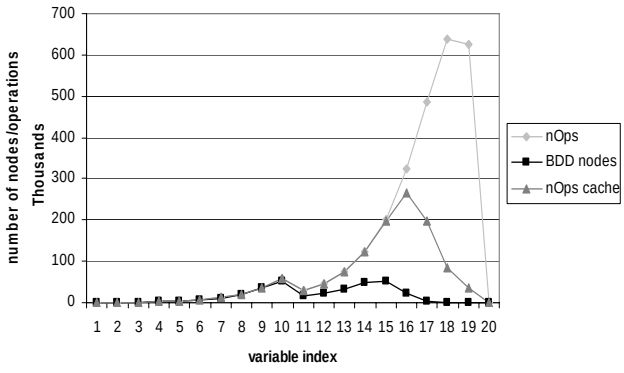

Fig. 3. 10 bit multiplier BDD properties

the largest amount of operations is processed at the lowest variable indexes (interval *[17,19]* in Fig. 3). Since all the multiplier outputs have these indexes (as noted in section III) in their cover, the operations issued at the lowest BDD levels are repeated very frequently. The operations involving higher variable indexes are computed less often because BDD nodes at these levels are not referenced as many times.

There is no upper bound for the number of operations at a given variable index whether the cache is used or not. Theoretically, if infinite cache is available [6], the number of BDD operations computed is limited by the size of the operands. The number of BDD nodes at the given level is generally limited only by the expression:

$$nOfNodes = 2^{2^{n+1-i}} \quad (3)$$

where i is the level, and n denotes the total number of variable indexes. The expression (3) expresses the number of possible Boolean functions over the cover of $n+1-i$ variables.

V. VARIABLE ORDERING EFFECT ON MULTIPLER BDD CHARACTERISTICS

Variable ordering is a prerequisite for the canonicity of the BDDs. The choice of variable ordering can be determined before the BDD creation (static variable ordering) or can be changed at any point of the computation (dynamic variable ordering) [4]. Dynamic variable ordering is based on the locality of the swap operation. When two adjacent variable indexes are swapped it is possible to keep the same data structures in which BDD nodes are stored in order to avoid updating of references [5].

Although there is no efficient variable ordering which can be used to decrease exponential complexity of the multipliers BDD representation, we tried to analyze the properties of the different orderings. Trying all the variable orderings would not be feasible even for small multipliers since the amount of orderings is given by the equation:

$$nOfpermuta\ tions = \frac{(2n)!}{2}$$

where n is the number of bits in the operands. The number of all the input bit permutations is divided by the factor of two since multiplication is a commutative operation. Because of the large number of different variable orderings we tested multipliers with only a limited number of ordering permutations (*nOrderings=1000)*.

In order to ensure a proper distribution of permutations over the set of all the permutations we used the heuristic algorithm presented in Fig. 4. The vector *a* contains pointers to BDD nodes representing variable indexes, while *n* represents the number of indexes. The function *random(i,n)* returns random number from the closed interval *[i,n]*, while the function *Swap(x,y)* swaps the contents of objects *x* and *y*. The creation of random numbers is done by the linear congruential generator from the standard *C* language library. We tested the random permutation generator by producing large numbers of permutations on only few variable indexes. The result was that all the possible permutations were generated and the distribution of occurences for each permutation was uniform.

```
Random_Permutation(vector a, int n)
Begin
      For i=1 to n-1 do
      Begin
            swapIndex=random(i,n)
            Swap(a[i],a[swapIndex])
      End
End.
```

Fig. 4. Random permutation algorithm

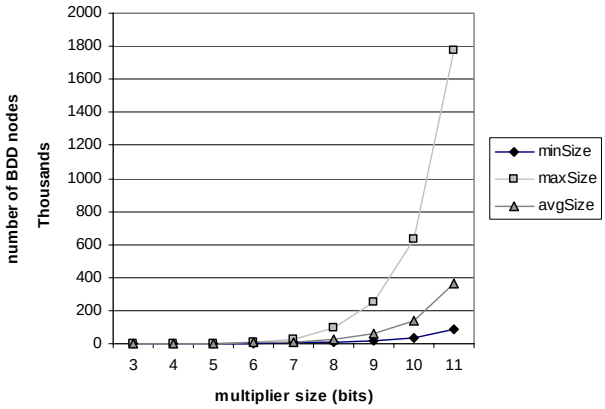

Fig. 5. Variable ordering impact on BDD size

For each of the generated permutations and given multiplier we measured the size of the resulting BDD and the number of generated operations. In addition, we noted the BDD sizes of all the multipliers output, and we specifically outlined the output whose BDD representation contains the greatest number of nodes.

The results of the size measurements are presented in Fig. 5. The curve *maxSize* denotes the highest number of BDD nodes generated during the computation of all tested variable ordering permutations for the given multiplier. The curve *minSize* denotes the minimal number of BDD nodes given in the described manner, while *avgSize* is the average multiplier BDD size for all the computed orderings.

It can be observed from Fig. 5. that in all three scenarios (minimal, maximal and average) the size of BDD representation is exponential in the number of multiplier bits. The sf_i factor (2) is different for the each curve. The factor for *minSize* range of values (for each multiplier) averages at *2,06*. The average value of the *avgSize* and the *maxSize* factor is *2,43* and *3,10* respectively.

When comparing these factors with the factors obtained for the variable ordering given in (1), both *minSize* and *avgSize* factors outperform it. The worst case variable ordering scenario has a factor *3,10* which is only slightly higher than the one stated in the previous section. This leads to the conclusion that the variable ordering given in (1) is close to the worst experimentally measured variable ordering. It should be noted that sf_i factor values for random variable orderings do not converge but are rather slowly growing with the number of multiplier bits. That could be due to the small number of calculated permutations (considering the number of possible orderings). As the number of bits becomes higher, the

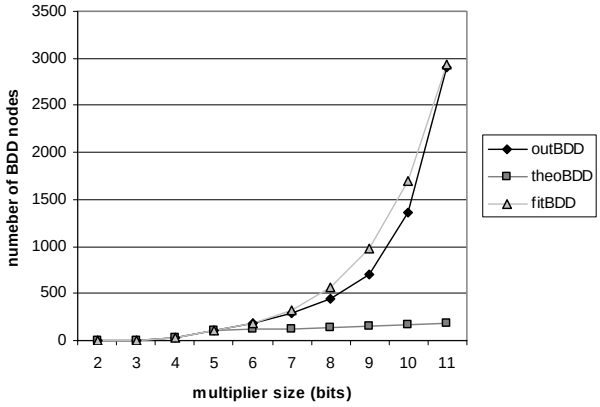

Fig. 6. The size of the largest multiplier output

percentage of tried orderings is lower, which results in lower probability in finding both the best and the worst variable ordering.

Experimental results for the number of BDD operations computed for different variable orderings are perfectly matched with the numbers gathered while analyzing the size of the resulting BDD. From the computational complexity analysis it can be noticed that the variable ordering given in (1) is very close to the worst generated ordering.

The third series of analysis focused on finding the multiplier output (among all the outputs) whose BDD representation is the largest, but for the permutation that minimizes such an output. The idea behind this is in the theoretical lower bound [3] for the complexity on the at least one of the multiplier outputs. This complexity (expressed in terms of the multiplier size) evaluates to $\Omega(1,09^n)$, where n is the number of multiplier bits. The output with largest representation is chosen as the worst case scenario since exact data could be gathered only with all the variable orderings available.

The comparison between theoretical and experimentally tested bounds is given in Fig. 6. The series of values *theoBDD* represents the exponential theoretical complexity $\Omega(1,09^n)$. The results for the output with the largest BDD representation for the best found variable ordering is denoted by *outBDD*. In order to express experimentally obtained results in terms of exponential size complexity we find the complexity of $\Omega(1,74^n)$ to be an approximation of the *outBDD* values. This complexity is shown in Fig. 6. as the *fitBDD* set of values. The base of the exponential function is taken as the average ratio between all the adjacent *outBDD* values. We believe that the misalignment of the measured values and the exponential function is due to the small number of variable orderings tested.

It can be concluded that the gap between the theoretical lower bound and measured values is not negligible. In addition, it should be noted that the variable ordering can improve BDD manipulation performance and therefore should be employed while representing multipliers.

VI. CONCLUSION

In this paper we analyzed BDD representation of digital multiplier, a common industrial design. BDDs are shown to be efficient representation for large number of commonly used combinatorial circuits, but not for the multipliers.

Our BDD analysis included BDD size and computational complexity inside the multiplier and among multipliers of different size. We also showed the impact of cache system on the BDD computation.

Since variable ordering is an important BDD issue we tested the multiplier against the number of randomly generated orderings. It is discovered that the initial variable ordering was far from the best ordering, and that the variable ordering algorithms should be employed even on the designs with exponential BDD complexity.

The comparison of theoretical lower bound on the size of the multiplier output and the experimentally obtained lower bound revealed a large misalignment of these two values. This leads to the conclusion that the best variable orderings are not found in our random search.

It is obvious that BDDs are not the best solution for the representation of circuitry which are proved to be exponentially large. Perhaps it is possible to achieve a speedup in manipulation of the BDDs by using distributed computation environment.

REFERENCES

[1] C. A. J. van Eijk, “A BDD-based verification engine for combinational equivalence checking“, *Proc. CSSP-97, 8th Annual ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing*, Mierlo, Netherlands, 27-28 November 1997, p. 155-162.

[2] A. Kuehlmann and F. Krohm, “Equivalence Checking using Cuts and Heaps”, Proc. 34th ACM/IEEE Design Automation Conference, pp. 263-268, 1997

[3] Randal E. Bryant, “On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication”, *IEEE Trans. on Computers*, vol. 40, no. 2, p. 205, February 1991.

[4] O. Grumberg, S. Livne, S. Markowitch, “Learning to Order BDD Variables in Verification”, *Journal of Artificial Intelligence Research 18(2003)* p. 83-116, Jan 2003.

[5] R. Rudell, “Dynamic Variable Ordering for Ordered Binary Decision Diagrams”, *In Proceedings of the International Conference on Computer Aided Design*, p. 42-47, 1993.

[6] K.S. Brace, R.L. Rudell, R.E. Bryant, “Efficient implementation of a BDD package”, *In Proc. of the 27th Design Automation Conference*, Orlando, USA. p. 40-45., June 1990.

[7] K. Milvang-Jensen and A. J. Hu, “BDDNow: A Parallel BDD Package”, *In Proceedings of Formal Methods in Computer Aided Design*, p. 501, 1998.